МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

В. І. Панченко, Г. В. Гейко, М. І. Главчев, В. В. Скородєлов

ОПЕРАЦІЙНІ СИСТЕМИ. УПРАВЛІННЯ ПРОЦЕСАМИ

Навчальний посібник
для студентів спеціальності
123 «Комп'ютерна інженерія»
денної та заочної форм навчання

Затверджено
редакційно-видавничою
радою НТУ «ХПІ»,
протокол № 1 від 13.02.2025 р.

Харків
НТУ «ХПІ»
2025

УДК 004.451
О-60

Рецензенти:
Л. Чорногор, проф., д-р фіз.-мат. наук, зав. каф. космічної радіофізики факультету радіофізики, біомедичної електроніки та комп'ютерних систем, ХНУ ім. В. Н. Каразіна;
С. Шульга, проф., д-р фіз.-мат. наук, декан факультету радіофізики, біомедичної електроніки та комп'ютерних систем, ХНУ ім. В. Н. Каразіна;
М. Резуненко, доц., канд. техн. наук, зав. каф. вищої математики та фізики, Український державний університет залізничного транспорту

О-60 Операційні системи. Управління процесами : навчальний посібник для студентів спеціальності 123 «Комп'ютерна інженерія» денної та заочної форм навчання / В. І. Панченко, Г. В. Гейко, М. І. Главчев, В. В. Скородєлов. – Харків : НТУ «ХПІ», 2025. – 350 с.

ISBN 978-617-05-0543-9

У навчальному посібнику розглянуто основні теоретичні та практичні відомості з архітектури операційних систем та механізмів управління процесами: розглянуто питання розвитку, архітектурних особливостей побудови операційних систем, організації управління процесами та планування процесорного часу. Наводяться всі необхідні терміни та визначення. Навчальний посібник містить опис реалізацій низки класичних та сучасних операційних систем. Виклад супроводжується великою кількістю ілюстративного матеріалу, корисного для застосування знань у реальних системах.

Призначено для студентів спеціальності 123 «Комп'ютерна інженерія» денної та заочної форм навчання за навчальними дисциплінами «Системне програмне забезпечення» та «Операційні системи», а також може бути корисним при курсовому та дипломному проєктуванні.

Іл. 84. Табл. 23. Бібліогр. 46 назв.

УДК 004.451
ISBN 978-617-05-0543-9

ЗМІСТ

Вступ 6

1. Основні поняття та визначення 9

1.1. Системне програмне забезпечення 9

1.2. Визначення та класифікація ОС 13

1.3. Коротка історія розвитку ОС 22

1.4. Складові частини ОС 49

1.5. Архітектура ОС 53

1.5.1. ОС з монолітною структурою 63

1.5.2. ОС з багаторівневою архітектурою 70

1.5.3. ОС з мікроядерною архітектурою 76

1.5.4. Гібридні ядра 83

1.5.5. ОС на базі екзоядра 97

1.5.6. ОС на базі наноядра 100

1.5.7. Віртуальні машини 103

Контрольні запитання 114

2. Процеси 116

2.1. Основні визначення 116

2.2. Модель процеса 121

2.3. Стани процеса 127

2.3.1. Модель процеса з двома станами 128

2.3.2. Модель процеса з п’ятьма станами 129

2.3.3. Модель процеса з сьома станами 136

2.4. Управління процесами 139

2.4.1. Створення і завершення процесів 139

2.4.2. Механізми переривання виконання процесів 143

2.4.3. Механізм системних викликів 146
2.4.4. Підтримка життєвого циклу процесів 161
2.5. Нитки 187
2.5.1. Багатониткова модель процеса 190
2.5.2. Програмування ниток 199
2.6. Стани процесів (ниток) у деяких ОС 208
2.6.1. Стани ниток в ОС Windows 208
2.6.2. Стани процесів в ОС UNIX 210
2.6.3. Стани процесів в ОС Linux 213
2.6.4. Стани ниток в ОС Solaris 217
Контрольні запитання 219
3. Управління процесорним часом 221
3.1. Планувальник та диспетчер процесорного часу 222
3.2. Критерії ефективності дисциплін планування 225
3.3. Дисципліни планування процесорного часу 229
3.3.1. Алгоритм FCFS 230
3.3.2. Алгоритм RR 234
3.3.3. Алгоритми пріоритетного планування 239
3.3.4. Алгоритм SPN 241
3.3.5. Алгоритм SRTF 245
3.3.6. Алгоритм HRRN 247
3.3.7. Алгоритм SRR 250
3.3.8. Алгоритм VRR 253
3.3.9. Алгоритм MLQ 256
3.3.10. Алгоритм MLFQ 258
3.3.11. Алгоритм лотерейного планування 264
3.3.12. Алгоритм гарантованого планування 266

3.3.13. Алгоритм справедливого планування FSS 267
3.3.14. Традиційне планування UNIX 274
3.3.15. Статичний алгоритм планування RMS 277
3.3.16. Динамічний алгоритм планування EDF 284
3.3.17. Алгоритм LLF 288
3.3.18. Перший алгоритм планування Linux 292
3.3.19. Алгоритм планування O(1) Linux 297
3.3.20. Алгоритми RSS і RSDS 301
3.3.21. Алгоритм повністю справедливого планування CFS 305
3.3.22. Алгоритми BFS і MuQSS 312
3.3.23. Алгоритм EEVDF 319
3.3.24. Альтернативні сучасні алгоритми планування процесорного часу в ОС Linux 322
3.3.25. Алгоритм планування ULE для FreeBSD 324
3.3.26. Планувальник Zircon Fair Scheduler 328
3.3.27. Планувальник Windows 332
3.4. Особливості планування в багатопроцесорних системах 337
Контрольні запитання 344
Список літератури 346

ВСТУП

Операційні системи є найбільшою складовою системного програмного забезпечення комп'ютерних систем, вони виконують функцію посередника між апаратним забезпеченням і користувачами. Операційні системи задіяні у всіх процесах розробки програмного забезпечення, адміністрування систем, проєктування архітектури та забезпечення безпеки.

Розуміння архітектури операційних систем, принципів їхнього функціонування дозволяє оптимізувати власні розробки, адаптувати їх під специфічні вимоги бізнесу і вирішувати складні проблеми, пов'язані з продуктивністю і стабільністю.

Операційні системи управляють всіма компонентами обчислювальних систем: від апаратних частин, таких як процесор і пам'ять, до програмного забезпечення, що працює в користувацькому просторі. Саме завдяки операційній системі можна писати застосунки, не заглиблюючись у деталі побудови апаратних платформ, а користувачі отримують можливість взаємодіяти з комп'ютерами через зручний та зрозумілий інтерфейс.

Сучасні операційні системи розроблені та використовуються для різноманітних пристроїв: серверів, суперкомп'ютерів, персональних комп'ютерів, смартфонів, вбудованих систем тощо.

Маючи на сучасному етапі розвитку комп'ютерної техніки величезну кількість операційних систем, виникає потреба розуміння їх функціонування, побудови, апаратної та програмної сумісності, призначення тощо, для професійного користування ними, розробки

системного та прикладного програмного забезпечення, налаштування та адміністрування.

Навчальний посібник призначено для студентів, які навчаються за спеціальністю «Комп'ютерна інженерія», але може бути корисним як професіоналам та розробникам програмного забезпечення, так і користувачам комп'ютерної техніки. Матеріал у начальному посібнику поділений на три розділи, кожен з яких присвячено детальному аналізу різних аспектів операційних систем.

У першому розділі наводяться основні терміни та поняття з теорії системного програмного забезпечення, дається визначення та класифікація операційних систем, ресурсів, програм та процесів. Також наведено коротку історію розвитку операційних систем з прикладами конкретних систем та описом їх особливостей. Приклади систем супроводжуються вказівкою версій та посиланнями на ресурси в мережі Інтернет. Для базового ознайомлення з побудовою операційних систем наведені основні складові частини систем та задачі, які ними вирішуються. Наводиться розширений опис сучасних вимог до операційних систем та основні архітектурні напрямки їх побудови з аналізом відповідності до наведених вимог. Розуміння архітектурних концепцій дозволяє професіоналам приймати обґрунтовані рішення при виборі операційних систем для специфічних проєктів. Для кожного архітектурного напрямку наводиться кілька прикладів структур реальних систем.

Другий розділ присвячено процесам та управлінню ними. Управління процесами є однією з найбільш важливих функцій операційної системи. Розглянуто моделі процесів на прикладах реальних систем. Виконано послідовний опис станів процесів від простої абстрактної до найбільш детальної моделі. Описано основні режими роботи операційних систем, механізми перемикання між режимами. Особливу увагу приділено опису реалізації механізму системних викликів у сучасних операційних системах. Розглянуто життєвий цикл процесів та його підтримку: описані основні системні виклики для створення, завершення, очікування, завантаження образу виконуваного файла, зміни пріоритету процеса. Детально описано моделі багатониткових процесів та питання програмування ниток.

Наприкінці розділу наведені приклади станів процесів та ниток у кількох реальних операційних системах.

Третій розділ присвячено одному з основних завдань операційних систем – управлінню чергами процесів, які очікують виконання, та розподілу процесорного часу між ними. Розглянуті базові принципи роботи планувальника та диспетчера процесорного часу, критерії оцінки ефективності алгоритмів планування. Описані основні теоретичні алгоритми планування процесорного часу, наведені приклади їх поведінки та розрахунки характеристик. Розглянуті особливості планування в операційних системах реального часу. Наведені описи реалізацій алгоритмів у конкретних класичних та сучасних операційних системах. Професійні системні адміністратори та розробники повинні розуміти ці алгоритми, щоб вміти оцінити, як вибір того чи іншого підходу може вплинути на продуктивність системи в умовах різних навантажень. В останній частині розділу розглянуті питання планування в багатопроцесорних системах.

Кожен із розділів містить контрольні запитання для закріплення матеріалу.

Усі посилання на ресурси в мережі Інтернет, версії та дати випусків останніх версій наведених операційних систем перевірені та актуальні станом на жовтень 2024 року.

Під час підготовки цього видання були зібрані та структуровані ключові матеріали, які можуть стати для читачів корисним інструментом у навчанні. Якщо у Вас виникнуть питання, уточнення або пропозиції щодо змісту посібника, звертайтеся за адресою: panchenko.vladimir@gmail.com.

1. ОСНОВНІ ПОНЯТТЯ ТА ВИЗНАЧЕННЯ

1.1. Системне програмне забезпечення

Роботу обчислювальної системи неможливо уявити без програмного забезпечення. Програмне забезпечення (ПЗ) (англ. Software) – це комплекс програм (англ. Program), що забезпечує обробку або передачу даних та призначений для багаторазового використання. Згідно стандарту ISO/IEC/IEEE 12207:2017 «Systems and software engineering – Software Life Cycle Processes» ПЗ – комп'ютерні програми, процедури, а також документація й дані, що з ними асоційовані, які стосуються функціонування комп'ютерної системи.

Залежно від функцій, що виконує ПЗ, його прийнято поділяти на системне, прикладне та інструментальне. Інструментальне ПЗ у багатьох випадках розглядається як частина системного ПЗ – тоді розподіл ПЗ виконується лише на дві складові. Розглянемо докладніше вказані складові.

Прикладне ПЗ (англ. Application software) складається з окремих прикладних програм (використовуються автономно), пакетів прикладних програм (проблемно-орієнтованих, загального призначення, інтегрованих пакетів) і автоматизованих систем, які створені на основі пакетів прикладних програм. Основне призначення прикладного ПЗ – вирішення різноманітних задач користувачів. Як приклад прикладного ПЗ можна назвати бухгалтерські, фінансові та експертні системи, системи автоматизованого проєктування (CAD), редактори (текстові, графічні) та ін.

Системне ПЗ (англ. System software) – це сукупність програм і програмних комплексів, призначених для забезпечення роботи обчислювальної системи та мережі. Системне ПЗ повинно забезпечувати ефективне управління компонентами обчислювальної системи, такими як процесор, оперативна пам'ять, канали введення-виведення, мережне устаткування. Системне ПЗ виконує роль інтерфейсу між апаратурою та прикладним ПЗ. Серед задач системного ПЗ можна виділити: створення операційного середовища функціонування інших програм; забезпечення надійної і ефективної роботи обчислювальної системи та мережі;

діагностика і профілактика апаратури обчислювальної системи та мережі; виконання допоміжних технологічних процесів (копіювання, архівація, відновлення даних) тощо.

На відміну від прикладного ПЗ, системне ПЗ не вирішує конкретні прикладні задачі, а лише забезпечує роботу інших програм, управляє апаратними ресурсами обчислювальної системи тощо.

До системного ПЗ відносяться:

♦ операційні системи (ОС) (англ. Operating System, OS);

♦ інтерфейсні оболонки (англ. Shell) для організації взаємодії користувачів з ОС (для забезпечення більш слушного й наочного засобу взаємодії);

♦ операційні середовища (англ. Operating environment) (сукупності програм, що забезпечують можливість управляти обчислювальними процесами і файлами). За допомогою операційних середовищ можна, наприклад, управляти процесами і файлами в стандартній ОС засобами графічного користувацького інтерфейсу (GNOME, KDE тощо);

♦ драйвери (англ. Driver) (програми, призначені для управління периферійними пристроями);

♦ утиліти (англ. Utility або Tool) (допоміжні або службові програми, які надають користувачу ряд додаткових послуг). Наприклад, диспетчери файлів або файлові менеджери, засоби динамічного стиску даних, засоби перегляду та програвання, засоби діагностування, засоби контролю, засоби комунікацій для організації обміну інформацією, засоби забезпечення безпеки.

Необхідно відмітити, що частина утиліт є складовою частиною ОС, а частина – автономна.

Розробкою, експлуатацією та супроводженням системного ПЗ займаються системні програмісти. Місце системного ПЗ в обчислювальній системі наведено на рис. 1.1, з якого видно, що найважливішим компонентом системного ПЗ є ОС – необхідне доповнення апаратних засобів.

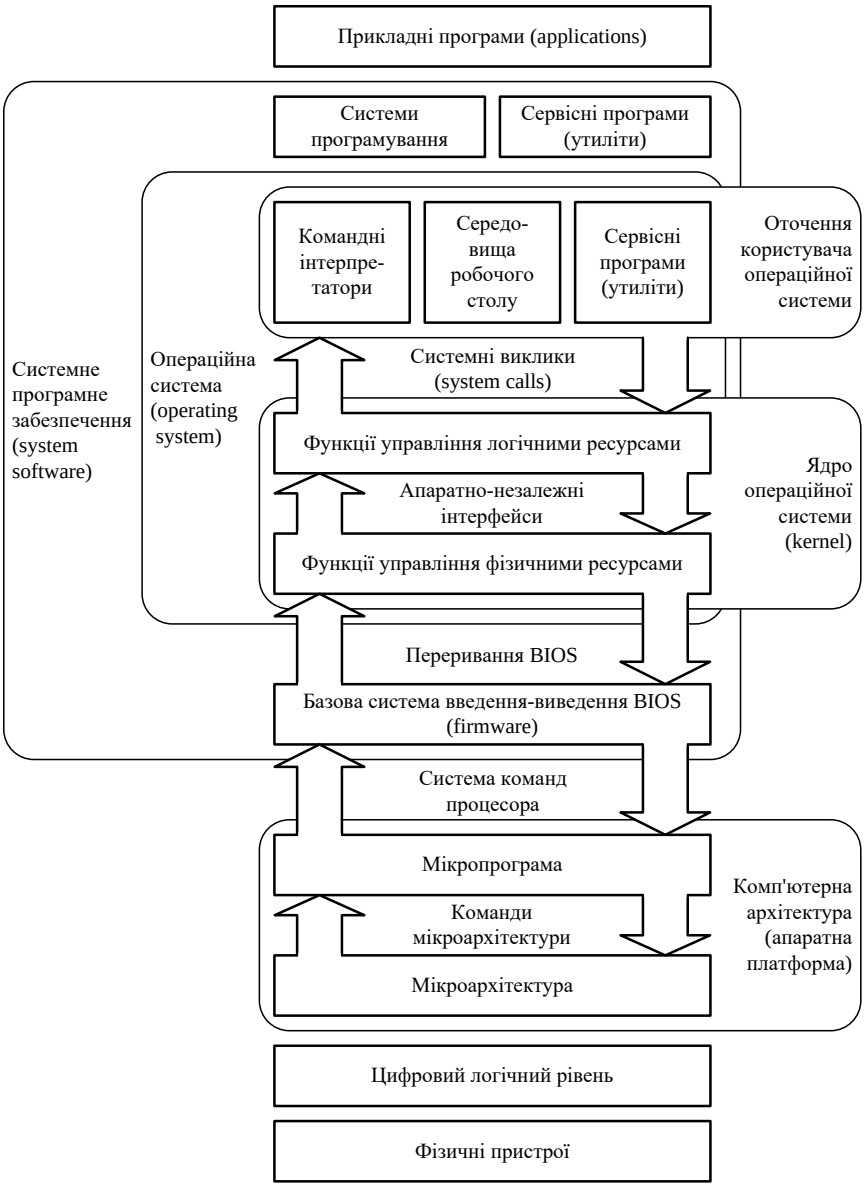


Рисунок 1.1 – Структура обчислювальної системи

Інструментальне ПЗ (системи програмування) (англ. Software tools) забезпечує розробку нових програм мовами програмування.

До складу інструментального ПЗ входять наступні компоненти:

♦ текстові редактори для створення, редагування, перегляду, пошуку фрагментів тексту, друку файлів з вихідними текстами програм. Часто текстові редактори дозволяють виділяти синтаксичні конструкції різними кольорами, виконувати автодоповнення коду та багато іншого. Приклади текстових редакторів: Sublime Text, Atom, Notepad++ тощо;

♦ спеціалізовані редактори вихідних текстів – аналоги текстових редакторів, але можуть бути вбудованими в інтегроване середовище розробки (англ. IDE – Integrated Development Environment);

♦ транслятори (англ. Translator) для виконання трансляції програми;

♦ компілятори (англ. Compiler) для переводу вихідного тексту програми в проміжний об’єктний код. Приклади компіляторів: GCC, Clang, Microsoft Visual C++;

♦ інтерпретатори (англ. Interpreter) для аналізу команд або операторів програми і для їх виконання. Інтерпретатори виконують вихідний код програми мовою програмування, переводячи його в машинний код під час виконання програми. Інтерпретатори не вимагають попередньої компіляції вихідного коду, що дає змогу швидше отримати результат. Приклади інтерпретаторів: Python, Ruby, Perl та ін.;

♦ редактори зв’язків (компонувальники) (англ. Linker, Linkage editor) для зв’язування об’єктних модулів і формування модуля для виконання;

♦ препроцесори (англ. Preprocessor) вихідних текстів для попередньої обробки тексту програми перед компіляцією;

♦ налагоджувачі (зневаджувачі) (англ. Debugger) для пошуку помилок у програмі;

♦ бібліотеки підпрограм, що використовується для розробки ПЗ. Вони містять функції та класи, які можуть бути викликані програмним кодом. Приклади бібліотек: Boost, Qt, .NET Framework;

♦ засоби візуального програмування для автоматизації в середовищах швидкого проєктування;

♦ системи керування версіями (англ. Source Code Management, SCM) для контролю версій вихідного коду програми: дозволяють відстежувати зміни в коді, створювати гілки розробки, поєднувати зміни з різних гілок та багато іншого. Приклади систем керування версіями: Git, SVN, Mercurial та ін.

IDE є комплексним програмним забезпеченням, що поєднує в собі редактор вихідного коду, компілятор, налагоджувач, а також безліч інших інструментів, необхідних для розробки ПЗ. Приклади таких систем: Visual Studio, Eclipse, NetBeans та ін.

Як видно, склад інструментального ПЗ має багато рис системного ПЗ, тому в багатьох випадках системи програмування вважаються частиною системного ПЗ.

1.2. Визначення та класифікація ОС

На даний час немає загальноприйнятого визначення ОС. Тому є певний смисл надати можливі визначення ОС, що дозволить зробити власні висновки.

ОС – це комплекс програм, що організує обчислювальний процес в обчислювальній системі. Це визначення є дуже загальним і не дає повного опису ОС. Під таке визначення можуть попасти і звичайні програми, і програми, які по суті є попередниками ОС – службові програми (завантажувачі та монітори), а також бібліотеки підпрограм. Службові програми мінімізують операції користувача з апаратурою, а бібліотеки дозволяють виключити багатократне програмування одних й тих же дій по виконанню обчислення математичних функцій, операцій введення-виведення тощо.

Наступне визначення ОС – це набір програм (звичайних і мікропрограм), які забезпечують інтерфейс між процесами (програмами на стадії виконання; більш детально процеси будуть розглянуті в другому розділі) користувача і апаратурою комп’ютера. ОС забезпечує користувачеві зручний інтерфейс з ПЗ та пристроями комп’ютера, а також виконує допоміжні дії, такі як копіювання, друк файлів тощо. ОС дозволяє

абстрагуватися від деталей реалізації апаратного забезпечення та постачає розробникам ПЗ мінімально необхідний набір функцій. Але насправді в ОС реалізовано щонайменше 2 інтерфейси: перший – між користувачем (програмами, запущеними на виконання) та ОС (інтерфейс процесів) і між ОС та апаратурою (інтерфейс апаратури) (рис. 1.2).

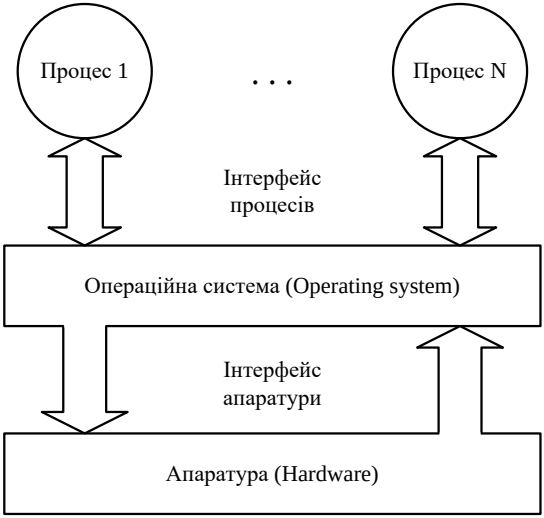


Рисунок 1.2 – ОС як інтерфейс між користувачем і апаратурою

Виконання функції абстрагування дозволяє розглядати ОС як віртуальну машину, для управління якої використовуються команди більш високого рівня, ніж для управління реальною обчислювальною системою. Для вирішення своїх задач користувач може обійтись без знання апаратної частини комп'ютера. Наприклад, при роботі з диском достатньо уявити його у вигляді деякого набору файлів, над якими можна виконувати операції відкриття, читання, запису, закриття тощо.

З точки зору управління, ОС можна описати як сукупність програм, що управляє апаратурою або іншими програмами. Тобто виконує розподіл ресурсів. Ресурси – це час, апаратні, програмні та інші засоби, які можуть бути надані обчислювальною системою або її окремими компонентами

обчислювальному процесу або користувачеві. Ресурси можуть бути первісними (апаратними) або вторинними (програмними, такими що створює сама ОС). Згідно з цим визначенням, ОС враховує та розподіляє ресурси: управляє центральним процесором, пам'яттю, введенням-виведенням, забезпечує виконання операцій над файлами, виступає в ролі диспетчера, який запускає на виконання прикладні програми, забезпечує взаємодію програм з пристроями і користувачем. З цієї точки зору ОС забезпечує безпеку: користувач не працює напряму з реальним пристроєм (той же диск), всі дії з ним виконує ОС, і, відповідно, користувач (його програма) не може пошкодити пристрій (або інший ресурс) та не може розпоряджатися ним без контролю з боку ОС.

У цілому можна зробити висновки, що ОС – це комплекс програм, який забезпечує:

- ♦ виконання інших програм;
- ♦ розподіл ресурсів;
- ♦ планування;
- ♦ введення-виведення даних;
- ♦ управління даними;
- ♦ взаємодію з користувачем.

Відповідно до вказаного вище можна сформулювати визначення ОС таким чином: ОС – це базовий комплекс системних програм, що розширює можливості обчислювальної системи, а також забезпечує управління її ресурсами, завантаження і виконання прикладних програм, взаємодію з користувачами.

Згідно з наведеним визначенням та самостійним аналізом сучасних ОС можна перелічити функції, які повинна виконувати ОС:

- ♦ завантаження програм в оперативну пам'ять та їх виконання;
- ♦ забезпечення стандартизованого доступу до периферійних пристроїв;
- ♦ розподіл оперативної пам'яті між процесами, організація віртуальної пам'яті;

♦ управління доступом до даних на зовнішніх носіях (жорсткий диск (англ. Hard Disk Drive, HDD), флеш-накопичувач (англ. Flash Drive), компакт-диск (CD, DVD, Blu-Ray Disk) та ін.);

♦ організація інтерфейсу користувача;

♦ підтримка мережі;

♦ паралельне або псевдопаралельне виконання задач;

♦ організація взаємодії між процесами;

♦ захист самої системи, а також даних і програм користувача від неправомірних дій інших користувачів або програм;

♦ організація розмежування прав доступу в багатокористувацькому режимі роботи.

Відповідно до перерахованих функцій можна визначити деякі ситуації, коли виникає необхідність у використанні ОС:

♦ збереження результатів роботи програм або обмін даними між програмами;

♦ виконання великих кількостей однотипних послідовностей машинних команд;

♦ захист даних і програм від несанкціонованого доступу;

♦ імітація паралельного виконання програм;

♦ надання можливості користувачу управляти ходом виконання програм.

Цей перелік можна розширювати достатньо довго, і на основі отриманих результатів зробити висновки про важливе місце ОС у системному ПЗ. Далі розгляд матеріалів стосовно системного ПЗ буде розглядатися в ракурсі вивчення основних принципів побудови та роботи ОС.

Залежно від задач, які вирішує конкретна ОС, вона може бути класифікована за ознаками:

♦ призначення;

♦ режим обробки;

♦ спосіб взаємодії користувача з системою;

♦ спосіб побудови.

Призначення ОС визначається ситуаціями, коли виникає необхідність у використанні ОС. Ці ситуації можуть бути поділені на:

♦ організація ефективних і надійних обчислень;

♦ створення різноманітних інтерфейсів для взаємодії з процесом обчислення і самою обчислювальною системою.

З таким розподілом ситуацій ОС можуть бути:

♦ загального призначення;

♦ спеціального призначення.

У свою чергу, ОС спеціального призначення поділяються на:

♦ ОС для переносних комп'ютерів і вбудованих систем;

♦ ОС для організації і ведення баз даних;

♦ ОС для вирішення задач реального часу та ін.

ОС також можуть бути розділені по режиму обробки задач:

♦ однопрограмний режим (наприклад, MS-DOS);

♦ мультипрограмний режим (наприклад, OS/2, UNIX, Windows).

Мультипрограмування (англ. Multiprogramming) – режим організації роботи обчислювальної системи, при якому одночасно виконується декілька програм, які поперемінно використовують одні й ті ж її ресурси. На однопроцесорній обчислювальній системі створюється видимість одночасного виконання декількох задач (основний ресурс – процесор використовується поперемінно різними процесами). Будь яка затримка у виконанні одного процеса використовується для виконання інших процесів.

Мультипрограмний і багатозадачний режими не є синонімами. Мультипрограмний режим забезпечує паралельне виконання декількох процесів, при цьому програміст, який створив програми, не повинен турбуватися про механізм організації їх паралельної роботи. Ці функції виконує ОС, яка розподіляє між процесами ресурси обчислювальної системи, забезпечує синхронізацію і взаємодію процесів. Багатозадачний режим передбачає, що організацію паралельного виконання і взаємодії процесів виконує програміст. Сучасні ОС реалізують і мультипрограмний і багатозадачний режим.

По числу одночасно працюючих користувачів ОС поділяються на однокористувацькі (однотермінальні) і багатокористувацькі (мультитермінальні).

У багатокористувацьких ОС з однією обчислювальною системою одночасно можуть робити декілька користувачів, кожний зі свого терміналу, при цьому для кожного користувача складається враження, що він працює на власній обчислювальній системі. Для організації мультитермінального доступу необхідна підтримка мультипрограмного режиму роботи обчислювальної системи.

У багатокористувацьких ОС існують засоби захисту інформації кожного користувача від несанкціонованого доступу інших користувачів. Більшість сучасних ОС є багатокористувацькими і забезпечують механізми розподілу повноважень.

З точки зору апаратури ОС поділяються на однопроцесорні і багатопроцесорні.

Однією з важливих властивостей ОС є наявність у ній засобів підтримки багатопроцесорної обробки даних. Багатопроцесорна обробка – організація обчислювального процесу в системах з кількома процесорами, при цьому декілька процесів можуть одночасно виконуватись на різних процесорах обчислювальної системи. Засоби багатопроцесорної обробки, наприклад, були реалізовані в таких сімействах ОС: OS/2, NetWare, Windows на базі ядра NT. Багатопроцесорні ОС встановлюються на серверах і потужних комп’ютерах. Багатопроцесорні ОС розділяють на симетричні й асиметричні. У симетричних ОС обробка повністю децентралізована, процес може бути виконаний на будь-якому процесорі. При цьому кожному з процесорів доступна вся пам’ять. В асиметричних ОС процесори нерівноправні. Зазвичай існує головний процесор і підлеглі, завантаження й характер роботи яких визначає головний процесор.

Ще однією з важливих ознак класифікації ОС є розподіл їх на локальні і мережні. Локальні ОС застосовуються на автономних комп’ютерах або комп’ютерах, що використовуються в комп’ютерних мережах в якості клієнтів. До складу локальних ОС входить клієнтська частина ПЗ для організації доступу до віддалених ресурсів. Мережні ОС призначені для

управління ресурсами комп'ютерів, підключених до мережі з метою сумісного використання ресурсів. У таких ОС повинні бути реалізовані надійні засоби розмежування доступу до інформації, її цілісності і інші можливості використання мережних ресурсів.

У залежності від області використання ОС також поділяються на 4 категорії (рис. 1.3):

- системи пакетної обробки;
- системи з розділенням часу;
- системи реального часу;
- гібридні ОС.

Рисунок 1.3 – Поділ ОС за областями використання

Пакетні системи (англ. Batch Operating Systems) з'явились з самого початку розвитку комп'ютерів, які використовували для введення даних перфокарти або плівку. Введення завдань виконувалося шляхом зчитування через пристрій зчитування колоди перфокарт – пакета. На цей час пакетні системи не обмежуються введенням даних з карт або стрічки, але процеси в них виконуються послідовно, без взаємодії з користувачем. Системи пакетної обробки призначені для вирішення задач, які не потребують швидкого отримання результатів.

Ефективність системи виміряється в пропускній здатності (англ. Throughput) – кількості завдань, виконаних у певний період часу (наприклад, 70 завдань на годину) та в часі оберту (англ. Turnaround time) – статистично усередненому часі між моментом надходження завдання в систему до отримання результату його виконання. Також важливим є підтримка постійного навантаження процесора (англ. CPU utilization) для максимізації корисної роботи. Пакетний режим передбачає наявність черги програм на виконання, і ОС для підтримки навантаження процесора може забезпечувати завантаження програми з зовнішніх носіїв даних в оперативну пам'ять без очікування завершення виконання попередньої програми. Головною метою ОС пакетної обробки є максимальна пропускна здатність або вирішення максимального числа завдань в одиницю часу. Ці системи забезпечують високу продуктивність при обробці великих обсягів інформації, але знижують ефективність роботи користувача в інтерактивному режимі. Першими системами пакетної обробки були однозадачні пакетні системи (1955-1965 рр.), пізніше з'явились багатозадачні пакетні ОС, які мають більш складні функції по управлінню пам'яттю, плануванню завдань і по захисту завдань від впливу одне на інше. В якості прикладів ранніх пакетних систем можна вказати FORTRAN Monitor System (FMS) (1955 р.) та IBSYS (1960 р.) для IBM 7090/7094 (однозадачні), та групу ОС OS\360 (1966 р.) для платформ IBM S/360, S/370 (багатозадачні).

Інтерактивні системи або системи з розділенням часу (англ. Time Sharing Systems) мають більш швидкий час оберту, ніж у пакетних систем, але повільніший, ніж у систем реального часу. Вони були введені, щоб задовольнити потреби користувачів, яким необхідний швидкий час оберту, коли вони виконують налагодження своїх програм. Системи з розділенням часу потребують наявності в своєму складі ПЗ, яке дозволяє кожному користувачеві взаємодіяти безпосередньо з системою за допомогою команд, що вводяться з клавіатури. Інтерактивна ОС забезпечує негайний зворотний зв'язок з користувачем, а час відгуку (англ. Response time) може вимірюватися в хвилинах або секундах, залежно від числа активних користувачів. У системах з розділенням часу для виконання кожного завдання надається невеликий проміжок часу, і жодне з завдань не займає

процесор надовго. Якщо цей проміжок часу обрано мінімальним – складається враження одночасного виконання декількох завдань. Пропускна здатність у системах з розділенням часу менше ніж у системах пакетної обробки, але вони забезпечують високу ефективність роботи користувача в інтерактивному режимі. Однією з перших систем, що працює в режимі розділення часу є CTSS (Compatible Time-Sharing System) Массачусетського технологічного інституту (MIT) (1961 р.) для IBM 7094. До систем з розділенням часу відносяться сучасні системи для персональних комп'ютерів класу UNIX/Linux, Windows.

Системи реального часу (англ. Real-Time Operating System, RTOS) є найшвидшими з вказаних чотирьох категорій і використовуються в середовищах, де критичні часові характеристики, тобто в середовищах, в яких дані повинні бути оброблені надзвичайно швидко, тому що результат буде викликати негайне вирішення. Системи реального часу використовуються для управління технологічним процесом або технічним об'єктом, наприклад, літальним об'єктом, станком, складною медичною технікою, розподілом електроенергії, телефонної комутації тощо. Час відгуку в системах реального часу вимірюється в частках секунди, хоча це не завжди можливо реалізувати на практиці. ОС реального часу поділяються на 2 типи: жорсткого реального часу (англ. Hard Real Time) і м'якого реального часу (англ. Soft Real Time). Системи жорсткого реального часу повинні забезпечувати необхідний час виконання завдань у будь-яких умовах. Якщо ОС реального часу може забезпечити необхідний час виконання завдань реального часу в середньому, тоді вона має назву ОС м'якого реального часу.

Оскільки дотримання термінів має вирішальне значення в системах жорсткого реального часу, інколи ОС є просто бібліотекою, пов'язаною з прикладними програмами, де все тісно поєднано без реалізації захисту між частинами системи.

Прикладом такого типу системи реального часу є безкоштовна eCos[1] та комерційна eCosPro[2].

Іншими прикладами ОС реального часу є системи жорсткого реального часу RTLinux на основі Linux, FreeRTOS[3] для мікроконтролерів, Keil RTX5[4] для пристроїв на базі процесора Arm Cortex-M, Cesium[5] для вбудованих систем (16 різних платформ), та системи м'якого реального часу: KURT на основі Linux, QNX (Neutrino RTOS та BlackBerry)[6], однозадачна RT-11[7] для 16-розрядних комп'ютерів серії PDP-11, Windows IoT[8] (та її попередник – Windows Embedded) та ін.

Гібридні системи являють собою комбінацію пакетних і інтерактивних систем. Інтерактивна складова полягає тільки в тому, що окремі користувачі мають доступ до системи через термінали і отримують швидкі відповіді, але система фактично приймає і виконує пакетні програми, які виконуються у фоновому режимі на відміну від інтерактивного режиму. Гібридні системи використовують вільний час між запитами користувачів на виконання програм, які не потребують значної допомоги оператора.

1.3. Коротка історія розвитку ОС

Еволюція ОС паралельна еволюції комп'ютерів, тому що ОС призначена насамперед для управління апаратурою. Саме тому далі не ставиться задачею перерахувати всі ОС за часом їх появи та описати їх особливості, а тільки наводяться основні риси, характерні для кожного

[1] Остання версія 3.0 від 30.03.2009 р., ресурс у мережі Інтернет – https://ecos.sourceware.org/

[2] Остання версія 4.1 від 28.07.2017 р., ресурс у мережі Інтернет – https://www.ecoscentric.com/

[3] Остання версія 11.1.0 від 22.04.2024 р., ресурс у мережі Інтернет – https://www.freertos.org/

[4] Ресурс у мережі Інтернет – https://developer.arm.com/

[5] Ресурс у мережі Інтернет – https://weston-embedded.com/

[6] Ресурс у мережі Інтернет – https://blackberry.qnx.com/

[7] Остання версія 5.7 за жовтень 1998 р.

[8] Ресурс у мережі Інтернет – https://azure.microsoft.com/en-us/products/windows-iot/

періоду розвитку. Крім того, більшість введених нових термінів в опису розвитку ОС будуть детально розглянуті в інших, відповідних темах. Більш детальну інформацію про розвиток ОС можна знайти у відповідній літературі, присвяченій вивченню ОС [9, 13, 15, 18, 21, 29, 34-37, 44, 45].

Перше покоління електронних обчислювальних машин (ЕОМ) (1940-1955 рр.) було часом технології вакуумних трубок (електронних ламп) і комп'ютери займали площу кількох кімнат. Кожен комп'ютер був унікальним за своєю структурою та призначенням. Потреби в стандартних ОС майже не було, оскільки кожен комп'ютер був обмежений роботою кількох фахівців математичного, наукового або військового напряму, і всі вони були добре інформовані про особливості використовуваного апаратного забезпечення.

Типова програма на той час мала всі інструкції, які комп'ютер повинен був виконувати. Команди дозволяли безпосередньо працювати з апаратурою: пристроєм зчитування перфокарт, процесором, пристроєм виведення.

Програміст управляв машиною з основної консолі і мав прямий доступ до вмісту кожного регістру, мав можливість призупиняти роботу процесора та робити корегування значень комірок пам'яті. У результаті таких ручних операцій комп'ютери часто простоювали, процесор займався обробкою тільки невелику частину часу, і вся система простоювала між виконанням програм.

З часом апаратне та програмне забезпечення стало більш стандартним, для виконання програм стала потрібна менша кількість кроків і менші знання архітектури комп'ютера: з'явились компілятори та транслятори; перші ОС почали складатися з бібліотечних програм, стандартних підпрограм і допоміжних програм; підпрограми-драйвери були написані для стандартизації використання пристроїв введення та виведення. Але недоліком програм було те, що вони були призначені для використання реальних ресурсів у монопольному режимі.

Друге покоління комп'ютерів (1955-1965 рр.) було розроблено для задоволення потреб на той час нової галузі – бізнесу. Комп'ютери в той час були дуже дорогими, наприклад IBM 7094 для ОС FMS та IBSYS коштувала

близько 6 млн. доларів (в цінах 1995 року). У комп'ютерах цього покоління було прийнято два вдосконалення: по перше, оператори комп'ютера наймалися для полегшення експлуатації машин і, по друге, вперше почали використовувати планування завдань. Планування завдань являло собою схему підвищення продуктивності роботи, яка об'єднує програми зі схожими вимогами. Наприклад, програми на Фортрані виконувалися доти, поки компілятор Фортран був завантажений у пам'ять. Або, спочатку виконувалися всі роботи по введенню даних з пристрою читання з перфокарт, а тоді – всі роботи по введенню даних зі стрічкового накопичувача. Але така робота з пристроями введення була визнана неефективною – було запропоновано поперемінне використання пристроїв: поки виконуються підготовчі роботи з пристроєм введення з перфокарт – виконувалося зчитування зі стрічки, та навпаки.

Планування завдань потребувало додати до складу ОС мову управління завданнями (англ. JCL – Job Control Language), яка допомагала ОС управляти ресурсами шляхом їх визначення для кожного завдання та перерозподілом з метою підвищення ефективності.

Але навіть при використанні пакетного режиму з оформленням кожного завдання за допомогою JCL у роботі комп'ютерів другого покоління був низький рівень узгодження швидкості роботи між процесором і пристроями введення-виведення. Наприклад, для читання 1600 перфокарт може бути потрібно 79 секунд, і лише 5 секунд процесорного часу для компіляції коду, зчитаного з цих перфокарт. Це означає, що процесор простоює 94 % часу і тільки 6 % часу він виконує корисну роботу. Ефективність роботи процесора з часом підвищилась, але не за рахунок нових алгоритмів управління, а за рахунок розвитку апаратури – появи швидкісних дискових пристроїв введення-виведення та організації буферизації операцій введення-виведення. Буферна пам'ять (англ. Buffer memory) є пам'яттю для тимчасового зберігання даних: повільний пристрій введення передає дані в буфер, а при заповненні буфера дані швидко передаються на центральний процесор.

В ОС другого покоління стали використовуватися переривання від таймеру для захисту процесора від зациклення на помилкових інструкціях і

для забезпечення спільного використання завдань. Фіксований обсяг часу виконання було виділено для кожної програми при надходженні її в систему. Час виконання контролювався ОС. Якщо будь-які програми вичерпували свій час, але не завершувались, вони припинялись примусово, а користувачі отримували повідомлення про помилку.

У другому поколінні ОС програми як і раніше, працювали в пакетному режимі – одна програма в один момент часу. Наступним кроком на шляху до більш ефективного використання ресурсів системи був рух до розподіленої обробки (використання розділення часу).

Як було вказано вище, представниками ОС другого покоління є FMS та IBSYS для IBM 7094.

Комп'ютери третього покоління (1965-1980 рр.) були розроблені на базі більш швидких процесорів, але їх швидкість викликала проблеми з відносно повільними пристроями введення-виведення. Рішенням цього недоліку було впровадження мультипрограмування.

Перші мультипрограмні ОС обслуговували програми по черзі, але якщо програма видавала запит на виконання команди введення-виведення, процесор звільнявся і починав виконання наступного завдання. Це було названо пасивним мультипрограмуванням, оскільки ОС не контролювала виконання програм, а лише чекала переривання для кожного завдання. Це не було ідеальним рішенням, оскільки якщо якась програма не виконувала запитів на операції введення-виведення, вона займала процесор протягом тривалого часу, а всім іншім завданням доводилося чекати.

Для усунення цього недоліку ОС стала виконувати більш активну роль з появою активного мультипрограмування. ОС дозволяла кожній програмі використовувати тільки заданий проміжок часу процесора. Коли час минав, то робота програми переривалась, і іншій програмі дозволялось розпочати виконання. Перервана програма чекала, поки їй не буде дозволено повернутися до виконання пізніше. Ідея розділення часу стала звичайним явищем у багатьох ОС з розділенням часу і використовується досі.

Планування завдань, яке було почато в ОС другого покоління, було ускладнене тим, що в оперативній пам'яті знаходилось багато завдань. Було прийнято рішення сортувати програми по групах, а потім завантажувати

програми відповідно до заданих налаштувань. Групи, як правило, визначались за пріоритетами або вимогами до пам'яті. Це дозволило отримати більш ефективне використання ресурсів.

Додатково до планування завдань, обробки переривань і виділення пам'яті ОС довелося вирішувати конфлікти між завданнями, які запрошували виділення одного й того ж пристрою одночасно.

ОС третього покоління складалися з багатьох модулів, які користувач мав можливість самостійно обирати, тобто, ОС могла бути налаштована відповідно до потреб користувача. Найбільш часто використовувані модулі були зроблені резидентними (постійно присутніми в оперативній пам'яті), інші модулі розміщалися в зовнішній пам'яті і могли бути викликані тільки в разі необхідності.

Комп'ютери, розроблені наприкінці 1970-х років, використовували ще більш швидкі процесори, у результаті цього невідповідність між швидкістю обробки даних та введенням-виведенням стала ще більшою. Схеми мультипрограмування, метою яких було підвищення ефективності використання центрального процесора, стали ще складнішими, тому що фізичний обсяг оперативної пам'яті був обмеженим і сама пам'ять була дуже коштовною. Вирішенням цього фізичного обмеження була віртуальна пам'ять (англ. VM – Virtual Memory), принцип роботи якої полягає в тому, що вся програма не повинна перебувати в пам'яті по причині того, що процесор виконує в один момент часу тільки одну команду. ОС з підтримкою віртуальної пам'яті може розділити програму на сегменти і тримати їх у зовнішній пам'яті, а завантажувати їх в оперативну пам'ять тільки за потребою.

Прикладами ранніх ОС третього покоління можна назвати OS/360 для комп'ютерів IBM/360 (перша багатозадачна ОС), CTSS (перша система з розділенням часу), Multics (Multiplexed Information and Computer Service) для комп'ютерів GE-645, та з 1973 р. – Honeywell 6180 (повноцінна система з розділенням часу).

Multics зробила великий вплив на розвиток ОС. У системі Multics вперше реалізовані: сегментно-сторінкова віртуальна пам'ять; відображення файлів на адресний простір процесів; динамічне під'єднання бібліотек до

виконуваної програми. У ній застосовувалися навіть більш складні та гнучкі механізми взаємодії процесів через спільні сегменти, ніж у сучасних системах. Використовувалося «гаряче» реконфігурування системи зі змінними процесорами, блоками пам'яті, жорсткими дисками без зупинки обслуговування користувачів. Multics – одна з перших мультипроцесорних систем. Multics була також однією з перших систем, в якій велика увага приділялась безпеці взаємодії між програмами та користувачами. Більше того, вона була самою першою ОС, реалізованою як «безпечна». Додатково до цього в Multics однією з перших була реалізована ієрархічна файлова система: імена файлів могли бути практично довільної довжини та містити будь-які символи. Файл або каталог могли мати декілька імен (коротке та довге), були доступні для використання символьні посилання між каталогами. В ОС Multics був вперше реалізований підхід використання стеків для кожного обчислювального процесу в ядрі системи: з окремим стеком для кожного рівня безпеки навколо ядра. Multics – одна з перших ОС, написаних на мові високого рівня PL/1. Ідея «хмарних» обчислень бере початок від системи Multics – комп'ютерного підприємства суспільного використання 60-70 років минулого століття.

Розробка Multics почалась у 1965 р. на ідеях та рішеннях, запозичених з ОС CTSS (ранні версії Multics не збереглись), з 1973 р. версіям стали присвоюватись номери з префіксом «MR» (MR 1.0 в 1974 р.). У 1992 р. було випущено останню класичну версію ОС (MR 12.5). Останній комп'ютер з ОС Multics був зупинений 31 жовтня 2000 року. З 2014 р. роботу над проєктом було відновлено: було створено емулятор DPS8M апаратури Honeywell Multics (36-bit GE Large Systems / Honeywell / Bull 600/6000-series mainframe computers) для ОС Windows, UNIX/Linux та macOS та випущено нову версію ОС для емулятора MR 12.6f (2016 р.)[9].

[9] Остання версія ОС Multics для емулятора – MR12.8 від 08.08.2023 р., ресурс у мережі Інтернет – https://www.multicians.org/

Поява мінікомп'ютерів (випуск корпорацією DEC у 1961 році комп'ютера PDP-1 і пізніше найбільш комерційно успішної моделі PDP-11) з відносно низькою ціною (120 тис. доларів, що складало лише 5 % від ціни мейнфрейма IBM 7094) і приблизно однаковою з мейнфреймами продуктивністю на нечислових завданнях сприяла дуже широкому їх поширенню, а разом з ними й ОС UNIX, розробленої співробітниками Bell Labs Кеном Томпсоном, Деннісом Рітчі і Дугласом Макілроєм. У 1969 році Кен Томпсон, намагаючись реалізувати ідеї, які були покладені в основі ОС Multics, на більш скромному апаратному забезпеченні (DEC PDP-7), створив першу нову однозадачну ОС, яку Браян Керніган назвав UNICS (Uniplexed Information and Computing Service – примітивна інформаційна та обчислювальна служба) – як тонкий «тролінг» багатозадачної системи Multics. Пізніше ця назва скоротилася до UNIX. UNICS вважається нульовою версію ОС UNIX. Ядро та програми PDP-7 UNIX були повністю написані на асемблері.

У листопаді 1971 року вийшла версія для PDP-11. Ця версія отримала назву «First Edition» (випуск 1) і була першою офіційною версією. До речі, системний час всі реалізації UNIX відраховують з 1 січня 1970 року. UNIX популяризувала запропоновану в Multics ідею ієрархічної файлової системи з довільною глибиною розміщення файлів. Ще одна інновація Multics, популяризована UNIX – інтерпретатор команд: тому що і інтерпретатор і команди ОС – звичайні програми, користувач міг би вибрати їх відповідно до своїх вподобань або навіть написати власну оболонку.

Перше видання працювало на PDP-11/20 без пристрою управління пам'яттю (англ. MMU – Memory Management Unit) та відповідно її апаратного захисту. Стабільність роботи та стійкість до збоїв була низькою. З мов програмування підтримувалися асемблер, B, BASIC, FORTRAN. Компілятор мови C був доданий у другий випуск ОС (червень 1972 р.). Захист пам'яті був доданий у третьому випуску (лютий 1973 р.). У четвертому випуску ядро було переписане мовою C (розмір ОС збільшився одразу на третину) і мову B було видалено з системи (листопад 1973 р.). Сьома версія є останньою єдиною версією UNIX (1979 р.), далі відбувся

поділ на дві основні гілки: System V від AT&T і BSD (Berkeley Software Distribution) від Каліфорнійського університету в Берклі.

Істотними особливостями UNIX були повна орієнтація на текстове введення-виведення і припущення, що розмір машинного слова кратний восьми бітам. Орієнтація на восьмибітний байт зробила UNIX більш масштабованою та переносною, ніж інші ОС. UNIX сприяла широкому поширенню регулярних виразів, які були вперше реалізовані в текстовому редакторі *ed*. Можливості, що надаються програмами UNIX, стали основою стандартних інтерфейсів операційних систем (POSIX – **P**ortable **O**perating **S**ystem **I**nterface for Uni**x**). Серед прикладів відомих подібних ОС UNIX можна назвати: BSD, Solaris, Linux, Minix (в 1987 р. Ендрю Таненбаум випустив цей невеликий клон UNIX для освітніх цілей; функціонально Minix дуже схожий на UNIX, включаючи підтримку POSIX), Android, MeeGo, NeXTSTEP, macOS.

У період четвертого покоління ОС (з 1980 року по 1990 роки) становиться популярним ПЗ управління базами даних – зводиться до мінімуму надмірність і спрощується оновлення і доступ до даних. Запити на обробку оформляються таким чином, щоб навіть починаючий користувач міг отримати доступ до бази даних. Такі запити, як правило, здійснюються через термінали, а це, у свою чергу, призвело до розширення підтримки терміналів і розробки ПЗ передачі даних.

Програмісти стали більш віддалені від складності архітектури комп'ютерів і в прикладних програмах почали використовувати слова англійської мови в якості конструкцій мови програмування. Ця тенденція використання стандартів призвела до поліпшення управління програмами, оскільки їх обслуговування стало простішим і швидшим.

Цей етап пов'язаний з появою великих інтегральних схем (ВІС) (англ. LSI – Large Scale Integration). У ці роки відбулося різке зростання ступеня інтеграції та здешевлення мікросхем: різко покращало співвідношення «вартість / продуктивність» комп'ютерних комплектуючих. Ціна обладнання стала низькою і частини ОС стали частиною самого обладнання – прошивкою (англ. Firmware). Це слово використовується для позначення того, що програма постійно знаходиться в постійному

запам'ятовуючому пристрої (ПЗП, англ. Read-only memory, ROM) на заміну від зберігання в зовнішній пам'яті.

Робота програміста дуже змінилася, оскільки багато функцій зі складу програм були перенесені до складу ОС, отже програмування стало простішим і більш віддаленим від роботи з устаткуванням комп'ютерів.

Комп'ютер став доступний окремій людині – настала ера персональних комп'ютерів. З погляду архітектури персональні комп'ютери нічим не відрізнялися від класу мінікомп'ютерів типу PDP-11, але ціна в них суттєво відрізнялася. Якщо мінікомп'ютер дав можливість мати власну обчислювальну машину відділу підприємства чи університету, персональний комп'ютер зробив це можливим для окремої людини. Комп'ютери стали широко використовуватися нефахівцями, що вимагало розробки «дружнього» програмного забезпечення.

На ринку ОС домінували дві системи: MS-DOS (Microsoft Disc Operating System)[10] та UNIX. Однопрограмна однокористувацька ОС MS-DOS широко використовувалася для комп'ютерів, побудованих на базі мікропроцесорів Intel 8088, а потім 80286, 80386 і 80486. Мультипрограмна багатокористувацька ОС UNIX домінувала в середовищі комп'ютерів, побудованих на базі RISC-процесорів.

Попередником системи MS-DOS була система 86-DOS від Seattle Computer Products (було випущено 2 версії: 86-DOS 0.42 від 25.02.1981 р. та 86-DOS 1.00 від 28.04.1981 р.), яка в свою чергу побудована на базі ОС CP/M (Control Program/Monitor та пізніше – Control Program for Microcomputers)[11] – першій ОС для персональних комп'ютерів на базі процесорів Intel 8080, Intel 8085, Zilog Z80, Zilog Z8000, Intel 8086 та Motorola 68000.

ОС для перших мікрокомп'ютерів повністю ґрунтувалися на введенні команд із клавіатури. Першою успішною комерційною реалізацією

[10] Перша версія – 12.08.1981 р., остання версія 6.22 (самостійна) за квітень 1994 р., та 8.0 (у складі ОС Windows Me) від 14.09.2000 р.

[11] Першу версію випущено в 1974 р., остання версія – 3.1 за 1983 р., ресурс у мережі Інтернет – https://www.digitalresearch.biz/CPM.HTM

графічного інтерфейсу (англ. GUI – Graphical User Interface) для персональних комп'ютерів була система Apple Macintosh, виготовлена під керівництвом Стіва Джобса в 1984 році.

ОС для персонального комп'ютера Macintosh 128K на процесорі Motorola 68000 не мала офіційної назви, та просто називалась – Системне програмне забезпечення (System Software).[12]

Під впливом успіхів Apple компанія Microsoft у 1985 випускає свою реалізацію GUI – систему Windows[13]. Протягом 10 років система Windows виконувала роль графічної оболонки, що запускалася на виконання з ОС MS-DOS. Підтримка версій 1-3 продовжувалась до кінця 2001 р.

З 1993 р. було розроблено ще дві лінійки настільних операційних систем. Сімейство Windows 9x включає Windows 95 (перший випуск 24.08.1995 р.), Windows 98 (перший випуск 25.06.1998 р.) і Windows ME (перший випуск 14.09.2000 р.). Воно було проміжним сімейством 32-розрядних систем. Сімейство Windows NT – сучасні версії операційної системи Windows, що беруть початок від системи Windows NT 3.1 (перший випуск 27.07.1993 р.). Найбільш популярні версії операційних систем цього сімейства: Windows NT 4.0 (перший випуск 29.07.1996 р.); Windows 2000 (NT 5.0) (перший випуск 17.02.2000 р.); Windows XP (NT 5.1) (перший випуск 25.10.2001 р.); Windows Vista (NT 6.0) (перший випуск 30.01.2007 р.); Windows 7 (NT 6.1) (перший випуск 22.10.2009 р.); Windows 8 (NT 6.2) (перший випуск 26.10.2012 р.); Windows 10 (NT 10.0) (перший випуск 29.07.2015 р.); Windows 11 (NT 10.0 – внутрішній номер версії залишився старим: як вказують деякі джерела, він, починаючи з Windows 10, не грає ролі) (перший випуск 05.10.2021 р.).

[12] Перша версія від 24.01.1984 р., з версії 7 (червень 1991 р.) з'явилась офіційна назва - Mac OS, після версії 9.2.2 (грудень 2001 р.) стала називатись Mac OS X, у 2012 р. перейменовано в OS X, у 2016 р. перейменовано в macOS. Остання стабільна версія – 15.0.1 від 03.10.2024 р., офіційний ресурс у мережі Інтернет – https://apple.com/macos

[13] Перша версія – 20.11.1985 р., друга версія – 09.12.1987, третя версія – 22.05.1990 р., остання версія з групи – 3.11 від 11.08.1993 р.

У 1983 р. було оголошено про створення GNU (GNU's Not UNIX), UNIX-подібної ОС. Проєкт з'явився під впливом ідей засновника проєкту Річарда Столмана про необхідність створення вільно поширюваної ОС та ПЗ з відкритим вихідним кодом.

Фактично це дало поштовх і для відкриття кодів ОС UNIX. На початку 1993 року ОС 386/BSD змінив свою назву на NetBSD. У грудні 1993 року з'явилась ОС FreeBSD[14], призначена для простих користувачів. Також відомі розробки на основі BSD: SunOS[15] від компанії Sun Microsystems та OpenBSD[16].

У 1997 році компанія Apple шукала основу для своєї нової ОС і вибрала NeXTSTEP[17] – операційну систему, яка вільно поширюється під керуванням, розроблену компанією NeXT. У 2000 році Apple випускає відкриту POSIX-сумісну ОС Darwin. Вона містить код, написаний самим Apple, отриманим від NeXTSTEP, FreeBSD і інших безкоштовних проєктів. ОС Darwin являє собою набір основних компонентів, які використовуються в Mac OS X і Apple iOS.

На основі UNIX System V компанією Oracle в 1992 р. було випущено ОС Solaris[18], компанією Novell у цьому ж році було випущено ОС UnixWare[19] та компанією Hewlett-Packard випущено ОС HP-UX[20].

14 червня 2005 р. був відкритий вихідний код ОС Solaris. Цей проєкт отримав назву OpenSolaris, та в 2008 році з'явилась перша офіційна

[14] Остання версія 14.1 від 04.06.2024 р., ресурс у мережі Інтернет – https://www.freebsd.org/

[15] Перша версія випущена в 1982 р., остання версія 4.1.4 у листопаді 1994 р.

[16] Перший випуск 18.10.1995 р., остання версія 7.5 за 05.04.2024 р., ресурс у мережі Інтернет – https://www.openbsd.org/

[17] Основана на BSD і мікроядрі Mach, перший випуск 18.09.1989 р., остання версія 3.3 за 1995 р.

[18] Перша версія – червень 1992 р., остання версія 11.4 SRU72 від 20.08.2024 р., ресурс у мережі Інтернет – https://www.oracle.com/solaris/

[19] Остання версія 7.1.4 за червень 2008 р., ресурс у мережі Інтернет – https://www.sco.com/products/unixware714/

[20] Остання версія HP-UX 11i Version 3 за 31.05.2024 р. ресурс у мережі Інтернет – https://www.hpe.com/us/en/servers/hp-ux.html

версія 2008.05. На основі OpenSolaris було випущено більше десяти ОС, найбільш відомі з них – BeleniX[21] і Nexenta OS[22].

Оригінальна версія MINIX також розвивалася – у 2005 р. було випущено версію MINIX 3[23]. MINIX 3 має здатність виявляти та замінювати несправні або навіть збійні модулі (наприклад, драйвери пристроїв введення/виведення) без перезавантаження та не заважаючи запущеним програмам.

Прагнення до безкоштовної робочої (на відміну від освітньої) версії MINIX змусило фінського студента Лінуса Торвальдса зайнятись розробкою Linux (з 1991 р.). Ця система була безпосередньо розроблена на основі MINIX і спочатку підтримувала різні її функції (наприклад, файлову систему MINIX).

Перша версія ОС Linux[24] була випущена в 1994 р. З самого початку і на цей час Linux поширюється як безкоштовне ПЗ з ліцензією GPL (GNU General Public License). Це означає, що вихідний код ОС може побачити і допрацювати будь-який користувач. Єдина умова – змінений, модифікований код має бути доступним для всіх і розповсюджуватися також за ліцензією GPL. Це важливо, тому що дає можливість розробникам використовувати код і в той же час не мати проблем із-за авторських прав. Зміна коду ОС привела до появи чисельних варіантів ОС Linux, які мають назву – дистрибутив Linux. Дистрибутив Linux – це визначення ОС, яка використовує ядро Linux і яку можна встановити на комп'ютері користувача. У дистрибутивах зазвичай міститься не тільки ядро і сама ОС, але і корисні застосунки: редактори, програвачі, інструменти для роботи з базами даних і інше ПЗ, що розроблюється в рамках GNU. Станом на 2024 р. існує більше

[21] Остання версія 0.7.1 від 19.07.2008 р.

[22] Остання версія 3.1.5 від 10.01.2013 р.

[23] Остання версія 3.3.3 від 16.09.2014 р., ресурс у мережі Інтернет – https://www.minix3.org/

[24] Остання версія ядра ОС Linux – 6.11.2 від 04.10.2024 р., ресурс у мережі Інтернет https://kernel.org/

ніж 600 дистрибутивів Linux, у таблиці 1.1 наведені найбільш популярні дистрибутиви станом на початок 2024 року.

Досвідчені фахівці ОС UNIX/Linux віддають перевагу командному інтерфейсу перед графічним інтерфейсом користувача, але звичайним користувачам, які не потребують широких знань і вмінь при роботі з ОС, більш зручно працювати з графічним інтерфейсом. Тому в майже всіх системах UNIX/Linux було додано підтримку GUI системи на основі вікон. Основна та сама розповсюджена розробка – X Window System[25] від компанії M.I.T.

Ця система забезпечує базове керування вікнами, дозволяючи користувачам створювати, видаляти, переміщувати та змінювати розміри вікон за допомогою миші. Часто повне робоче середовище на основі графічного інтерфейсу користувача, таке як Gnome або KDE, доступне для запуску над X Window System (X11), надаючи UNIX/Linux вигляд і поведінку, чимось схожі на Macintosh або Microsoft Windows: X Window System не визначає деталі інтерфейсу користувача – цим займаються менеджери вікон. З цієї причини зовнішній вигляд програми в середовищі X Window System може дуже сильно відрізнятися в залежності від можливостей і налаштувань конкретного віконного менеджера.

Таким чином, головними конкурентами систем Windows у персональних обчисленнях стають різні похідні системи UNIX (у дужках вказані ОС, на базі яких було їх розроблено): сімейство ОС Apple MacOS (BSD), Google Android (Linux), Ubuntu (Linux). Для серверних застосунків та високопродуктивних обчислень операційна система UNIX (зокрема, Linux) є домінуючою ОС: Linux забезпечує величезну частку серверів у центрах обробки даних і формує основу ОС Android.

[25] Перша версія випущена в червні 1984 р., остання – X11R7.7 від 06.06.2012 р., ресурс у мережі Інтернет https://x.org/

Таблиця 1.1 – Популярні дистрибутиви Linux

ОС	Основа для побудови	Дата випуску першої версії	Остання стабільна версія	Ресурс у мережі Інтернет
Debian	ядро Linux	16.08.1993 р.	12.7 від 31.08.2024 р.	https://www.debian.org/
Ubuntu Desktop	Debian	20.10.2004 р.	24.04.1 LTS від 04.09.2024 р.	https://ubuntu.com/
Pop!_OS	Ubuntu	27.10.2017 р.	22.04 LTS від 25.04.2022 р.	https://pop.system76.com/
Drauger OS	Ubuntu	04.05.2018 р.	7.7 від 22.07.2024 р.	https://draugeros.org/
Tails	Debian	23.06.2009 р.	6.7 від 10.09.2024 р.	https://tails.net/
Kali Linux	Debian	13.03.2013 р.	2024.3 від 11.09.2024 р.	https://www.kali.org/
Linux Mint	Ubuntu, Debian, Kubuntu	27.08.2006 р.	22 від 25.07.2024 р.	https://linuxmint.com/
elementary OS	Ubuntu	31.03.2011 р.	7.1 від 03.10.2023 р.	https://elementary.io/
Puppy Linux	Ubuntu, Debian, Slackware	19.06.2003 р.	9.5 від 17.09.2020 р.	https://puppylinux.com/
CentOS Stream	Red Hat	24.09.2019 р.	9 від 03.12.2021 р.	https://centos.org/
ArchLinux	ядро Linux	березень 2002 р.	2024.10.01 від 01.10.2024 р.	https://archlinux.org/

У 80-х роках минулого сторіччя крім окремо працюючих персональних комп’ютерів стали бурхливо розвиватися мережі персональних комп’ютерів, які працюють під управлінням мережевих чи розподілених ОС.

У мережевих ОС користувачі повинні бути обізнані про наявність інших комп’ютерів і повинні робити логічний вхід до іншого комп’ютера, щоб скористатися його ресурсами, переважно файлами. Кожна машина в мережі виконує свою власну локальну ОС, що відрізняється від ОС автономного комп’ютера наявністю додаткових засобів, що дозволяють

комп'ютеру працювати в мережі. Мережева ОС не має фундаментальних відмінностей від ОС однопроцесорного комп'ютера. Вона обов'язково містить програмну підтримку для мережних інтерфейсних пристроїв (драйвер мережевого адаптера), а також засоби для віддаленого входу в інші комп'ютери мережі та засоби доступу до віддалених файлів, проте ці доповнення істотно не змінюють структуру ОС.

Врешті-решт, промисловість перейшла на багатопроцесорні системи, були розроблені більш складні мови програмування для управління множиною процесорів у межах одного завдання. У результаті стало можливим паралельне виконання програм. З'явились розподілені ОС (глобальні ОС у масштабах обчислювальної мережі, які забезпечують для користувача більш прозорий доступ до мережних ресурсів, наприклад, глобальне управління процесами і глобальна файлова система, як в ОС Sun Cluster / Oracle Solaris Cluster[26]).

П'яте покоління (з 1990 р. по теперішній час) характеризується розвитком мобільних комп'ютерів та технологій хмарних обчислень. Насправді, перший справжній мобільний телефон з'явився в 1946 р. і важив близько 40 кілограмів, а перший кишеньковий телефон був випущений 03.04.1973 р. компанією Motorola і при вазі 1.1 кг вважався дуже легким, акумулятор працював 15 хвилин, після чого потрібен був 10-годинний цикл заряджання. Цей телефон відноситься до покоління 0G (Zero Generation) – стандарт, який було розроблено ще в 1946 р.

Зараз охоплення мобільним зв'язком у розвинених країнах наближається до 90 % населення. Хоча ідея поєднання телефонії та комп'ютерів у пристрої, схожому на телефон, існує ще з 1970-х років, перший смартфон з'явився лише в середині 1990-х років, коли Nokia випустила модель N9000, який буквально поєднав два, здебільшого окремих пристрої: телефон і персональний цифровий помічник. Сам термін

[26] Остання версія 4.4 для Oracle Solaris 11.4 за травень 2021 р., ресурс у мережі Інтернет – https://www.oracle.com/solaris/solaris11/

«смартфон» було введено компанією Ericsson у 1997 р. для свого GS88 «Penelope». На пристроях N9000 та GS88 працювала 16-розрядна ОС GEOS (Graphic Environment Operating System)[27].

До одних з перших мобільних ОС також відносять Newton OS (ОС для КПК – кишенькового персонального комп'ютера (англ. PDA – Personal Digital Assistance) Apple Newton (1993 – 1997 рр.). Вона була повністю написана на C++, характеризувалась низьким споживанням енергії та ефективним використанням пам'яті: частина її програм була попередньо встановлена в ПЗП Newton) та Zaurus для КПК Simon від IBM (сформувала базовий функціонал для мобільних пристроїв: календар, годинник, щоденник, нотатки, калькулятор та ін.).

Взагалі, ОС для мобільних пристроїв повинна виконувати всі функції звичайних ОС, але з деякими додатковими можливостями:

♦ управління ресурсами: ОС контролює доступ до різних ресурсів пристрою, таких як процесор, пам'ять, мережа тощо. Вона визначає, які програми можуть використовувати кожен ресурс і встановлює пріоритети, коли є сперечання за ресурси;

♦ управління процесами: ОС координує виконання різних процесів на пристрої, щоб гарантувати, що вони працюють ефективно та не конфліктують один з одним. Вона також управляє перемиканням між різними програмами, коли користувач переміщується між ними;

♦ управління файловою системою: ОС забезпечує доступ до сховища даних на пристрої та керує файлами та каталогами. Вона дозволяє користувачам створювати, змінювати та видаляти файли, та забезпечує безпеку цих операцій;

♦ управління пристроями введення/виведення: ОС керує різними пристроями введення/виведення, такими як сенсорний екран, клавіатура, динаміки та мікрофони. Вона забезпечує зв'язок між програмами та

[27] Розробник Berkeley Softworks, перша версія – 1990 р., остання версія 4.1.3 від 25.08.2005 р., ресурс у мережі Інтернет – https://github.com/bluewaysw/

пристроями введення/виведення, щоб користувачі могли ефективно працювати з програмами та керувати пристроєм;

♦ управління безпекою: ОС забезпечує безпеку пристрою, захищаючи його від різних загроз, таких як віруси та несанкціоновані втручання. Вона також захищає особисті дані користувачів, такі як контакти, повідомлення та ін.;

♦ управління енергоспоживанням: ОС контролює споживання енергії пристрою, керуючи роботою різних компонентів, таких як екран, процесор та бездротові з'єднання. Вона також оптимізує роботу цих компонентів, щоб продовжити час роботи пристрою та збільшити час між заряджаннями.

У перше десятиліття після появи більшість смартфонів працювали під керуванням ОС Symbian[28] розробника Nokia. Це була ОС для таких популярних брендів, як Samsung, Sony Ericsson, Motorola та, відповідно, Nokia. Розробникам Symbian вдалося отримати значну економію пам'яті, поліпшення кешування коду і, як наслідок, прискорення роботи програм при знижених вимогах до енергоспоживання. З точки зору розробки відмінною особливістю системи є повноцінна її об'єктно-орієнтована архітектура.

Корпорація Microsoft також випустила ОС для мобільних пристроїв: Windows CE[29] – багатозадачна багатониткова багатоплатформна ОС з підтримкою реального часу. Перша версія була скороченою версією ОС Windows 95, наступні версії створені на базі ядра NT та спроєктовані так, щоб була сумісність з ОС класу Windows для настільних комп'ютерів. Також у 2000 р. було випущено ОС Windows Mobile[30] для власних апаратних платформ Pocket PC та Smartphone.

У 2001 р. на ринок було випущено смартфон Kyocera QCP 6035, який працював під управлінням ОС Palm OS[31]. Ця ОС була реалізована на широкому спектрі мобільних пристроїв, включаючи смартфони, наручні

[28] Перший випуск 05.06.1998 р., остання версія – 11.10.2012 р.
[29] Перший випуск у 1996 р., останній – 14.06.2013 р.
[30] Перший випуск 19.04.2000 р., остання версія 6.5.3 від 02.02.2010 р.
[31] Перший випуск у 1996 р., остання версія випущена в 2007 р.

годинники, кишенькові ігрові консолі, зчитувачі штрих-кодів і GPS-пристрої. Прикладами компаній, які використовувалися в своїх пристроях ОС Palm OS, можна назвати такі відомі компанії як Sony, Kyosera, Samsung, Lenovo, Garmlin. Особливість цієї ОС полягає в тому, що її ядро багатозадачне, хоча для користувача ОС однозадачна (можна запускати лише один застосунок), але з можливістю виконання фонових процесів (програвання музики та ін.). Останній смартфон на Palm OS був представлений у кінці 2007 р., а в січні 2009 р. компанія Palm випустила основану на Linux нову ОС з відкритим кодом та назвою webOS[32] (з 2010 р. розробка виконується компанією Hewlett-Packard, а з 2013 р. – компанією LG Electronics для телевізорів: її встановлено більш ніж на 70 мільйонів пристроїв).

Серед інших ОС на початку 2000-х років можна назвати Blackberry OS[33] розроблену компанією RIM (Research In Motion Limited) для своїх смартфонів. Особливістю ОС була підвищена система безпеки, що давала можливість шифрувати дані та застосунки, а також віддалено керувати пристроєм у разі втрати або крадіжки.

У 2007 р. було випущено першу ОС для мобільних пристроїв компанії Apple – iOS (повна назва iPhone OS). Спочатку ОС була розроблена тільки для iPhone та iPod touch, проте пізніше система стала доступною і для таких пристроїв, як iPad та Apple TV. Ядро iOS майже ідентичне ядру операційної системи Apple OS X. Слід також зауважити, що взаємна інтеграція OS X та iOS в останні роки зросла, і компанія Apple розглядає ці дві ОС як єдину платформу. Інтерфейс користувача заснований на функції «мультитач» (англ. Multi-Touch), тобто сенсорної системи введення.

У 2010 р. вийшла ОС Windows Phone на базі Windows Mobile але не сумісна з нею (остання версія була випущена разом з Windows 10 та отримала назву Windows 10 Mobile).

---

[32] Перший випуск у 2009 р., остання версія webOS OSE 2.26.0 від 05.06.2024 р., ресурс у мережі Інтернет – https://www.webososе.org/

[33] Перший випуск 19.01.1999 р., остання версія 7.1.0.2931 за листопад 2013 р.

Windows Phone та iOS від Apple почали захоплювати частку ринку Symbian. Ринкова частка Symbian різко впала: якщо ще осінню 2011 р. на Symbian працювало 16.9 % пристроїв, то через рік вже тільки 2.6 %. У 2011 р. Nokia відмовилася від Symbian і оголосила, що зосередиться на Windows Phone як своїй основній платформі.

А вже 09.10.2017 р. у компанії Microsoft заявили про припинення створення нових пристроїв та оновлень Windows 10 Mobile і в 2019 р. власникам мобільних пристроїв під управлінням цієї системи було рекомендовано перейти на пристрої з Android чи iOS.

Певний час ОС від Apple і RIM були популярними, але не знадобилося багато часу, щоб Android[34], операційна система на базі Linux, випущена Google у 2008 р. (спочатку розробка велась компанією Android Inc., яку придбала Google), випередила всіх своїх суперників. Основні риси системи, які сприяли цьому:

♦ Android – платформа з відкритим вихідним кодом;

♦ підтримка програм сторонніх розробників за допомогою надійного та стабільного API (англ. Application Programming Interface);

♦ дозвіл всім програмам сторонніх розробників конкурувати на рівних умовах;

♦ реалізовано модель безпеки застосунків, коли сама ОС захищає користувача від неправильної поведінки застосунків, причому не лише програм із помилками, які можуть спричинити її збій, але й неправильного використання пристрою та даних користувача на ньому;

♦ типова взаємодія з користувачем, коли користувач часто проводить короткий час у багатьох програмах – систему оптимізовано для таких випадків за допомогою реалізації швидкого запуску та перемикання програм;

[34] Перший випуск 23.09.2008 р., остання версія 15 від 03.09.2024 р., ресурс у мережі Інтернет – https://android.com/

♦ спрощена взаємодія з застосунками – користувачам немає потреби турбуватися про закриття застосунків після завершення їх роботи;

♦ можливість для застосунків взаємодіяти та працювати спільно багатьма безпечними способами;

♦ Android – повноцінна ОС загального призначення. Дизайн достатньо функціональний та потужний, як у традиційних ОС.

Для виробників телефонів Android мав перевагу в тому, що він був відкритим вихідним кодом і доступний за вільною ліцензією. Як наслідок, виробники могли з легкістю попрацювати з ним і адаптувати його до свого апаратного забезпечення. Крім того, у нього є величезна спільнота розробників, які пишуть програми, переважно на мові програмування Java. Незважаючи на це, останні роки показали, що домінування може тривати недовго, і конкуренти Android прагнуть повернути собі частину ринку.

Ще деякі відомі ОС для мобільних пристроїв:

♦ HarmonyOS[35] – ОС, розроблена компанією Huawei. Вона призначена для використання на різних пристроях, включаючи смартфони, планшети та інші пристрої. Основні переваги HarmonyOS полягають у здатності забезпечувати високу продуктивність при низькому енергоспоживанні, а також ефективному механізму перемикання між застосунками;

♦ Tizen[36] – ОС на базі ядра Linux, розроблена компаніями Samsung і Intel. Вона може бути використана на різних пристроях, включаючи смартфони, телевізори та інші пристрої. Tizen пропонує високу продуктивність та безпеку;

[35] Перший випуск 09.13.2009 р., остання версія NEXT.0.0.70 від 27.09.2024 р., ресурс у мережі Інтернет – https://developer.harmonyos.com/en/

[36] Перший випуск 30.04.2012 р., остання версія 8.0 M2 від 31.10.2023 р., ресурс у мережі Інтернет – https://www.tizen.org/

♦ Sailfish OS[37] – багатозадачна ОС на базі ядра Linux, розроблена компанією Jolla. Вона має свій власний інтерфейс і може запускати програми Android. Використовується на деяких телефонах Sony;

♦ Ubuntu Touch[38] – ОС, розроблена компанією Canonical Ltd. Вона заснована на Ubuntu Linux і може працювати на різних пристроях, включаючи смартфони та планшети. Ubuntu Touch має власний інтерфейс і підтримує запуск програм з Ubuntu Software Center. Представлена на деяких телефонах Meizu, Volla, Fairphone та ін.

Сучасні ОС для смартфонів надають такі сервіси:

♦ оплата за допомогою мобільного телефону. Ця тенденція знаходиться на стадії зростання, оскільки користувачі все більше прагнуть спростити оплату товарів та послуг та здійснювати її за допомогою своїх мобільних телефонів;

♦ Progressive Web Apps – програми, які запускаються на мобільних пристроях, але не потребують встановлення через App Store або Google Play. Використання PWA дозволяє підвищити продуктивність та чутливість застосунків, а також покращити їхню безпеку та доступність;

♦ 5G технологія мобільного зв'язку, яка забезпечує більш високу швидкість передачі даних та скорочений час відгуку в порівнянні з мережею LTE;

♦ розпізнавання обличчя та голосу – ця технологія використовується для спрощення процесу ідентифікації користувачів та забезпечує більш високий рівень безпеки в мобільних застосунках;

♦ голосові та перекладацькі асистенти (наприклад, Siri або Google Assistant) – допомагають полегшити роботу з мобільними пристроями та застосунками. Більш того, поява нових мовних модулів та машинного

[37] Перший випуск 16.11.2013 р, остання версія 4.6.0.15 від 20.09.2024 р., ресурс у мережі Інтернет – https://sailfishos.org/

[38] Перший випуск 21.02.2013 р., остання версія 20.04 OTA-5 від 30.07.2024 р., ресурс у мережі Інтернет – https://ubuntu-touch.io/

навчання дозволяє асистентам ставати все більш точними та зручними у використанні;

♦ доповнена реальність (англ. AR – Augmented Reality) та віртуальна реальність (англ. VR – Virtual Reality) – ці технології використовуються для створення інтерактивних застосунків та ігор, які мають високий потенціал для залучення нової аудиторії;

♦ мобільна аналітика – включає процеси збору, аналізу та використання даних для оптимізації продуктивності застосунків і підвищення задоволеності користувачів;

♦ блокчейн-технології – для забезпечення безпеки посвідчень лояльності, онлайн-банкінгу та багатьох інших застосунків, які потребують високого ступеня захисту та конфіденційності.

Варто вказати, що для мобільних ОС існують ще користувацькі інтерфейси (англ. UI – User Interface), які розширюють функціональні можливості, а для самих мобільних пристроїв – вбудовані ОС реального часу, які працюють разом з більшістю наведених мобільних ОС.

Користувацький інтерфейс для мобільних ОС не є окремим застосунком – це змінений варіант основної ОС, в який додано додаткові функції, візуальні ефекти, компоненти системи. Саме тому більшість таких інтерфейсів також називають мобільними ОС. Серед сучасних інтерфейсів можна перерахувати наступні:

♦ MIUI[39] інтерфейс користувача, який використовується в операційній системі Android на смартфонах Xiaomi. MIUI має безліч налаштувань інтерфейсу користувача, які роблять його дуже гнучким. Але є певні особливості: MIUI часто споживає багато ОЗП та вбудованої пам’яті, що призводить до зниження продуктивності; наявність реклами у вбудованих програмах; велика кількість попередньо встановлених партнерських

[39] Перший випуск 16.08.2010 р., остання версія V14.0.23.10.8.DEV від 17.08.2023 р., ресурс у мережі Інтернет https://www.mi.com/global/miui, остання версія OS2.0.240823.1.VNLCNXM (HyperOS 2.0) від 23.08.2024 р., ресурс для Hyper OS – https://www.mi.com/global/hyperos

застосунків, набір яких залежить від регіону. 26.10.2023 р. було випущено нову ОС-оболонку Xiaomi Hyper OS на заміну MIUI;

♦ ColorOS[40] – користувацький інтерфейс, який використовується в операційній системі Android на смартфонах Oppo. Він має багато функцій і опцій налаштування інтерфейсу користувача, таких як теми, жести та ін. При роботі з пам'яттю фонові процеси не вивантажуються з часом з оперативної в зовнішню пам'ять як в ОС Android, що дозволяє їм виконуватись без обмежень до примусової зупинки;

♦ EMUI[41] (Emotion UI, Magic UI) – інтерфейс користувача, який використовується в ОС Android та HarmonyOS на смартфонах Huawei. EMUI має свій власний стиль і дизайн, а також пропонує багато функцій і налаштувань інтерфейсу користувача;

♦ One UI[42] – користувацький інтерфейс, який використовується в операційній системі Android смартфонів Samsung. One UI має зручний і зрозумілий інтерфейс, а також надає багато можливостей для налаштування інтерфейсу користувача.

Як було вказано раніше, у смартфонах крім мобільної ОС працює ще вбудована ОС реального часу. Ця ОС зберігається в прошивці та працює на комунікаційному процесорі. Так, у комунікаційних процесорах Qualcomm (наприклад, MSM6280) RTOS називається AMSS – на зміну їй у сучасних процесорах (платформа Snapdragon) використовується ОС QuRT, для процесорів MTK (MediaTek) використовується ОС Nucleus:

♦ AMSS (Advanced Mobile Subscriber System) заснована на власному пропрієтарному (англ. Proprietary) закритому ядрі REX та виконує до 69 одночасних процесів, які відповідають за функціонування модулів від USB до GPS. AMSS виконує безліч функцій, таких як керування модулем зв'язку,

[40] Перший випуск 23.09.2013 р., остання версія 14 від 16.11.2023 р., ресурс у мережі Інтернет – https://www.oppo.com/en/coloros14/

[41] Перший випуск 30.07.2012 р., остання бета версія 14.2 від 25.04.2024 р., ресурс у мережі Інтернет – https://consumer.huawei.com/en/emui-13/

[42] Перший випуск 07.11.2018 р., остання версія 6.1.1 від 10.07.2024 р., ресурс у мережі Інтернет – https://www.samsung.com/global/galaxy/apps/one-ui/

керування живленням, керування апаратним забезпеченням та ін. Система працює на процесорі ARMv5;

♦ ОС REX (Real-time Executive Operating System) розроблена Qualcomm для розробки дворежимної абонентської станції (DMSS) мобільного телефону на основі процесора ARM або розширеного програмного забезпечення для абонента (AMSS). REX – це комбінація двох ОС: вбудованого мікроядра L4A Pistachio та Iguana з великими модифікаціями та розширеннями від Qualcomm і HTC. Вона має наступні функції: багатозадачність з витісненням, управління завданнями, синхронізація завдань, ексклюзивне блокування, таймер, управління перериваннями, використовує менше 5 КБ ПЗП. REX не надає функцій захисту пам'яті, але є можливості її керування;

♦ ОС QuRT має наступні функції: багатонитковість, м'ютекси, семафори, таймери, обробка переривань, керування пам'яттю, а також дозволяє програмам і ниткам виконуватися в окремих захищених адресних просторах для підвищення безпеки та стабільності системи. Розробники мобільних ОС можуть писати користувацькі програми, призначені для використання QuRT на C/C++ та/або асемблері;

♦ ОС Nucleus від Mentor Graphics складається з ядра, сервісів, модуля зв'язку, файлової системи, модуля мережі, IoT Framework (набір протоколів, інструментів і стандартів, які забезпечують розробку та підтримку Інтернету речей), модуля бездротового зв'язку, модуля безпеки, графічного інтерфейсу. Вона має ядро реального часу з пріоритетним плануванням, підтримку динамічного зв'язування за допомогою завантажувальних модулів, інтерфейси для C++, інтерфейс POSIX, підтримку SMP (англ. Symmetric Multiprocessing) та 64-розрядних архітектур, масштабування відповідно до пристроїв з обмеженою пам'яттю, вбудовану структуру керування живленням.

В останні роки відбувається зростання популярності хмарних обчислень – це дозволяє користувачам отримувати доступ до застосунків та даних через Інтернет. Хмарні обчислення (англ. Cloud Computing) є одним з найпопулярніших напрямів розвитку IT. «Хмара» (англ. Cloud) – це представлення сервісів, що надаються через Інтернет або іншу

комунікаційну мережу. Хмарні обчислення – модель обчислень, заснована на динамічно масштабованих і віртуалізованих ресурсах (даних, застосунках, ОС та ін.), які доступні і використовуються як сервіси через Інтернет і реалізуються за допомогою високопродуктивних центрів обробки даних.

З точки зору користувачів існує сукупність «хмар» (загальнодоступні, корпоративні, приватні та ін.), що надаються різними компаніями, для використання потужних обчислювальних ресурсів, яких немає в індивідуального користувача. Як правило, «хмарні» сервіси платні.

Головною перевагою використання хмарних обчислень, яка покладена в основу технології, є балансування робочого навантаження, за рахунок чого досягається більш ефективне використання ресурсів обчислювальної системи.

Недолік хмарних обчислень у тому, що користувач виявляється повністю залежним від використовуваної їм «хмари» (у якій доступні використовувані ним дані і програми) і не може управляти не лише роботою «хмарних» комп'ютерів, але навіть резервним копіюванням своїх даних. У зв'язку з цим виникає ціла низка важливих запитань щодо безпеки хмарних обчислень, збереження конфіденційності призначених для користувача даних тощо.

Хмарна ОС (англ. Cloud OS) – це ОС, спеціально розроблена для роботи в хмарному обчислювальному та/або віртуалізаційному середовищі, вона керує роботою ресурсів хмарних обчислень. Вона розроблена для роботи у віртуальному середовищі «хмари», забезпечуючи користувачам і застосункам доступ до ресурсів через Інтернет. Хмарна ОС служить інтерфейсом між зовнішніми сервісами та користувачами, покращуючи процес управління зберіганням, обчислювальною потужністю та мережевими можливостями в «хмарі».

Переваги хмарної ОС: ПЗ коштує недорого, покращена продуктивність, оновлення ПЗ відбувається без припинення роботи та непомітно для користувачів, підвищення надійності даних.

Недоліки хмарної ОС: потрібен постійний доступ до мережі Інтернет, швидкість з'єднання впливає на загальну продуктивність, кількість доступних функцій обмежена.

Хмарна ОС складається з трьох основних частин:

1) клієнтська ОС для ПК користувача, призначена для завантаження застосунків з Інтернету або локальної мережі;

2) програмна платформа на стороні сервера, яка забезпечує повну інфраструктуру для налаштування служб хмарних обчислень;

3) серверна ОС для середовища хмарних обчислень. Це може бути безпосередньо ОС або монітор віртуальної машини (VMM) або обидва компонента одразу.

Зазвичай для користувача доступ до програмної платформи на стороні сервера виконується за допомогою будь-якого Інтернет-браузера незалежно від використовуваних пристроїв (смартфона, планшета, настільного комп'ютера). Наведемо деякі з перших хмарних ОС.

♦ Joli OS (проєкт закрито) – заснована на ядрі Linux. «Хмару» Jolicloud[43] представляла Joli OS, яку можна було або встановити на локальній обчислювальній системі (настільна версія), або використовувати через браузер (веб-версія). Вимоги до обладнання досить низькі, що дозволяло встановлювати систему навіть на слабких конфігураціях, але інтерфейс був побудований на сучасних технологіях HTML5, що забезпечувало високу швидкість роботи. Joli OS пропонувала повноцінний досвід взаємодії з хмарними системами. Joli OS могла зберігати і використовувати застосунки в будь-який момент. Jolicloud мала більше 15000 веб-застосунків: всередині хмарного сервісу пропонувався вибір автоматичного підключення популярних Інтернет-сервісів – Box, Dropbox, Google Drive, SkyDrive, Instagram, Vimeo, YouTube, SoundCloud, Flickr та ін. Користувач міг централізовано переглядати документи, фотографії, відео та музику з

[43] Перший випуск у липні 2010 р., фінальна версія 1.2 від 09.03.2011 р., ресурс у мережі Інтернет – https://www.jolicloud.com/jolios/ (проєкт закрито)

авторизованих соціальних сервісів. Однак на цьому всі активні дії закінчувалися: редагувати або змінювати файли не можна.

♦ Glide OS (проєкт закрито) – крос-платформний робочий стіл, повністю побудований на Adobe Flash. Сервіс не мав окремих застосунків на смартфонах, але мав окрему мобільну версію на HTML5 з обмеженим функціоналом. Підтримував три режими відображення: робочий стіл (десктоп), Glide HD та Інтернет-портал. У режимі робочого столу доступно 15 програм: текстовий редактор, фоторедактор, календар, відео- та аудіоплеєр, поштовий клієнт, різні утиліти.

♦ AstraNOS[44] – створена за подобою ОС Mac OS. Сервіс написаний на мові програмування HTML і JavaScript. В AstraNOS відсутнє хмарне сховище і взаємодія з настільним комп'ютером або мобільним пристроєм. Основною перевагою AstraNOS є її безкоштовність – проєкт повністю некомерційний.

♦ SilveOS (проєкт закрито) розроблено за допомогою Silverlight. Її можна було запускати в будь-якому браузері на пристрої, де встановлений Silverlight. Підтримувалась певна кількість вбудованих застосунків, які дозволяли писати повідомлення, слухати музику, робити замітки. Сервіс містив безліч обмежень – немає збереження файлів з персонального комп'ютера в «хмару», в інтерфейсі не передбачено контекстне меню та ін.

♦ Cub Linux (проєкт закрито) – хмарна ОС, основана на Ubuntu, основна ідея якої – імітувати зовнішній вигляд і функціональність Chrome OS, ця ОС була повноцінною – локально можна було встановлювати будь-які застосунки Linux.

На сучасному етапі розвитку хмарних обчислень вже не ведеться розробка окремих хмарних ОС, вони переросли в PaaS (Platform as a Service) – платформи, які надають хмарне ПЗ як сервіс через веб-інтерфейс – SaaS (Software as a Service). На хмарну ОС можна завантажувати своє ПЗ та використовувати його. Хмарна ОС має серверну віртуалізовану

[44] Ресурс у мережі Інтернет – http://astranos.org

інфраструктуру IaaS (Infrastructure as a Service), таку як обчислювальні потужності, сховища та мережі.

Найпопулярніша «хмарна» платформа – Microsoft Windows Azure[45]. Один з компонентів платформи Azure Services Platform – це ОС Windows Azure, що керує дисковим простором, застосунками і мережами. Концепція хмарних обчислень – це використання обчислювальних потужностей, дискового простору і каналів зв'язку «обчислювальної хмари» для виконання трудомістких завдань. Навантаження між комп'ютерами, що входять у цю хмару, розподіляється автоматично. Більшість хмарних застосунків працюють у браузері.

Іншою відомою платформою є Amazon Web Services (AWS)[46]. Її технологія дозволяє абонентам мати в своєму розпорядженні повноцінний віртуальний кластер комп'ютерів, який завжди доступний через Інтернет. Віртуальний комп'ютер AWS має більшість атрибутів реального комп'ютера (процесор, відеокарту, локальну та оперативну пам'ять, жорсткий диск або SSD-накопичувач, операційну систему на вибір, мережу, попередньо встановлені прикладні програми). Кожна система AWS віртуалізує консольне введення/виведення, що дозволяє користувачам AWS підключатися до своєї системи AWS за допомогою браузера.

1.4. Складові частини ОС

ОС можна розглядати як систему, яка складається з декількох підсистем, кожна з яких управляє своїми категоріями ресурсів обчислювальної системи. Один з варіантів поділу ОС на підсистеми пов'язаний з виділенням чотирьох категорій ресурсів: оперативна пам'ять, центральний процесор, зовнішні пристрої і файли. Відповідні чотири підсистеми названі підсистемою управління пам'яттю, підсистемою

[45] Перший випуск 27.10.2008 р., останні версії: для Android за 20.09.2024 р., для iOS за 23.09.2024 р., ресурс у мережі Інтернет – https://azure.microsoft.com/

[46] Перший випуск у березні 2006 р., ресурс у мережі Інтернет – https://aws.amazon.com/

управління процесором, підсистемою управління введенням-виведенням, і підсистемою управління файловими системами. Всі підсистеми між собою взаємодіють (рис. 1.4). Кожна підсистема реалізована у вигляді менеджерів – модулів ОС, виконуючих операції по управлінню вказаними типами ресурсів.

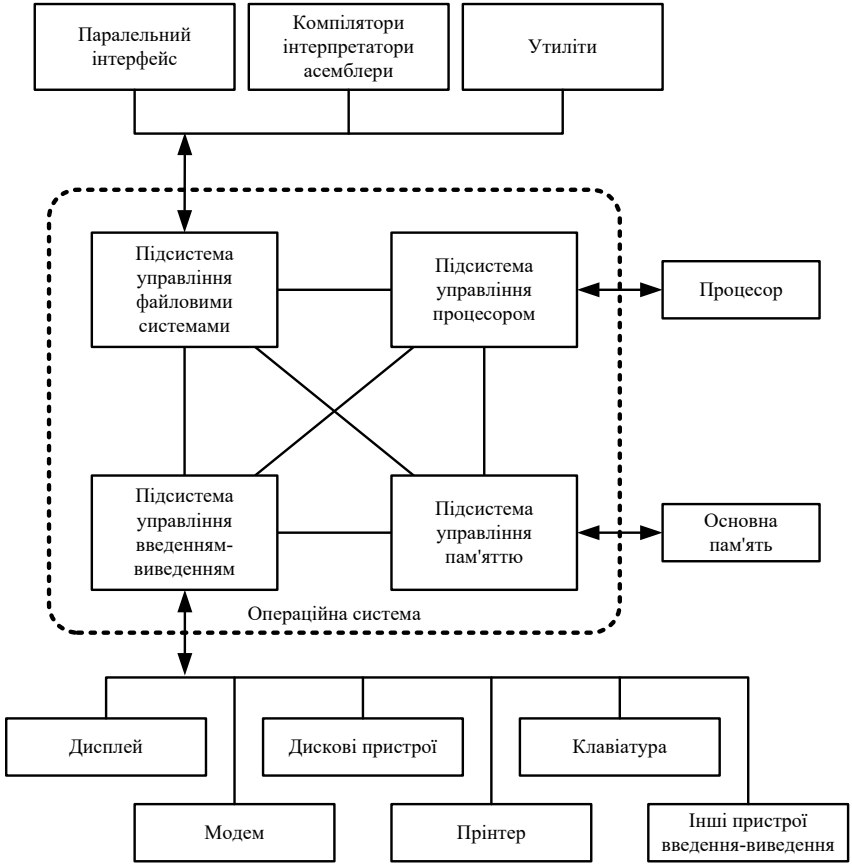


Рисунок 1.4 – Підсистеми операційної системи

Незалежно від розміру або конфігурації ОС, кожен з менеджерів повинен виконувати такі задачі:

1) виконувати постійний контроль за ресурсами;

2) забезпечувати відстеження запитів на ресурси (кому, скільки і на який час виділені ресурси);

3) виділяти ресурси в разі можливості;

4) звільняти або перерозподілювати ресурси в разі можливості.

Наприклад, менеджер пам'яті відповідає за управління оперативною пам'яттю. Він перевіряє вагомість кожного запиту на виділення пам'яті, і, у разі прийняття рішення на дозвіл, менеджер пам'яті виділяє блок пам'яті необхідного розміру з нерозподіленого простору. У багатокористувацькій ОС також ведеться таблиця для відстеження, який блок пам'яті кому належить. Нарешті, коли наступає час звільнити пам'ять, менеджер пам'яті виконує операції по її звільненню та поверненню в нерозподілений простір для можливого подальшого використання.

Але основною задачею менеджера пам'яті є збереження простору в оперативній пам'яті, який займає сама ОС – недопустимо дозволити, щоб будь-яка її частина була випадково або навмисно змінена.

Менеджер процесора (підсистема управління процесором) вирішує, як розподілювати ресурс центрального процесора. Важливою функцією менеджера процесора є відстеження статусу кожного процеса. Він управляє переходами процесів від одного стану виконання до іншого, його робота може бути порівняна з диспетчером. Як тільки менеджер процесора виділяє процесорний час будь-якому процесу, він встановлює необхідні регістри і таблиці, а також, коли робота закінчена або вичерпано максимальний час роботи, він перерозподіляє процесорний час іншому процесу.

Менеджер процесора має два рівня управління: перший – управління завданнями, тому що вони надходять у систему, і другий – управління кожним з процесів у межах завдання. Завдання (англ. Job) – це сукупність програм та їх даних, яка оброблюється ОС як одне ціле. Першою частиною управляє планувальник завдань, високорівнева частина менеджера процесорного часу, який приймає або відхиляє завдання, які надходять у систему. Другою частиною управляє планувальник процесів, низькорівнева частина менеджера процесорного часу, і він вирішує, який процес отримує процесорний час та на який період.

Менеджер пристроїв (підсистема управління введенням-виведенням) контролює кожний пристрій, канал, блок управління. Його задача – обрати самий ефективний шлях розміщення всіх пристроїв системи (дисплей, клавіатура, принтер, дисководи, модем та ін.) згідно з алгоритмами, що реалізовані в ОС. Менеджер пристроїв розміщує, запускає роботу та звільняє пристрої.

Четверта частина, файловий менеджер (підсистема управління файловими системами), відстежує кожен файл у системі, серед яких сервісні програми; компілятори, інтерпретатори, компонувальники; файли даних і прикладних програм. Використовуючи політики доступу, він забезпечує обмеження доступу до кожного файла. Файловий менеджер також контролює рівень доступу кожного користувача для роботи з файлами (наприклад, тільки читання на відміну від читання і запису, або повноваження по створенню або видаленню). Файловий менеджер також виділяє ресурс при відкритті файла і звільнює його при закритті файла.

Наведена структурна організація ОС на основі різних програмних компонентів (підсистем) визначає її архітектуру. У загальному вигляді ОС можна поділити на дві групи компонентів: ядро (англ. Kernel) і зовнішні (допоміжні) модулі.

Ядро ОС – частина ОС, яка постійно знаходиться в оперативній пам’яті та виконує найбільш важливі функції по організації обчислювального процесу і підтримці ПЗ: визначає черговість виконання процесів центральним процесором, розподіляє оперативну пам’ять і інші ресурси обчислювальної системи, займається обробкою переривань та виключних ситуацій. До допоміжних модулів відносяться компілятори, налагоджувачі, редактори, архіватори, різноманітні бібліотеки і оболонки користувача. Важливою властивістю ядра є його робота в привілейованому режимі (режимі ядра) процесора (англ. Kernel mode), що дозволяє в повному обсязі контролювати доступ до пам’яті, регістрів, пристроїв введення-виведення, виконувати перемикання процесора з процеса на процес.

Більшість програм, як системних (що входять до складу ОС), так і прикладних, виконуються в непривілейованому користувацькому режимі роботи (англ. User mode) процесора і отримують доступ до пристроїв (і, при

необхідності, до інших ресурсів ядра, а також ресурсів інших програм) тільки за допомогою системних викликів (англ. System calls), як зображено на рис. 1.1.

1.5. Архітектура ОС

Архітектурні особливості ОС відрізняються одна від одної. Вони визначаються побудовою підсистем ОС та організацією інтерфейсу між підсистемами та процесами. Серед найбільш поширених архітектурних рішень можна виділити наступні:

- ♦ монолітне ядро;
- ♦ багаторівнева система;
- ♦ мікроядерна система;
- ♦ гібридне ядро;
- ♦ екзоядро;
- ♦ наноядро;
- ♦ віртуальна машина.

Для того, щоб оцінити вказані архітектурні особливості ОС необхідно виконати аналіз ефективності їх побудови. На сучасному етапі розвитку ОС сформовано цілий ряд вимог до них, перевірка на відповідність цим вимогам дозволить оцінити якість реалізації ОС [44, 45].

Вимоги до ОС поділяються на вимоги функціональної повноти і експлуатаційні вимоги. До вимог функціональної повноти відносяться: ефективність, зручність. Ефективність основана на тому, що ОС – це складний комплекс програмних засобів, який використовує значну частину апаратних ресурсів для своїх власних потреб. Тобто сама система повинна бути як можна більш економічною, щоб більша частина ресурсів була в розпорядженні користувачів. Крім того, система повинна управляти ресурсами користувачів так, щоб мінімізувати час простою (максимізувати завантаженість ресурсів). Зручність ОС визначається тим, що ОС повинна бути достатньо гнучкою і зручною для користувача. Крім того, сучасна ОС повинна підтримувати мультипрограмну обробку, віртуальну пам’ять, багатовіконний графічний інтерфейс користувача.

Наступний перелік формує експлуатаційні вимоги:

1. Розширюваність (англ. Extensibility). Код ОС повинен бути написаний таким чином, щоб зміни, які вносяться, не порушували цілісність системи. Зміни, які вносяться, в основному стосуються отримання нових властивостей, таких як підтримка нових типів пристроїв або мережних технологій. Розширюваність дозволяє покращувати ОС з оновленням периферійного обладнання. Наприклад, сучасні модифікації в ОС пов'язані з новими технологіями зберігання даних (накопичувачі SSD), мережевої взаємодії (бездротові мережі, оптичні канали високої пропускної здатності), новими інтерфейсами (введення за допомогою жестів, голосове введення, сенсорні панелі). Збереження цілісності коду ОС з огляду на те, що відбувається постійна модернізація апаратури, є однією з цілей проєктування ОС.

2. Переносимість (англ. Portability). Код ОС повинен легко переноситися з однієї апаратної платформи на іншу. У тому випадку, якщо ОС має декілька варіантів реалізації для різних апаратних платформ, вона називається багатоплатформною (кросплатформною, англ. Multi-platform OS або Cross-platform OS). Вимога переносимості тісно пов'язана з розширюваністю: якщо розширюваність дозволяє покращувати ОС, то переносимість дає можливість переміщувати всю систему на іншу машину, з іншим процесором, з мінімальними змінами в коді. Залежність від апаратури полягає в розбіжностях на різних апаратних платформах низькорівневих механізмів маніпулювання регістрами процесора, апаратно-залежних структур даних (контекст процеса, дескриптори сторінок та сегментів), звернення до контролерів периферійного обладнання. Для можливості переносимості необхідно, щоб код був написаний мовою високого рівня (наприклад – мова С кросплатформна на рівні компіляції). Програми, написані мовою асемблера, можуть бути переносимими тільки у випадках повної сумісності в командах. Більшість переносимих ОС написано мовою С, асемблер використовується тільки для реалізації низькорівневих операцій з апаратурою. Слід мінімізувати або, по можливості, виключити частини коду, які безпосередньо взаємодіють з апаратурою. Для переносимості асемблерний код ізолюється в невеликих

модулях, що дозволяє замінювати його аналогічними модулями залежно від апаратної платформи. Але переносимість не завжди може бути реалізована. Так, якщо ОС побудована на 32-бітових адресах, то вона не може бути перенесена на апаратну платформу з 16-бітовими адресами.

3. Сумісність (англ. Compatibility). ОС повинна мати засоби для виконання ПЗ, написаного для інших ОС або більш ранніх версій даної ОС. Сумісність може бути реалізована на рівні двійкових кодів і на рівні вихідних текстів.

Двійкова сумісність дозволяє запускати одну й ту саму програму в різних ОС, але для цього необхідна сумісність на рівні команд процесора, системних викликів та викликів бібліотечних функцій. Двійкова сумісність між процесорами, що базуються на різних архітектурах, вимагає написання спеціальних емуляторів та використання прикладних середовищ у зв'язку з різними машинними командами, які використовуються цими процесорами. Для виконання коду в гостьовій ОС потрібні: процедура завантаження, адаптована під формат виконуваного файла; інтерпретація команд цільового процесора на гостьовому процесорі (якщо процесорні архітектури відмінні); імітація системних викликів з використанням заздалегідь написаної бібліотеки функцій аналогічного призначення.

Приклади середовищ для забезпечення сумісності:

♦ Wine[47] (абр. Wine Is Not an Emulator) – середовище для запуску в ОС Linux програм, розроблених для Microsoft Windows – реалізація Windows API з підтримкою 64-бітних застосунків. Wine приймає системні виклики застосунків Windows до бібліотек ОС та підміняє їх своїми;

♦ Cygwin[48], навпаки, є інструментом для перенесення програм ОС UNIX/Linux у Windows. Cygwin складається з бібліотеки DLL (*cygwin1.dll*), яка працює як емулятор, надаючи функціональність у вигляді середовища

[47] Перший випуск у червні 1993 р., остання стабільна версія 9.0 від 16.01.2024 р., ресурс у мережі Інтернет – https://www.winehq.org/

[48] Перший випуск у 1995 р., остання версія 3.5.4-1 від 25.08.2024 р., ресурс у мережі Інтернет – https://cygwin.com/

системних викликів POSIX API, та колекції інструментальних засобів, які відтворюють звичне середовище Linux.

Сумісність на рівні вихідних текстів потребує перекомпіляції текстів у нові програми, тобто необхідний відповідний компілятор у складі ПЗ. Також потрібна сумісність на рівні бібліотек та системних викликів. Засобами забезпечення сумісності на рівні вихідних кодів є стандартизована мова високого рівня та стандартизовані інтерфейси системних викликів. Прикладом стандартизованої процедурної мови програмування є мова С (останній стандарт для мови ISO/IEC 9899:2018 вийшов у червні 2018 р., станом на початок 2024 р. ведеться розробка нового стандарту на його заміну: ISO/IEC DIS 9899). Найбільш відомим набором стандартів, що описують інтерфейси між операційною системою і прикладною програмою, є переносний інтерфейс ОС UNIX POSIX, який створено для забезпечення сумісності різних UNIX-подібних ОС та переносимості прикладних програм на рівні вихідного коду. Використання стандарту POSIX (останні версії: IEEE Std 1003.1-2017, ISO/IEC/IEEE 9945:2009, станом на початок 2024 р. ведеться розробка нових стандартів IEEE Std 1003.1-202x та ISO/IEC DIS 9945) дозволяє створювати програми в стилі UNIX, які можуть легко переноситися з однієї обчислювальної системи до іншої.

Для сімейства ОС, що базуються на Linux, є стандарт сумісності LSB (Linux Standard Base), остання версія сімейства стандартів ISO/IEC 23360-x-x:2021 за жовтень 2021 р. LSB визначає: стандартні бібліотеки, кілька команд та утиліт на додаток до стандарту POSIX, структуру ієрархії файлової системи, рівні запуску та різні розширення системи X Window System.

Сумісність на рівні вихідних текстів є важливою для розробників застосунків. Для кінцевих користувачів практичне значення має лише двійкова сумісність, завдяки якій програмні продукти можна використовувати в різних операційних середовищах і різних машинах.

Слід зауважити, що сумісність також має на увазі підтримку інтерфейсів користувача інших ОС.

4. Надійність (англ. Reliability) і відмовостійкість (англ. Fault Tolerance). Основна ідея – система повинна бути такою надійною, як і

апаратура, на якій вона працює. При виникненні помилки в ПЗ або апаратурі система повинна виявити помилку і спробувати виправити її, або звести збитки до мінімуму. Дії ОС завжди повинні бути передбачуваними. Надійність і відмовостійкість ОС залежать від її архітектурної побудови і від якості реалізації.

Помилки в роботі можуть бути викликані:

♦ збоями програмного характеру, які можуть виникати з різних причин: наприклад, вірусна атака, необережне поводження з системними файлами, конфлікт драйверів пристроїв та багато іншого;

♦ апаратними збоями, які можуть виникати через недотримання умов експлуатації обладнання, через некоректний монтаж плат на системну плату та ін.

Відмовостійкість означає здатність системи продовжувати безперебійну роботу, незважаючи на відмову одного або кількох її компонентів. Здатність ОС відновлюватися після відмов та ігнорувати помилки і працювати без збоїв може бути забезпечена апаратним забезпеченням, програмним забезпеченням або комбінованим рішенням.

Відмовостійкі системи гарантують відсутність перерв у роботі завдяки використанню резервних компонентів, які автоматично замінюють несправні компоненти. Вони можуть містити:

♦ апаратні системи з ідентичними або еквівалентними резервними ОС. Наприклад, сервер з ідентичним відмовостійким сервером, що віддзеркалює всі операції резервного копіювання, які виконуються паралельно. Усуваючи одиничні відмови, апаратна відмовостійкість у формі резервування може зробити будь-який компонент або систему набагато безпечнішою та надійнішою. Наприклад, на рівні контролера накопичувача використання надлишкового масиву недорогих дисків (RAID) є логічною стратегією підвищення відмовостійкості, яку можна без складнощів реалізувати;

♦ системи програмного забезпечення, резервне копіювання в яких здійснюється іншим програмним забезпеченням;

♦ резервні джерела живлення (джерела безперебійного живлення – UPS, генератори) можуть допомогти уникнути несправності: якщо

альтернативні джерела автоматично починають працювати під час збоїв живлення, то це гарантує відсутність втрати обслуговування.

Одним з варіантів побудови відмовостійкої архітектури є використання Візантійської відмовостійкості (англ. Byzantine Fault Tolerance, BFT). Системи BFT запобігають простою, навіть якщо певні вузли в системі виходять з ладу або намагаються зашкодити іншим (наприклад, у результаті несанкціонованого доступу або вірусної атаки). Вони важливі для авіації, блокчейну, атомної енергетики та космічної промисловості.

Будь-яка ОС містить програми, що запобігають більшості стандартних збоїв, у тому числі засоби моніторингу стану апаратних компонентів, але для запобігання збоїв, що виникають через навмисні дії користувачів, необхідно встановлювати додаткове ПЗ.

5. Безпека (англ. Security). ОС повинна бути захищена від несанкціонованого доступу. Правила безпеки визначають способи захисту ресурсів користувача та встановлюють квоти за ресурсами для запобігання захопленню користувачем усіх системних ресурсів (наприклад, пам'яті).

Між відмовостійкістю та безпекою є прямий зв'язок. Погано спроєктована система може бути легко виведена з ладу під час атаки. Наприклад, брандмауер, який не є відмовостійким, становить загрозу безпеці для сайту.

Деякі ключові моменти, що стосуються забезпеченню безпеки в ОС (але не обмежуються ними):

- управління виправленнями (англ. Patch Management);
- контроль доступу (англ. Access Control) та автентифікація (англ. Authentication);
- безпечна конфігурація (англ. Secure Configuration);
- управління вразливостями (англ. Vulnerability Management);
- реєстрація та моніторинг активності;
- навчання та підвищення обізнаності користувачів;
- реагування на інцидент;
- фізична безпека та ін.

Будь-яка сучасна ОС повинна мати елементи управління безпекою – це дії або механізми, які запобігають, виявляють або реагують на загрози безпеці ОС. Вони поділяються на три категорії:

♦ превентивні засоби контролю, такі як брандмауери, антивіруси, автентифікація, шифрування та захист;

♦ детективні засоби контролю, які відстежують або сповіщають про діяльність або події ОС, наприклад журналювання, аудит (англ. Auditing), система виявлення вторгнень (англ. Intrusion Detection System, IDS), управління інформаційною безпекою та подіями безпеки (англ. Security information and event management, SIEM);

♦ коригувальні елементи керування, які відновлюють ОС після випадків порушення безпеки, такі як резервне копіювання, виправлення та ізоляція.

Необхідно контролювати й оновлювати безпеку ОС, щоб вона відповідала стандартам безпеки ОС. Це не одноразова діяльність, а постійний процес.

Стандарти безпеки ОС – це рекомендації і керівництва, що визначають вимоги щодо забезпечення безпеки ОС. Ці стандарти розроблені для забезпечення конфіденційності, цілісності та доступності системних ресурсів і даних, а також для нівелювання різних ризиків і загроз безпеки. Вони допомагають організаціям і приватним особам встановити базові показники для безпечного налаштування, розгортання та обслуговування ОС.

Основи стандартів в області безпеки були закладені в документі «Trusted Computer System Evaluation Criteria» (1983 р.) або «Помаранчева книга» (за кольором обкладинки) – критерії оцінки надійних комп'ютерних систем. Цей стандарт входить у серію стандартів інформаційної безпеки («веселкова серія» – всі стандарти мали свої кольори обкладинок), яку було спочатку опубліковано Міністерством оборони США (пізніше – Центром національної комп'ютерної безпеки США). Згідно цього стандарту в більшості популярних ОС гарантується ступень безпеки даних, яка відповідає рівню С2.

Аналогом «Помаранчевої» книги є міжнародний стандарт «ISO/IEC 15408-1:2022. Information security, cybersecurity and privacy protection. Evaluation criteria for IT security», вперше опублікований у 1999 р. (станом на 2024 р. ведеться робота по оновленню: нова версія ISO/IEC WD 15408-1 на стадії розробки). Це більш універсальний і досконалий стандарт, але всупереч поширеній помилці, він не замінив собою «Помаранчеву книгу» через різну юрисдикцію документів – «Помаранчева книга» використовується виключно Міністерством Оборони США, тоді як ISO/IEC 15408 ратифікували багато країн.

У цьому стандарті для характеристики основних критеріїв інформаційної безпеки застосовують модель тріади CIA – три основні характеристики інформаційної безпеки:

♦ конфіденційність (англ. Confidentiality) – загрози, що відносяться до несанкціонованого ознайомлення з інформацією, становлять загрози конфіденційності. Якщо існують вимоги щодо обмеження можливості ознайомлення з інформацією, то відповідні послуги відносяться до критеріїв конфіденційності;

♦ цілісність (англ. Integrity) – загрози, що відносяться до несанкціонованої модифікації інформації, становлять загрози цілісності. У випадку, якщо існують вимоги щодо обмеження можливості модифікації інформації, то їх відносять до критеріїв цілісності;

♦ доступність (англ. Availability) – загрози, що відносяться до порушення можливості використання комп'ютерних систем або оброблюваної інформації, становлять загрози доступності. Якщо існують вимоги щодо захисту від відмови в доступі або захисту від збоїв, то їх відносять до критеріїв доступності.

Ще деякі відомі стандарти безпеки ОС: «NIST SP 800-123. Guide to General Server Security» – посібник із загальної безпеки серверів, «CIS Benchmarks» – інструкції з конфігурації для різних платформ ОС, «PCI DSS. Payment Card Industry Data Security Standard» – стандарти безпеки для індустрії і даних платіжних карток.

Важливо відзначити, що безпека ОС – це неперервний процес, і стандарти безпеки можуть розвиватися протягом усього часу для усунення виникаючих загроз і вразливостей.

6. Продуктивність (англ. Performance). ОС повинна забезпечувати швидкодію, відповідну до апаратної платформи. Продуктивність ОС залежить від архітектурних особливостей її побудови, кількості функцій, які вона виконує, якості її коду. Для користувача ОС повинна бути передбачуваною: користувач повинен мати очікувані терміни отримання результатів роботи ПЗ на основі термінів виконання аналогічних програм або попередніх запусків однієї і тієї ж програми.

ОС – складний комплекс ПЗ, що управляє безліччю фізичних пристроїв, що постійно змінюються та оновлюються, і різними прикладними робочими навантаженнями. При випуску нових версій ОС додаються нові можливості щодо збільшення продуктивності певних робочих навантажень, а вузькі місця, що виникають, усуваються в міру масштабування систем. Деякі зміни, наприклад, для усунення вразливості Meltdown[49], також можуть негативно позначатися на продуктивності. Аналіз та робота над поліпшенням продуктивності ОС – це безперервний процес.

Типова мета аналізу продуктивності ОС – покращити взаємодію з кінцевим користувачем за рахунок зменшення затримок та зниження витрат на обчислення. Зниження витрат можна досягти за рахунок усунення неефективності, підвищення пропускної спроможності та загального налаштування системи.

При аналізі продуктивності враховуються такі параметри [19]:

♦ кількість операцій введення/виведення за секунду (англ. Input/Output Operations Per Second, IOPS) – міра частоти виконання операцій передачі даних. Для дискового введення/виведення під кількістю операцій

[49] Апаратна вразливість у мікропроцесорах Intel, яка дає несанкціонований доступ до захищеної віртуальної пам'яті. Інформацію про вразливість оприлюднено в 2018 р.

введення/виведення мається на увазі сума операцій читання та запису за секунду;

♦ пропускна здатність (англ. Throughput) – швидкість виконаної роботи. Зокрема, у галузі зв'язку цей термін використовується для позначення швидкості передачі даних (байтів або бітів за секунду). У деяких контекстах (наприклад, у базах даних) під пропускною здатністю може матися на увазі швидкість роботи (кількість операцій або транзакцій за секунду);

♦ час відгуку (англ. Response time) – час від початку до завершення операції. Складається з часу очікування, часу обслуговування (англ. Service Time) і часу передачі результату;

♦ затримка (англ. Latency) – час, протягом якого операція очікує на обслуговування. У деяких контекстах під затримкою мається на увазі весь час виконання операції, тобто час відгуку;

♦ споживання (англ. Utilization) – для ресурсів, необхідних для обслуговування, споживання визначає міру зайнятості ресурсу. Для ресурсів, які забезпечують зберігання даних, під коефіцієнтом використання (споживанням) може матися на увазі спожитий обсяг (наприклад, використаний обсяг пам'яті);

♦ насиченість (англ. Saturation) – ступінь заповнення черги запитів на використання ресурсу, які не можна виконати негайно;

♦ вузьке місце (англ. Bottleneck) – ресурс, що обмежує продуктивність системи. Основна робота з підвищення продуктивності системи саме полягає у виявленні та усуненні вузьких місць;

♦ робоче навантаження (англ. Workload) – вхідні дані для системи. Наприклад, для бази даних робочим навантаженням є потік запитів та команд, які надсилаються клієнтами;

♦ кеш (англ. Cache) – область швидкодіючого сховища, де можуть зберігатися копії, або буфери обмеженого обсягу даних. Кеш використовується, щоб запобігти операціям з більш повільними сховищами і тим самим підвищити продуктивність. З економічних причин кеш часто менше основного та повільнішого сховища.

1.5.1. ОС з монолітною структурою

Монолітне ядро (англ. Monolithic kernel) має найдовшу історію розвитку. Це класична архітектура ядер ОС. Усі частини монолітного ядра виконуються в одному адресному просторі. Монолітне ядро складається з множини процедур, які працюють у привілейованому режимі та можуть викликати одна одну. Всі компоненти ОС є частками одного цілого: використовують загальні структури даних, взаємодіють між собою через безпосередній виклик процедур. Це визначає переваги монолітних ядер: швидкість роботи та спрощена розробка їх складових частин. У монолітних ОС ядро співпадає з усією системою. Це, а також використання єдиного адресного простору, приводить до зниження надійності – збій в одному з компонентів може привести до непрацездатності системи в цілому.

У багатьох ОС з монолітним ядром компіляція ядра виконується окремо для кожного комп'ютера. Це також єдиний спосіб змінити склад ядра ОС: додати необхідні або видалити непотрібні компоненти. Оскільки ядро ОС постійно знаходиться в оперативній пам'яті, то видалення непотрібних компонентів є важливою задачею: зменшення обсягу ядра ОС не тільки приводить до економії пам'яті, але й підвищує надійність ОС. Однак монолітні ядра мають дуже складну структуру взаємозв'язків між компонентами, а це ускладнює операції по модифікації коду.

Компоненти монолітної ОС організовані безсистемно і будь-який модуль може звертатися до будь-якого іншого модуля без будь-яких застережень. За аналогією з іншими ОС, ПЗ відокремлено від самої ОС. Це означає, що код ОС виконується в привілейованому режимі процесора та має доступ до системних даних і апаратного забезпечення, а програми запускаються в непривілейованому режимі процесора, з обмеженим набором інтерфейсів доступу до даної системи. Монолітна структура ОС у режимі ядра наведена на рис. 1.5.

Коли програма в режимі користувача викликає системний сервіс, реалізований в інтерфейсі системних викликів, процесор генерує виключення, у результаті якого виконується перехід у режим ядра та викликається відповідна процедура ядра ОС. Завершення системного виклику виконує перемикання назад у режим користувача.

Монолітна структура не примушує приховувати дані в ОС. Це забезпечує кращу продуктивність, але розширення такої системи може бути важкою роботою, оскільки зміна процедури може внести помилки в частини системи, на перший погляд які не пов’язані між собою.

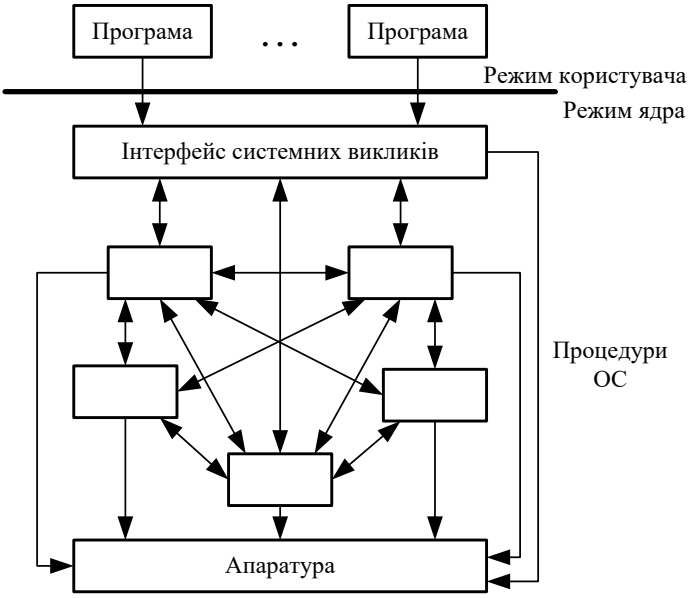


Рисунок 1.5 – Монолітна структура ОС

Прикладами ОС з монолітним ядром є традиційні UNIX (BSD), Linux, MS-DOS, CP/M, Oracle Solaris, MenuetOS[50] (написана мовою асемблера FASM), KolibriOS[51] (також написана мовою асемблера FASM, базується на

[50] Перший випуск 16.05.2000 р., остання 32-бітна версія 0.86B від 02.09.2019 р. (відкритий вихідний код), остання 64-бітна версія 1.54.40 від 04.10.2024 р. (закритий код), ресурс у мережі Internet – https://www.menuetos.net/

[51] Перший випуск у 2004 р., остання версія 0.7.7.0+8770 від 01.08.2024 р., ресурс у мережі Internet – https://kolibrios.org/

MenuetOS), Visopsys[52] (маленька 32-розрядна багатозадачна ОС з вбудованим в ядро графічним інтерфейсом).

На рис. 1.6 наведено структуру ОС MS-DOS. На базову систему введення/виведення (англ. Basic Input/Output System – BIOS), яка не входить до складу ОС та розміщується в енергонезалежній пам'яті, покладено такі функції: під час завантаження ОС – контроль працездатності пристроїв комп'ютера (тестування) та ініціалізація процесу завантаження програм ОС, тобто зчитування інформації з диска та розміщення її в оперативній пам'яті комп'ютера; керування роботою стандартних зовнішніх пристроїв комп'ютера (монітор, клавіатура, зовнішній диск).

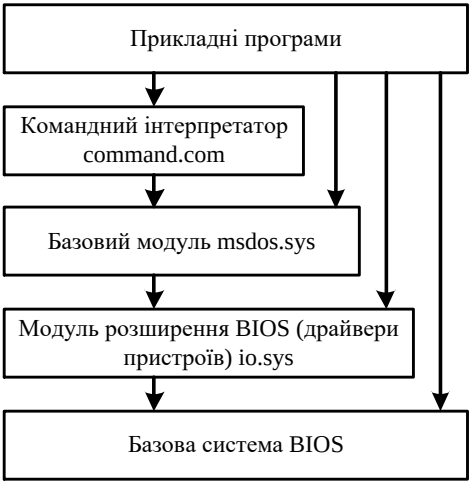


Рисунок 1.6 – Структура ОС MS-DOS

Модуль розширення BIOS – надбудова над BIOS, яка виконує такі функції: у процесі завантаження ОС заміна драйверів, що зберігаються в BIOS, і підключення, якщо потрібно, нових драйверів; організація

[52] Перший випуск у 1997 р., остання версія 0.92 від 21.09.2023 р., ресурс у мережі Internet – https://visopsys.org/

інтерфейсу з BIOS. Модуль розширення BIOS зберігається на системному диску як файл з ім'ям *io.sys* і є невід'ємною частиною ОС.

Базовий модуль MS-DOS реалізує основні функції з управління всіма ресурсами комп'ютера та програмами. Базовий модуль зберігається на системному диску як файл з ім'ям *msdos.sys*. Після завантаження базового модуля він постійно перебуває в оперативній пам'яті комп'ютера (є резидентним).

Командний процесор (інтерпретатор) призначений для організації взаємодії користувача з комп'ютером, тобто користувач дає вказівку MS-DOS виконання тих чи інших дій у вигляді введення з клавіатури відповідних команд. Командний процесор зберігається на системному диску як файл з ім'ям *command.com*.

Інтерфейси та рівні функціональності в ОС MS-DOS недостатньо розділені. Наприклад, прикладні програми можуть отримувати доступ до базових процедур введення-виведення для запису безпосередньо на зовнішні пристрої, такі як дисплей і диски. Це робить MS-DOS вразливою до помилок або зловмисних програм, що спричиняє збої для всієї системи під час збоїв або помилок у програмах користувача. Відсутність механізмів захисту, крім особливостей реалізації ОС, також була обумовлена апаратним забезпеченням. Так, процесор Intel 8088, для якого вона була написана, не забезпечує захист на апаратному рівні.

ОС MS-DOS відноситься до класичної монолітної архітектури – всі процедури ОС зібрані в один файл (*msdos.sys*). Модульне ядро – сучасна, вдосконалена модифікація архітектури монолітних ядер, в якій код ядра поділяється на модулі, що окремо компілюються і завантажуються.

Модульні ядра надають механізм завантаження модулів ядра, які підтримують апаратне забезпечення (наприклад, драйверів). Завантаження модулів може бути як динамічним (без перезавантаження ОС), так і статичним (виконується при перезавантаженні ОС). Модулі ядра, які завантажуються динамічно, в ОС UNIX називаються спільними бібліотеками (англ. Shared Libraries), в ОС Windows – бібліотеками DLL (Dynamic-Link Libraries). Модулі ядра працюють в адресному просторі ядра і можуть користуватися всіма функціями, що надаються ядром. Тому

модульні ядра залишаються монолітними. Модульність ядра здійснюється лише на рівні бінарного образу, а не на архітектурному рівні ядра, оскільки модулі динамічного завантаження розміщуються в адресному просторі ядра й надалі працюють як його частина.

Приклад ОС з монолітною модульною структурою – оригінальна ОС UNIX [24, 34, 37], яка складається з двох окремих частин: ядра та системних програм. Ядро, в свою чергу, поділено на серію інтерфейсів і драйверів пристроїв, які додавались та розширювались протягом багатьох років з розвитком системи, організацію традиційної ОС UNIX наведено на рис. 1.7.

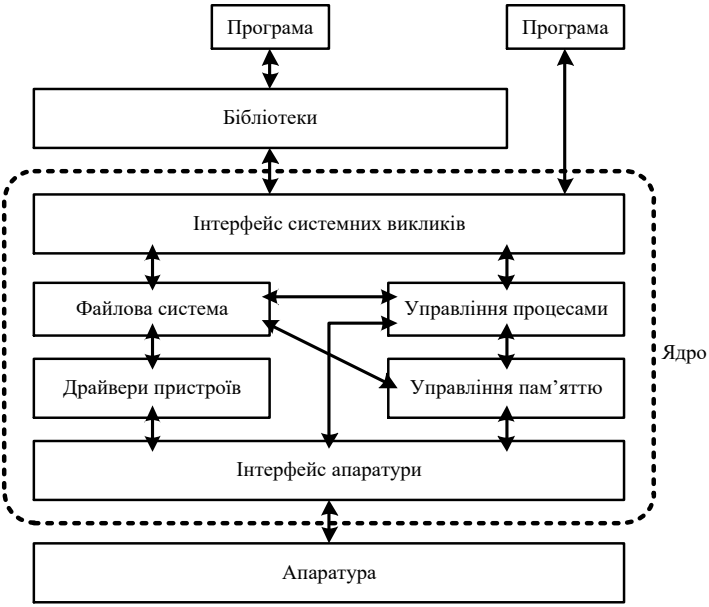


Рисунок 1.7 – Структура ОС UNIX

Усе, що знаходиться нижче рівня інтерфейсу системних викликів та вище фізичного обладнання, є ядром. Ядро забезпечує файлову систему, планування центрального процесора, керування пам'яттю та інші функції ОС через системні виклики. Кількість рівнів невелика, кожен рівень – монолітний, між рівнями немає інкапсуляції: функції та послуги, що

надаються на різних рівнях, доступні всій системі. По суті ядро ОС UNIX – набір процедур, які можуть викликати будь-які інші процедури. В ядрі реалізовано величезну кількість функціональності і всі вони об'єднані в єдиному адресному просторі.

ОС Linux [7, 17, 25, 39] побудована на базі UNIX і має аналогічну модульну структуру, як наведено на рис. 1.8.

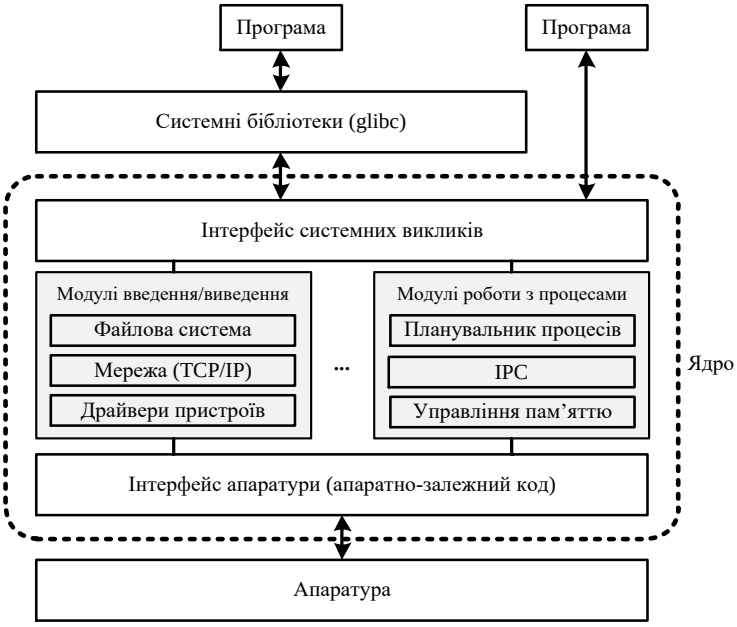


Рисунок 1.8 – Структура ОС Linux

ОС Linux складається з трьох основних компонентів:

♦ ядро – складається з різних модулів і безпосередньо взаємодіє з основним обладнанням. Ядро забезпечує необхідну абстракцію, щоб приховати деталі апаратного забезпечення низького рівня для системних або прикладних програм;

♦ системні бібліотеки – спеціальні функції або програми, за допомогою яких прикладні програми або системні утиліти отримують доступ до

функцій ядра. Ці бібліотеки реалізують більшість функцій ОС і не потребують прав доступу до коду модуля ядра;

♦ системні утиліти – відповідають за виконання спеціалізованих завдань індивідуального рівня.

Програми зазвичай використовують стандартну бібліотеку C *glibc* (GNU C Library) під час обміну даними з інтерфейсом системних викликів. Архітектура ядра ОС Linux, як було вказано, дотримується модульного підходу. Кожен модуль у ядрі – це частина коду, написана мовою С. Ядро Linux є монолітним, оскільки воно повністю працює в режимі ядра в єдиному адресному просторі, але за рахунок модульної конструкції є можливість його змінювати безпосередньо під час роботи системи.

Solaris – ОС на базі UNIX, розроблена компанією Sun Microsystems [26], після придбання компанією Oracle, відома як Oracle Solaris. Структура ядра – монолітна з модулями, що динамічно завантажуються. Загальна схема ОС Solaris наведена на рис. 1.9.

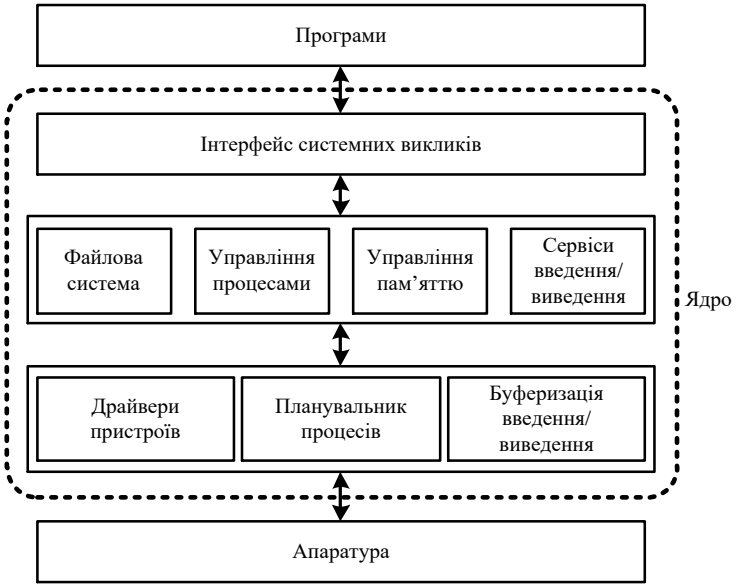


Рисунок 1.9 – Структура ОС Solaris

Однією з важливих рис ОС Solaris є її здатність забезпечувати високу надійність і безпеку. Solaris має потужні механізми безпеки, які дозволяють користувачам забезпечувати високий рівень безпеки своїх систем. Крім того, Solaris також є масштабованою системою. Вона підтримує багатопроцесорні системи та кластери, що дозволяє використовувати її для обробки великих обсягів даних та виконання складних завдань.

Незважаючи на уявну простоту монолітних ядер, їх важко реалізувати та розширити. Однак монолітні ядра мають явну перевагу в продуктивності: в інтерфейсі системних викликів дуже мало накладних витрат, а зв'язок між компонентами всередині ядра швидкий. Таким чином, незважаючи на недоліки монолітних ядер, їх швидкість і ефективність пояснюють те, чому ця архітектура (або її частки) використовується і в сучасних версіях ОС UNIX, Linux і частково Windows (Windows має іншу архітектуру).

Головні висновки:

♦ переваги монолітного ядра:

- простота – монолітні ядра простіше проєктувати та реалізувати порівняно з мікроядрами;

- продуктивність – оскільки всі компоненти є частиною одного цілого, зв'язок між ними швидкий і ефективний, що забезпечує кращу продуктивність;

♦ недоліки монолітного ядра:

- масштабованість – із зростанням розміру ОС розмір монолітного ядра також зростає, що ускладнює його підтримку та налагодження;

- стабільність – помилка в будь-якій частині монолітного ядра потенційно може привести до збою всієї системи, зробивши її менш стабільною;

- безпека – оскільки всі компоненти є частиною одного цілого, порушення безпеки в одній частині системи може потенційно поставити під загрозу всю систему.

1.5.2. ОС з багаторівневою архітектурою

Оскільки ОС ставали більшими та складнішими, монолітні архітектури ставали громіздкими. Багаторівневий підхід до ОС намагається вирішити цю

проблему шляхом групування компонентів, які виконують подібні функції, у рівні.

ОС з багаторівневою архітектурою (англ. Layered Operating Systems) організовані у вигляді ієрархії рівнів. Взаємодія між рівнями виконується за допомогою міжрівневих інтерфейсів. Робота компонентів одного рівня базується на роботі компонентів нижчих рівнів. На основі функцій низьких рівнів будуються більш складні функції високих рівнів. На рис. 1.10 наведена базова структура ОС з багаторівневою архітектурою.

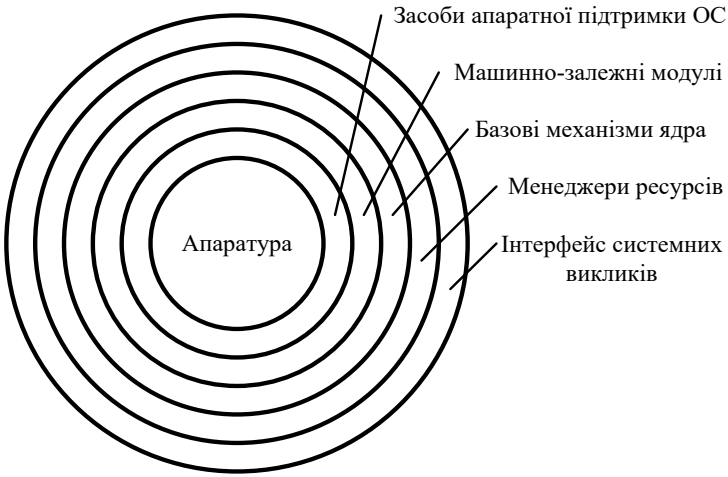


Рисунок 1.10 – Структура ОС з багаторівневою архітектурою

До складу ОС можуть входити такі рівні:

♦ засоби апаратної підтримки ОС: засоби підтримки привілейованого режиму, система переривань, засоби перемикання контекстів процесів, засоби захисту областей пам’яті тощо;

♦ машинно-залежні компоненти ОС: програмні модулі, які враховують особливості апаратної платформи комп’ютера. Цей рівень повністю приховує деталі управління апаратурою. Приклад такого модуля – шар HAL (англ. Hardware Abstraction Level) в ОС Windows на базі ядра NT;

♦ базові механізми ядра: програмне перемикання контекстів процесів, диспетчеризація переривань, переміщення сторінок з пам'яті на диск та навпаки тощо;

♦ менеджери ресурсів: модулі управління основними ресурсами обчислювальної системи – менеджери процесів, введення-виведення, файлової системи, віртуальної пам'яті;

♦ інтерфейс системних викликів, який утворює інтерфейс прикладного програмування API. Цей рівень самий верхній в ядрі ОС і безпосередньо взаємодіє з ПЗ.

Основні переваги багаторівневого підходу – це простота побудови та налагодження. Розробники ОС мають більше свободи в зміні внутрішніх частин системи. Рівні обираються так, що кожен з них використовує функції і послуги тільки нижчих рівнів. Такий підхід спрощує налагодження та систему контролю: перший рівень можна налагоджувати без відношення до іншої частини системи, оскільки, за визначенням, він використовує тільки апаратні ресурси для реалізації своїх функцій. Після того як налагоджено перший рівень, можна налагоджувати другий і т.д. Якщо під час налагодження певного рівня виявлено помилку, вона має бути саме на цьому рівні, оскільки рівні під ним уже налагоджені. Таким чином, спрощується розробка та впровадження системи.

Кожен рівень базується тільки на тих функціях, які реалізовані на нижчих рівнях, при цьому внутрішня реалізація цих функцій для нього не важлива. Таким чином, кожен рівень приховує певні структури даних, особливості виконання операцій та устаткування для інших рівнів. Ця особливість дозволяє легко модифікувати ОС: достатньо замінити один рівень, але залишити міжрівневі інтерфейси з нижчим та вищим рівнем.

Проблемою багаторівневої архітектури є множинність і розмитість інтерфейсів між рівнями, оскільки складно виконати однозначне угруповання функцій за рівнями, необхідне ретельне планування. Наприклад, драйвер пристрою для резервного сховища (дисковий простір, який використовується у віртуальній пам'яті) має бути на нижчому рівні, ніж підпрограми керування пам'яттю, оскільки керування пам'яттю вимагає можливості використовувати резервне сховище. При цьому ж драйвер

резервного сховища зазвичай знаходиться над підсистемою управління процесором, тому що драйверу може знадобитися очікування введення-виведення, а процесор можна перепланувати протягом цього часу. Однак підсистема управління процесором в якісь моменти може мати більше інформації про всі активні процеси, ніж може вмістити пам’ять – цю інформацію доведеться переміщати в резервне сховище, тобто, підпрограма драйвера резервного зберігання повинна бути нижче підсистеми управління процесором.

Основним недоліком багаторівневих ОС є те, що вони, як правило, менш ефективні, ніж інші види ОС. Наприклад, коли користувач запускає програму введення-виведення, він виконує системний виклик, який потрапляє на рівень управління введенням-виведенням та пам’яттю, який, у свою чергу, викликає рівень планування процесора, з якого вже передається управління до апаратури (рис. 1.11).

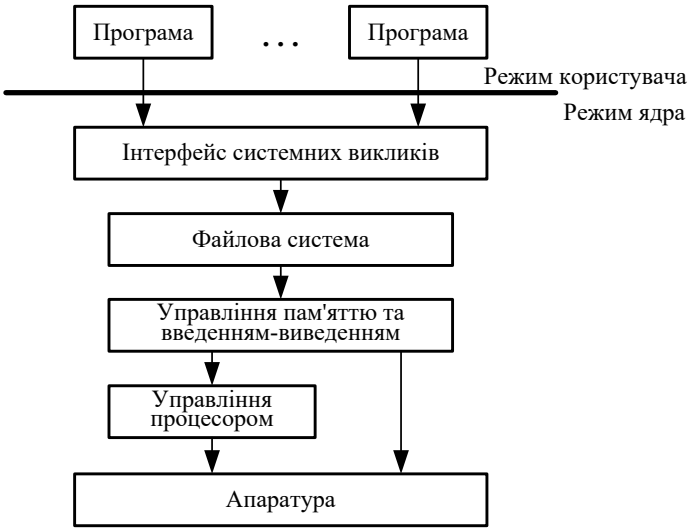


Рисунок 1.11 – Обробка запиту на введення-виведення

На кожному рівні параметри виклику можуть бути змінені, дані перетворені тощо. Кожен рівень додає накладні витрати до системного

виклику і отримання результату виконання займає більше часу, ніж при обробці аналогічного виклику в ОС з монолітною структурою.

Багаторівнева модель надає дві основні переваги. По-перше, це захист від збоїв системи. Помилки у вищих рівнях (з меншим доступом) зазвичай можна відновити. Це пов'язано з тим, що лише самі нижні рівні мають прямий доступ до пам'яті та центрального процесора, тому, якщо процес, що виконується на верхньому рівні, аварійно завершує роботу, його можна перезапустити без втрати даних або спричинення помилки в ОС. По-друге, забезпечується підвищена безпека. Щоб виконати інструкції, які потребують більшого ніж є доступу до ресурсів, процес має запитати дозвіл в ОС, яка вирішує, задовольнити запит чи відхилити його. Це допомагає захистити систему від небажаної чи зловмисної поведінки процесів. Ще одна з важливих переваг багаторівневої архітектури – абстрагування, що забезпечує відсутність у процесів прямого доступу до апаратного забезпечення, минаючі проміжні рівні.

Вперше багаторівневий підхід був використаний при розробці системи THE (нід. Technische Hogeschool Eindhoven) multiprogramming system (THE OS) Дейкстрою в 1965-68 рр. Іншим прикладом багаторівневої ОС є Multics.

Система THE OS [14] – пакетна система для комп'ютера Electrologica X8 (мав обсяг лише 32 тисячі 27-бітних слів). Система мала шість рівнів:

♦ рівень 0, який виконував управління процесором, перемикання між процесами (при перериваннях або таймеру) – базове мультипрограмування центрального процесора;

♦ рівень 1, який керував пам'яттю. Він ділив простір для процесів в основній пам'яті та на барабані слів (зовнішня пам'ять) обсягом 512 КБ, який використовувався для зберігання частин процесів (сторінок), для яких не було місця в основній пам'яті;

♦ рівень 2, який обробляв зв'язки між процесами і консоллю оператора (користувача). Кожен процес фактично мав власну консоль оператора;

♦ рівень 3, який керував пристроями введення/виведення та виконував буферизацію даних для обміну з ними;

♦ на рівні 4 працювали програми користувача;

♦ на рівні 5 було розташовано процес системного оператора.

Кожен вищий рівень працював з попереднім та всі дії на попередньому рівні для нього вважалися абстракцією: наприклад, програмам користувача (рівень 4) не потрібно було турбуватися про процеси, пам'ять, консоль або керування введенням/виведенням.

Багаторівнева архітектура THE OS була лише допоміжним механізмом в ОС, оскільки всі частини системи були пов'язані в єдину виконувану програму. Однак, кожен рівень міг бути наділений привілеями і виконувати системні виклики з контролем параметрів для звернення до інших рівнів.

Подальший розвиток концепція рівнів отримала в системі Multics. Замість рівнів у Multics використовується термін «кільце». Система має серію концентричних кілець, причому внутрішні є більш привілейованими, ніж зовнішні. ОС Multics складається з восьми захисних кілець[53], що підтримуються апаратним забезпеченням, межі яких можна перетинати лише за допомогою спеціальних інструкцій. Коли процедура в зовнішньому кільці викликає процедуру у внутрішньому кільці, вона робить еквівалент системного виклику, тобто інструкцію TRAP, параметри якої ретельно перевіряються, перш ніж виклик буде дозволено продовжити. Хоча вся ОС Multics є частиною адресного простору кожного процеса користувача, апаратне забезпечення дозволяло призначати окремі процедури (окремі сегменти пам'яті) як захищені від читання, запису або виконання.

---

[53] Механізм захисних кілець також використовується в сучасних ОС Microsoft Windows, UNIX та ін.:

♦ кільце 0 (найбільш привілейоване) доступне для ядра ОС і має повний доступ (код працює в режимі ядра) до всіх ресурсів. Процеси, запущені в режимі ядра, можуть впливати на всю систему. Це кільце має прямий доступ до центрального процесора та системної пам'яті, тому будь-які інструкції, які вимагають їх використання, виконуються на цьому рівні;

♦ кільце 3, найменш привілейоване кільце, доступне для процесів користувача (код виконується в режимі користувача). Кільце не має прямого доступу до центрального процесора чи пам'яті, і тому має передавати будь-які інструкції, що їх стосуються, до кільця 0;

♦ кільця 1 і 2 мають особливі привілеї, яких не має кільце 3. Кільце 1 використовується для керування обладнанням, підключеним до комп'ютера, та взаємодії з ним. Кільце 2 використовується для інструкцій, які виконують введення/виведення – вони передбачають переміщення даних у робочу пам'ять або з неї. Наприклад, завантаження документа Microsoft Office в ОС Windows зі сховища відбувається в кільці 2, а перегляд і його редагування – у кільці 3.

Головні висновки:

♦ переваги багаторівневої ОС:

- абстракція – рівні не пов'язані з функціонуванням інших рівнів у структурі, що робить їх придатними для налагодження окремо;

- модульність – ОС розділена на кілька блоків, і кожен блок окремо ефективно виконує свої функції;

- краще технічне обслуговування – будь-які внесені оновлення чи зміни обмежуються поточним рівнем і жодним чином не впливають на інші рівні;

- налагодження – його можна виконати добре, якщо рівень, який налагоджується, буде виправлено, за умови, що попередні рівні функціонують належним чином;

♦ недоліки багаторівневої ОС:

- відсутність прямого зв'язку – зв'язок між несусідніми рівнями ОС завжди відсутній, що ускладнює передачу даних;

- функціональність – коли функціональні можливості взаємопов'язані, їх неможливо відокремити одна від одної, і в цьому випадку складно виконати поділ на рівні;

- повільна обробка – рівні відповідають за надходження будь-якого запиту, а перехід до іншого рівня є трудомістким процесом, на відміну від монолітних ядер. Чим більше рівнів, тим менше ефективність;

- складність проєктування – структура ОС має складну конструкцію, яка має бути ретельно реалізована, оскільки служби, які використовуються поточним рівнем, мають розташовуватися нижче цього рівня.

1.5.3. ОС з мікроядерною архітектурою

Поява нової концепції в області ОС – мікроядра (англ. Microkernel) спрямована на міграцію традиційних сервісів ОС з монолітного ядра на рівень процесів користувача. Ідея полягає в розділенні ОС на кілька процесів, кожен з яких реалізує окремий набір сервісів, наприклад, сервер введення-виведення, сервер управління пам'яттю, сервер управління процесами тощо. Кожен сервер виконується в режимі користувача та надає сервіси на запити клієнта. Клієнт, який може бути іншим компонентом ОС

(мікроядром або іншим сервером) або програмою, виконує запит на обслуговування, відправивши повідомлення на сервер. Ядро ОС, яке працює в режимі ядра, доставляє повідомлення до відповідного сервера, той виконує необхідну операцію, і мікроядро повертає результати клієнту в іншому повідомленні. Принцип роботи мікроядерної ОС наведений на рис. 1.12.

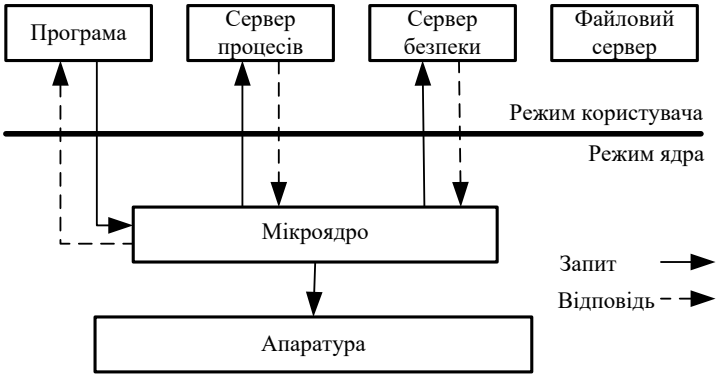


Рисунок 1.12 – Мікроядерна структура ОС

Загалом, сервіси традиційно є невід'ємною частиною файлових систем, служб безпеки тощо, тобто всіх допоміжних модулів, які взаємодіють з ядром та одне з одним і раніше вказані як підсистеми ОС. Сервери могли би спілкуватися безпосередньо між собою, але спілкуються через повідомлення, які передаються через ядро: воно перевіряє повідомлення, передає їх між компонентами, і надає доступ до обладнання. Коли мікроядро отримує повідомлення від процеса, воно може обробити його безпосередньо або надіслати в інший процес.

Весь код, залежний від процесора, ізольований у мікроядрі. Для запуску ОС на новому процесорі необхідно виконати набагато менше змін у мікроядрі ніж в ядрі з монолітною або багаторівневою архітектурою. Це робить можливим запускати ОС більш ніж на одній апаратній платформі.

Підсистеми, що працюють вище мікроядра, можуть бути налаштовані для задоволення потреб нової платформи, що підвищує мобільність системи.

Суперечливим питанням у мікроядерних ОС є розподіл функцій між мікроядром і допоміжними модулями (підсистемами, серверами).

З точки зору складності коду було б невірно вважати мікроядро нескладним та невеликим. Мікроядерні ОС, які знаходяться на комерційному ринку, по складності перевищують монолітні системи.

Мікроядро є мінімальною реалізацією функцій ядра ОС. Класичні мікроядра надають лише невеликий набір низькорівневих системних функцій для реалізації базових сервісів ОС:

- ♦ управління адресним простором оперативної пам'яті;
- ♦ управління адресним простором віртуальної пам'яті;
- ♦ управління процесами і нитками;
- ♦ міжпроцесні комунікації.

Набір функцій мікроядра зазвичай відповідає функціям рівня базових механізмів ядра багаторівневих систем.

У мікроядерній ОС можна без переривання її роботи завантажувати і вивантажувати нові модулі: драйвери, файлові системи тощо. Компоненти ОС небагато відрізняються від програм користувачів, тому для їх налагодження можна застосовувати звичайні засоби.

Перевагами мікроядерної архітектури ОС є:

- ♦ розширюваність за рахунок можливості додавати нові сервіси, або модифікувати та видаляти старі;
- ♦ переносимість за рахунок необхідності зміни тільки коду мікроядра;
- ♦ надійність за рахунок відокремлення серверів один від одного: у разі краху вони можуть бути перезапущені без впливу на інші сервери;
- ♦ підтримка розподілених систем за рахунок єдиного інтерфейсу запитів і підтримки механізму обміну повідомленнями.

Недоліком ОС з мікроядерною архітектурою є низька продуктивність: створення, передача повідомлення та отримання відповіді займає більше часу порівняно з безпосереднім викликом сервісу, додатково потребується при виконанні дій по обробці запитів численні перемикання в режим ядра та назад. Для того щоб підвищити швидкість роботи системи, необхідно

досконало планувати розподіл ОС на компоненти з урахуванням кількості взаємодій між ними.

Найвідомішою ілюстрацією ОС, в якій застосовано мікроядерні технології, є Darwin – компонент ядра ОС macOS та iOS. Фактично, Darwin є гібридним ядром, та складається з двох ядер [23], одним з яких у свою чергу є мікроядро Mach[54]. За винятком macOS звичайні настільні ОС не використовують мікроядра. Однак, вони домінують у системах реального часу, які є критично важливими та мають дуже високі вимоги до надійності. Кілька найбільш відомих мікроядер (крім вказаного мікроядра Mach): Integrity[55] для однопроцесорних вбудованих систем (модифікація Integrity-178B використовується у військових та цивільних літаках), K42, L4[56] (розмір коду ядра L4 становить лише 14 кілобайт і містить лише 7 системних викликів), PikeOS[57] (для пристроїв IoT – Інтернет речей (англ. Internet of Things)), QNX (ядро має розмір 7 кілобайт та виконує такі функції: передача повідомлень між процесами, планування процесів, обробка апаратних переривань та мережевий зв'язок нижнього рівня), Symbian і Minix 3.

Мікроядро Mach [34] – перше в світі мікроядро для ОС, воно розроблено в якості бази, на основі якої можна емулювати UNIX та інші ОС. Ця емуляція забезпечується програмним рівнем, який працює в режимі користувача. Кілька емуляторів можуть працювати одночасно: наприклад, можна виконувати програми 4.3BSD, System V і MS-DOS на одній машині в один і той же час. Ядро Mach забезпечує управління процесами, управління пам'яттю, комунікаціями та функціями введення-виведення. Функції управління файлами, каталогами та інші традиційні для ОС функції

[54] Перший випуск у 1985 р., остання стабільна версія – 3.0 за 1994 р., ресурс у мережі Internet – https://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html

[55] Остання стабільна версія – 29.09.2009 р., ресурс у мережі Internet – https://www.ghs.com/

[56] Перший випуск в 1993 р., остання версія L4/Fiasco – червень 2010 р., ресурс у мережі Internet – https://os.inf.tu-dresden.de/L4/, у листопаді 2023 р. анонсовано ОС SkyOS для електромобілів

[57] Остання версія – 5.1 у січні 2021 р., ресурс у мережі Internet – https://www.sysgo.com/pikeos

виконуються в режимі користувача. Ядро управляє п'ятьма основними абстракціями (але є і інші абстракції – менш важливі):

♦ задачі (англ. Tasks) – відіграють роль середовищ, в яких відбувається виконання ниток. Вони мають адресний простір, що містить текст програми та дані, і зазвичай один або більше стеків. Задача – базова одиниця для розподілу ресурсів. Наприклад, комунікаційний канал завжди належить всій задачі;

♦ нитки в Mach є одиницями виконання. Кожна з них має лічильник команд і набір пов'язаних з нею регістрів. Кожна нитка є частиною задачі;

♦ об'єкти пам'яті представляють собою структури даних, які можуть бути відображені в адресному просторі задачі. Об'єкти пам'яті займають одну або кілька сторінок і утворюють основу для системи управління віртуальною пам'яттю Mach. Коли задача посилається на об'єкт пам'яті, якого немає у фізичній пам'яті, це викликає сторінкове переривання. Як і в інших ОС, ядро перехоплює сторінкове переривання. Однак, на відміну від інших систем, ядро Mach для завантаження відсутньої сторінки посилає повідомлення відповідному серверу, який працює в режимі користувача, а не виконує цю операцію самостійно;

♦ порти – захищені конвеєри (поштові скриньки) для взаємодії між задачами;

♦ повідомлення. Взаємодія між задачами в Mach основана на передачі повідомлень. Для того, щоб отримати повідомлення, нитка просить ядро створити захищену поштову скриньку, яка називається портом. Порт зберігається всередині ядра і здатний підтримувати впорядковану чергу повідомлень. Нитка може надати іншій задачі можливість надсилати (або отримувати) повідомлення в одному з портів, що їй належать.

ОС Minix 3 [38] – це POSIX-сумісна система з відкритим кодом, яка доступна у вільному доступі. Структура ОС Minix 3 наведена на рис. 1.13.

Мікроядро Minix 3 містить близько 15 тис. рядків коду мовою С і 1400 рядків коду мовою асемблера для функцій низького рівня, таких як перехоплення переривань і перемикання процесів. Код мовою С керує та планує процесами, обробляє міжпроцесний зв'язок (використовується механізм передачі повідомлень) і реалізує набір із приблизно 40 викликів

ядра (Sys Task), необхідних для роботи решти ОС. Драйвер пристрою для годинника (Clock Task) також знаходиться в ядрі, оскільки планувальник тісно взаємодіє з ним. Інші драйвери пристроїв (Audio Driver, Disk Driver, Printer Driver та ін.) працюють як окремі процеси в режимі користувача.

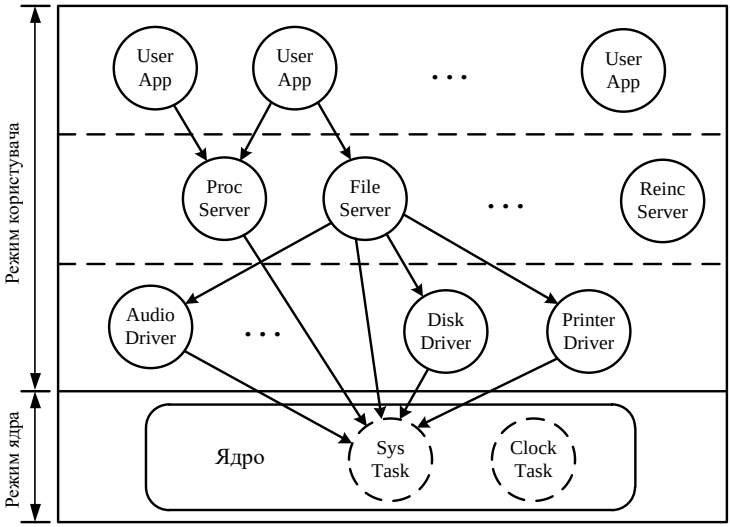


Рисунок 1.13 – Спрощена структура ОС MINIX 3

Поза ядром система структурована як три рівні процесів, усі вони виконуються в режимі користувача. Найнижчий рівень містить драйвери пристроїв. Оскільки вони працюють у режимі користувача, вони не мають фізичного доступу до портів введення/виведення та не можуть безпосередньо видавати команди введення-виведення. Замість того, щоб запрограмувати пристрій введення/виведення, драйвер створює структуру, яка вказує, які значення записувати до портів введення/виведення, і здійснює виклик ядра, вказуючи ядру виконати запис. Цей підхід дозволяє ядру контролювати виконувані операції. Наприклад, аудіодрайвер з помилками не може випадково записати дані на диск.

Над драйверами знаходиться наступний рівень ОС у режимі користувача, який містить сервери, що виконують більшу частину роботи ОС. Один або кілька файлових серверів (File Server) керують файловими системами, менеджер процесів (Proc Server) керує процесами (створення, знищення процесів тощо). Програми користувача (User App) отримують послуги ОС, надсилаючи короткі повідомлення до серверів із запитами на виконання системних викликів POSIX. Одним із серверів є сервер реінкарнації (Reinc Server), завдання якого полягає в перевірці правильності роботи інших серверів і драйверів. У разі виявлення несправного модуля, він автоматично замінюється без втручання користувача, що дає змогу системі самовідновлюватися під час роботи і, відповідно, значно підвищити її надійність.

Головні висновки:

♦ переваги мікроядра:

- масштабованість – оскільки мікроядро менше і включає лише основні компоненти, його легше підтримувати та розширювати;

- стабільність – якщо компонент виходить з ладу, це впливає лише на цей компонент, а решта системи продовжує працювати;

- безпека – оскільки компоненти працюють в окремих процесах режиму користувача, порушення безпеки в одному компоненті не може вплинути на всю систему;

♦ недоліки мікроядра:

- продуктивність – зв’язок між компонентами відбувається повільніше, оскільки вони розділені ядром і мають проходити через системні виклики;

- складність – мікроядра є складнішими для розробки та реалізації порівняно з монолітними ядрами;

- інтеграція – поєднання компонентів між собою може бути складнішим, оскільки вони відокремлені від ядра і потребують більш ретельного налагодження.

1.5.4. Гібридні ядра

На практиці дуже небагато ОС мають одну, строго визначену структуру. Натомість вони поєднують різні структури, така реалізація має назву – гібридні системи, вона вирішує проблеми продуктивності, безпеки та зручності використання. Найбільш поширене гібридне ядро (англ. Hybrid Kernel), яке є комбінацією монолітної та мікроядерної архітектури. Воно поєднує в собі переваги обох і намагається подолати недоліки кожного. Цей архітектурний підхід полягає в розміщенні апаратно залежних сервісів у режимі ядра (управління процесами, пам'яттю, введенням-виведенням; безпека). Розмір гібридного ядра більше ніж мікроядра, але менше, ніж монолітного ядра, при цьому зберігається орієнтований на повідомлення механізм взаємодії клієнт-серверної архітектури.

Основною перевагою підходу є значне підвищення швидкодії системи. Це відбувається тому, що не потрібно часто переключатися з режиму ядра в режим користувача при передачі повідомлень між серверами, що реалізують складний системний виклик.

Недоліком є деяка втрата гнучкості та надійності. Як приклад можна розглянути ОС NT версій 3.x, драйвер графічної апаратури в яких був оформлений як окремий сервер, що працює в режимі користувача. Це викликало низьку швидкість при роботі з графічними пристроями і повільний графічний інтерфейс користувача. У версії Windows NT 4 графічну підсистему разом з драйвером апаратури (USER – менеджер інтерфейсу користувача, GDI – менеджер графіки та драйвер графічної плати) перенесли в ядро ОС для підвищення швидкості роботи системи. Перенесення драйверів графічних плат у режим ядра призвело до значного зменшення надійності ОС, але, починаючи з Windows XP Professional, Microsoft забезпечила кілька програмних інструментів для розробки драйверів, і компоненти режиму ядра стали стабільнішими і скоротилася кількість причин для появи «синього екрану смерті» (англ. Blue Screen of

Death) в термінах ОС Windows (в термінах ОС UNIX/Linux подібна ситуація має назву – «паніка ядра» (англ. Kernel Panics))[58].

У деяких джерелах гібридні ядра відносять до монолітної модульної побудови ОС. Ця неправильна класифікація пов'язана з тим, що гібридні ядра підтримують архітектуру завантажувальних модулів ядра, властиву модульним ядрам, але є важлива відмінність – модулі ядра, які завантажуються, і інші компоненти ядер розташовуються в пам'яті, яку можна витіснити, і взаємодіють один з одним шляхом передачі повідомлень, як це робиться в мікроядерних ОС (в модульній побудові всі модулі знаходяться в адресному просторі ядра і викликаються як звичайні процедури).

Прикладами ОС із гібридними ядрами є Microsoft Windows NT (ядро NT kernel) і macOS (ядро Darwin XNU) та iOS від Apple, Android від Google.

Mac OS X від Apple – це комерційно доступна ОС, що забезпечує графічний інтерфейс користувача на основі ядра UNIX. Darwin, ядро Mac OS X з відкритим вихідним кодом, складається з різноманітних технологій, включаючи засоби мікроядра Mach 3, служби ОС на основі FreeBSD 5, розширення ядра KEXT (Kernel Extensions) (високопродуктивні мережеві засоби NKE (Network Kernel Extensions) та ін.), фреймворк для написання драйверів I/O Kit та чисельні інтегровані файлові системи. Структура ОС Mac OS X наведена на рис. 1.14.

Основний вміст рівнів ОС:

[58] «Паніка ядра» – це системна помилка, яка виникає, коли ядро ОС не може впоратися з серйозною проблемою. Це призводить до припинення роботи всіх системних функцій і може призвести до збою системи. На паніку часто вказує повідомлення, що відображається на екрані з написом «Kernel Panics». Причиною «паніки ядра» можуть бути різні фактори, включаючи помилки ПЗ, апаратні збої та проблеми з конфігурацією системи. Точне повідомлення та інформація, які відображаються під час «паніки ядра», можуть відрізнятися залежно від ОС, але кінцевий результат зазвичай однаковий: система більше не стабільна і її потрібно перезапустити. У більшості випадків «паніки ядра» трапляються нечасто, і їх можна вирішити шляхом оновлення ОС, драйверів або апаратних компонентів. Однак у деяких випадках вони можуть вказувати на більш серйозну проблему.

♦ Рівень взаємодії з користувачем. Цей рівень визначає програмний інтерфейс, який дозволяє користувачам взаємодіяти з апаратурою. Mac OS використовує інтерфейс користувача Aqua, розроблений для миші чи трекпада, тоді як iOS використовує інтерфейс користувача Springboard, який призначений для сенсорних пристроїв;

♦ Рівень програмних інтерфейсів застосунків (API). Цей рівень включає фреймворки Classic, Carbon, Cocoa та Cocoa Touch, які надають API для мов програмування Objective-C і Swift. Підсистема Classic забезпечує виконання традиційних застосунків. Carbon є «засобом перехідного періоду» і дозволяє створювати програми, що однаково добре працюють як на «класичній» Mac OS, так і на Mac OS X. Підсистема Cocoa – повністю об'єктно-орієнтований набір сервісів та API. Основна відмінність між Cocoa і Cocoa Touch полягає в тому, що перший використовується для розробки застосунків Mac OS, а другий – iOS для забезпечення підтримки апаратних функцій, унікальних для мобільних пристроїв, таких як сенсорні екрани. Carbon і Cocoa можуть використовувати всі переваги Mac OS X, проте з точки зору створення нових програм набір інтерфейсів Cocoa є кращим;

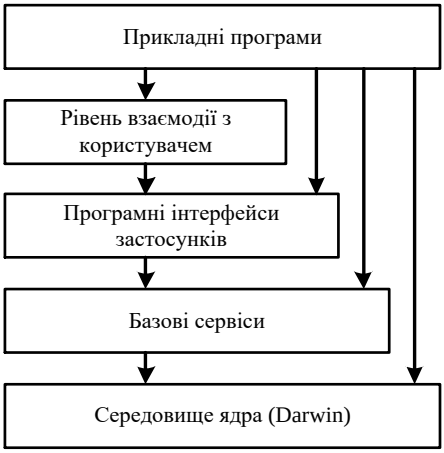


Рисунок 1.14 – Загальна структура ОС Mac OS X та iOS

♦ Базові сервіси. Цей рівень визначає інфраструктури, які підтримують графіку та медіа: підсистеми Quartz, OpenGL та QuickTime. Quartz – засіб візуалізації двовимірних даних, OpenGL забезпечує роботу з тривимірними зображеннями, а QuickTime – з мультимедійними даними;

♦ Середовище ядра Darwin, яке складається з мікроядра Mach і ядра BSD UNIX.

Як наведено на рис. 1.14, застосунки можуть бути розроблені так, щоб використовувати переваги функцій взаємодії з користувачем або обходити їх і взаємодіяти безпосередньо з фреймворком застосунків (рівень програмних інтерфейсів застосунків) або основним фреймворком (базові сервіси). Крім того, програма може повністю відмовитися від фреймворків і спілкуватися безпосередньо з середовищем ядра (наприклад, програма написана мовою С без інтерфейсу користувача, яка здійснює системні виклики POSIX).

Структура ядра Darwin наведена на рис. 1.15.

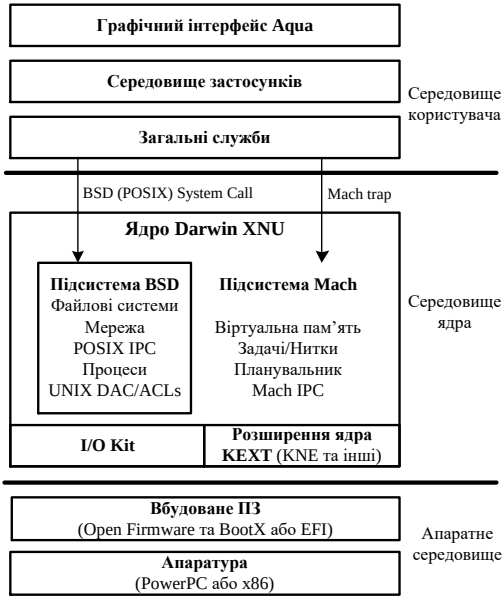


Рисунок 1.15 – Спрощена структура Darwin

Darwin [23] по суті є самостійною ОС. Сюди не входять графічне середовище та бібліотеки, потрібні для роботи віконних застосунків (наприклад, такі як Quartz, QuickTime, Cocoa, Carbon або OpenGL), але входять ядро, драйвери, мережевий стек, набір системних утиліт і утиліт користувача командного рядка, система запуску служб і застосунків. За бажанням Darwin можна встановити як самостійну мінімалістичну ОС з текстовим інтерпретатором команд.

У той час як більшість ОС забезпечують єдиний інтерфейс системних викликів для ядра, наприклад через стандартну бібліотеку C в ОС UNIX і Linux, Darwin надає два інтерфейси системних викликів: системні виклики Mach, відомі як пастки (англ. Mach traps), і системні виклики BSD (POSIX System Calls). Інтерфейс цих системних викликів – набір бібліотек, який включає не лише стандартну бібліотеку C, але й бібліотеки, які забезпечують роботу в мережі, безпеку та підтримку мов програмування.

Значна частина функціональних можливостей Mach доступна через абстракції ядра, які включають задачі (процеси Mach), нитки, об'єкти пам'яті та порти (використовуються для IPC). Як приклад, програма може створити новий процес за допомогою системного виклику BSD POSIX, а Mach, у свою чергу, використовуватиме абстракцію ядра – «задача» для представлення процеса в ядрі.

Основні функції підсистеми Mach:

- нетипізована міжпроцесна взаємодія (IPC);
- виклики віддалених процедур (RPC);
- підтримка планувальника для симетричної багатопроцесорної обробки (SMP);
- підтримка сервісів у реальному часі;
- підтримка віртуальної пам'яті;
- підтримка модульної архітектури.

Підсистема BSD – ретельно спроєктована ОС з багатьма можливостями. Фактично більшість сучасних комерційних UNIX і UNIX-подібних ОС містять велику кількість коду BSD. BSD також надає набір стандартних API. Основні функції підсистеми BSD:

- підтримка файлових систем;

♦ підтримка мережевих протоколів (крім рівня апаратного пристрою);

♦ реалізація моделі безпеки UNIX;

♦ підтримка моделі процеса BSD, включаючи ідентифікатори процесів та сигнали API ядра FreeBSD;

♦ підтримка ядра для моделі ниток POSIX (*pthreads*);

♦ підтримка різних API POSIX.

Технології I/O Kit та розширення ядра KEXT були спроєктовані Apple, щоб скористатися перевагами розширених можливостей, що надаються моделлю об'єктно-орієнтованого програмування.

I/O Kit забезпечує основу для спрощеної розробки драйверів, що підтримує багато категорій пристроїв. I/O Kit є об'єктно-орієнтованою архітектурою введення-виведення.

Компонент I/O Kit забезпечує:

♦ справжній Plug & Play;

♦ динамічне керування пристроєм;

♦ динамічне завантаження драйверів;

♦ керування живленням для настільних систем та портативних пристроїв;

♦ можливості багатопроцесорної роботи.

Розширення ядра KEXT – це динамічно завантажуваний пакет виконуваного коду, який виконується в просторі ядра. Можна створити KEXT для виконання задач низького рівня, які неможливо виконати в просторі користувача. Розширення ядра зазвичай належать до однієї з трьох категорій:

♦ низькорівневі драйвери пристроїв;

♦ мережеві фільтри;

♦ файлові системи.

Ядро XNU (X is Not Unix) є сучасним гібридним ядром, що поєднує в собі переваги як монолітних ядер, так і мікроядер, зокрема, можливості передачі повідомлень мікроядер для підвищення модульності системи і захисту пам'яті різних модулів і високу швидкість монолітних ядер у деяких критичних завданнях. У даний час Darwin XNU може працювати на

процесорах з архітектурою ARM, x86, x86-64 (підтримка PowerPC закінчилася починаючи з версії Mac OS X 10.6).

iOS – мобільна ОС, розроблена для смартфонів iPhone і планшетних комп'ютерів iPad. Архітектурно Mac OS та iOS мають багато спільного, деякі істотні відмінності між ними полягають у такому:

♦ iOS розроблена для мобільних пристроїв і, отже, скомпільована для архітектури на основі ARM. Подібним чином ядро iOS було дещо модифіковано для вирішення конкретних функцій і потреб мобільних систем, таких як керування живленням і агресивне керування пам'яттю. Крім того, iOS має більш суворі налаштування безпеки, ніж Mac OS.

♦ iOS, як правило, набагато більше обмежена для розробників, ніж Mac OS, і навіть може бути закритою для розробників. Наприклад, iOS обмежує доступ до API POSIX і BSD, тоді як вони відкрито доступні для розробників на Mac OS.

ОС Android [35, 37] є стеком ПЗ, який розділений на кілька рівнів, як наведено на рис. 1.16.

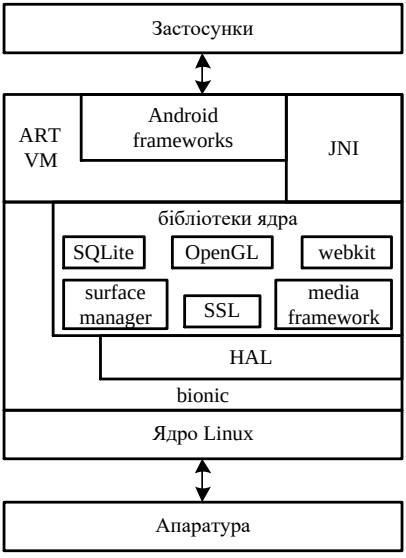


Рисунок 1.16 – Архітектура ОС Android

ОС Android була розроблена альянсом Open Handset Alliance під керівництвом Google для смартфонів і планшетних комп'ютерів. У той час як iOS була розроблена для роботи на мобільних пристроях Apple і має закрите джерело, Android працює на різноманітних мобільних платформах і має відкритий код.

Багатошаровий стек ПЗ забезпечує великий набір фреймворків, що підтримують графічні, аудіо та апаратні функції. Ці функції, у свою чергу, забезпечують платформу для розробки мобільних програм.

У нижній частині стека ПЗ знаходиться ядро Linux (хоча воно було змінено Google). Linux використовується в основному для підтримки процесів, пам'яті та драйверів пристроїв для апаратного забезпечення. Його було також розширено для підтримки особливих потреб мобільних систем, наприклад – керування живленням. Також були внесені зміни в керування та розподіл пам'яті та додано нову форму IPC – Binder.

Для розробки ПЗ для Android використовується мова Java з окремим Android API. Програми Java скомпільовані у форму, яка може виконуватися на Android RunTime ART VM[59] – віртуальній машині, розробленій для Android і оптимізованій для мобільних пристроїв з обмеженою пам'яттю та можливостями процесора. Тоді як багато віртуальних машин Java виконують динамічну компіляцію JIT (англ. Just In Time) для підвищення ефективності програми, ART виконує компіляцію перед виконанням AOT (англ. Ahead Of Time). Компіляція AOT дозволяє ефективніше виконувати програми, а також знижувати енергоспоживання.

Також при розробці програм може використовуватись інтерфейс JNI (Java Native Interface), що дозволяє обходити віртуальну машину та замість цього писати програми на Java, які мають доступ до певних апаратних

[59] У ранніх версіях Android використовувалась Dalvik Virtual Machine (DVM), яка була спроєктована саме для мобільних пристроїв, вона використовувала динамічну компіляцію JIT. Починаючи з версії Android 4.4 (Kitkat), ART VM була представлена як середовище виконання, а в Android 5.0 (Lollipop) ART повністю замінив Dalvik.

функцій. Програми, написані з використанням JNI, зазвичай не можна переносити з одного апаратного пристрою на інший.

Набір власних бібліотек, доступних для програм Android, містить фреймворки для розробки веб-браузерів (webkit), підтримку бази даних (SQLite) і мережеву підтримку, наприклад безпечні сокети (SSL).

Для збільшення кількості апаратних пристроїв, на яких може працювати ОС Android, фізичне обладнання абстраговане через рівень абстракції обладнання HAL. Абстрагуючи все апаратне забезпечення, таке як камера, чіп GPS та інші датчики, HAL надає застосункам стандартизоване представлення незалежно від конкретного обладнання. Це дозволяє розробляти програми, які легко переносяться на різні апаратні платформи.

Стандартною бібліотекою С, яка використовується в системах Linux, є GNU C library (*glibc*). Для Android було розроблено стандартну бібліотеку C Bionic, яка не тільки займає менше пам'яті, ніж *glibc*, але також розроблена для повільніших процесорів, характерних для мобільних пристроїв.

Архітектура ОС Windows зазнала низку змін у процесі еволюції. Перші версії системи мали мікроядерну структуру. У результаті поступового подолання основного недоліку мікроядра (додаткові накладні витрати, пов'язані з передачею повідомлень) ядро ОС Windows стало містити як елементи мікроядерної архітектури так і елементи монолітного ядра.

Архітектура ОС Windows [1, 13, 34, 37, 42] наведена на рис. 1.17, вона складається з кількох функціональних рівнів, кожен із яких користується сервісами нижчого рівня.

У режимі ядра працюють:

♦ рівень абстрагування від обладнання HAL, який приховує особливості апаратури. Це надає можливість потенційного перенесення системи з однієї платформи на іншу. HAL надає вищим рівням апаратні пристрої в абстрактному вигляді, що дозволяє ізолювати ядро, драйвери та виконавчу систему ОС Windows від специфіки обладнання. Обсяг коду на цьому рівні невеликий, міститься у файлі *hal.dll*;

♦ ядро, яке постійно знаходиться в пам'яті і містить низькорівневі функції ОС: диспетчеризація переривань і винятків, планування ниток, а також багатопроцесорна синхронізація. Також ядро надає набір процедур і

базових об'єктів, які решта виконавчої системи використовує для реалізації конструкцій вищого рівня. Код ядра розташовано у файлі *Ntoskrnl.exe* (ядро ОС NT – NT OS kernel), написано мовою С, а частини, що дають найбільше навантаження на процесор – мовою асемблера;

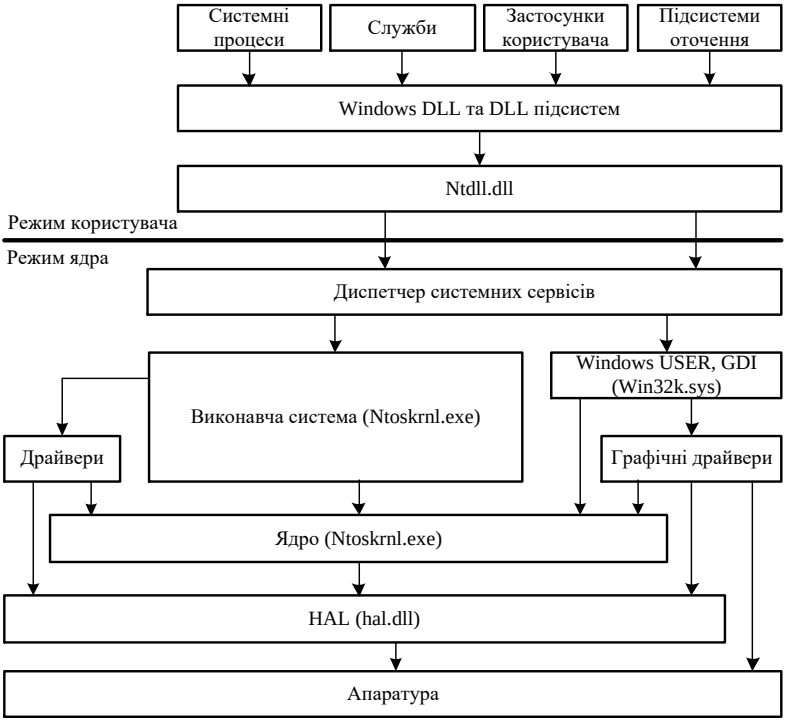


Рисунок 1.17 – Архітектура ОС Windows NT (починаючи з версії ядра 5.1)

♦ драйвери пристроїв, які дозволяють Windows NT взаємодіяти з апаратними компонентами, вони містять апаратно-залежний код і забезпечують трансляцію викликів користувача в запити, специфічні для конкретних пристроїв;

♦ виконавча система (англ. Executive), яка забезпечує управління пам'яттю, процесами та нитками, захист, введення-виведення та взаємодію між процесами. Основні модулі у виконавчій системі:

- диспетчер введення/виведення – керує всіма операціями введення/виведення в ОС; підтримує драйвери файлової системи, апаратних пристроїв і мережеві драйвери. Диспетчер введення/виведення працює в поєднанні з диспетчером кеша;
- диспетчер процесів – керує створенням і видаленням процесів. Працює в поєднанні з монітором контролю безпеки та диспетчером пам'яті для забезпечення захисту;
- диспетчер пам'яті – відображає віртуальні адреси в адресному просторі процеса на фізичні в пам'яті комп'ютера; приховує фізичну організацію пам'яті від процесів;
- монітор контролю безпеки SRM (англ. Security Reference Monitor) – відповідає за виконання політики перевірки доступу та генерації аудиту, визначеної локальною підсистемою безпеки; відповідає за контроль того, які об'єкти мають дозволи на ресурси;
- диспетчер кеша – обробляє кешування для всієї системи введення-виведення; надає служби кешування для всіх файлових систем і мережевих компонентів під керуванням диспетчера введення/виведення;
- диспетчер Plug-and-Play – підтримує виявлення та встановлення пристроїв під час завантаження. Його завдання полягає в тому, щоб зіставити фізичні пристрої з програмним забезпеченням, яке ними керує, і встановити канали зв'язку між кожним фізичним пристроєм і його драйвером;
- диспетчер об'єктів – надає правила для збереження, іменування та безпеки об'єктів – всіх ресурсів ОС;
- диспетчер живлення Power Manager – виконує вимкнення живлення, перехід у режим очікування та сплячий режим. Підтримуються стандарти APM – Advanced Power Management і ACPI – Advanced Configuration and Power Interface, технологія Wake-on-LAN);

- диспетчер конфігурації – дозволяє перевіряти стан апаратних пристроїв і оновлювати драйвери пристроїв для обладнання, встановленого на комп'ютері;
- механізм виклику локальних процедур LPC (англ. Local Procedure Call) – реалізує засіб передачі повідомлень для спрощеного IPC між процесами);

♦ менеджер вікон та графіки (Windows USER, GDI), який реалізує функції графічного інтерфейсу користувача (GUI) такі як робота з вікнами, елементи керування інтерфейсом користувача та графіки. Перенесення графічної підсистеми в ядро не зменшує швидкодію, але не дає змінити графічну оболонку без перезавантаження системи (наприклад, в Linux Ubuntu можна перемикатись між графічними оболонками X11 та Wayland);

♦ диспетчер системних сервісів, що знаходиться у верхній частині виконавчої системи, його функція полягає в наданні інтерфейсу між виконавчою системою та процесами користувача за допомогою Native API (недокументований інтерфейс програмування, призначений для внутрішнього використання). Диспетчер приймає системні виклики і передає управління необхідним частинам виконавчої системи для їх виконання.

У режимі користувача працюють:

♦ системні процеси – спеціальні процеси підтримки системи (WinLogon – процес реєстрації користувача в системі, Smss – менеджер сеансів, Lsass – підсистема локального центру безпеки, Userinit – процес запуску користувацького середовища, Services – диспетчер управління службами та ін.);

♦ служби – процеси, що автоматично запускаються системою при запуску Windows і виконуються у фоновому режимі (Windows Audio, Windows Installer, Print Spooler, Event Logger та ін.). Служби мають спільні риси з процесами «демонами» в UNIX;

♦ застосунки користувача;

♦ підсистеми оточення (англ. Environment Subsystems) – дозволяють Windows використовувати коди різних стандартів. Даний механізм є

набором віртуальних ядер сторонніх ОС і дозволяє швидко адаптувати під Windows будь-який сторонній код. Використовуються дві підсистеми. Підсистема Windows – за допомогою цієї підсистеми виконуються 32-розрядні програми Windows (Win32), а також 16-розрядні програми Windows (Win16), програми MS-DOS і консольні програми. За підсистему Windows відповідає системний процес *Csrss.exe* та драйвер режиму ядра *Win32k.sys*. Підсистема POSIX – для UNIX-застосунків (починаючи з Windows Server 2003 R2 компонент, що реалізує цю підсистему, має назву SUA (Subsystem for UNIX-based Applications));

♦ *ntdll.dll* – спеціальна бібліотека системної підтримки, призначена для використання бібліотек Windows та підсистем. У ній містяться функції-заглушки, що забезпечують переходи від диспетчера системних сервісів до виконавчої системи та допоміжні внутрішні функції, які використовуються підсистемами та DLL-бібліотеками підсистем. Більш детально про *ntdll.dll* описано в розділі 2.4.3.

При роботі в ОС Windows застосунки можуть звертатись через бібліотечні функції мови програмування (Language Runtime Library) до функцій Windows API, або через виклик функції Windows API та далі через документований інтерфейс Windows API до однієї з бібліотек DLL. У свою чергу, ця бібліотека за внутрішнім протоколом (документація для сторонніх розробників не доступна) звертається до *ntdll.dll* і далі за протоколом Native API інформація передається через диспетчер системних сервісів ядру *Ntoskrnl.exe*, як зображено на рис. 1.18.

Так, наприклад, для завершення роботи в застосунку можна скористуватись функцією *exit()* з бібліотеки CRT (C RunTime library), або викликом функції Windows API *ExitProcess()*, або викликом функції Native API *NtTerminateProcess()*. Реалізація функції *exit()* у свою чергу звертається до *ExitProcess()*, а *ExitProcess()* – до *NtTerminateProcess()*. Після обробки в режимі ядра та використання драйверів дані передаються бібліотеці *Hal.dll* для керування обладнанням.

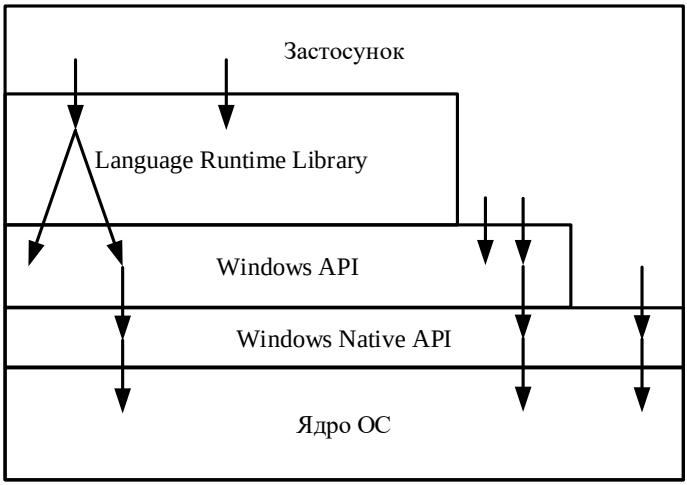


Рисунок 1.18 – Шляхи звернень застосунків до механізмів ОС Windows

Головні висновки:

♦ переваги гібридного ядра:

- продуктивність – гібридне ядро поєднує швидкий зв'язок між компонентами монолітного ядра з перевагами стабільності та безпеки мікроядра;

- масштабованість – менший розмір ядра порівняно з аналогічним монолітним ядром полегшує його підтримку та розширення;

- гнучкість – більше ядро порівняно з мікроядром дозволяє включити та інтегрувати більше компонентів;

♦ недоліки гібридного ядра:

- складність – гібридні ядра є складнішими, ніж монолітні та мікроядра, оскільки вони повинні збалансувати переваги та недоліки обох;

- інтеграція – поєднання компонентів може бути складним, оскільки ними потрібно ретельно керувати та збалансувати їх із розміром і функціональністю ядра;

- компроміс щодо продуктивності – хоча гібридне ядро більш швидке порівняно з мікроядром, але воно повільніше, ніж монолітне ядро.

1.5.5. ОС на базі екзоядра

У традиційних ОС ядро надає не тільки набір сервісів, що забезпечують виконання процесів, але й велику кількість високорівневих абстракцій для використання ресурсів ОС. На відміну від них, ОС на основі екзоядра (англ. Exokernel) надає лише набір сервісів для взаємодії між процесами, а також необхідний мінімум функцій, пов'язаних із захистом: виділення і звільнення ресурсів, контроль прав доступу тощо. Екзоядро не повинно здійснювати абстрагування первісних (апаратних) ресурсів, для цього призначені непривілейовані прикладні бібліотеки – бібліотечна ОС (англ. libOS – library Operating System). Структура ОС на базі екзоядра наведена на рис. 1.19.

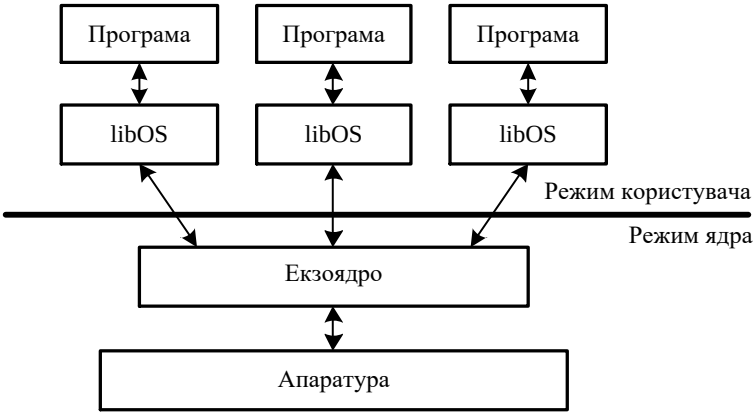


Рисунок 1.19 – Структура ОС на базі екзоядра

У рамках кожного конкретного процеса може бути реалізована своя libOS. Таким чином, ресурсами фактично повинні управляти самі процеси, а інтерфейси ресурсів мають бути як можна ближче до апаратної платформи. Чим нижче рівень, на якому функціонує інтерфейс, тим менше потрібно втручання з боку екзоядра і тим більший контроль над ресурсом отримують процеси.

Архітектура на основі екзоядра є подальшим розвитком і удосконаленням мікроядерної архітектури. На відміну від ОС на основі мікроядра, ОС, що базуються на екзоядрі, забезпечують набагато більшу

ефективність за рахунок відсутності необхідності в перемиканні між процесами при кожному зверненні до апаратури.

Оскільки для реалізації безпечного розподілу ресурсів не потрібні ні складні алгоритми, ні великий об'єм коду, екзоядро може бути невеликим і легко зрозумілим. Отже, і помилок воно містить відносно мало (що має велике значення). Також легко усувається проблема сильної залежності всієї ОС і процесів від реалізації абстракцій апаратури – екзоядро з мінімумом сервісів простіше модифікувати та завдяки вживанню надспеціалізованих абстракцій вирішується проблема низької продуктивності – процеси отримують тільки ті ресурси, в яких у них дійсно є потреба, і маніпулюють ними фактично без яких-небудь обмежень з боку ядра ОС.

Прикладами систем з екзоядерною архітектурою є: екзоядро XOK з бібліотечною ОС ExOS[60], Nemesis[61].

XOK є невеликим безпечним розподільником ресурсів x86 сумісних машин. У ньому застосовується механізм «безпечних зв'язків» (англ. Secure Bindings), який відокремлює авторизацію від фактичного використання ресурсу, у результаті також підвищується продуктивність системи. По-перше, перевірки, що здійснюються при зв'язуванні, – це прості та швидкі операції ядра (або навіть обладнання при їх апаратній реалізації). По-друге, при безпечному зв'язуванні отримання дозволу відбувається один раз, а подальше управління виконується незалежно. Оскільки саме прикладне ПЗ відповідає за багато складних високорівневих об'єктів (наприклад, файли, каталоги, мережеві з'єднання), екзоядро має «вникати» в їх семантику тільки під час зв'язування, для реалізації перевірок у процесі доступу цього не потрібно. Отже, екзоядро захищає ресурси без детального уявлення про них.

Бібліотечна ОС ExOS є не що інше, як «UNIX у бібліотеці». Реалізації UNIX-абстракцій були обрані з ядра ОС 4.4BSD, завдяки чому вони

[60] Перший випуск у 1995 р., остання версія в 2010 р., ресурс у мережі Internet – https://pdos.csail.mit.edu/archive/exo/

[61] Остання версія – 26.04.1999 р., ресурс у мережі Internet – https://www.cl.cam.ac.uk/research/srg/netos/projects/archive/nemesis/

дозволяють запускати без модифікації багато програм, включаючи досить складні. Для економії ресурсів використовуються бібліотеки, що розділяються. У реалізації відсутня підтримка груп процесів, віконного графічного інтерфейсу та механізму підкачки, оскільки низькорівневий інтерфейс XOK передбачає його прикладну реалізацію – підкачка може виконуватися з локального диска, по мережі або за допомогою регенерації даних, з різними перетвореннями, такими як компресія, верифікація вмісту за допомогою цифрових сигнатур, шифрування.

ExOS забезпечує розширювану реалізацію операційної системи UNIX на рівні користувача. Вимірювання продуктивності показують, що ExOS працює принаймні так само добре, як OpenBSD і FreeBSD: наприклад, веб-сервер Cheetah, створений на основі XOK, працює в три-чотири рази швидше, ніж веб-сервер з набору IIS (Internet Information Services), що працює на Windows NT Enterprise Edition [30].

Nemesis – ОС, з частковою сумісністю з POSIX, призначена для чутливих до часу програм, що потребують узгодженої «якості обслуговування» (англ. Quality of Service, QoS), наприклад, мультимедійних. Nemesis розроблено для платформ на базі x86, робочих станцій DEC Alpha і мережевих комп'ютерів на базі StrongARM SA-110. ОС Nemesis забезпечує виділення застосункам гарантованих часток всіх системних ресурсів, включаючи центральний процесор, пам'ять та ін. У мікроядерній ОС для виконання програми задіяні кілька процесів, більшість з яких є серверами (частинами ОС). Це, крім зменшення швидкодії, призводить до величезних труднощів з обліку використання ресурсів. Керівний принцип розробки Nemesis полягав у структуруванні ОС таким чином, щоб більшість коду могла виконуватися в самому процесі програми. Тому Nemesis має надзвичайно маленьке ядро та виконує більшість функцій ОС у спільних бібліотеках, які виконуються в режимі користувача. Забезпечення QoS для програм, чутливих до часу, обов'язково потребує більш частого перемикання контексту. Використовуючи єдиний адресний простір, Nemesis значно зменшує витрати на перемикання контексту, пов'язані з системою пам'яті. Єдиний адресний простір також позбавляє від необхідності копіювати мультимедійні дані.

Головні висновки:

♦ переваги екзоядра:

- покращена продуктивність – екзоядро зменшує накладні витрати на реалізацію системних викликів та взаємодії між процесами;

- покращена безпека – екзоядро ізолює системні служби в окремі процеси, зменшуючи площу атак та полегшуючи виявлення та усунення вразливостей безпеки;

- гнучкість – екзоядро дозволяє легко додавати, змінювати або видаляти системні служби без необхідності перекомпілювати все ядро;

♦ недоліки екзоядра:

- складність – екзоядро складніше, ніж традиційні монолітні ядра, що ускладнює його розуміння та підтримку;

- накладні витрати на ресурси – додаткові процеси, які використовує екзоядро, споживають додаткові системні ресурси, що в деяких випадках призводить до зниження продуктивності;

- незріла технологія – екзоядро є дослідницьким проєктом, що не має широкого використання.

1.5.6. ОС на базі наноядра

Термін «наноядро» (англ. Nanokernel) або «пікоядро» відноситься до ядра ОС, де загальний обсяг коду, що виконується в привілейованому режимі апаратного забезпечення, дуже малий [40]. Термін «пікоядро» іноді використовувався, щоб ще більше підкреслити малий розмір.

Зараз наноядро вважається таким, якщо воно відповідає не за всі важливі завдання ОС, а за одну конкретну задачу – воно обробляє апаратні переривання, які генерують інші компоненти комп'ютера. По завершенню обслуговування переривання програми від апаратних блоків комп'ютера, наноядро відправляє результати обслуговування програмі, що знаходиться вище за рівнем, використовуючи спосіб переривання програми. За своєю суттю ідеологія наноядра схожа на ідеологію рівня HAL, яка надає програмному забезпеченню вищих рівнів зручні методи абстракцій апаратних блоків та методи обслуговування переривань від них.

Розміри наноядра можуть бути настільки невеликими і примітивними, що навіть найважливіші апаратні модулі, які знаходяться прямо на материнській платі або на платах контролерів різних встановлених блоків, таких як, наприклад, таймери або контролери переривань з можливістю їх програмування, обробляються різними драйверами апаратних модулів, а не самим ядром.

Найчастіше в сучасних комп'ютерах наноядра використовуються для віртуалізації апаратного забезпечення реальних комп'ютерів або забезпечення можливості запуску «старої» ОС на новому, несумісному апаратному забезпеченні без її повного переписування. Віртуалізація апаратного забезпечення дозволяє організувати спільну роботу кількох операційних систем на одному пристрої. Причому ОС у такій реалізації можуть працювати «поруч одна з одною» або «одна система всередині іншої». За допомогою наноядера можна організувати незалежну ОС. Тобто наноядро дозволяє створити ОС, яка працюватиме на будь-якому залізі, незалежно від його апаратного або програмного складу. Таким чином, сучасну ОС можна запустити на старому обладнанні, на якому сучасні ОС з традиційними структурами ядер не працюють.

Як приклад можна навести KeyKOS[62] – перша ОС, в основі якої лежить наноядро. Особливість структури наноядра полягає в тому, що вона дозволяє переносити ОС на різну апаратну основу, і, крім цього, забезпечує можливість встановлення діючої версії ОС на несумісне апаратне обладнання без її суттєвої переробки.

Також, компанія Apple Computer застосовувала наноядро Mac OS Classic[63] для PowerPC, з метою трансляції апаратних переривань комп'ютерів на основі процесора PowerPC у формат для процесора Motorola 680x0. Пізніше, у версії Mac OS 8.6, наноядро виконувало додатково

[62] Перший випуск у 1977 р., остання версія в 1099 р., ресурс у мережі Інтернет – http://cap-lore.com/CapTheory/KK/

[63] Перший випуск 24.01.1984 р., остання версія 9.2.2 від 05.12.2001 р.

віртуалізацію мультипроцесорних можливостей, що дозволило забезпечити роботу симетричної багатопроцесорності SMP у системі.

Інший приклад застосування наноядерної структури – наноядро Adeos[64] (Adaptive Domain Environment for Operating Systems), яке працює як модульне ядро для Linux та дозволяє працювати паралельно з нею іншій ОС. Воно відрізняється від інших наноядер тим, що це не лише низькорівневий рівень для зовнішнього ядра, але ще воно призначене для одночасного запуску кількох ядер. Adeos дозволило забезпечити SMP, покращену віртуалізацію, можливість налагодження ядра без встановлення виправлень та систему реального часу. Adeos можна завантажувати як окремий модуль ядра Linux, щоб дозволити іншій ОС працювати з ним.

Головні висновки:

♦ переваги наноядра:

- модульність – наноядро легко модифікувати та налаштовувати, оскільки функції можна додавати або видаляти за потреби без необхідності перекомпілювати все ядро;

- безпека – завдяки делегуванню несуттєвих функцій процесам у режимі користувача зменшується площа атак на наноядро;

♦ недоліки наноядра:

- складність – хоча наноядро може бути простішим за традиційне монолітне ядро, воно вимагає певного рівня досвіду для розуміння;

- обмежена функціональність – через мінімальний дизайн наноядро може не містити всіх функцій, необхідних для деяких випадків використання, що призводить до необхідності додаткових процесів у режимі користувача;

- незріла технологія – наноядра все ще є відносно новою сферою, яка швидко розвивається, і не так широко використовується, як традиційні монолітні ядра.

---

[64] Перший випуск 03.06.2002 р., ресурс у мережі Інтернет – https://www.opersys.com/adeos/

1.5.7. Віртуальні машини

Багаторівневий підхід побудови ОС був доведений до логічного завершення в концепції віртуальних машин (англ. Virtual machine) [32, 44]. Основною ідеєю віртуальної машини є абстрагування апаратних засобів одного комп'ютера (процесор, пам'ять, жорсткі диски, мережні карти тощо) для кількох різних середовищ виконання, тим самим створюючи ілюзію, що кожне окреме середовище виконання управляє власним комп'ютером.

Використовуючи планування процесора і технологію віртуальної пам'яті, основна ОС (віртуальна машина) може створити ілюзію, що процес має свій власний процесор зі своєю власною (віртуальною) пам'яттю. Як правило, процес має додаткові можливості, такі як системні виклики і файлову систему, які не надаються апаратно. Підхід на основі віртуальних машин не забезпечує таку додаткову функціональність, а являє собою інтерфейс, що збігається з інтерфейсом апаратури. Кожен процес забезпечується (віртуальною) копією реальної апаратури (рис. 1.20).

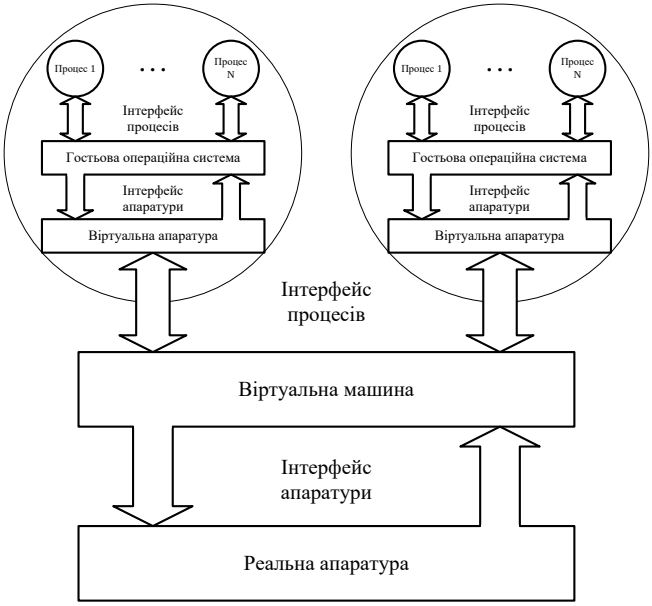


Рисунок 1.20 – Модель віртуальної машини

Є кілька причин для створення віртуальних машин, всі з яких фундаментально відносяться до створення можливості розділяти одне й те ж обладнання серед різних середовищ виконання (тобто, різних гостьових ОС) одночасно.

Хоча концепція віртуальних машин вигідна, її важко здійснити. Потребується багато роботи по створенню точного дублікату основної машини. Гостьова ОС може виконуватися тільки в режимі користувача. Але для створення повної ілюзії роботи з реальною апаратурою вона повинна мати віртуальний режим користувача і віртуальний режим ядра, обидва з яких працюють у реальному режимі користувача. Ті дії, які викликають перехід з режиму користувача в режим ядра на реальній машині (наприклад, системні виклики або спроба виконати привілейовану інструкцію), також мають передбачати перехід з віртуального режиму користувача у віртуальний режим ядра на гостьових ОС.

Такий перехід може здійснюватися за наступною схемою. Коли системний виклик, наприклад, генерується в програмі (гостьовій ОС), запущеній на віртуальній машині у віртуальному режимі користувача, це призводить до переходу в монітор віртуальної машини (МВМ) (англ. CP – Control Program або VMM – Virtual Machine Monitor), який управляє реальною апаратурою. Коли МВМ одержує управління, він може змінювати вміст регістрів і програмного лічильника для віртуальної машини для імітації ефекту системних викликів.

Основна відмінність у віртуальних машинах від звичайних ОС полягає у витратах часу. У той час, як реальна операція введення-виведення може зайняти 100 мс, віртуальна операція може зайняти менше часу (тому що вона не виконується з реальним обладнанням) або більше в декілька разів (тому що вона інтерпретується МВМ). Крім того, більшість машинних інструкцій гостьових ОС також моделюється (інтерпретується), щоб забезпечити справжню віртуальну машину.

Однак, концепція віртуальних машин має вагомі переваги: у віртуальних машинах реалізовано повний захист різних ресурсів системи. Кожна віртуальна машина повністю ізольована від всіх інших віртуальних машин, і тому немає проблем захисту. У той же час, немає прямого обміну

ресурсів, що може бути обійдене за допомогою віртуального міні-диску (який пов'язаний з частиною реального диску або з оперативною пам'яттю), через який можна обмінюватися файлами. Також можна визначити мережу віртуальних машин, кожна з яких дозволяє передавати інформацію через віртуальну мережу зв'язку (мережа моделюється як фізична мережа зв'язку, але реалізована програмно).

Віртуальні машини – ідеальний засіб для наукових досліджень і розробок у галузі ОС. Як правило, модифікація ОС – важке завдання: ОС складається з великої кількості модулів, пов'язаних між собою, і важко бути впевненим, що зміни в одному модулі, не будуть причиною незрозумілих помилок в іншому модулі. Крім того, помилки при проведенні змін можуть катастрофічно вплинути на всю обчислювальну систему (наприклад, неправильна зміна покажчика може призвести до помилки, яка знищить всю файлову систему). Віртуальна машина може усунути більшу частину подібних проблем: розробка системи виконується на віртуальній машині, а не на фізичній машині, і, відповідно, помилки не впливають на реальні ресурси і не можуть їх пошкодити.

Згідно з наведеними характеристиками віртуальних машин можна визначити випадки, в яких їх використання є доцільним:

♦ для захисту інформації і обмеження можливостей процесів;

♦ для дослідження продуктивності ПЗ або нової комп'ютерної архітектури;

♦ для емуляції різної архітектури (наприклад, емулятора ігрової приставки);

♦ з метою оптимізації використання ресурсів потужних комп'ютерів;

♦ для моделювання інформаційних систем з клієнт-серверною архітектурою на одній ЕОМ (емуляція комп'ютерної мережі за допомогою декількох віртуальних машин).

Першою системою такого роду була система CP/CMS (більш пізнє найменування – VM/370) для IBM/370. Цей проєкт з'явився в 1964 році з назвою CP-40, він дозволяв запускати кілька екземплярів клієнтських ОС CMS. У 1967 році на основі проєкту CP-40/CMS розроблено ОС CP-67/CMS, яка була розрахована на роботу кількох користувачів у режимі

поділу часу. CP-67/CMS працювала на апаратному мейнфреймі IBM System/360-67 і складалася з двох компонентів: CP (Control Program) – програма керування віртуалізацією та CMS (Cambridge Monitor System) – одна з найбільш поширених однокористувацьких ОС для запуску у віртуальному оточенні CP (гостьова ОС). Користувачі підключалися до гостьових ОС за допомогою спеціальних пристроїв введення-виведення (терміналів).

Вже тоді віртуалізація мала суттєві переваги над концепцією поділу часу:

♦ збільшена надійність та безпека за рахунок ізоляції користувачів;

♦ запуск будь-яких програм (не лише пристосованих до концепції поділу часу) за рахунок симуляції окремого комп'ютера для кожного користувача;

♦ збільшена продуктивність за допомогою використання легковажних гостьових ОС.

У 1980-х роках сервери та персональні комп'ютери на базі архітектури x86 стали доступнішими, внаслідок чого мейнфрейми з віртуалізацією та терміналами втратили свої позиції. У 1988 році компанія Insignia Solutions представила емулятор програмного забезпечення SoftPC, за допомогою якого можна було запускати програми MS-DOS на робочих станціях UNIX. У 1997 році компанія Connectix створила програму Virtual PC для запуску під Mac ОС Windows. У 1999 році компанія VMware представила VMware Workstation, яка дозволила запускати цілу низку ОС у рамках віртуальних машин.

На початку 2000-х років стали з'являтися продукти для серверної віртуалізації. Ці рішення дали можливість запускати кілька ізольованих гостьових ОС у віртуальному середовищі на одному фізичному сервері, що спрощувало адміністрування інфраструктури, підвищувало її стійкість до відмов та знижувало простоювання серверного обладнання. У 2001 р. VMware представила ESX Server та GSX Server. У 2003 р. Microsoft придбала компанію Connectix і перезапустила проєкт Virtual PC, який став попередником Microsoft Virtual Server та сучасного Microsoft Hyper-V. У 2007 р. компанією Innotek було представлено VirtualBox. Також у 2007 р.

на ринок корпоративної віртуалізації вийшла компанія Citrix, яка купила компанію XenSource і почала розвивати проєкт з відкритим вихідним кодом Xen, надаючи для клієнтів комерційну версію продукту Citrix XenServer (зараз перейменований на Citrix Hypervisor).

Крім серверної віртуалізації існують також продукти для віртуалізації робочих столів (нове втілення зв'язки мейнфрейм-термінал з 1960-70 рр.), застосунків та ін.

Згідно функції ОС абстрагування ресурсів, можна виділити основні види віртуалізації:

♦ апаратна віртуалізація – дозволяє створювати незалежні та ізольовані один від одного віртуальні комп'ютери за допомогою програмної імітації ресурсів (процесора, пам'яті, мережі, диска та ін.) фізичного сервера. Фізичний сервер називають хостом (англ. Host), віртуальні комп'ютери – гостями (англ. Guest). Гіпервізор (англ. Hypervisor) – це програма, яка керує фізичними ресурсами обчислювальної машини та розподіляє ці ресурси між кількома різними ОС, дозволяючи запускати їх одночасно. Гіпервізор створює з одного фізичного комп'ютера кілька копій, клонів його апаратних ресурсів і кожен клон видно з боку користувача як окремий пристрій. На кожну ВМ можна встановити гостьову ОС користувача, не прив'язану до апаратури хоста. Гіпервізор ізолює запущені ОС одну від одної так, щоб кожна монопольно використовувала виділені їй ресурси. Але при необхідності гіпервізор дозволяє ОС віртуальних машин взаємодіяти між собою. Механізмом зв'язку між ОС може бути загальний доступ до певних файлів та обмін даними по локальній мережі. На практиці на віртуальних машинах можуть використовуватись різні ОС для різних цілей – наприклад, Windows Server під контролер домену Active Directory та Debian під веб-сервер NGINX;

♦ віртуалізація робочих столів – дозволяє відокремити логічний робочий стіл (набір програм, що працює під ОС) від фізичної інфраструктури (наприклад, персональних комп'ютерів). Однією із найпоширеніших форм віртуалізації робочих столів є VDI (англ. Virtual Desktop Infrastructure) – інфраструктура віртуальних робочих столів. Кожен користувач VDI має програмну імітацію ОС із необхідним набором програм

на фізичному сервері під керуванням гіпервізора та може підключатися до нього через мережу. На практиці VDI може використовуватися для роботи великої кількості співробітників на «віддаленні» для того, щоб не закуповувати для них окремі робочі станції та керувати централізованою інфраструктурою;

♦ віртуалізація на рівні ОС – дозволяє запускати програмне забезпечення ізольованих на рівні операційної системи просторах. Найбільш поширеною формою віртуалізації на рівні ОС є контейнери – наприклад, Docker[65]. Контейнери легковаговіші, ніж віртуальні комп'ютери, оскільки вони спираються на функціонал ядра ОС і їм не потрібно взаємодіяти з апаратним забезпеченням. На практиці контейнери являють собою ізольоване середовище для запуску будь-якої програми з усіма його залежностями та налаштуваннями.

Процес поділу фізичного сервера на кілька унікальних та ізольованих віртуальних комп'ютерів (серверів) за допомогою ПЗ гіпервізора має назву «віртуалізація серверів». На кожному віртуальному сервері можуть незалежно виконуватися власні ОС.

Віртуалізація серверів дозволяє:

♦ оптимізувати витрати на придбання серверного обладнання – під кожну задачу виділяється віртуальний сервер з необхідною кількістю ресурсів (центральний процесор, основна пам'ять та ін.), простоювання обладнання мінімізується;

♦ спростити супровід інфраструктури – створення, видалення або обслуговування віртуального комп'ютера як правило простіше і швидше, ніж аналогічні операції з фізичним сервером;

♦ підвищити стійкість до відмови інфраструктури – віртуальні комп'ютери ізольовані один від одного, програмний збій на одному з них не призведе до втрати працездатності сервісів і застосунків на інших.

[65] Перший випуск 13.03.2013 р., остання версія 27.3.1 від 20.09.2024 р., ресурс у мережі Інтернет – https://www.docker.com/

Гіпервізор забезпечує ізольоване середовище виконання для кожного віртуального комп'ютера, а також керує доступом ВМ та гостьових ОС до апаратних ресурсів фізичного сервера: забезпечує паралельне і незалежне функціонування декількох ОС на одному комп'ютері. У класичному підході гіпервізори групуються за двома типами: гіпервізори першого типу запускаються безпосередньо на апаратному забезпеченні комп'ютера (тип 1, X); гіпервізорам другого типу для роботи необхідна наявність основної ОС (тип 2, V). В останні кілька років до класифікації додався гібридний тип гіпервізорів (тип 1+), який поєднує характеристики першого і другого типів.

Гіпервізор першого типу (англ. Native or Bare-metal Hypervisor) виконується як контрольна програма безпосередньо на апаратній частині комп'ютера і не вимагає для своєї роботи основної ОС (рис. 1.21). У цій архітектурі гіпервізор керує розподілом обчислювальних ресурсів і сам контролює всі звернення віртуальних комп'ютерів до реальних пристроїв.

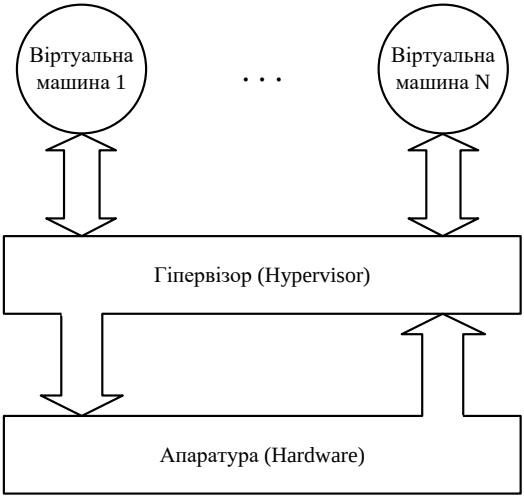


Рисунок 1.21 – Гіпервізор типу 1

Гіпервізор першого типу називають ще тонким гіпервізором або автономним гіпервізором, що виконується на «чистому залізі». Гіпервізор

першого типу найпростіше сприймати як специфічну компактну ОС, яка встановлюється прямо на апаратну частину і має основні ознаки ОС:

♦ замість невпорядкованого набору апаратного забезпечення надає абстрактний набір ресурсів для прикладних програм;

♦ управляє набором ресурсів – розподіляє процесорний час, пам'ять, пристрої введення-виведення між програмами, які претендують використання ресурсів комп'ютера.

Він надає гостьовим ОС, запущеним під його керуванням на верхньому рівні, абстракцію, тобто службу віртуальної машини. У результаті кожна гостьова ОС отримує собі від гіпервізора ілюзію повноправного розпорядження всіма ресурсами комп'ютера – аналогічно тому, якби ОС працювала на реальному устаткуванні в привілейованому режимі ядра, режимі супервізора.

Такі гіпервізори показують високу швидкодію, проте мають недолік – необхідність підтримувати драйвери пристроїв призводить до звуження списку сумісного апаратного забезпечення.

У ПЗ гіпервізора першого типу є дуже важлива особливість – розмір його коду в сотні разів менший, ніж у більшості сучасних ОС. Це забезпечує значно меншу кількість можливих помилок. Збій у роботі гостьової ОС на одній із ВМ користувача не повинен вплинути на роботу всіх сусідніх машин на тому самому фізичному обладнанні.

Однією з найважливіших вимог до гіпервізора є безпека, оскільки гіпервізор отримує повне управління апаратними ресурсами комп'ютера, на яких виконується віртуалізація, не дозволяючи гостьовій ОС:

♦ блокувати переривання;

♦ модифікувати таблиці відображення сторінок віртуальної пам'яті на фізичну для комп'ютера;

♦ змінювати дані в комірках пам'яті, виділених іншим запущеним процесам (крім випадків, коли це заздалегідь передбачено логікою роботи, обміну даними між ними).

Системні виклики також перехоплюються та виконуються всередині гіпервізора, але з боку кожної гостьової ОС все виглядає так, як має бути при звичайному виконанні інструкцій у режимі ядра.

Гіпервізор виявляється єдиним програмним забезпеченням, яке запущено в режимі максимальних привілеїв. Така властивість гіпервізора називається еквівалентністю – поведінка програм користувача не відрізняється при роботі на віртуальній машині та на фізичному обладнанні за винятком часових характеристик.

При цьому час виконання коду суттєво відрізняється – гіпервізор забирає частину процесорного часу для своїх потреб, перехоплення та аналізу інструкцій гостьової ОС, а також емуляції виконання деяких із них. Крім того, ресурси фізичного обладнання зазвичай розділені між кількома віртуальними машинами і кожна з них отримує на вимогу лише частину процесорного часу. Однак цього достатньо для повноцінної роботи більшості процесів, оскільки не всі вони постійно та рівномірно завантажені. Частина може простоювати в очікуванні дій користувача чи завершення роботи повільного периферійного устаткування. Цей час ефективно використовується, оскільки система перерозподіляє його для інших активних процесів у багатозадачному режимі.

Приклади гіпервізорів 1 типу: VMWare ESXi[66], KVM (англ. Kernel-based Virtual Machine або Proxmox Virtual Environment)[67] – може бути також віднесено до другого типу, Xen (Xenserver[68], Citrix Hypervisor[69]) та Hyper-V (продукт від компанії Microsoft, вбудований у Windows 8, 8.1, 10, 11 випусків Pro або Enterprise) – можуть бути також віднесені до гібридного типу.

Гіпервізор другого типу (англ. Hosted Hypervisor) є додатковим програмним шаром, розташованим над основною ОС. Він управляє гостьовими ОС, тоді як емуляцією і управлінням фізичними ресурсами займається основна ОС (рис. 1.22). Фактично гіпервізор другого типу працює як один із процесів, що виконуються основною ОС: він управляє

[66] Перший випуск 23.03.2001 р, остання версія 8.0 Update 3b від 17.09.2024 р., ресурс у мережі Internet https://www.vmware.com/products/esxi-and-esx.html

[67] Перший випуск 15.04.2008 р, остання версія 8.2 від 24.04.2024 р., ресурс у мережі Internet – https://www.proxmox.com/en/

[68] Остання версія 8 від 10.01.2024 р., ресурс у мережі Internet – https://www.xenserver.com/

[69] Остання версія 8.2 від 30.10.2023 р., ресурс у мережі Internet – https://www.citrix.com/

гостьовими ОС, а емуляцію і управління фізичними ресурсами виконує основна ОС.

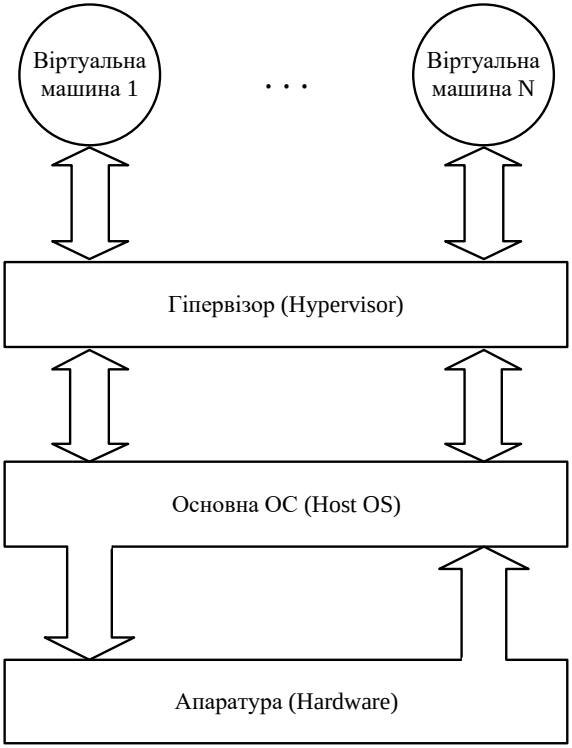


Рисунок 1.22 – Гіпервізор типу 2

Гіпервізори другого типу мають меншу відносно гіпервізорів першого типу швидкодію і рідше використовуються в промисловій експлуатації, проте відмінно підходять для завдань навчання та розробки ПЗ.

До таких гіпервізорів відносяться Oracle VM VirtualBox[70], VMWare Workstation[71] та вказаний раніше KVM (Proxmox Virtual Environment).

Гібридний гіпервізор (англ. Hybrid Hypervisor) поєднує в собі характеристики гіпервізорів першого і другого типів – він виконується в спеціалізованій сервісній ОС. Сервісна ОС називається батьківським розділом (англ. Parent Partition) у термінології Hyper-V або доменом (англ. Domain dom0) у термінології Xen. Після установки гіпервізора ядро ОС переходить у режим підтримки віртуалізації і передає управління ресурсами процесора і пам'яті гіпервізору. При цьому сервісна ОС бере на себе функцію обробки звернень до драйверів пристроїв та операцій введення-виведення.

Гібридний гіпервізор є продовженням розвитку ОС на базі екзоядра.

Даний підхід зручний з точки зору сумісності з обладнанням: не потрібно додавати в гіпервізор драйвери пристроїв (розширюється список сумісного апаратного забезпечення). Також гіпервізор звільняється від обробки викликів до драйверів пристроїв – ці звернення обробляє сервісна ОС.

В якості прикладів можна вказати Xen (XenServer, Citrix Hypervisor), Hyper-V – що можуть також бути віднесені до першого типу; oVirt[72]; Oracle VM Server[73].

Головні висновки:

♦ переваги віртуальних машин:

- захист – кожна віртуальна машина повністю ізольована від усіх інших віртуальних машин;

[70] Перший випуск 17.01.2007 р, остання версія 7.1.2 від 26.09.2024 р., ресурс у мережі Інтернет – https://www.virtualbox.org/

[71] Перший випуск в 1999 р., остання версія 17.6 від 02.09.2024 р., ресурс у мережі Інтернет – https://www.vmware.com/

[72] Остання версія 4.5.5 від 01.12.2023 р., офіційний ресурс у мережі Інтернет https://www.ovirt.org/

[73] Остання версія 3.4.7, січень 2022 р., для апаратної платформи x86 та версія 3.6.2, листопад 2023 р., для апаратної платформи SPARC, ресурс у мережі Інтернет https://www.oracle.com/

- абстрагування – віртуальна машина може забезпечувати архітектуру, яка відрізняється від реальних комп'ютерів;

- зручність – просте обслуговування, доступність, відновлення, резервне копіювання та клонування;

♦ недоліки віртуальних машин:

- продуктивність – коли кілька віртуальних машин одночасно працюють на головному комп'ютері, на продуктивність однієї віртуальної машини можуть впливати інші запущені віртуальні машини;

- ефективність – віртуальні машини не такі ефективні, як реальні при доступі до обладнання.

Контрольні запитання

1. Чим відрізняється інструментальне ПЗ від системного ПЗ?
2. Що входить до складу інструментального ПЗ?
3. Що входить до складу системного ПЗ?
4. Роз'ясніть поняття мультипрограмність і багатозадачність.
5. Надайте визначення ОС.
6. Які основні функції ядра ОС?
7. Які основні вимоги висуваються до ОС?
8. Чим відрізняються між собою ОС жорсткого та м'якого реального часу?
9. У чому полягає різниця між монолітним модульним ядром та мікроядром?
10. У чому полягає різниця між монолітним модульним ядром та багаторівневою організацією ОС?
11. У чому полягає різниця між монолітним модульним ядром та гібридним ядром?
12. У чому полягає різниця між мікроядром та гібридним ядром?
13. Чим відрізняється наноядро та гіпервізор?
14. Чим відрізняються гіпервізори типів 1 та 2?
15. Яка відмінність між режимом ядра та режимом користувача?
16. Які додаткові функції виконують мобільні ОС по відношенню до класичних?

17. Які особливості використання в якості гостьових ОС у віртуальних машинах ОС реального часу?
18. Які основні функції виконує мікроядро?
19. Які функції виконують екзоядро та наноядро? Чим вони відрізняються?
20. Які архітектурні підходи використовуються в сучасних найбільш розповсюджених ОС?
21. За що відповідає рівень ОС HAL?
22. За якими ознаками можна класифікувати ОС?
23. Наведіть найбільш відомі приклади ОС для кожної з архітектур (монолітне ядро, монолітне модульне ядро, багаторівневе ядро, мікроядро, гібридне ядро, екзоядро, наноядро, віртуальні машини).
24. Назвіть основні етапи розвитку ОС UNIX.
25. Назвіть основні етапи розвитку ОС Windows.
26. Назвіть основні етапи розвитку ОС Mac OS.
27. Порівняйте побудову ядра ОС UNIX та Linux.
28. Порівняйте побудову ядро ОС Windows NT та Mac OS X.

2. ПРОЦЕСИ

2.1. Основні визначення

Найважливішою частиною ОС, що безпосередньо впливає на функціонування обчислювальної системи, є підсистема управління процесором. Ця підсистема планує виконання процесів, тобто розподіляє процесорний час між кількома процесами, що одночасно існують у системі, а також займається створенням і знищенням процесів, забезпечує процеси необхідними системними ресурсами, підтримує взаємодію між процесами.

Перед тим як розглядати питання управління процесами необхідно визначити деякі з термінів, що мають відношення до цієї теми.

Процесор, або центральний процесор (ЦП) (англ. CPU – Central Processing Unit) – частка апаратного забезпечення ЕОМ, яка робить обчислення і виконує програми. Процес (англ. Process) – абстракція, що описує програму під час її виконання, тобто це – єдиний екземпляр програми, що виконується (наприклад, виконання математичного обчислення). Програма – статичний об’єкт, що є файлом або сукупністю файлів з кодами і даними. Взагалі, процес можна представити як сукупність даних ядра системи, необхідних для опису відображення програми в пам’яті і управління її виконанням.

Згідно зі стандартами програма та процес визначаються наступним чином.

Програма – комбінація комп’ютерних інструкцій і визначень даних, які дозволяють апаратному забезпеченню комп’ютера виконувати обчислювальні чи керуючі функції (ISO/IEC/IEEE 24765:2017 Systems and software engineering – Vocabulary).

Програма – фактично це дані, призначені для управління конкретними компонентами системи обробки інформації з метою реалізації певного алгоритму.

Процес обробки даних – сукупність взаємозв’язаних і взаємодіючих дій, що перетворюють вхідні дані у вихідні (стандарт ISO/IEC/IEEE 12207:2017 «Systems and software engineering – Software life cycle processes»).

У програмі міститься пасивна послідовність інструкцій, процес, на відміну від програми, це – активна система дій, що реалізує певну функцію в системі обробки інформації.

Будь-який процес характеризується станами, які визначаються присутністю тих або інших ресурсів у його розпорядженні і, відповідно, можливістю фактично виконувати дії, які до нього відносяться. Перерозподіл ресурсів, який виконується керуючою програмою, впливає лише на тривалість процесу обробки даних, але не на його кінцевий результат. В ОС процес оформлюється за допомогою спеціальних структур керуючих даних, якими маніпулює керуючий механізм.

У конкретних системах обробки інформації зустрічаються різновиди процесів, які розрізняються способом оформлення і складом ресурсів, що призначаються та віднімаються в процеса, у різних системах вони мають спеціальні назви, наприклад, для ЄС ЕОМ (Єдина Серія Електронних Обчислювальних Машин – серія комп'ютерів, які випускалися в період з 1971 р. до середини 1990-х рр.) – задача, для IBM – task, для UNIVAC (UNIVersal Automatic Computer – виробництво США, використання з 1951 р. по 1970 р.) – activity.

Для ОС процес є одиницею роботи, заявкою на споживання системних ресурсів.

Слід зазначити, що програма не являється процесом. Програма – пасивний об'єкт (наприклад, файл на диску, що містить список інструкцій), а процес – активний об'єкт, він має лічильник команд, що конкретизує наступну інструкцію для виконання, і набір пов'язаних з ним ресурсів. Програма стає процесом, коли файл з нею завантажується в пам'ять та в ОС створюються необхідні системні структури даних для підтримки виконання процеса. Таким чином, можна сказати, що процес – це програма в стадії виконання. Два екземпляри програми, які були запущені на виконання, не є двома екземплярами одного процеса – це два процеси, пов'язаних з однією і тією ж програмою, але з двома окремими послідовностями дій. Процес – це більше ніж програмний код (секція TEXT – кодовий сегмент), він, крім цього, характеризується поточною активністю, що визначається значенням програмного лічильника (англ. Program Counter) і змістом регістрів

процесора; має стек, який містить тимчасові дані (параметри функцій, адреси повернення з функцій, локальні змінні); має секцію DATA (сегмент даних), яка містить глобальні змінні; а також може мати купу (англ. Heap), що є пам'яттю, яка динамічно виділяється під час виконання процеса.

Умовно процеси можна поділити на три категорії:

- системні;
- фонові;
- прикладні.

Системні процеси є часткою ядра ОС і завжди розташовані в оперативній пам'яті. Інструкції і дані цих процесів знаходяться в пам'яті, яку займає ядро ОС, тому такі процеси можуть викликати функції ОС безпосередньо і звертатися до даних, недоступних для решти процесів (наприклад, звертатися до диспетчерів сторінкового заміщення, пам'яті ядра, буферного кеша та ін.).

Фонові процеси (в ОС класу UNIX вони мають назву – «демони» (англ. Daemon)) – це неінтерактивні процеси, які зазвичай запускаються при ініціалізації ОС (після ініціалізації ядра) і забезпечують роботу різних її підсистем. Наприклад, підсистеми термінального доступу, системи друку, системи мережевого доступу та ін. «Демони» не пов'язані з призначеними для користувача сеансами роботи і не можуть безпосередньо управлятися користувачами.

Прикладні процеси, як правило, породжуються в рамках призначеного для користувача сеансу. Вони можуть виконуватися як в інтерактивному, так і у фоновому режимах.

У будь-якій ОС на вимогу існуючого процеса проводиться робота по створенню нових процесів. Процес, що задає дану вимогу, називають таким що породжує, а створюваний на вимогу – породженим. Якщо породжений процес, у свою чергу, на інтервалі свого існування видає вимогу на породження іншого процеса, то він одночасно стає також таким, що породжує. Ці два типи процесів можуть бути незалежними один від одного або мати відносити типу «батько-нащадок». У випадку незалежних процесів – створює процеси та управляє ними тільки сама ОС. Для ОС з організацією процесів «батько-нащадок» процес, що породжує новий

процес, називається батьківським (англ. Parent process), а породжений процес називається дочірнім (англ. Child process). Дочірній процес успадковує більшість атрибутів (наприклад, відкриті файли) від свого батька. Батьківські процеси можуть управляти дочірніми процесами. В UNIX-подібних системах підтримуються відносини типу «батько-нащадок» (тому оточення батьківського процеса майже повністю відповідає оточенню нащадка), в ОС на базі ядра NT (Microsoft Windows) це відношення не підтримується (хоча деяку інформацію про батьківський процес має нащадок, а інформацію про нащадка має батьківський процес).

Процеси також можна класифікувати за результатами виконання. Два процеси, які мають однаковий кінцевий результат обробки одних і тих же вихідних даних по одній і тій же або навіть різним програмам на одному й тому або на різних процесорах, називають еквівалентними. Якщо в кожному з еквівалентних процесів обробка даних відбувається по одній і тій самій програмі, але траси (послідовність і тривалість перебування процеса в кожному зі своїх станів) при цьому в загальному випадку не збігаються, то такі процеси називають тотожними. При збігу трас у тотожних процесів їх називають рівними. У всіх інших випадках процеси завжди різні.

Під час виконання процеси можуть відрізнятися за динамічними ознаками: якщо інтервали виконання двох процесів не перетинаються в часі, то такі два процеси називають послідовними один відносно другого. Якщо на певному інтервалі часу існують одночасно два процеси, то вони на цьому інтервалі є паралельними один щодо другого. Якщо на певному інтервалі часу знайдеться хоча б один момент, в якому існує один процес, але не існує інший, і хоча б один момент, в якому обидва процеси існують одночасно, то такі два процеси називають комбінованими.

Процеси незалежно від їх виду можуть бути взаємозалежними або ізольованими один від другого. Два процеси є взаємозалежними, якщо між ними за допомогою системи управління процесами підтримуються будь-якого роду зв’язки: функціональні, просторово-часові, керуючі, інформаційні та ін. В іншому випадку вони є ізольованими, тобто процесами зі слабкими зв’язками, однак при відсутності явних зв’язків вони можуть

бути пов’язані між собою опосередковано і, певним чином, впливати на виконання один одного.

Якщо два взаємозалежні процеси при виконанні спільно використовують деякі ресурси, але інформаційно між собою не пов’язані (немає обміну інформацією), то такі процеси називають інформаційно-незалежними. Зв’язок між такими процесами може бути або функціональним, або просторово-часовим. При наявності інформаційних зв’язків між двома процесами їх називають взаємодіючими. Коли необхідно підкреслити зв’язок між взаємозалежними процесами по ресурсах, їх називають конкуруючими.

Процеси в ОС можуть мати пріоритети (англ. Priority). Пріоритет процеса визначає, як часто даний процес, у порівнянні з іншими процесами, що стоять у черзі на виконання до процесора, буде виконуватися процесором.

Пріоритет може виражатися цілим чи дробовим, позитивним чи негативним значенням. У деяких ОС прийнято, що пріоритет тим вищий, чим більше (в арифметичному сенсі) число, що з ним пов’язане. В інших системах, навпаки, менше число означає вищий пріоритет.

Система може привласнювати процесам пріоритети автоматично або вони можуть призначатися ззовні. Пріоритети можуть бути заслуженими або купленими. Вони можуть бути статичними або динамічними. Вони можуть призначатися за якимось раціональним принципом або присвоюватися в ситуаціях, коли системі просто необхідно якимось чином розрізняти процеси.

Статичні пріоритети не змінюються, такий механізм пріоритетів досить простий і не пов’язаний з великими витратами. Проте слід враховувати, що такий механізм недостатньо гнучкий, тому що не реагує на зміну ситуації в ОС. Динамічні пріоритети дозволяють підвищити реактивність системи, тому що реагують на зміну ситуації, і початкове значення пріоритету процеса може бути змінено на нове, більш відповідне значення.

Куплені пріоритети дають можливість користувачеві за рахунок придбання коштів підвищити пріоритет завдання і отримати більш високий рівень обслуговування.

2.2. Модель процеса

На рис. 2.1 наведено приклад типового образу процеса для ОС Linux, створеного з програми, написаної мовою С, в оперативній пам’яті.

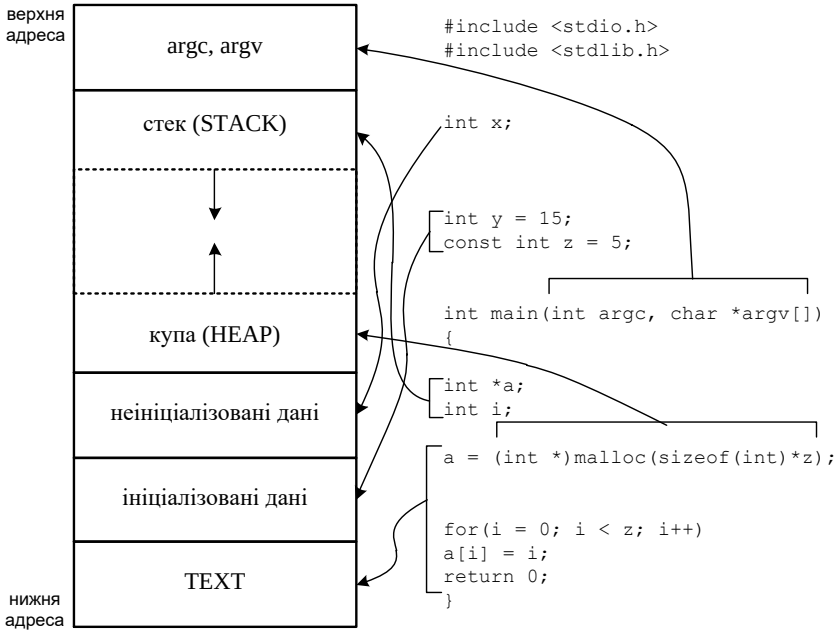


Рисунок 2.1 – Типовий образ процеса в пам’яті

1. Текстовий сегмент, або секція коду «TEXT», є одним із розділів програми в об’єктному файлі або в пам’яті, який містить виконувані інструкції. Як область пам’яті сегмент тексту може бути розміщений під купою або стеком, щоб запобігти його перезапису при збільшенні купи або переповненні стека. Зазвичай текстовий сегмент доступний для спільного використання, тому доцільно в пам’ять завантажувати лише одну копію для програм, що часто виконуються, таких як текстові редактори, компілятор, оболонки тощо. Крім того, текстовий сегмент часто доступний лише для читання, щоб програма не могла випадково змінити свої інструкції.

2. Ініціалізований сегмент даних або секція даних «DATA». Сегмент даних — це частина віртуального адресного простору програми, яка містить значення всіх зовнішніх, глобальних, статичних і постійних змінних, значення яких ініціалізуються під час оголошення змінної в програмі. Цей сегмент можна класифікувати на ініціалізовану область лише для читання та ініціалізовану область читання-запису залежно від змісту.

У прикладі на рис. 2.1 змінні *y* і *z* оголошено поза межами функції *main*(), через що вони зберігаються в частині ініціалізованого сегмента даних для читання та запису, але глобальна змінна *z* оголошується за допомогою ключового слова «*const*» і, отже, зберігається в частині ініціалізованого сегмента даних, яка доступна лише для читання.

3. Неініціалізований сегмент даних або секція «BSS» (block starting symbol). Дані в цьому сегменті ініціалізуються ядром в арифметичний 0 (усі статичні та глобальні змінні) перед початком виконання процеса. Оскільки значення змінних, що зберігаються в «BSS», можна змінювати, цей сегмент даних має дозволи на читання та запис. Наприклад, змінна, оголошена «*int x;*» (рис. 2.1) буде міститись в секції «BSS» та буде ініціалізована значенням 0.

4. Сегмент стека «STACK» реалізований за принципом LIFO (останній прийшов, перший вийшов) і зростає до нижчої адреси (але це залежить від архітектури комп'ютера). Стек росте в напрямку, протилежному купі. У сегменті стека зберігаються значення локальних змінних і значення параметрів, переданих у функції, а також деяка додаткова інформація, наприклад, адреса інструкції, яка має бути виконана після повернення з функції. Набір значень, розміщених у стеці, для одного виклику функції, називається «фреймом стека», який складається як мінімум з адреси повернення. Стек також використовується для рекурсивних функцій. Кожного разу, коли функція рекурсивно викликає себе, створюється новий фрейм стека, який дозволяє набору змінних одного фрейму стека не заважати іншим змінним іншого екземпляра функції. Коли функція сама себе викликає велику кількість разів, це може спричинити переповнення стека, що призводить до помилки сегментації в програмі.

У прикладі на рис. 2.1 змінні a та i розміщуються в одному фреймі стека після виклику функції main.

5. Купа «HEAP» використовується для пам'яті, яка виділяється під час виконання (динамічно розподілена пам'ять). Купа зазвичай починається в кінці секції «BSS», і вона збільшується в напрямку до стека. Такі команди, як *malloc, calloc, free, realloc* тощо, використовуються для керування розподілом пам'яті в купі. Область купи є спільною для всіх спільних бібліотек і динамічно завантажуваних модулів у процесі.

При виконанні рядка «*a = (int *)malloc(sizeof(int)*z);*» (рис. 2.1) змінна *a* буде містити адресу блоку пам'яті в купі.

6. Аргументи командного рядка. Значення змінних *argv* та *argc*, переданих як параметри запуску процеса з консолі, та інші змінні середовища оточення зберігається в цій структурі пам'яті.

Впродовж існування процеса в системі його виконання може бути багато разів перерване і продовжене. Для відновлення виконання процеса необхідно відновити стан його середовища виконання. Стан середовища виконання процеса відображається станом регістрів і програмного лічильника, режимом роботи процесора, покажчиками на відкриті файли, інформацією про незавершені операції введення-виведення, кодами помилок системних викликів, що викликаються даним процесом і так далі. Ця інформація називається контекстом процеса (англ. Process Context) [2].

Крім цього, ОС для реалізації планування процесів потрібна додаткова інформація: ідентифікатор процеса, стан процеса, дані про ступінь привілею процеса, місце знаходження кодового сегменту та інша інформація. У деяких ОС (наприклад, в ОС UNIX) інформацію такого роду називають дескриптором процеса (БУП – блок управління процесом; англ. PCB – Process Control Block або TCB – Task Control Block). Дескриптор процеса в порівнянні з контекстом містить більш оперативну інформацію, яка має бути легко доступна підсистемі планування процесів. Контекст процеса містить менш актуальну інформацію і використовується ОС тільки після того, як прийнято рішення про відновлення виконання перерваного процеса.

Кожен процес в ОС має свій БУП, який створюється під час створення процеса (при запуску програми на виконання), і саме за допомогою нього представляється в системі.

Для реалізації моделі процесів в ОС підтримується системна таблиця процесів (масив структур). Кожний елемент таблиці містить БУП або покажчик на БУП (у Linux, наприклад, використовується масив покажчиків з назвою «task array» або «task list»). У дескрипторі процеса безпосередньо або через покажчики на пов'язані з процесом структури (контекст процеса) міститься інформація про стан процеса, його пріоритет, ідентифікатор, параметри планування, дані про розташування образу процеса в оперативній пам'яті і на диску, про очікувані процесом події, а також інша оперативна інформація, що необхідна ядру системи протягом всього життєвого циклу процеса, незалежно від того, чи знаходиться процес в активному або пасивному стані.

Чергами процесів є дескриптори окремих процесів, об'єднані в списки. Таким чином, кожен дескриптор, крім усього іншого, містить, принаймні, один покажчик на інший дескриптор, що є сусідом з ним у черзі. Така організація черг дозволяє легко їх перевпорядковувати, включати і виключати процеси в/з черги, переводити процеси з одного стану в інший. Всі БУП, таблиці процесів та черги процесів розташовуються в адресному просторі ядра ОС.

На рис. 2.2 наведено типовий БУП (*task_struct*) для ОС Linux [5, 7, 25]. Структура *struct task_struct* визначена у файлі <*linux/sched.h*> і містить повну інформацію про процес, який виконується (наведено фрагмент, повний розмір займає приблизно 1.7 Кбайт для 32-розрядної архітектури):

```
/* Спрощене представлення структури task_struct в ядрі Linux */
struct task_struct {
volatile long state; // Стан процеса
struct thread_info *thread_info;
struct exec_domain *exec_domain;     // Інформація про домен
                                     // виконання (не підтримується)
struct mm_struct *mm; // Інформація про керування пам'яттю
                      // (адресний простір)
struct fs_struct *fs; // Інформація про файлову систему
```

```
struct files_struct *files; // Таблиця дескрипторів файлів
struct signal_struct *signal;        // Обробники сигналів і
                                     // сигнали в очікуванні
struct sighand_struct *sighand; // Інформація про обробку сигналів
/* Різні інші поля */
};
```

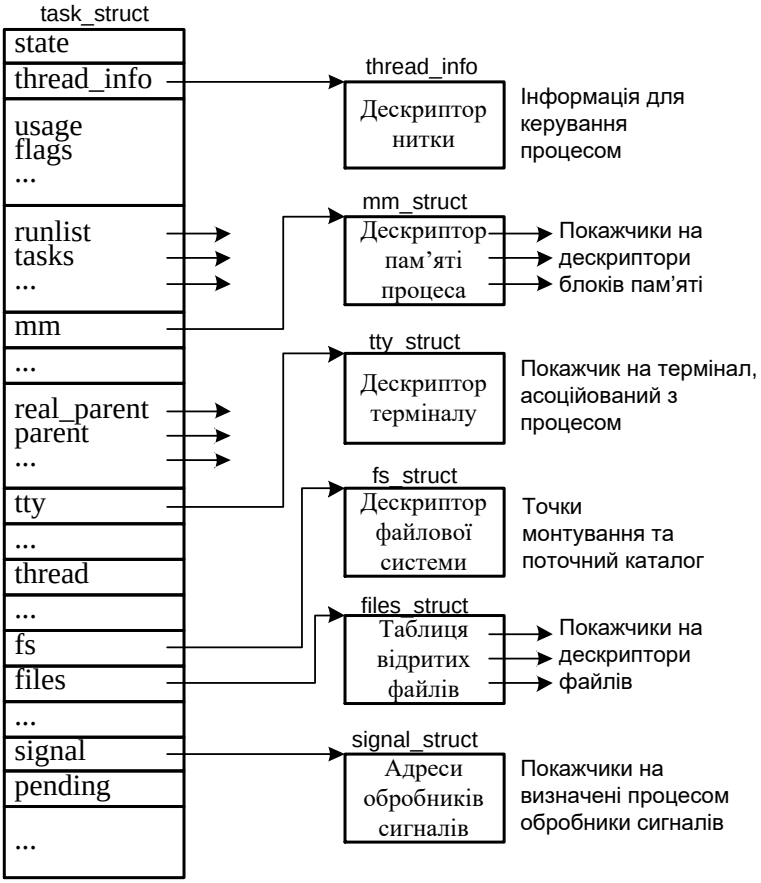


Рисунок 2.2 – Типовий БУП в ОС Linux

В ОС Windows БУП міститься в основній структурі з назвою *EPROCESS* та ще кількох пов'язаних з нею [42]. Розмір і навіть значення структури змінюються не тільки між версіями ОС, але й між пакетами оновлень однієї версії ОС Windows. Відносно до версії ОС Windows 10 БУП наведено на рис. 2.3.

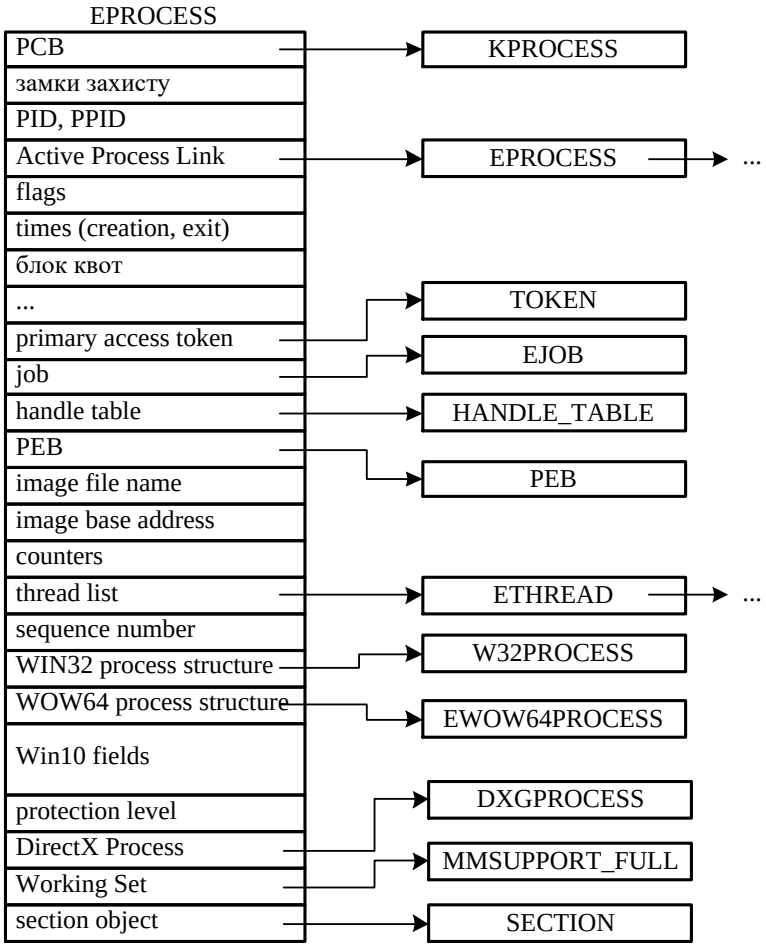


Рисунок 2.3 – Типовий БУП в ОС Windows

Підпрограми виконавчої системи зберігають інформацію в *EPROCESS*, а диспетчер, планувальник і код обліку переривань/часу, будучи частиною ядра ОС, використовують пов'язану структуру *KPROCESS*. Поле «*Active Process Link*» – покажчики для утворення двоспрямованого кільцевого списку структур *EPROCESS* процесів у системі. Структури *KPROCESS* для всіх процесів у системі також утворюють окремий список. Структура *PEB* (Process Environment Block) містить блок змінних оточення та розміщується в адресному просторі користувача.

Фрагмент опису структури *EPROCESS* (визначена в <*include/ddk/ntifs.h*>):

```
typedef struct _EPROCESS
{
    KPROCESS Pcb;
    EX_PUSH_LOCK ProcessLock;
    LARGE_INTEGER CreateTime;
    LARGE_INTEGER ExitTime;
    EX_RUNDOWN_REF RundownProtect;
    PVOID UniqueProcessId;
    LIST_ENTRY ActiveProcessLinks;
    ULONG QuotaUsage[3];
    ULONG QuotaPeak[3];
    ULONG CommitCharge;
    ULONG PeakVirtualSize;
    ULONG VirtualSize;
...
    EX_FAST_REF Token;
    PEJOB Job;
    PPEB Peb;
...
} EPROCESS;
```

2.3. Стани процеса

Будь-який процес в ОС може перебувати в одному зі станів, який визначається наявністю тих чи інших ресурсів, йому виділених [9, 13, 35, 37, 44, 45]. Залежно від кількості ресурсів у системі в кожному стані одночасно

можуть перебувати один або декілька процесів. У тому випадку, коли в деякому стані знаходяться кілька процесів, вони формуються в черги. ОС визначає, як і коли переводити процеси з одного стану в інший. Перемикання процесів між станами виконується за допомогою диспетчерів.

2.3.1. Модель процеса з двома станами

Найпростішу модель диспетчеризації, тобто – схему чергування виконання процесів, можна побудувати, використовуючи модель процеса з двома станами: активний (виконується) та пасивний (не виконується). У такій моделі не враховується можливість блокування процесів через очікування завершення операцій введення-виведення або виділення необхідних ресурсів – всі процеси вважаються як процеси чисто обчислювального характеру.

На рис. 2.4 наведено модель процеса з двома станами.

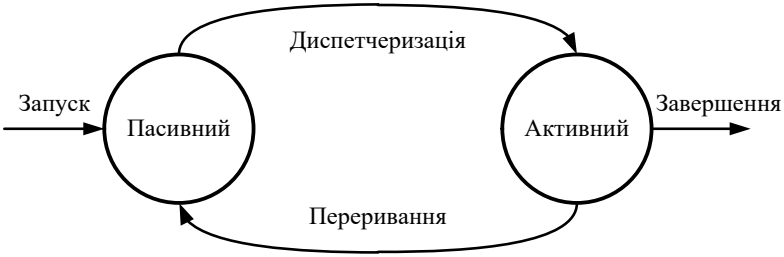


Рисунок 2.4 – Модель процеса з двома станами

Коли ОС створює новий процес, він переводиться в пасивний стан. Виконання процеса може бути перервано при необхідності виконати операцію введення-виведення або по закінченню виділеного часу на його виконання. У цьому випадку для виконання диспетчер обирає інший процес, що знаходиться в пасивному стані. Перерваний процес переводиться з активного стану в пасивний, а обраний для виконання процес, навпаки – з пасивного стану в активний.

Для однопроцесорної системи в активному стані може знаходитися лише один процес, у пасивному – декілька. Для управління множиною процесів, що знаходяться в пасивному стані, використовується черга процесів. Коли процес надходить до системи або переривається, він розміщується в хвості цієї черги. Якщо в активному стані немає жодного процеса, обирається інший процес з черги за певним алгоритмом (з голови або за іншими міркуваннями).

На рис. 2.5 наведено схему обслуговування черги процесів.

Рисунок 2.5 – Схема обслуговування черги процесів

2.3.2. Модель процеса з п’ятьма станами

У моделі процеса з двома станами не враховується очікування виділення ресурсів – всі процеси: і готові до виконання, і заблоковані, знаходяться в одній черзі. Більш адекватною моделлю є модель процеса з п’ятьма станами (рис. 2.6).

Кожний зі станів процеса характеризується такими властивостями:

♦ новий (англ. New) – процес створено: створені та заповнені керуючі структури даних у пам’яті, але сам процес може бути ще не завантажений у пам’ять та йому ще не виділені інші ресурси. У новому стані можуть одночасно перебувати кілька процесів, організованих у чергу;

♦ готовий (англ. Ready) – процес отримав всі ресурси для свого виконання крім ресурсу центрального процесора. У стані готовності можуть одночасно перебувати кілька процесів, організованих в одну або кілька черг (залежно від алгоритму диспетчеризації);

♦ активний (англ. Active) – процес виконується, йому виділено всі ресурси, у тому числі ресурс центрального процесора – процесорний час. В активному стані в однопроцесорних системах може перебувати тільки один

процес, у багатопроцесорних – декілька (кількість процесів обмежено кількістю процесорів);

♦ блокований або очікування (англ. Waiting) – процес чекає на деяку подію: завершення операції введення-виведення або виділення ресурсу. У стані очікування можуть одночасно знаходиться декілька процесів, організованих в одну загальну чергу, або в окремі черги для кожного виду подій;

♦ завершений (англ. Terminated) – процес завершив своє виконання.

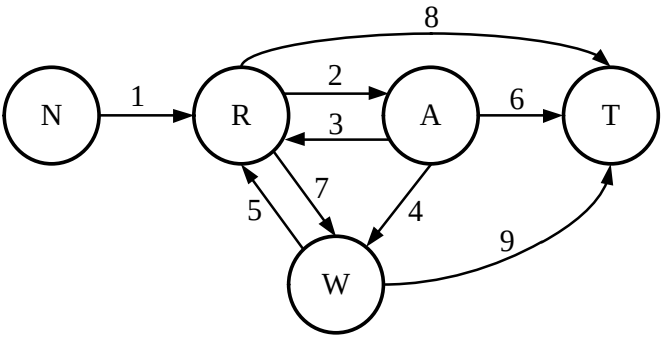


Рисунок 2.6 – Модель процеса з п’ятьма станами:
N – новий, R – готовий до виконання, A – активний,
W – блокований (стан очікування) та T – завершений

Перемикання процесора з одного процеса на інший називається перемиканням контексту (англ. Context switch), для нього характерні такі властивості:

♦ стан виконання активного процеса зберігається і стан виконання наступного процеса завантажується (перемикається контекст). Час для збереження старого і завантаження нового стану виконання процеса називається часом перемикання контексту;

♦ при перемиканні система не виконує корисну роботу, час перемикання контексту – накладні витрати, і їх необхідно зменшувати;

♦ час перемикання залежить від апаратного забезпечення.

Наприклад, для архітектури Intel x86 при перемиканні контексту виконується зберігання та завантаження наступних даних:

- ♦ контекст регістрів загального призначення,
- ♦ контекст стану сопроцесора з рухомою комою,
- ♦ стан регістрів MMX/SSE,
- ♦ стан сегментних регістрів,
- ♦ стан деяких регістрів управління (CR3 та ін.).

В ОС Linux перемикання контексту реалізоване в макросі «switch_to» (вихідний код розташовано у файлі *arch/x86/kernel/process_64.c*). Він виконує кілька дій між працею планувальника задач (процесів) і планувальника ЦП:

- ♦ перевказівка робочого простору: відновлення стека (SS:SP);
- ♦ пошук наступної інструкції: відновлення IP (CS:IP);
- ♦ відновлення стану задачі: відновлення регістрів загального призначення;
- ♦ свопінг адресних просторів пам'яті: оновлення каталогу сторінок (CR3);
- ♦ оновлення стану математичного сопроцесора FPU (англ. Floating Point Unit), структур даних ОС, регістрів налагодження, апаратних обхідних шляхів тощо.

Відновлення роботи процеса виконується за допомогою інструкції *iret* після повернення з планувальника. На рис. 2.7 наведено хід перемикання контексту в ОС Linux.

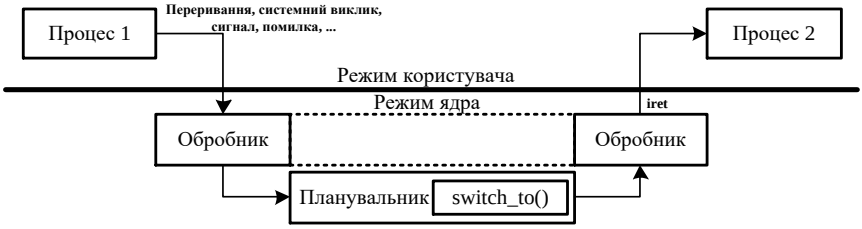


Рисунок 2.7 – Реалізація перемикання контексту в ОС Linux

У версії Linux 0.01 цей макрос викликався в останньому рядку кода планувальника (функція *schedule()*) та виглядав наступним чином:

```
/** include/linux/sched.h */
#define switch_to(n) {
struct {long a,b;} __tmp;
__asm__("cmpl %%ecx,_current\n\t"
  "je 1f\n\t"
  "xchgl %%ecx,_current\n\t"
  "movw %%dx,%1\n\t"
  "ljmp %0\n\t"
  "cmpl %%ecx,%2\n\t"
  "jne 1f\n\t"
  "clts\n"
  "1:"
  ::"m" (*&__tmp.a),
  "m" (*&__tmp.b),
  "m" (last_task_used_math),
  "d" _TSS(n),
  "c" ((long) task[n]));
}
```

Вхідний аргумент n – порядковий номер наступного процеса (від 0 до 63). Далі резервується 8 байтів у стеку (*struct {long a,b;} __tmp;*). Сама реалізація перемикання реалізована на асемблері: спочатку перевіряється, чи є процес, на який потрібно перемикнутись, поточним, якщо так – перемикати контекст не потрібно. Якщо ні – виконується оновлення глобального контексту для нового процеса (*xchgl %%ecx,_current*). Далі, у простір, зарезервований на початку, за допомогою макроса *_TSS* отримується покажчик сегмента на наступний процес. Інструкція *ljmp %0* викликає апаратне перемикання контексту.

У сучасних версіях Linux розмір макроса «*switch_to*» збільшився, розділився на 2 частини (асемблерна частина «*prepare_switch_to*» та частина мовою програмування С) , та в ньому виконується такі головні дії:

♦ отримує необхідні дані з БУП (*task_struct*);

♦ отримує номер процесора, потрібний для управління глобальною таблицею дескрипторів GDT (англ. Global Descriptor Table) для даних

сегменту стану задачі TSS (англ. Task State Segment), локального сховища ниток і стану FPU;

- ♦ зберігає поточний стан FPU;
- ♦ зберігає значення регістрів FS і GS;
- ♦ перевантажує GDT для локального сховища ниток нового процеса;
- ♦ зберігає значення регістрів ES та DS і завантажує нові значення;
- ♦ завантажує нові значення в регістри FS і GS;
- ♦ виконується ініціалізація FPU для нового процеса;
- ♦ виконується оновлення БУП (*task_struct*) для поточного процеса. Оновлюються FPU і частина сегмента стека користувацького процеса – область даних процеса PDA (англ. Per-process Data Area), в якій зберігається інформація про поточний каталог, відкриті процесом файли та ін.;
- ♦ оновлюється покажчик на вершину стека ЦП (перевантажується елемент апаратного стека процесора sp1) та перевіряється коректність нового стека;
- ♦ оновлюються регістри налагодження і бітові карти введення-виведення;
- ♦ міняються місцями біти привілеїв I/O для паравіртуалізації гіпервізора Xen;
- ♦ оновлюються дескриптори сегментів у регістрах ES та DS;
- ♦ виконуються задачі по очищенню ресурсів: для архітектури Intel оновлюються засоби моніторингу та контролю RMID (Resource Monitoring ID) і CLOSid.

Як можна побачити – при виконанні перемикання контексту виконується багато дій: програмних та апаратних, що призводить до витрат часу. Час, витрачений на перемикання контексту, не використовується обчислювальною системою для здійснення корисної роботи і є накладними витратами, що знижують продуктивність системи. Він залежить від конкретної апаратної платформи і зазвичай коливається в діапазоні від 1 до 1000 мікросекунд.

Перехід процеса з одного стану до іншого називається перемиканням процесів (англ. Process switch). Відповідно до рис. 2.6 можливі наступні переходи між станами процеса:

1. N→R (New→Ready). Перехід ініціюється планувальником завдань по деякій наперед визначеній політиці. На цьому етапі перевіряється наявність достатньої кількості оперативної пам'яті і інших первинних та вторинних ресурсів;

2. R→A (Ready→Active). Перехід ініціюється планувальником процесорного часу, за деяким визначеним алгоритмом. Прийняте планувальником рішення реалізується диспетчером процесорного часу;

3. A→R (Active→Ready). Перехід обробляється планувальником та диспетчером процесорного часу згідно з наперед заданим обмеженням по часу, або іншим критеріям, наприклад, перериванню по пріоритету;

4. A→W (Active→Waiting). Перехід виконується, коли для продовження роботи процеса потрібне настання деякої події (очікування завершення обробки запитів на введення-виведення, повідомлень від інших процесів, виділення додаткових ресурсів, звернення до відсутніх у пам'яті даних тощо);

5. W→R (Waiting→Ready). Перехід оброблюється менеджером ресурсів, та ініціюється настанням події, що очікувалась процесом;

6. A→T (Active→Terminated). Перехід оброблюється планувальником та диспетчером процесорного часу разом з планувальником завдань, коли процес успішно завершено або ОС сигналізує про те, що сталася помилка і процес припиняється достроково;

7. R→W (Ready→Waiting). Перехід виконується за ініціативою ОС – у разі нестачі ресурсів для більш пріоритетних процесів, ресурси можуть бути відібрані в менш пріоритетного процеса, а сам такий процес буде очікувати повторного виділення цих ресурсів;

8. R→T (Ready→Terminated). Перехід також виконується за ініціативою ОС – процес примусово завершується, наприклад, при нестачі ресурсів або в тупиковій ситуації ОС завершує один або декілька процесів для стабілізації ситуації. Також по завершенню батьківського процеса, дочірній процес може бути припиненим;

9. W→T (Waiting→Terminated). Умови виконання переходу такі самі, як і для попереднього випадку.

Прямий перехід W→A (Waiting→Active) – неможливий: для того, щоб процес міг отримати процесорний час, він спочатку повинен отримати необхідні ресурси та потрапити в чергу готових процесів, та тільки тоді бути обраним для виконання за певним алгоритмом планування.

При перемиканні процесів операційною системою реалізуються задачі планування. Існує довгострокове, середньострокове та короткострокове планування.

Планування завдань використовується в якості довгострокового планування процесів (англ. Long-term scheduling). Воно відповідає за породження нових процесів у системі, визначаючи її ступінь мультипрограмування. Якщо ступінь мультипрограмування системи підтримується на постійному рівні, тобто середня кількість процесів у комп'ютері не змінюється, то нові процеси можуть бути запущені на виконання тільки після завершення раніше завантажених. Тому довгострокове планування здійснюється досить рідко, між появою нових процесів можуть проходити хвилини і навіть десятки хвилин. Рішення про вибір для запуску того чи іншого процеса впливає на функціонування обчислювальної системи протягом достатньо тривалого часу. У деяких ОС довгострокове планування зведено до мінімуму або відсутнє зовсім. Планування завдань також називається плануванням на верхньому рівні.

У деяких обчислювальних системах буває вигідно для підвищення продуктивності тимчасово видалити будь-який частково виконаний процес з оперативної пам'яті в зовнішню пам'ять, а пізніше повернути його назад для подальшого виконання. Коли і який саме з процесів треба вивантажити і повернути назад, вирішується додатковим проміжним рівнем планування процесів – середньостроковим (англ. Medium-term scheduling). Планувальник цього рівня визначає, яким процесам буде дозволено конкурувати за захоплення ЦП, тобто які процеси буде призупинено, а які пробуджено для забезпечення рівномірного завантаження системи. Цей вид планування також називається плануванням на проміжному рівні. На цьому рівні працюють менеджери різноманітних ресурсів, які займаються

плануванням, виділенням та обліком ресурсів обчислювальної системи. Тому цей вид планування також має назву – планування ресурсів.

Планування використання процесора застосовується як короткострокове планування процесів (англ. Short-term scheduling), метою якого є вибір нового процеса на виконання. Це планування проводиться, наприклад, при зверненні активного процеса до пристроїв введення-виведення або просто по завершенню певного інтервалу часу. Короткострокове планування здійснюється, як правило, не рідше одного разу на 100 мілісекунд. Воно також називається плануванням на нижньому рівні.

2.3.3. Модель процеса з сьома станами

Узагальненням моделі з п’ятьма станами є модель з сьома станами, яка дозволяє більш адекватно змоделювати реалізацію процесів в ОС. У даній моделі вводиться додатково поняття призупиненого процеса і два нових стани: блокований/призупинений (англ. BS – Blocked/Suspended) – процес вивантажено в зовнішню пам’ять, але він очікує певну подію; готовий/призупинений (англ. RS – Ready/Suspended) – процес знаходиться в зовнішній пам’яті, але вже готовий до виконання, для цього його необхідно тільки завантажити в основну пам’ять.

Призупинений процес є процесом, який відсутній в основній пам’яті і не може бути запущений негайно. Причинами призупинки процеса можуть бути:

♦ необхідність звільнити основну пам’ять для оптимізації продуктивності віртуальної пам’яті;

♦ періодичний режим виконання процеса (наприклад, програма аналізу використання ресурсів комп’ютера);

♦ припинення дочірніх процесів батьківським для їх перевірки або координації;

♦ припинення призначеного для користувача процеса для налагоджування програми;

♦ припинення фонових процесів операційною системою в разі виявлення проблем.

На рис. 2.8 наведено діаграму станів процеса з сьома станами.

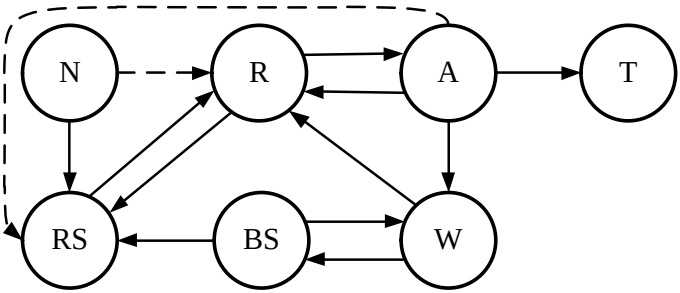


Рисунок 2.8 – Модель процеса з сьома станами

Переходи між доданими в модель станами (порівняно до моделі з п'ятьма станами) наступні:

1. W→BS (Waiting→Blocked/Suspended). Якщо жоден процес не готовий до виконання, то блокований процес вивантажується з пам'яті, щоб звільнити місце для іншого процеса, який не є блокованим. Цей перехід виконується також у випадку, коли ОС визначить, що для деякого процеса, управління до якого перейде в найближчим часом, потрібно збільшити обсяг основної пам'яті для забезпечення адекватної продуктивності;

2. BS→RS (Blocked/Suspended→Ready/Suspended). Процес у стані блокованого/призупиненого переходить у стан готового/призупиненого процеса, якщо відбувається подія, якої чекав цей процес;

3. RS→R (Ready/Suspended→Ready). Коли в основній пам'яті немає готових до виконання процесів, ОС для продовження обчислень потрібно завантажити процес у пам'ять. Також цей перехід виконується, якщо в готового/призупиненого процеса виявиться більш високий пріоритет, ніж в інших готових до виконання процесів;

4. R→RS (Ready→Ready/Suspended). Перехід виконується при необхідності звільнити досить великий блок основної пам'яті. ОС може також замість блокованого процеса з більш високим пріоритетом призупинити готовий до виконання процес з більш низьким пріоритетом, якщо блокований процес досить скоро буде готовий до виконання;

5. N→RS (New→Ready/Suspended). Перехід виконується аналогічно переходу N→R (New→Ready), але процес не розміщується в основній пам'яті. У моделі з сьома станами новий процес може бути доданий замість черги готових/призупинених процесів одразу в чергу готових до виконання процесів – при цьому він одразу завантажується в основну пам'ять (лінія переходу на рис. 2.8 зображена пунктиром);

6. BS→W (Blocked/Suspended→Waiting). Перехід виконується в такій ситуації: у черзі блокованих/призупинених процесів є процес, пріоритет якого вище, ніж у будь-якого процеса з черги готових/призупинених процесів та ОС планує в найближчому часі зняти блокування з цього більш пріоритетного процеса;

7. A→RS (Active→Ready/Suspended). При наявності процеса з вищим пріоритетом, який перебував у черзі заблокованих/призупинених процесів і щойно був розблокований, ОС може віддати перевагу саме йому. Щоб звільнити частину основної пам'яті, вона може перевести активний процес безпосередньо в стан готового/призупиненого процеса (на рис. 2.8 лінія переходу зображена пунктиром).

Також у даній моделі з довільного стану може бути виконаний перехід у завершений стан (Terminated), причини переходу повністю відповідають причинам для моделі з п'ятьма станами (на рис. 2.8 такі переходи не зображено).

На рис. 2.9 наведено уточнену схему обслуговування черг процесів для моделі процеса з сьома станами. Черги процесів формуються для кожного зі станів (New, Ready, Waiting, Blocked/Suspended та Ready/Suspended). Залежно від реалізованих алгоритмів планування в кожному зі станів процеси можуть розташовуватись в одній черзі або кількох взаємопов'язаних чергах. Зафарбованими прямокутниками на рис. 2.9 відокремлено області дій планувальників. Планувальник завдань відповідає за черговість надходження нових процесів у систему: пакетні процеси обслуговуються за певним алгоритмом, інтерактивні завантажуються в систему поза чергою. Планувальник ресурсів відповідає за черговість виділення ресурсів: коли при виконанні процес потребує додаткові ресурси або чекає на якусь подію, він переводиться в стан очікування, де потрапляє

в одну з чисельних черг (або одразу в кілька черг, якщо потребує кілька різних ресурсів) відповідно до потрібних ресурсів. Планувальник процесора відповідає за черговість отримання готовими до виконання процесами часток процесорного часу. Перехід з активного стану в чергу готових процесів на рис. 2.9 вказано по перериванню від таймера, як найчастіше, але це не одна причина (детальніше можливі причини описані в розділі 3).

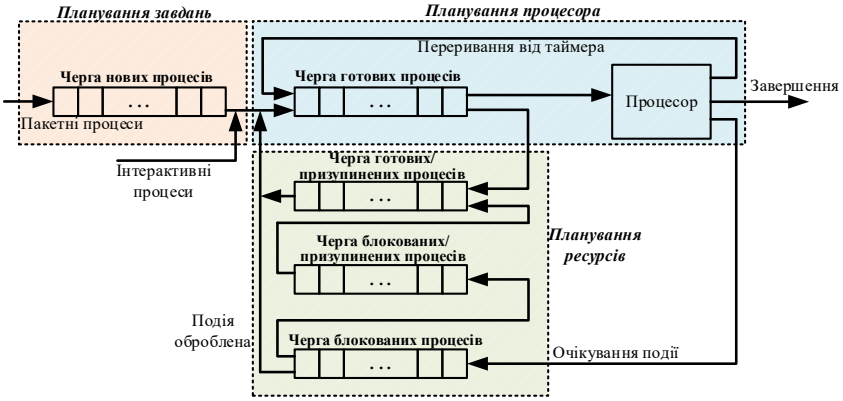


Рисунок 2.9 – Уточнена схема обслуговування черг процесів

2.4. Управління процесами

Управління процесами здійснюється ОС у режимі ядра і включає наступні функції [2, 3, 12, 15, 27, 31]:

- ♦ створення і завершення процесів;
- ♦ планування і диспетчеризація процесів;
- ♦ перемикання процесів;
- ♦ синхронізація і обмін інформацією між процесами;
- ♦ організація блоків управління процесами БУП.

2.4.1. Створення і завершення процесів

Процеси створюється за наступними основними подіями:

- ♦ ініціалізація системи;

♦ виконання системного виклику створення процеса від існуючого процеса;

♦ запит користувача на створення процеса;

♦ початок роботи пакетного завдання.

При старті ОС зазвичай створюється декілька процесів. Деякі з цих процесів – інтерактивні, які потребують обміну даними з користувачем, інші – фонові, які не взаємодіють з користувачами і виконують певні спеціальні функції. Під час роботи ОС може також створювати процеси для забезпечення поточних потреб у фоновому режимі (наприклад, створити процес управління друком). Процеси під час своєї роботи можуть ініціювати створення нових процесів, наприклад, для розгалуження обчислень.

Для створення нового процеса в ОС необхідно виконати певну послідовність дій:

♦ привласнити новому процесу унікальний ідентифікатор, тобто занести новий запис у таблицю процесів;

♦ виділити адресний простір для всіх елементів образу процеса;

♦ ініціалізувати БУП (інформація про стан процесора зазвичай ініціалізується нульовими значеннями, за винятком лічильника команд (що містить точку входу в програму) і покажчиків системного стека (які задають межі стека процеса);

♦ розташувати процес у списку готових або готових/призупинених процесів;

♦ завантажити частку кодів і даних процеса в оперативну пам’ять (ОС повинна виявити місце розташування програми на диску, перерозподілити оперативну пам’ять і виділити пам’ять виконуваній програмі нового процеса. У системах з віртуальною пам’яттю в початковий момент може завантажуватися тільки частка кодів і даних процеса. При виконанні всіх цих дій підсистема управління процесами тісно взаємодіє з підсистемою управління пам’яттю і файловою системою).

Так, при створенні процеса ОС Linux виконує такі основні дії (більш детальний опис дій при створенні процеса наведено в пункті 2.4.3):

1. Виділяється місце в таблиці процесів для нового процеса.

2. Генерується для створюваного процеса унікальний ідентифікатор.

3. Виділяється пам’ять для образу процеса – копія батьківського процеса, за винятком спільної пам’яті.

4. Збільшується значення лічильників файлів, відкритих у батьківському процесі (створюваний процес наслідує відкриті файли).

5. Створюваному процесу призначається стан готовності (Ready).

6. Повертається ідентифікатор створеного процеса батьківському процесу та значення 0 створеному процесу.

Всі дії виконуються в режимі ядра в батьківському процесі. Після цього виконується передача управління за одним з варіантів:

♦ батьківському процесу – виконується повернення в режим користувача в точку кода програми після переривання;

♦ створеному дочірньому процесу – дочірній процес починає виконуватися в тому ж місці коду, яка йде після переривання батьківського процеса;

♦ іншому процесу – батьківський і створений дочірній процеси залишаються в стані готовності.

В ОС Windows створення процеса виконується за такими кроками:

1. Перевіряються та конвертуються параметри та прапорці підсистеми Windows у необхідний формат.

2. Відкривається файл з програмою, яку потрібно виконати всередині процеса.

3. Створюється об’єкт процеса Windows.

4. Створюється основна нитка (стек, контекст і об’єкт нитки Windows).

5. Виконується ініціалізація процеса.

6. Основна нитка процеса переводиться в стан готовності (якщо не вказано прапорець CREATE_SUSPENDED – тоді переводиться в стан очікування).

7. У контексті нового процеса та нитки завершується ініціалізація адресного простору (наприклад, завантажуються необхідні DLL). Виконання процеса після отримання процесорного часу починається з точки входу програми.

Процес стає завершеним, коли він виконує останню інструкцію свого коду. Ресурси, які були йому виділені, повертаються системі, інформація про процес видаляється з системних таблиць, БУП знищується, пам'ять, що була для нього виділена, помічається як вільна. Слід зазначити, що процес може бути завершений нормально, за помилкою або знищений іншим процесом.

Процеси завершуються за такими основними причинами:

♦ нормальне завершення – процес виконує системний виклик, щоб повідомити ОС про своє завершення;

♦ перевищення граничного ліміту часу – процес виконується довше, ніж встановлено обмеження за часом;

♦ перевищення ліміту відведеної пам'яті – процес потребує пам'яті більше ніж система може виділити;

♦ арифметичні помилки – процес намагається виконати неприпустиме обчислення, наприклад, виконує ділення на нуль або намагається оперувати числами більшими ніж дозволяє апаратура;

♦ тайм-аут – процес чекає певну подію більше визначеного в системі максимального часу очікування;

♦ критична помилка введення-виведення – ситуації неможливості знайти файл, нездатність читати і писати після того, як використано максимальну кількість спроб (наприклад, по причині дефектних областей), або неприпустимі операції (наприклад, читання з пристрою друку);

♦ частина даних має невірний тип або не ініційована;

♦ процес намагається використовувати інструкції, які зарезервовані для ОС;

♦ порушення захисту – процес намагається використовувати ресурс або файл, на який не має прав, або намагається використовувати його неналежним чином, наприклад, спроба записати у файл, доступний тільки для читання;

♦ помилкова інструкція – процес намагається виконати неіснуючу команду (часто в результаті переходу в область даних і спроби виконати дані);

♦ вихід за межі адресного простору – процес намагається звернутися до пам'яті, до якої немає доступу;

♦ втручання користувача, адміністратора або ОС – може бути з різних причин, наприклад, для розв'язання тупикової ситуації;

♦ завершення батьківського процеса – в ОС з організацією процесів за схемою «батько-нащадок» завершення батьківського процеса може приводити до завершення всіх його дочірніх процесів.

Замість завершення в деяких ситуаціях процес може бути призупинений. Після призупинки процес може продовжити своє виконання з міста зупинки і без втрати результатів роботи. Зупинка виконується в таких випадках:

♦ вивантаження в зовнішню пам'ять – ОС звільняє пам'ять для іншого процеса, готового для виконання;

♦ ОС може призупинити фоновий процес або процес, який є причиною проблем для збільшення ефективності роботи системи в цілому;

♦ запит користувача – користувач може призупинити процес з метою його налагодження або з метою перерозподілу ресурсів;

♦ синхронізація – процес може виконуватися в певні проміжки часу (наприклад, процес моніторингу системи) і призупиняється в очікуванні часу наступної активації;

♦ запит батьківського процеса – батьківський процес може призупинити виконання нащадка для перевірки або модифікації його в стані призупинення, а також для координації виконання різних нащадків.

2.4.2. Механізми переривання виконання процесів

Для того, щоб виконати переривання виконання процесів, в ОС повинні бути реалізовані певні механізми. Основними механізмами є: переривання (англ. Interrupt), пастка (англ. Trap) та виклик супервізора (звернення до ОС; англ. SVC – Supervisor call).

Механізм переривань є зовнішнім до виконання поточної інструкції, він використовується як реакція на асинхронну зовнішню подію. Переривання – це подія, яка змінює нормальний хід виконання процеса і може бути згенерована апаратними пристроями або процесором. У разі переривання

поточний процес зупиняється і запускається обробник переривання. Переривання бувають синхронні – викликані поточним контекстом виконання та асинхронні – викликані ззовні.

Переривання також можуть бути:

♦ масковані – запити на переривання IRQ (англ. Interrupt Request) можуть бути «замасковані» і не викликати переривання виконання процесів;

♦ немасковані NMI (англ. Non Maskable Interrupt) – завжди повинні бути оброблені, існує лише невелика кількість таких переривань.

Під час переривання ЦП зупиняє поточний процес і передає керування спеціальній процедурі – обробнику переривань ISR (англ. Interrupt Service Routine). Обробник переривань обробляє переривання та передає керування назад до раніше зупиненого процеса. Можна розділити переривання на три типи:

♦ програмні переривання – коли програмне забезпечення сигналізує ЦП, що потрібно звернутися до ядра ОС. Ці переривання зазвичай використовуються для системних викликів;

♦ апаратні переривання – коли відбувається апаратна подія, наприклад, натискання кнопки на клавіатурі;

♦ винятки (англ. Exception) – це програмно генеровані переривання, зазвичай викликані виконанням команди програми.

Приклади винятків:

♦ помилки з рухомою комою, такі як ділення на нуль;

♦ спроби виконати неприпустимі коди операцій;

♦ спроба отримати доступ до комірок пам’яті за межами дозволеної пам’яті процеса.

Винятки поділяються на такі типи:

♦ помилки (англ. Faults) – винятки, після закінчення обробки яких перервана команда повторюється;

♦ аварії (англ. Aborts) – винятки, при обробці яких ЦП не зберігає стан та не має можливості повернутися до місця, де відбувся виняток. Такі винятки перешкоджають продовженню виконання пошкодженого коду: їх

виникнення вказує на серйозну проблему, яка не дозволяє продовжувати виконувати код.

Кожне переривання та виняток в архітектурі Intel 80x86 ідентифікується 8-бітним унікальним беззнаковим числом – вектором переривання. Номер вектора може бути будь-яким числом від 0 до 255. Існує звичайна практика використовувати векторні номери 0-31 для винятків, а номери від 32 до 255 для переривань користувачів. ЦП використовує номер вектора як індекс у таблиці векторів переривань.

Термін «пастка» часто використовується як синонім винятків, але це не так. Пастки – це особливий тип винятків. Пастки – винятки, при обробці яких ЦП зберігає стан, що йде за командою, що викликала виняток, тобто після повернення з пастки процес продовжить своє виконання з наступної команди. Архітектура Intel 80x86 визначає пастку як ініційовану програмістом (і, отже, очікувану) передачу управління програмі-обробнику. Можна сказати, що пастки – справжні інструкції, навмисно закодовані в програмі, які викликають переривання, тоді як винятки – будь-які переривання, що генеруються програмним забезпеченням. До механізму пасток відносяться системні виклики.

Виклик супервізора завжди виконується явним чином (на відміну від попередніх механізмів), він є викликом до функції ОС. Виконується інструкція, яка перериває програму і передає управління диспетчеру для виконання операції, зазначеної в інструкції.

У мейнфреймах IBM, інструкція виклику супервізора (SVC) – це інструкція процесора, яка вказує передати управління комп'ютером програмі супервізора ОС. Більшість SVC – це запити до певної служби ОС від прикладної програми або іншої частини ОС. Розробники прикладних програм зазвичай використовують конструкцію мови програмування, щоб зробити запит (наприклад, щоб отримати більше пам'яті для роботи програми). Компілятор генерує інструкцію, яка містить конкретний запит SVC. Кожна служба має попередньо призначений номер SVC. Коли процесор комп'ютера виконує інструкцію, яка містить SVC, код викликає переривання програми і керування негайно передається програмі-

супервізору ОС. Потім супервізор передає управління процедурі SVC, яка виконує дії, які відповідають вказаному номеру SVC.

2.4.3. Механізм системних викликів

Відповідно до функції абстрагування, процеси не можуть виконувати операції з первинними ресурсами обчислювальної системи безпосередньо (у режимі користувача), тому для виконання, наприклад, обміну даними з пристроєм введення-виведення необхідно звернутися до ОС. Більшість подій в ОС виконується з використанням механізму переривань. При цьому управління передається від процеса (режим користувача) к відповідній процедурі обробки переривань ОС (режим ядра).

Механізм обробки переривання виконання процеса складається з таких етапів:

1) встановлення факту переривання (прийняття та ідентифікація сигналу на переривання);

2) зберігання контексту процеса, який переривається;

3) управління апаратно передається програмі обробки переривання;

4) обробка переривання;

5) відновлення інформації, що відноситься до перерваного процеса;

6) повернення в перерваний процес.

Системний виклик (англ. System call) дозволяє процесу звернутися до ОС для виконання деяких дій у привілейованому режимі. Для прикладного програміста системний виклик зовнішньо не відрізняється від виклику звичайної функції. Набір функцій, за допомогою якого можна звертатися до системних викликів, носить назву «інтерфейс прикладного програмування (API)». Для всіх системних викликів, звернення до яких виконується через функції API, формуються заглушки (англ. Stub) – кілька асемблерних рядків, необхідних для виконання переривання. Слід зазначити, що не всі функції API обов'язково звертаються до системних викликів – якщо немає потреби для перемикання в режим ядра, функція API повністю виконує всю роботу.

На системні виклики накладаються такі вимоги:

- ♦ перемикання в режим ядра;
- ♦ висока швидкість виклику функцій ОС;

♦ однакове звернення до системних викликів для всіх апаратних платформ, на яких працює ОС;

♦ можливість розширення набору системних викликів;

♦ контроль зі сторони ОС за коректним використанням системних викликів.

Дії для реалізації системних викликів обов'язково містять обробку переривань, тому в більшості ОС перші секції програм обробки переривань для обслуговування механізму системних викликів виносяться в окремий спеціальний системний модуль – супервізор переривань або диспетчер системних викликів SCH (англ. System Call Handler). Така реалізація (централізована схема обробки) використовує тільки один номер переривання, використання ж для кожного системного виклику свого вектору переривання (децентралізована схема обробки) обмежує кількість системних викликів кількістю можливих векторів переривань в обчислювальній системі (для Intel 80x86 – 256).

У будь-якій ОС системний виклик виконується деякою процесорною інструкцією, яка перериває послідовне виконання команд і передає управління коду диспетчера системних викликів (у Linux це функція з іменем *system_call*, у Windows або *KiSystemService* – у разі переривання, або *KiFastCallEntry* – у разі спеціальної інструкції процесора, адреса цієї функції для забезпечення швидкого виклику занесена в спеціальний регістр MSR (Model-Specific Register)). Це зазвичай деяка команда програмного переривання, залежно від архітектури процесора: *svc*, *emt*, *trap*, *int*, або спеціальна інструкція: *sysenter*, *syscall* тощо. Для архітектури Intel 80x86 команда програмного переривання з різним вектором традиційно закріплена для конкретної ОС: наприклад, MS-DOS – 21h, Windows – 2Eh, Linux – 80h, QNX – 21h, Minix 3 – 21h. Для повернення з процедури системного виклику, який був ініційований командою переривання, використовується команда *iret*. У сучасних системах підтримка системних викликів за допомогою механізму переривань використовується в основному для сумісності з попередніми версіями.

При передачі параметрів диспетчеру системних викликів можуть використовуватися різні механізми:

♦ через регістри (найшвидший механізм, але є жорстке обмеження по кількості переданої інформації);

♦ через область пам'яті, адреса якої заноситься в регістр (максимально можлива кількість параметрів);

♦ через стек процеса (не задіяний додатковий регістр, немає проблем з пам'яттю, оскільки параметри будуть одразу поміщені в доступну для процеса пам'ять).

На рис. 2.10 наведено схему реалізації системних викликів через механізм переривань.

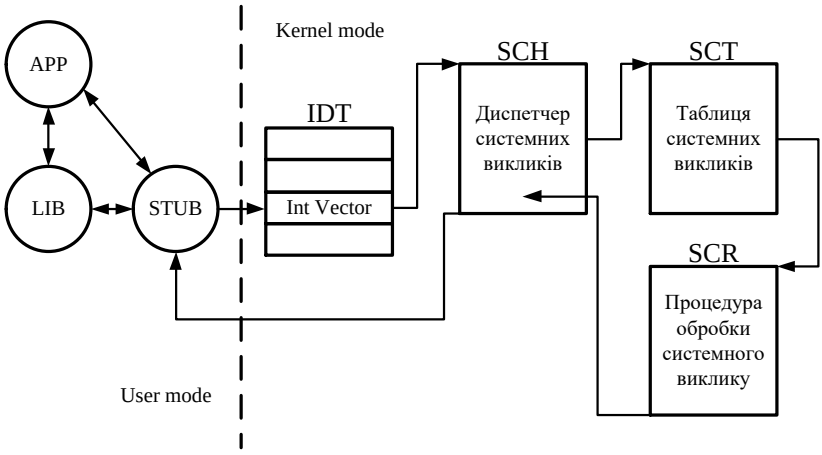


Рисунок 2.10 – Реалізація системних викликів

Коли процес (APP) намагається виконати дії з ресурсами ОС, він звертається до відповідної функції в бібліотеці LIB (яка в свою чергу формує системний виклик), або безпосередньо звертається до системного виклику. У режимі користувача виконується частина коду (Stub), яка підготовлює дані для передачі системному виклику та генерує необхідне переривання. У результаті переривання виконується перемикання в режим ядра ОС. У таблиці переривань (англ. IDT – Interrupt Descriptor Table) у відповідному векторі переривання (англ. Interrupt vector) знаходиться адреса диспетчера системних викликів (SCH), за якою виконується перехід. У диспетчері

системних викликів за таблицею системних викликів SCT (англ. System Call Table) визначається, який саме системний виклик відбувся і управління передається на відповідну процедуру для обробки системного виклику SCR (англ. System Call Routine). Після завершення обробки системного виклику диспетчер системних викликів підготовлює результат виконання для передачі процесу і виконує повернення в режим користувача.

На рис. 2.11 наведено приклад традиційної обробки через механізм переривань системного виклику створення процеса при виклику функції *fork()* для ОС Linux. При виклику переривання 80h параметром передається номер системного виклику. Номери всіх системних викликів знаходяться у вихідному файлі *arch/x86/entry/syscalls/syscall_32.tbl*, що має наступний вигляд (наведено приклад для архітектури Intel 80x86, для 64-розрядної архітектури номери, закріплені за системними викликами, та кількість системних викликів відрізняються та містяться у файлі *arch/x86/entry/syscalls/syscall_64.tbl*):

```
#
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly
# for sys_*() system calls and compat_sys_*() compat system calls
# if IA32_EMULATION is defined, and expect struct pt_regs *regs as
# their only parameter.
#
# The abi is always "i386" for this file.
#
0      i386    restart_syscall       sys_restart_syscall
1      i386    exit                  sys_exit
2      i386    fork                  sys_fork
3      i386    read                  sys_read
4      i386    write                 sys_write
5      i386    open                  sys_open       compat_sys_open
6      i386    close                 sys_close
...
```

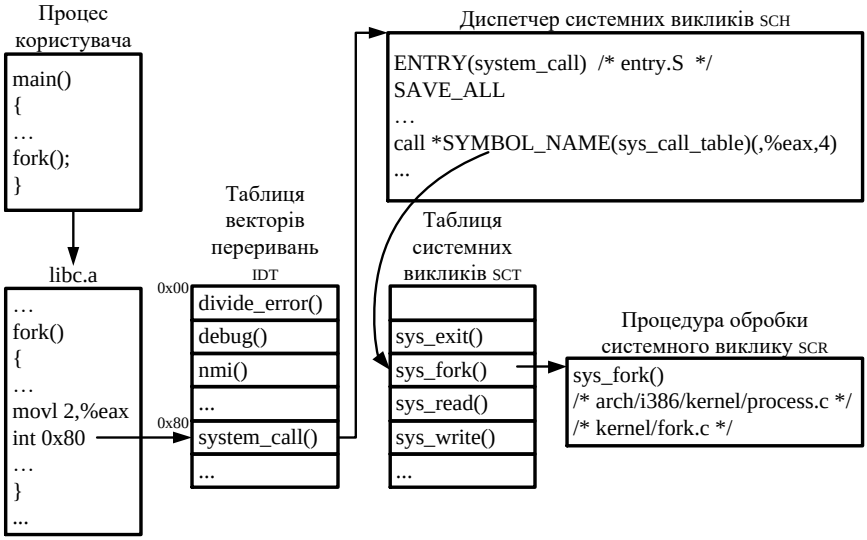


Рисунок 2.11 – Традиційна обробка системного виклику *fork*() у Linux

Диспетчер системних викликів SCH (system_call) до версії ядра 2.6.24 для архітектури Intel 80x86 знаходиться в модулі *arch/i386/kernel/entry.S*, у секції *.data* якого знаходиться таблиця системних викликів:

```
.data
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
        .long SYMBOL_NAME(sys_write)
        .long SYMBOL_NAME(sys_open)
        .long SYMBOL_NAME(sys_mincore)
        .long SYMBOL_NAME(sys_madvise)
...
```

На рис. 2.12 наведено приклад обробки системного виклику через команду *syscall* для 64-розрядної архітектури Intel сучасних версій ОС Linux. Організація викликів через команду *syscall* реалізована за допомогою

бінарного програмного інтерфейсу ABI (англ. Application Binary Interface). ABI визначає спосіб доступу до підпрограми в машинному коді (залежно від апаратного забезпечення), тоді як API визначає подібний доступ у вихідному коді (незалежно від апаратного забезпечення). Якби системні виклики Linux були реалізовані за допомогою стандартного API мовою C, кожна програма повинна була б викликати їх, як функції C. ABI усуває це обмеження, запитуючи компілятор або інтерпретатор згенерувати машинний код (тобто ініціалізувати регістри). Бінарний інтерфейс ABI System V є еталонною специфікацією, яка використовується основними UNIX-подібними ОС: номер системного виклику передається через регістр *rax*; аргументи передаються через регістри, кількість аргументів обмежена шістьма; при поверненні із *syscall*, регістр *rax* містить результат системного виклику або від'ємне значення з кодом помилки.

На рис. 2.12, як і в попередньому прикладі, викликається функція *fork()*, яка в свою чергу призводить до системного виклику. При виклику інструкції *syscall* параметри не передаються, необхідно в регістр *rax* занести лише номер виклику, для виклику *fork* для 64-розрядної архітектури його номер – 57 або константа *__NR_fork*, яка визначена у файлі *tools/arch/x86/include/uapi/asm/unistd_64.h*:

```
...
#ifndef __NR_fork
#define __NR_fork 57
#endif
...
```

Номери, закріплені за системними викликами, та кількість системних викликів містяться у файлі *arch/x86/entry/syscalls/syscall_64.tbl* :

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
```

```
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system
# calls
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
...
57      common  fork                    sys_fork
...
```

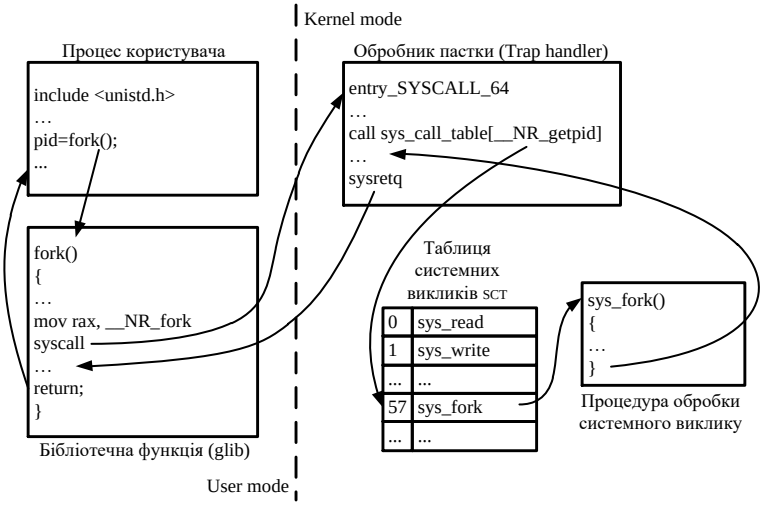


Рисунок 2.12 – Сучасна обробка системного виклику *fork*() у Linux

Відмінності в коді генерування системного виклику створення процеса на основі механізму переривань та спеціальної інструкції:

```
; fork (x86)
mov eax, 2
int 0x80
```

```
; fork (x86-64)
mov rax, 39
syscall
```

З точки зору реалізації:

♦ *int* 0x80 – програмне переривання. Ідея полягає в тому, щоб використовувати такий саме метод для переходу в режим ядра, який використовують апаратні переривання;

♦ *syscall* (і *sysenter*) – спеціальні інструкції процесора, розроблені для реалізації системних викликів;

♦ *syscall* виконує менше операцій (*syscall* не генерує програмне переривання) і, за результатами тестування, використання *syscall* приблизно в три рази швидше ніж використання *int* 0x80.

Взагалі, традиційне ядро Linux підтримує наступні можливості використання системних викликів:

1) інструкція *int* 0x80;

2) інструкція *syscall*

та нерозглянуті вище:

3) інструкція *sysenter* – спеціальна інструкція системного виклику, особливістю якої є те, що на апаратному рівні при її виконанні не робляться перевірки на дійсність дескрипторів, а також перевірки, які залежать від рівня привілеїв. Також інструкція спирається на те, що процес, який її виконав, використовує пласку модель пам'яті. Розробники ядра Linux не рекомендують використовувати *sysenter,* оскільки ABI для системних викликів може змінитися;

4) інструкція *vsyscall* – для покращення продуктивності системних викликів, вирішено їх обробляти в просторі користувача. Для цього було зарезервовано 8 Мбайтів пам'яті для відображення простору ядра в простір користувача, в який було розміщено кілька реалізацій часто виконуваних системних викликів у режимі «read-only». Це дозволило при виконанні викликів не робити перехід у режим ядра. Але фіксоване розміщення в адресному просторі є проблемою з точки зору безпеки, а фіксований розмір може негативно вплинути на розширення простору ядра;

5) механізм vDSO (Virtual Dynamic Shared Object) – реалізація системних викликів у вигляді відображення бібліотеки, яка динамічно підключається, до якої застосовано технологію рандомізації розміщення адресного простору ASLR (англ. Address Space Layout Randomization).

Бібліотека автоматично завантажується для кожного процеса та містить так само як і пам'ять для *vsyscall* реалізації кількох часто виконуваних системних викликів, які не потребують зміни пам'яті (наприклад, часто виконуваний системний виклик *gettimeofday* для читання та встановлення часу завдяки цій технології замінюється на виклик звичайної функції і доступ до пам'яті без необхідного для системних викликів перемикання в режим ядра).

Згідно ABI System V системні виклики повинні виконуватись за допомогою інструкції *syscall*. На практиці додатково використовуються виклики через vDSO. Підтримка системних викликів через *int* 0x80, *sysenter* і *vsyscall* залишилась для сумісності.

Як було вказано, останнім часом більшість ОС на апаратній платформі Intel x86 перейшла для реалізації системних викликів від використання програмного переривання *int* (та команди *iret* для повернення з процедури системного виклику) до команд процесора *sysenter* (та команди повернення *sysexit*) та *syscall* (та команди повернення *sysret*).

В ОС Windows, використання цих інструкцій залежить від платформи: 64-розрядне ядро з 64-розрядним простором користувача на архітектурі x86-64 завжди використовує *syscall*; 32-розрядна система використовує інструкцію *sysenter*, якщо вона доступна; підсистема WoW64 (Windows-on-Windows 64-bit: 32-розрядний простір користувача на 64-розрядному ядрі) використовує команду *call far*, яка транслюється в *syscall*.

Приклад формування системного виклику при звертанні в процесі до функції *CreateFile* (фрагменти з налагоджувача CDB.exe з Microsoft Windows SDK):

♦ Використання переривання *int* 0x2e

```
0:000> uf ntdll!NtCreateFile
ntdll!ZwCreateFile:
mov eax,25h
mov edx,offset SharedUserData!SystemCallStub (7ffe0300)
call dword ptr [edx]
ret 2Ch

0:000> uf poi(7ffe0300)
ntdll!KiIntSystemCall:
```

```
lea edx,[esp+8]
int 2Eh
ret
```

♦ використання *syscall* (для 64-розрядної архітектури):

```
0:000> uf ntdll!NtCreateFile
ntdll!ZwCreateFile:
mov r10,rcx
mov eax,52h
syscall
ret
```

♦ використання *sysenter* (для 32-розрядної архітектури):

```
0:000> uf ntdll!NtCreateFile
ntdll!ZwCreateFile:
mov eax,25h
mov edx,offset SharedUserData!SystemCallStub (7ffe0300)
call dword ptr [edx]
ret 2Ch

0:000> uf poi(7ffe0300)
ntdll!KiFastSystemCall:
mov edx,esp
sysenter
ret
```

Команда *int* 0x2e – застарілий спосіб виконання переходів користувача в режим ядра, який підтримується всіма ЦП x86, що існують сьогодні, при виклику *int* 0x2e виконується перехід у режим ядра та викликається підпрограма обслуговування переривань, зареєстрована в таблиці дескрипторів переривань (IDT) для вектора 0x2e (*nt!KiSystemService*). *int* 0x2e є програмним перериванням і має всі накладні витрати, пов'язані з обробкою переривань. Перехід у режим ядра – операція, яка виконується дуже часто (може відбуватися приблизно 5000-6000 разів на секунду).

Інструкція *sysenter*/*syscall* використовує спеціальний шлях для виклику обробника системної служби режиму ядра (*nt!KiFastCallEntry*). Ядро реєструє цю функцію в ЦП, зберігаючи покажчик на неї в спеціальному регістрі ЦП (MSR) SYSENTER_EIP_MSR (0x175) під час запуску системи. Швидкість переходу в ОС Windows у режим ядра за допомогою

sysenter/*syscall* так само як і в ОС Linux відбувається приблизно в три рази швидше, ніж за допомогою переривання.

Оскільки двійковий файл *ntdll.dll* може виконуватися на всіх типах ЦП x86, *ntdll.dll* має підтримувати всі механізми для виконання переходів з режиму користувача в режим ядра. Рішення про те, який саме механізм використовувати, приймається під час виконання залежно від типу ЦП у системі (біт 11 (SEP) регістра *edx* із результату інструкції *cpuid* повідомляє, чи підтримує процесор інструкції *sysenter*/*syscall*). Однак, у системах x64 завжди використовується інструкція *syscall*.

Приклад реалізації системного виклику *NtCreateFile* (відповідні виклики API функції створення файла в процесі користувача *CreateFileA* для кодування символів у форматі ANSI (8 біт на символ) та *CreateFileW* – для кодування символів у форматі UNICODE (16 біт на символ) з бібліотеки *kernel32*) в ОС Windows наведено на рис. 2.13.

Системні виклики в ОС Windows мають імена, які починаються з префікса *Nt* або *Zw* (наприклад, *NtCreateFile* і *ZwCreateFile*) – обслуговуються однією системною процедурою в режимі ядра. Для системних викликів із режиму користувача версії підпрограм *Nt* і *Zw* поводяться однаково – з обов'язковою перевіркою параметрів (перевірка виконується завжди, коли режим *PreviousMode* дорівнює *UserMode*). Для викликів із драйвера режиму ядра версії підпрограм *Nt* і *Zw* відрізняються обробкою значень параметрів, які передаються системній процедурі: при виклику версії *Nt* буде виконана перевірка параметрів (режим *PreviousMode* дорівнює *UserMode*), а при виклику версії *Zw* виклик буде передано до функції версії *Nt*, але без перевірки параметрів (в функції версії *Zw* для цього виконується зміна режиму *PreviousMode* з *UserMode* на *KernelMode*).

Програміст використовує функцію *CreateFile* (що є лише *define* токеном для *CreateFileW* або *CreateFileA*), яка об'являється у файлі *fileapi.h* і реалізована в бібліотеці *kernel32.dll*. Однак сама ця функція не вимагає створення файла, вона лише перевіряє аргументи на вході і викликає функцію *NtCreateFile* (префікс *Nt* свідчить про те, що функція нативна – відноситься до недокументованого інтерфейсу програмування, призначеного для внутрішнього використання). Дана функція описана у

файлі *winternl.h* і реалізована в *ntdll.dll*. Вона виконує підготовку до переходу в режим ядра, після чого здійснюється системний виклик для створення файла. Бібліотека *kernel32.dll* – лише обгортка для *ntdll.dll*, одна з причин такої реалізації – створення можливості змінити нативні функції без зміни API.

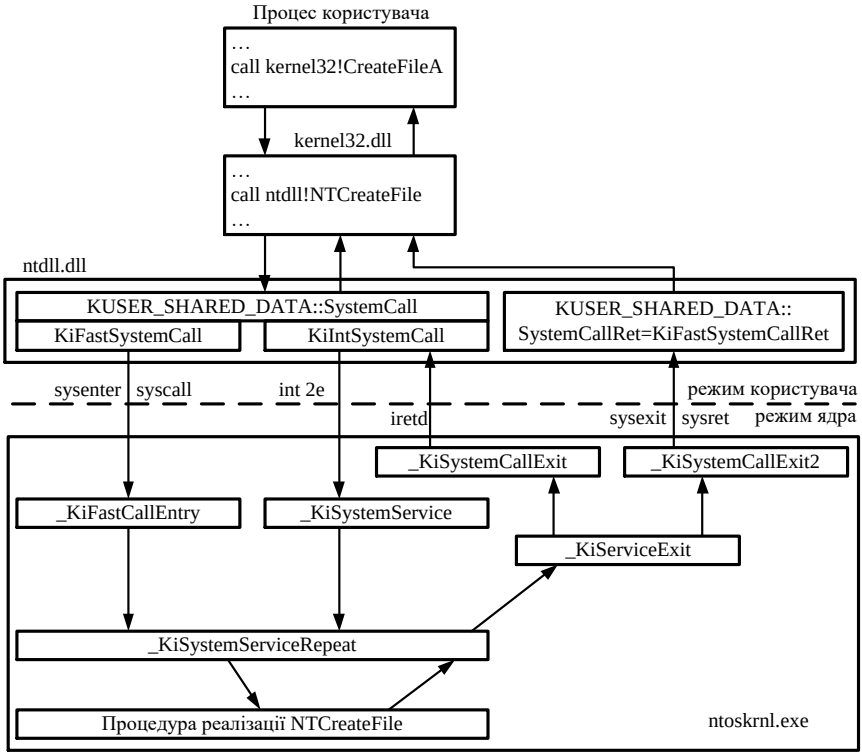


Рисунок 2.13 – Два шляхи обробки системного виклику *NtCreateFile* у Windows

Звісно, можна обійти обгортку *kernel32.dll*, але не рекомендується безпосередньо викликати нативні функції. Основна перевага нативних застосунків (англ. Windows Native Applications) полягає в тому, що *ntdll.dll* завантажується в систему значно раніше ніж *kernel32.dll*: *kernel32.dll* вимагає наявності *ntdll.dll* для своєї роботи. Відповідно, застосунки, що

користуються нативними функціями, можуть розпочинати роботу значно раніше, ніж застосунки, які використовують Windows API: їх можна запустити на ранньому етапі завантаження Windows. Приклад нативного застосунка: *autochk.exe*, який виконує утиліту *chkdisk* для перевірки диска на помилки ще до запуску основних сервісів.

Бібліотека *ntdll.dll* містить безліч функцій, таких як: завантажувачі образів (префікс в імені функції *Ldr*), диспетчер динамічної області пам'яті, функції обміну даними між процесами підсистеми Windows (префікс в імені функції *Csr*), загальні підпрограми бібліотеки часу виконання (префікс в імені функції *Rtl*), функції підтримки налагодження в режимі користувача (префікс в імені функції *DbgUi*), відстеження подій для Windows (*EventTracingforWindows*) (префікс в імені функції *Etw*), диспетчер асинхронних викликів процедур і виключень користувацького режиму (APC), невеликий піднабір підпрограм часу виконання мови C (CRT), який обмежений підпрограмами, що є частиною рядкових і стандартних бібліотек (наприклад, *memcpy*, *strcpy*, *itoa* тощо).

Як було вказано вище, бібліотека *ntdll.dll* також містить інтерфейси диспетчера системних викликів (англ. System service dispatch stubs), які містять специфічну для апаратної платформи команду переходу в режим ядра для звернення до диспетчера системних викликів.

До цієї групи належать більш ніж 400 функцій, серед яких *NtCreateFile(), NtSetEvent()* тощо. Основна частина можливостей, властивих даним функціям, доступна через Windows API. Але деякі можливості недоступні та призначені для використання лише всередині ОС. Для кожної з цих функцій у *ntdll.dll* міститься точка входу з ім'ям, що збігається з ім'ям функції. Код всередині функції містить інструкцію, що залежить від конкретної архітектури, та здійснює перехід у режим ядра для виклику диспетчера системних служб, який після перевірки параметрів викликає справжню системну службу режиму ядра, реальний код якої міститься у файлі *ntoskrnl.exe*.

Команди *sysenter*/*syscall* обробляються частиною диспетчера системних викликів *_KiFastCallEntry*, переривання *int* 0x2e – *_KiSystemService*, далі викликається однаковий код з *_KiSystemServiceRepeat*,

в якому визначається процедура обробки виклику через таблицю системних викликів (для функцій ядра ОС) *nt!KiServiceTable*, доступ до якої виконується через таблицю *KeServiceDescriptorTable* (доступ до таблиці *W32pServiceTable*, що пов'язана з графічною підсистемою, виконується через таблицю *KeServiceDescriptorTableShadow*). Таблиці системних викликів на рис. 2.13 не наведені. Далі передається управління процедурі реалізації системного виклику. Після завершення процедури обробки системного виклику *_KiSystemServiceRepeat* передає управління процедурі *_KiServiceExit* для підготовки перемикання в режим користувача і за допомогою функції *_KiSystemCallExit* (при використанні механізму переривань) або *_KiSystemCallExit2* (при використанні нових команд процесора) виконується повернення в режим користувача.

Слід зауважити, що не кожна функція з набору Windows API однозначно відображається на окремий системний виклик. Існують тисячі функцій Windows API, а кількість системних викликів для ядра NT становить 210 для Windows NT 4.0 та 486 – для Windows 11 23H2 (для сучасної ОС Linux кількість системних викликів для платформи x86 становить 384, для платформи x64 – 332). На рис. 2.14 наведено приклад відображення кількох функцій роботи з пам'яттю (бібліотечні та Windows API) на системний виклик.

Так, наприклад, для створення купи програміст використовує API функцію *HeapCreate()*, при її виклику відбувається перехід у бібліотеку *kernel32.dll*, з якої виконується перехід до функції *RtlCreateHeap(),* реалізованій у бібліотеці *ntdll.dll*. У свою чергу, у бібліотеці *ntdll.dll* функція *RtlCreateHeap()* звертається до нативної функції *NtAlocateVirtualMemory(),* яка є останнім етапом звернення до системного виклику в просторі користувача.

Частота генерації системних викликів є досить високою. Наприклад, в ОС Windows при активній роботі відбувається в середньому близько 2000 системних викликів у секунду, а при простоюванні системи – до 100 викликів у секунду. Будь-яка програма в стадії виконання призводить до десятків системних викликів.

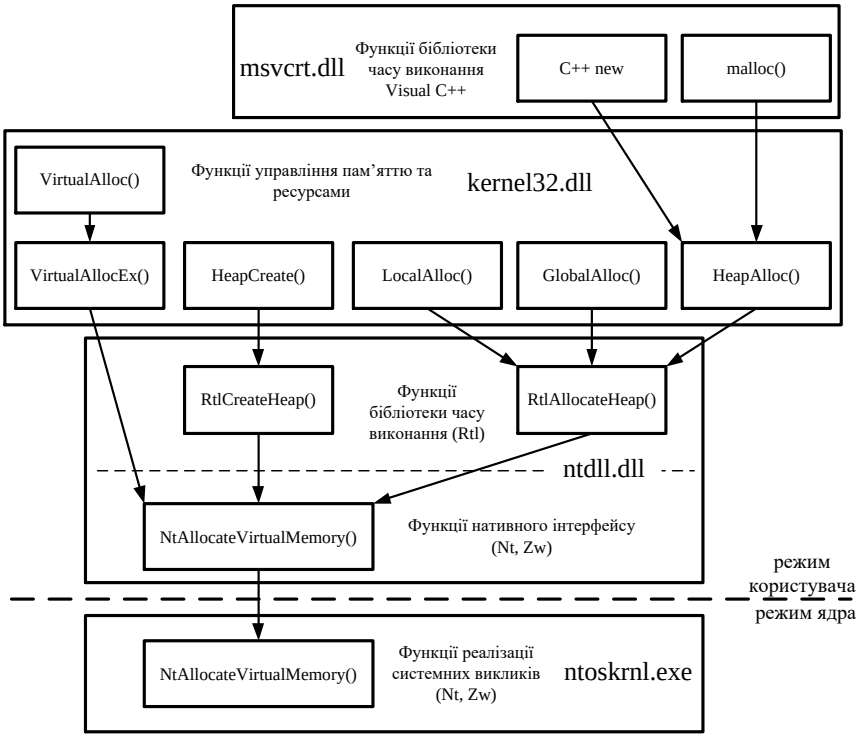


Рисунок 2.14 – Відображення функцій на системні виклики

Наприклад, для наступної простої програми *hello.c* для ОС Linux:

```
#include <stdio.h>
int main() {
   printf("Hello!\n");
   }
```

за допомогою утиліти *strace* можна відстежити кількість системних викликів (наведено результат з використанням параметру *-c* – отримання лише статистики):

```
pvi@pvi-pc:~/t1$ strace -c ./hello
Hello!
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 29,25    0,000043          10         4           mprotect
 13,61    0,000020          20         1           munmap
 10,20    0,000015          15         1           write
  8,84    0,000013           1         8           mmap
  8,84    0,000013           4         3           brk
  7,48    0,000011           3         3           newfstatat
  3,40    0,000005           2         2           close
  3,40    0,000005           5         1           set_tid_address
  3,40    0,000005           5         1           set_robust_list
  3,40    0,000005           5         1           getrandom
  2,72    0,000004           2         2         1 arch_prctl
  2,72    0,000004           4         1           prlimit64
  2,72    0,000004           4         1           rseq
  0,00    0,000000           0         1           read
  0,00    0,000000           0         4           pread64
  0,00    0,000000           0         1         1 access
  0,00    0,000000           0         1           execve
  0,00    0,000000           0         2           openat
------ ----------- ----------- --------- --------- ----------------
100,00    0,000147           3        38         2 total
```

У результаті аналізу з’ясовується, що для створення, завантаження в пам’ять та безпосередньо виконання процеса з програми, вихідний код якої наведено вище, необхідно виконати 38 системних викликів, більша частина з яких відноситься до додаткових операцій в ОС з підтримки виконання процесів.

2.4.4. Підтримка життєвого циклу процесів

Всі системні виклики, для яких потрібно виконувати перехід у режим ядра, можна умовно розділити на 4 основні категорії з такими головними функціями:

1) контроль за процесами:

- ♦ створення, завантаження, виконання, зупинка процеса;
- ♦ отримання/встановлення атрибутів процеса;

♦ виділення або звільнення пам’яті;

♦ очікування подій, подання сигналів про події та ін.;

2) керування файлами та пристроями:

♦ створення/відкриття, читання/запис, закриття/видалення файлів або пристроїв;

♦ отримання/встановлення їх атрибутів та ін.;

3) спілкування/зв’язок:

♦ створення/видалення комунікаційного з’єднання;

♦ надсилання/отримання повідомлень;

♦ отримання інформації про статус передачі;

♦ підключення або відключення віддалених пристроїв та ін.;

4) інформаційне обслуговування:

♦ отримання/встановлення системних даних;

♦ отримання/встановлення часу або дати та ін.

До основних системних викликів з групи контролю за процесами відносяться такі:

1. Створення процеса. У відповідь на запит створення ОС породжує новий процес із заданими за замовчуванням атрибутами та ідентифікатором. Деякі з параметрів, що визначаються на момент породження процеса, містять:

♦ рівень привілеїв (робота в режимі ядра або користувача);

♦ пріоритет процеса;

♦ розмір та вимоги до пам’яті;

♦ максимальний розмір пам’яті для даних та / або розмір стека;

♦ інформацію про захист пам’яті та права доступу;

♦ інші системно залежні дані.

В ОС класу UNIX використовується модель *fork-exec* – двокрокове породження процеса. На першому кроці за допомогою системного виклику *fork* створюється ідентична копія поточного процеса (для забезпечення швидкодії, як правило, одразу адресний простір не копіюється, а тільки при першій зміні, яку вносить дочірній процес – використовується механізм копіювання при запису COW (англ. Copy-On-Write)). На другому кроці за

допомогою операції *exec* завантажується нова програма (у цій моделі батьківський процес має можливість дочекатися завершення дочірнього процеса за допомогою системних викликів сімейства *wait*). Розбивка цієї операції на два кроки дає можливість легко породжувати ідентичні копії процеса (наприклад, для масштабування програми), а також гнучко керувати ресурсами, доступними дочірньому процесу.

2. Завершення процеса. Виконання системного виклику завершення процеса призводить до активації дій ОС, щоб знищити певний процес і видалити його з системи.

3. Скасування процеса. Використовується для примусового завершення процеса. Хоча процес цілком може перервати своє виконання сам, частіше цей виклик використовується для примусового звільнення, наприклад, для видалення несправного процеса в системі.

Для ОС, в яких одиницею виконання є нитка, можливе використання моделі *fork-join*. Вперше її використання з'явилося в якості примітивів для багатопроцесорних систем. Операція створення *fork* використовується для поділу послідовності інструкцій на дві такі самі послідовності, що виконуються паралельно. Операція *join* використовується для об'єднання двох послідовностей коду, поділених раніше за допомогою *fork*. Операція *join* може викликатися в батьківському процесі з метою синхронізації роботи з дочірнім процесом.

4. Призупинення виконання процеса. Призначений процес призупиняється на невизначений термін і переводиться в стан очікування. Процес може призупинити сам себе або бути призупиненим іншим процесом, який має достатньо прав для цього.

5. Відновлення виконання процеса. Цей запит відновлює виконання вказаного процеса (він переводиться у стан готових процесів), який імовірно був призупинений. Очевидно, сам призупинений процес не може відновити своє виконання.

6. Очікування. Цільовий процес призупиняється на зазначений період часу. Час може бути виражений в умовних одиницях ОС (такти) або в умовних одиницях часу (секунди, хвилини). Процес може перейти в

очікування самостійно або примусово за командою ззовні: при необхідності відкласти виконання деяких інших процесів.

7. Отримання атрибутів. Це запит до ОС, на який вона повертає поточні значення атрибутів процеса, або їх підмножину, отриману з дескриптору процеса – БУП.

8. Зміна пріоритету. Цей системний виклик дозволяє змінити пріоритет процеса. У системах, де пріоритет процеса є статичним або не використовується, даний системний виклик не реалізований.

Наведемо приклади основних системних викликів для роботи з процесами в ОС Linux [5, 7, 17, 39]. Всі вихідні коди системи зручно представлені на ресурсі https://elixir.bootlin.com, таблиці системних викликів для різних архітектур – https://syscalls.mebeim.net.

• Створення нового процеса (системні виклики *fork, vfork, clone*):

```
pid_t fork(void);
```

де *pid_t* – примітивний тип даних, який визначає ідентифікатор процеса або групи процесів. При виклику *fork()* породжується новий процес (дочірній процес), який майже ідентичний батьківському процесу, в якому був виконаний виклик *fork()*. Дочірній процес успадковує наступні ознаки батьківського процеса:

♦ сегменти коду, даних і стека програми;

♦ таблицю файлів, в якій знаходяться стани прапорців дескрипторів файлів, які вказують на тип операції з файлом (читання або запис). Крім того, у таблиці файлів міститься поточна позиція покажчиків запису / читання;

♦ робочий і кореневий каталоги;

♦ реальний і ефективний номер користувача і номер групи;

♦ пріоритет процеса;

♦ контрольний термінал;

♦ маску сигналів;

♦ обмеження по ресурсах;

♦ середовище виконання;

♦ спільні сегменти пам’яті.

Дочірній процес має наступні власні (не отримує від батьківського процеса) ознаки:

♦ ідентифікатор процеса PID (англ. Process Identifier), та ідентифікатор батьківського процеса PPID (англ. Parent Process Identifier);

♦ витрачений час ЦП (він обнуляється);

♦ сигнали батьківського процеса, що вимагають відповіді;

♦ блоковані файли.

При виклику *fork()* виникають два практично ідентичних процеси. Увесь код після *fork()* виконується двічі: як у дочірньому процесі, так і в батьківському.

Дочірній і батьківській процеси отримують різні коди повернення після виклику *fork()*. Батьківський процес отримує ідентифікатор PID дочірнього процеса. Якщо це значення негативне, це означає, що при створенні процесів сталася помилка. Дочірній процес отримує в якості коду повернення значення 0, якщо виклик *fork()* стався успішно. Таким чином, можна перевірити, чи був створений новий процес.

На рис. 2.15 наведено етапи створення нового процеса за допомогою виклику *fork()*. У результаті буде створено копію батьківського процеса. Одразу після виконання виклику обидва процеси (батьківський і новий) продовжать своє виконання з наступного оператора програми. Значення змінної *pid* у батьківському процесі буде більше 0 (наприклад, 3456), у дочірньому – 0. При виконанні аналізу змінної *pid* виконується розгалуження: у батьківському процесі перехід на функцію *ParentProcess()*, у дочірньому – на функцію *ChildProcess()*. Така організація створення процесів дозволяє розмістити код різних процесів в одному вихідному файлі та спростити передачу інформації від батьківського процеса до дочірнього.

Звісно, програміст у програмі звертається не безпосередньо до системного виклику *fork*, а тільки до функції *fork()*, яку реалізовано як обгортку в бібліотеці *glibc*.

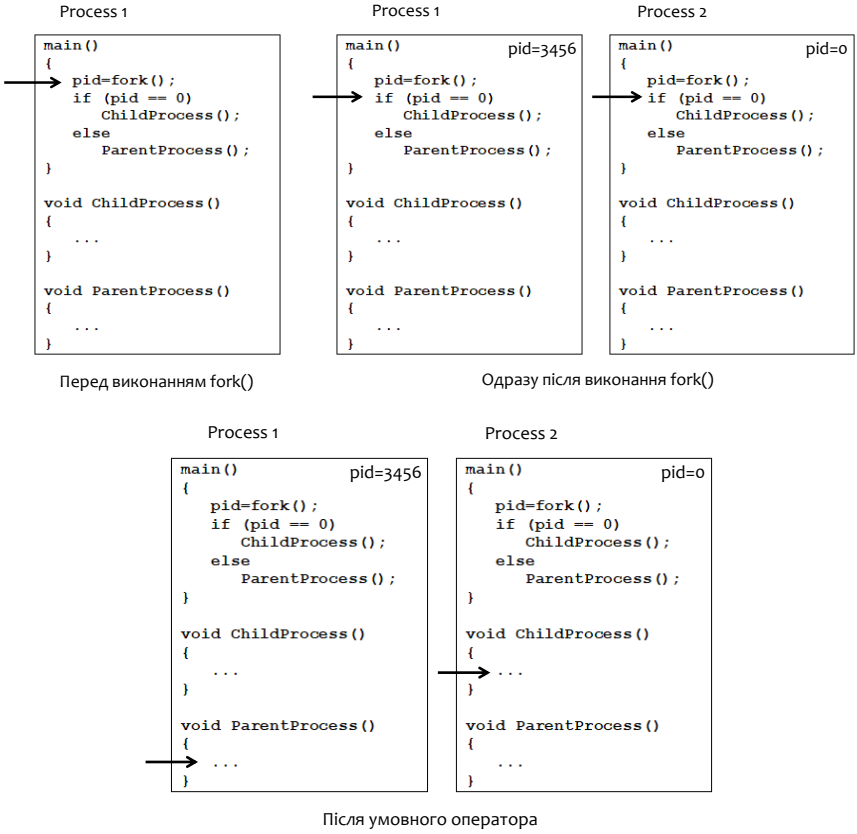


Рисунок 2.15 – Етапи створення процеса за допомогою *fork*()

Шлях від оператора в програмі до реалізації створення процеса в ОС Linux виглядає наступним чином (рис. 2.16):

оператор виклику функції в програмі *fork()* →

обгортка в бібліотеці *glibc* →

системний виклик (*syscall*) *fork* (номер у таблиці системних викликів – 0x39) →

перехід у режим ядра →

процедура реалізації системного виклику верхнього рівня *sys_fork()* →

процедура реалізації системного виклику нижнього рівня *_do_fork()* (версія ядра до 5.10) або *kernel_clone()* (версія ядра 5.10 та вище) →

процедура копіювання процеса *copy_process()*.

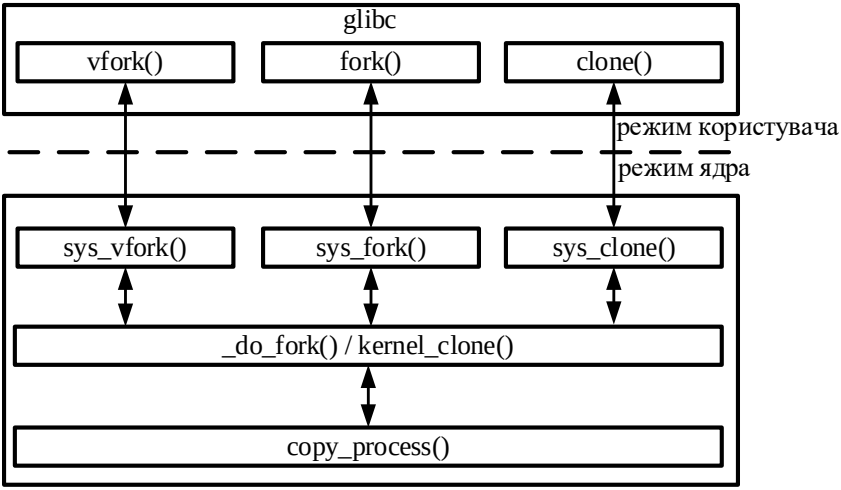


Рисунок 2.16 – Внутрішня реалізація виклику *fork*

У вихідних кодах ядра ОС реалізація *fork()* розташована у файлі */kernel/fork.c*.

Визначення системного виклику до версій ядра 5.10:

```
#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
        return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
#else
        /* помилка – не підтримується в режимі nommu */
        return -EINVAL;
#endif
}
#endif

#ifdef __ARCH_WANT_SYS_VFORK
```

```
SYSCALL_DEFINE0(vfork)
{
        return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
                        0, NULL, NULL, 0);
}
#endif

long _do_fork( /* параметри виклику - у прикладі не вказано */)
{
...
        /* створення нового процеса */
        p = copy_process(clone_flags, stack_start, stack_size,
                         child_tidptr, NULL, trace, tls);
        ...
        if (!IS_ERR(p)) {
                ...
                /* отримання pid створеного процеса */
                pid = get_task_pid(p, PIDTYPE_PID);
                nr = pid_vnr(pid);
                /* фіналізація виклику */
                wake_up_new_task(p);
        } else {
                /* помилка */
                nr = PTR_ERR(p);
        }
        /* повернення pid створеного процеса */
        return nr;
}
```

Визначення системного виклику починаючи з версії ядра 5.10:

```
#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
        struct kernel_clone_args args = {
                .exit_signal = SIGCHLD,
        };
        return kernel_clone(&args);
#else
        /* помилка – не підтримується в режимі nommu */
```

```
        return -EINVAL;
#endif
}
#endif

#ifdef __ARCH_WANT_SYS_VFORK
SYSCALL_DEFINE0(vfork)
{
        struct kernel_clone_args args = {
                .flags          = CLONE_VFORK | CLONE_VM,
                .exit_signal    = SIGCHLD,
        };
        return kernel_clone(&args);
}
#endif

pid_t kernel_clone(struct kernel_clone_args *args)
{
        ...
        /* створення нового процеса */
        p = copy_process(NULL, trace, NUMA_NO_NODE, args);
        ...
        /* помилка */
        if (IS_ERR(p))
                return PTR_ERR(p);
        ...
        /* отримання pid створеного процеса */
        pid = get_task_pid(p, PIDTYPE_PID);
        nr = pid_vnr(pid);
        ...
        /* фіналізація виклику */
        wake_up_new_task(p);
        put_pid(pid);
        /* повернення pid створеного процеса */
        return nr;
}
```

У програмі, написаною мовою С, також можна викликати функцію *syscall()*, яка дозволяє безпосередньо згенерувати системний виклик за його номером:

```
long syscall(long number, ...);
```

syscall() – невелика бібліотечна функція, яка робить системний виклик, чий інтерфейс вказується в параметрі *number*, з додатковими аргументами. Виконання *syscall()* потрібно, наприклад, для запуску системного виклику, який не має обгорткової функції в бібліотеці С. У двох наступних рядках коду результат виконання буде однаковий:

```
pid = fork();
pid = syscall(SYS_fork);
```

На рис. 2.16 та в прикладах коду вище також наведено системний виклик *vfork*, який декілька відрізняється від виклику *fork* (але його реалізовано в тих самих процедурах). У програмах відповідно використовується виклик функції *vfork()*:

```
pid_t vfork(void);
```

де *pid_t* – примітивний тип даних, який визначає ідентифікатор процеса або групи процесів.

Системний виклик *vfork* також використовується для створення нового процеса. Подібно до *fork*, тут також новий створений процес є дочірнім процесом, для процеса, який викликав *vfork*. Код дочірнього процеса також ідентичний коду батьківського процеса. На відміну від *fork* – як дочірній, так і батьківський процес спільно використовують один адресний простір (відповідно і не використовується копіювання під час запису COW) і дочірній процес призупиняє виконання батьківського процеса до того часу, поки не завершить своє виконання. Якщо дочірній процес змінить будь-які ресурси (зміст пам'яті, значення змінних, стек), ці зміни також відбудуться і для батьківського процеса. Оскільки при використанні *vfork* не створюється окремий адресний простір, то цей виклик має бути реалізований там, де дочірній процес одразу викликає *exec()* після створення.

У виклику *vfork* є наступні обмеження: поведінка функції не визначена, якщо створений за її допомогою процес здійснить хоча б одну з таких дій:

♦ повернеться з функції, в якій був викликаний *vfork()* (руйнування структури стека батьківського процеса);

♦ викликає будь-яку функцію, окрім *exit()* або *exec()* (руйнування структури стека батьківського процеса);

♦ змінить будь-які дані крім змінної, в якій зберігається значення, що повертається функцією *vfork()* (зміна даних батьківського процеса).

Всі ці обмеження пов'язані з тим, що дочірній процес після *vfork()* має спільний адресний простір з батьківським процесом (в прикладах коду вище можна побачити, що для *vfork()* задається прапорець CLONE_VM), тобто не створює копію батьківського процеса, а розділяє адресний простір з батьківським процесом, доки не буде викликана функція *exit()* або одна з функцій *exec()*. Іншими словами, після виклику *vfork()* обидва процеси будуть бачити одні і ті самі дані.

Інколи користуються цією властивістю, передаючи дані між процесами, хоча щодо стандарту POSIX поведінка таких процесів не визначена. Навіть при звичайній зміні даних (наприклад, змінюється якась змінна, щоб таким чином повідомити батьківському процесу про якісь дії) можуть виникнути менш очевидні проблеми:

♦ навіть якщо в коді мови високого рівня дочірній процес дійсно не змінює ніяких інших даних, у машинному коді це може бути не так (наприклад, по причині виявлення прихованих тимчасових змінних);

♦ при виклику *vfork()* батьківський процес зупиняється, але насправді зупиненою буде тільки поточна нитка, в якій було зроблено виклик. Це означає, що дочірній процес продовжує виконуватися паралельно з іншими нитками, які можуть, наприклад, змінити права батьківського процеса: отримуємо два процеси з різними правами в одному адресному просторі, що приводить до проблем з безпекою.

Слід звернути увагу на ще одну особливість використання виклику *vfork()*. При створенні процеса за допомогою *fork()* для нього обов'язково створюється новий адресний простір, що супроводжується виконанням додаткових операцій та, відповідно, витратами часу. Якщо після створення нового процеса виконати завантаження нової програми (виклик *exec()*), то потрібно знову створювати новий адресний простір – тобто перша операція

по створенню адресного простору при виклику *fork()* є надлишковою. При використанні пари *vfork()-exec()* новий адресний простір створюється лише один раз.

У сучасних версіях ОС використовується COW та запуск породженого процеса перед продовженням виконання батьківського процеса, відповідно в багатьох випадках (коли одразу використовується виклик *exec()*), створення адресного простору виконується також один раз під час виклику *exec()* (за винятком копіювання таблиць сторінок батьківського процеса – ці структури створюються два рази – під час *fork()* та під час *exec()*). Але швидкість виконання пари *vfork()-exec()* по зрівнянню з *fork()-exec()* все одно вища в кілька разів. Тому у випадках, коли одразу після створення процеса необхідно в нього завантажити іншу програму, з точки зору швидкості краще використовувати виклик *vfork()*.

Також слід звернути увагу на один з неявних недоліків *vfork()*: при помилці виконання *exec()* виникає небезпечна поведінка – тому що необхідно цю помилку обробити, але новий адресний простір не було створено і всі зміни відбудуться в адресному просторі батьківського процеса.

У табл. 2.1 вказані основні відмінності між викликами *fork()* та *vfork()*.

Приклад, який демонструє використання адресного простору батьківського процеса при здійсненні виклику *vfork()*:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void)
{
  pid_t pid;
  int var = 0;
  pid = vfork();
  if (-1 == pid) { return (-1); }
  if (pid == 0) { var = 1; exit(0); }
     else { printf("var=%d\n", var); }
  return (0);
}
```

Таблиця 2.1 – Порівняння викликів *fork*() та *vfork*()

fork()	*vfork*()
Дочірній і батьківський процеси мають окремі адресні простори.	Дочірній і батьківський процеси використовують один і той самий адресний простір.
Батьківський та дочірній процеси виконуються одночасно.	Батьківський процес залишається зупиненим, поки дочірній процес не завершить виконання.
Якщо дочірній процес змінює дані в адресному просторі, це не впливає на батьківській процес, оскільки адресний простір розділений.	Якщо дочірній процес змінює дані в адресному просторі, це впливає на батьківській процес, оскільки вони спільно використовують один і той же адресний простір.
Використовується копіювання під час запису, коли батьківський та дочірній процеси спільно використовують одну й ту саму пам’ять, доки один з них не почне її змінювати.	*vfork*() не використовує копіювання під час запису.
Повільний виклик за рахунок копіювання адресного простору.	Швидкій виклик – копіювання адресного простору не виконується.
Надійний виклик.	Небезпечна поведінка в багатьох випадках.

У результаті виконання прикладу значення змінної *var*, яка виводиться в батьківському процесі буде відмінним від початкового, тому що її змінює дочірній процес, який працює в тому самому адресному просторі. Виконання виклику *exit(0)* (або в інших випадках *exec()*) – обов’язкове, щоб батьківський процес коректно міг продовжити своє виконання (якщо його

прибрати – можна побачити, що батьківський процес працює некоректно – значення змінної *var* спотворюється).

Якщо в прикладі замінити виклик *vfork()* на *fork()* – можна побачити, що зміна значення змінної *var* у дочірньому процесі не впливає на батьківській (хоча дочірній процес починає працювати до початку продовження виконання батьківського (по причині, яка була вказана вище – COW та запуск породженого процеса перед продовженням виконання батьківського процеса), але в деяких випадках оператор *printf* у батьківському процесі може бути виконаний раніше, ніж відпрацює дочірній процес, тому бажано додати рядок *wait(NULL)* перед викликом *printf*. Також у даному випадку можна перевірити, що видалення виклику *exit(0)* у дочірньому процесі не приводить до некоректної поведінки батьківського процеса.

Системний виклик *clone* є оновленою версією виклику *fork*. Він створює дочірній процес і забезпечує точніший контроль над даними, які спільно використовуються батьківським і дочірнім процесами. Виклик може контролювати таблицю дескрипторів файлів, таблицю обробників сигналів і використання адресного простору.

Наведений також на рис. 2.16 виклик *clone* дозволяє розмістити дочірній процес у різних просторах імен. Завдяки гнучкості можна вибрати спільний доступ до адресного простору з батьківським процесом (емуляція системного виклику *vfork*), ділитися інформацією про файлову систему, відкритими файлами та обробниками сигналів

- Виконання програми в процесі (системний виклик *execve*).

Функція *exec()* завантажує і запускає інший програмний код у дочірній процес. Таким чином, нова програма повністю заміняє код поточного процеса. Нова програма починає своє виконання зі своєї точки входу. Всі файли, відкриті програмою перед викликом *exec()*, залишаються відкритими і доступними для нової програми. Існують кілька різних варіантів функцій (описані у файлі *unistd.h*) *exec()*: *execl*, *execv*, *execle*, *execve*, *execlp*, *execvp*, *fexecve*. Літери *l*, *v*, *p* і *e* наприкінці імен функцій визначають формат і обсяг

аргументів, а також каталоги, в яких потрібно шукати завантажену програму:

```
int execl(const char *pathname, const char *arg0, ...
          /* (char *)0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ...
          /* (char *)0, char *const envp[] */ );
int execve(const char *pathname, char *const argv[],
          char *const envp[]);
int execlp(const char *filename, const char *arg0, ...
          /* (char *)0 */ );
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

де *l* – аргументи командного рядка передаються у формі списку *arg0*, *arg1*, ... *argn*, *NULL*. Цю форму використовують, якщо кількість аргументів відома; *v* – аргументи командного рядка передаються у формі вектора *argv[]*. Окремі аргументи адресуються через *argv [0]*, *argv [1]*, ... *argv [n]*. Останній аргумент (*argv [n]*) повинен мати значення *NULL*; *p* – зазначений по імені файл шукається не тільки в поточному каталозі, але також у каталогах, визначених змінною середовища PATH; *e* – функція очікує список середовища у вигляді вектора (*envp[]*) і не використовує поточне середовище виконання. Функція *fexecve()* виконує ті самі дії, що і *execve()*, тільки виконуваний файл вказується за допомогою файлового дескриптора *fd*, а не через шлях до файла.

При успішному завершенні функції *exec()* не повертається жодного значення (після успішного виконання повернення не відбувається, замість цього починає виконуватись новий процес). При виникненні помилки функція *exec()* повертає значення «-1».

У табл. 2.2 наведено порівняння функцій сімейства *exec()*, а на рис. 2.17 наведено діаграму зв'язків між функціями сімейства *exec()* та внутрішню реалізацію відповідного системного виклику *execve* в ОС Linux.

Всі варіанти *exec()* зводяться до виклику функції *execve()*. Перехід 1 на рис. 2.17 супроводжується побудовою шляху до файла по файловому

дескриптору, перехід 2 – побудовою вектора *argv*, перехід 3 – додавання середовища виконання, перехід 4 – супроводжується пошуком файла в каталогах, вказаних у змінній середовища виконання PATH.

Таблиця 2.2 – Порівняння параметрів функцій exec()

функція	pathname	filename	fd	arg list	argv[]	environ	envp[]
execl	*			*		*	
execlp		*		*		*	
execle	*			*			*
execv	*				*	*	
execvp		*			*	*	
execve	*				*		*
fexecve			*		*		*
літера		p	f	l	v		e

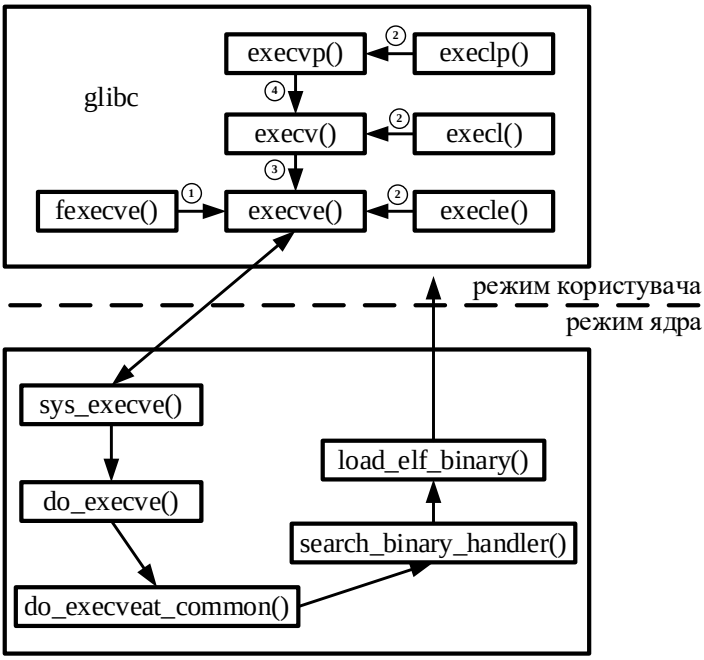


Рисунок 2.17 – Зв’язки між функціями та реалізація системного виклику *execve*

Перехід у режим ядра виконується за допомогою одного з раніше описаних варіантів (наприклад – команда *syscall* зі значенням 3b, яке заноситься в регістр *rax*) у процедуру *sys_execve* – основний обробник системного виклику *execve*, з якої викликається *do_execve* та *do_execveat_common*, що відкривають цільовий файл з образом процеса та зчитують фрагмент на початку файла для отримання основної інформації про нього. Далі викликається *search_binary_handler* – обробник двійкових файлів – механізм ядра ОС, який виконує спробу завантаження файла в різних форматах (за допомогою низки функції класу *load_binary*). Одна з таких функцій – *load_elf_binary*, що завантажує двійковий файл формату ELF (англ. Executable and Linkable Format), який, як правило, є вихідним файлом компілятора мови С. Після цього виконується повернення в режим користувача та передача управління коду завантаженого процеса.

• Очікування зміни стану виконання процеса (системні виклики *wait4*, *waitpid* та *waitid*).

Наступний набір функцій використовується для очікування зміни стану процеса і отримання інформації про процес, чий стан змінився. До змін стану відносяться: припинення роботи процеса, зупинка/відновлення роботи процеса по сигналу. Очікування припинення роботи процеса дозволяє системі звільнити ресурси, які були ним зайняті (опис функцій для ОС Linux розташовано у файлі *sys/wait.h*):

```
pid_t wait(int *_Nullable wstatus);
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
int waitid(idtype_t idtype, id_t id,
           siginfo_t *infop, int options);
```

Функція *wait()* призупиняє виконання поточного процеса (або нитки) до тих пір, поки не завершить своє виконання дочірній процес, який було запущено з нього. Функція повертає *pid_t* – ідентифікатор процеса, який припинив виконання, параметр *stat_addr* – адреса змінної цілого типу, яка після завершення виконання функції буде містити в старшому байті код

завершення дочірнього процеса, а в молодшому – індикатор причини його завершення.

Функція *waitpid()* призупиняє виконання поточного процеса (або нитки) до тих пір, поки не зміниться стан дочірнього процеса, який задано аргументом *pid*. За замовчуванням *waitpid()* очікує завершення роботи дочірнього процеса.

Виклик функції *wait(&wstatus)* еквівалентний до виклику функції *waitpid(-1, &wstatus, 0)*.

Виклик функції *waitid()* (введено в ОС Linux з версії 2.6.9) надає більш повний контроль над тим, якої зміни стану необхідно чекати в дочірнього процеса.

Реалізацію системних викликів наведено на рис. 2.18

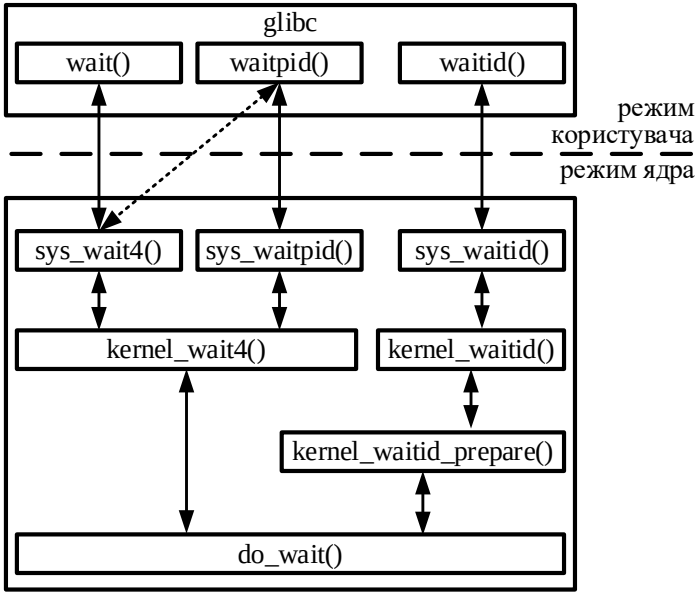


Рисунок 2.18 – Внутрішня реалізація викликів групи *wait*

В ОС Linux *wait(), waitpid()* та *waitid()* – бібліотечні функції (*glibc*), які реалізується через системні виклики *wait4, waitpid* та *waitid* (реалізовано в

процедурах ядра *kernel_wait4()*, *kernel_waitid()*, *kernel_waitid_prepare()* та *do_wait()*). Для деяких архітектур немає окремої реалізації системного виклику *waitpid*, його заміняє інтерфейс-обгортка бібліотеки С, яка викликає також *wait4*.

Слід зауважити, що в ОС класу UNIX/Linux, якщо не виконується очікування завершення процеса, то такий завершений дочірній процес залишається в системі в стані «зомбі» (англ. Zombie). Ядро ОС підтримує мінімальний набір інформації про процеси-зомбі (ідентифікатор процеса, стан завершення, використані ресурси), щоб дозволити батьківському процесу виконати функцію очікування для отримання цієї інформації. Будь-який завершений процес переводиться в стан «зомбі» і до виклику батьківським процесом функції очікування зміни стану виконання процеса займає простір у таблиці процесів, і якщо ця таблиця заповниться, неможливо буде створювати нові процеси.

• Завершення процеса (системний виклик *exit*).

Завершення процеса може бути викликано кількома подіями – звичайне завершення процеса, примусове через отримання сигналу або через явний виклик функції виходу. Але при звичайному завершенні процеса все одно виконується неявний виклик функції виходу.

Формат функції *exit()*, призначеного для завершення функціонування процеса:

```
void exit (int status);
```

Аргумент *status* є статусом завершення, який передається батьківському процесу і може бути їм отриманий за допомогою однієї з функцій очікування зміни стану виконання процеса.

При завершенні процеса (з явним або неявним викликом функції виходу) генерується системний виклик *exit*, процес завершується викликом функцій ядра *sys_exit()* та *do_exit()* (реалізовані у файлі */kernel/exit.c*). На рис. 2.19 наведено реалізацію системного виклику *exit*.

Метою *do_exit()* є видалення всіх посилань на поточний процес з ОС (для всіх ресурсів, які не є спільними). Спочатку встановлюється спеціальний прапорець, який враховується при роботі іншими процедурами ядра, щоб уникнути маніпулювання цим процесом під час його видалення.

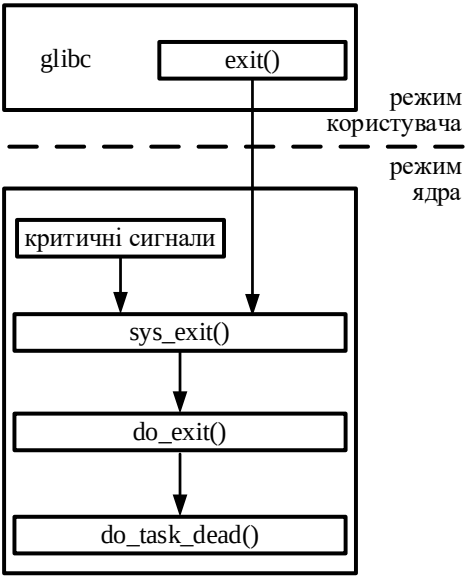


Рисунок 2.19 – Внутрішня реалізація виклику *exit*

Від'єднання процеса від різних ресурсів, які він отримав протягом свого існування, виконується за допомогою серії викликів функцій, таких як: *exit_mm()* (для видалення сторінок пам'яті), *exit_sem()* (для видалення з черги очікування семафорів IPC), *exit_files()* (для зменшення лічильників відкритих файлів), *exit_fs()* (для зменшення лічильників по файловій системі) та ін. Якщо який-небудь лічильник дорівнюватиме 0, то об'єкт видаляється з відповідної системної таблиці. Далі встановлюється код завершення процеса, його значення передається як аргумент функції *exit()*. Далі викликається функція ядра *exit_notify()*, яка відправляє сигнали батьківському процесу, і призначає новий батьківський процес (*reparent*) для всіх породжених процесів завершеним процесом. Останньою викликається

функція *do_task_dead()* (реалізована у файлі */kernel/sched/core.c*) для встановлення стану процеса, що завершується, у значення TASK_DEAD. Функції *do_exit()* та *do_task_dead()* описані як *__noreturn* – з них немає повернення (*do_task_dead()* завершується нескінченним циклом з викликом *cpu_relax()* – макроса з двох команд REP та NOP, який інтерпретується як команда PAUSE).

• Отримання ідентифікаторів PID, PPID (системні виклики *getpid*, *getppid*).

Для отримання власного ідентифікатора процеса використовується виклик *getpid()*, а для отримання ідентифікатора батьківського процеса – виклик *getppid()*:

```
pid_t getpid(void);
pid_t getppid(void);
```

Якщо батьківський процес було завершено до виклику *getppid()*, то буде повернуто ідентифікатор процеса *init* (значення 1). Виклик *getpid()* завжди повертає PID і ніколи не завершується з помилкою.

В ядрі ОС Linux для системних викликів *getpid* та *getppid* управління передається процедурі *sys_getpid()* і далі – *task_tgid_vnr()* (реалізовані у файлі */kernel/pid.c*).

• Зміна пріоритету процеса (системні виклики *nice*, *getpriority*, *setpriority*).

Процес може поміняти пріоритет собі, використовуючи виклик функції *nice()*. Виклики *getpriority()*, *setpriority()* визначають або змінюють пріоритети будь-яких процесів або груп процесів.

Функція-обгортка *nice()* описана у файлі *unistd.h* та її синтаксис наступний

```
int nice(int inc);
```

При зверненні до цієї функції генерується системний виклик *nice* (для архітектури IA-32 (скорочення від «Intel Architecture, 32-bit»), для 64-бітної x86 архітектури з інтерфейсом x86-64 ABI функція *nice()* є лише обгорткою, яка веде до виклику *setpriority*). Системний виклик *nice* збільшує поправку до пріоритету поточного процеса на величину *incr*. Поправка до пріоритету – невід'ємне число: чим воно більше, тим нижче пріоритет процеса.

При обробці системного виклику *nice* управління передається процедурам ядра *sys_nice()* та далі в *set_user_nice()*, реалізованим у файлі */kernel/sched/core.c*.

Функції *getpriority()* та *setpriority()* визначені у файлі *sys/resource.h*, їх опис:

```
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

Параметр *which* визначає, працювати з пріоритетом процеса, групи процесів або користувача. Параметр *who* задає ідентифікатор процеса, групи процесів або користувача (залежно від значення параметра *which*). Нульове значення *who* позначає, що виклик застосовується до поточного процеса, групи процесів або користувача. Параметр *prio* приймає значення від -20 до +20. За замовчуванням значення пріоритету дорівнює 0; менші його значення означають перевагу над іншими процесами. Виклик *setpriority* встановлює значення пріоритетів для всіх зазначених процесів на потрібну величину. Тільки суперкористувач (англ. Superuser) має право на зменшення *prio*.

Ядро реалізує ці системні виклики за допомогою службових процедур *sys_getpriority()* і *sys_setpriority()*, які реалізовані у файлі */kernel/sys.c*.

В ОС Windows використовуються схожі за призначенням можливості з підтримки життєвого циклу процесів. Так само, як і в Linux, програміст може користуватись різними шляхами для звернення до ядра ОС (як наведено на рис. 1.18). Таблиці системних викликів для різних версій ОС Windows можна знайти на ресурсі https://j00ru.vexillium.org/.

В якості прикладу наведемо основні системні виклики для створення нового процеса (системні виклики *NtCreateProcessEx, NtCreateUserProcess* та ін).

Windows API надає кілька функцій для створення процесів. Найпростішою є *CreateProcess()*, яка намагається створити процес з такими ж правами доступу, що й батьківській процес. Якщо потрібні інші права – використовується *CreateProcessAsUser()*.

Інші функції створення процеса: *CreateProcessWithTokenW()* і *CreateProcessWithLogonW()* (частина бібліотеки *advapi32.dll*) – викликають вторинний сервіс входу *SecLogon* у систему (бібліотека *SecLogon.dll*, яка міститься в *SvcHost.Exe*), здійснюючи виклик віддаленої процедури (англ. Remote Procedure Call, RPC) для фактичного створення процеса. У сервісі *SecLogon* виконується виклик внутрішньої функції *SlrCreateProcessWithLogon()* і далі – *CreateProcessAsUser()*. Дані функції реалізовано в утиліті командного рядка *runas*.

Після викликів функцій Windows API управління передається функції *CreateProcessInternal()*, яка починає фактичну роботу зі створення процеса Windows у режимі користувача. *CreateProcessInternal()* у свою чергу викликає *NtCreateUserProcess()* з бібліотеки *ntdll.dll*, щоб здійснити перехід у режим ядра та продовжити частину створення процеса в режимі ядра у виконавчій системі (Executive) у функції з такою ж назвою (*NtCreateUserProcess*).

Шляхи реалізації викликів створення процеса в ОС Windows наведені на рис. 2.20.

До версії ОС Windows Vista виклик Windows API функції *CreateProcess()* приводив до системного виклику *NtCreateProcessEx*, хоча він присутній і в нових версіях ОС для зворотної сумісності та створення мінімальних процесів (насправді існує пара викликів з назвами *NtCreateProcess* і *NtCreateProcessEx*, але перший є просто оболонкою, яка викликає другий). Мінімальним процесом у системі є по суті порожній процес: з порожнім адресним простором, без підключення будь-яких бібліотек, без заповнення блоків змінних оточення процеса та ниток (PEB та TEB), без створення головної нитки та без завантаження образу програми, та

такі процеси не можуть працювати в режимі користувача – вони призначені тільки для виконання певних дій з обслуговування системи в режимі ядра.

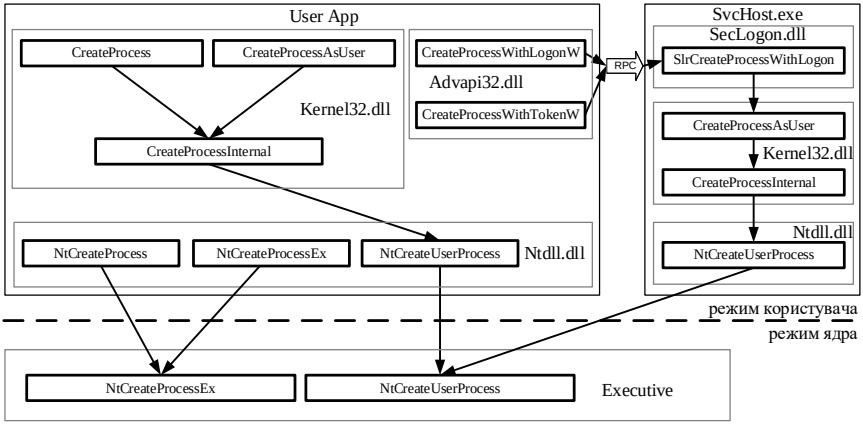


Рисунок 2.20 – Шляхи реалізації викликів створення процеса в ОС Windows

Якщо зіставити створення процеса, то можна побачити, що виклик функції *CreateProcess()* в ОС Windows еквівалентний послідовності викликів *fork()* та *execve()* в ОС Linux.

Для очікування в ОС Windows використовуються такі системні виклики: *NtWaitForMultipleObjects*, *NtWaitForMultipleObjects32*, *NtWaitForSingleObject* (відповідні функції Windows API: *WaitForMultipleObjects()*, *WaitForSingleObject()*), для завершення процесів – *NtTerminateProcess* (функції Windows API *ExitProcess()*, *TerminateProcess()*).

Другий приклад демонструє використання різних варіантів завершення роботи застосунку (згідно рис. 1.18).

Текст програми 1. Завершення через виклик бібліотечної функції *exit()*:

```
#include <stdlib.h>
int main()
{
        exit(20);
}
```

Отримати код повернення з застосунку можна в командному рядку за допомогою команди:

```
echo %errorlevel%
```

Текст програми 2.1. Завершення через виклик функції Windows API *ExitProcess()*:

```
#include <windows.h>
int main()
{
        ExitProcess(20);
}
```

Текст програми 2.2. Завершення через виклик функцій Windows API *GetCurrentProcess()* та *TerminateProcess()*:

```
#include <windows.h>
int main()
{
        auto self = GetCurrentProcess(); // отримання ProcessHandle
        TerminateProcess(self, 20);
}
```

Текст програми 3. Завершення через виклик функції Native API *NtTerminateProcess()*. Безпосередній виклик функції зробити неможливо, тому виконується отримання адреси цієї функції з бібліотеки *ntdll.dll* та виклик *NtTerminateProcess()* за отриманою адресою. Опис прототипу функції взято з ресурсу http://undocumented.ntinternals.net (*ProcessHandle* для поточного процеса вказувати на відміну від функцій Windows API не потрібно, тому функція *GetCurrentProcess()* не використовується):

```
#include <windows.h>
DWORD NtTerminateProcessAdr;
// прототип функції NtTerminateProcess
typedef NTSTATUS(NTAPI* NtTerminateProcess)(IN HANDLE
                        ProcessHandle, IN NTSTATUS ExitStatus);
```

```
int main()
{
        HMODULE ntdll = LoadLibrary(L"ntdll.dll");
// отримання адреси функції NtTerminateProcess з бібліотеки
        NtTerminateProcessAdr = (DWORD)GetProcAddress(ntdll,
                        "NtTerminateProcess");
// призначення отриманої адреси для _NtTerminateProcess
        NtTerminateProcess _NtTerminateProcess =
                        (NtTerminateProcess)NtTerminateProcessAdr;
// виклик функції NtTerminateProcess з бібліотеки ntdll.dll
        _NtTerminateProcess(0, 20);
}
```

Текст програми 4. Завершення через системний виклик *syscall*. Номер виклику для *NtTerminateProcess* отримано з таблиці номерів системних викликів *nt!KiServiceTable* (або можна скористатись ресурсом https://j00ru.vexillium.org/). Для розробки використано GNU асемблер AS під Windows[74]:

```
.globl main
.text
main:
    movq $0, %r10       # ProcessHandle – не вказуємо (=0) –
                        # завершення поточного процеса
    movq $20, %rdx      # код повернення
    movq $44, %rax      # 44 – номер виклику NtTerminateProcess
    syscall             # системний виклик
    ret
```

[74] Для Windows одним з найпопулярніших версій GCC є пакет засобів для розробки від некомерційного проєкту MSYS2. Офіційний ресурс - https://www.msys2.org/ . Для додавання набору компіляторів у встановлений пакет необхідно в терміналі виконати команду: pacman -S mingw-w64-ucrt-x86_64-gcc

2.5. Нитки

При мультипрограмуванні підвищується пропускна здатність системи, але окремий процес ніколи не буде виконано швидше, ніж, якщо б він виконувався в однопрограмному режимі. Однак задача, яка вирішується в рамках одного процеса, може мати властивості внутрішнього паралелізму, що дозволяє прискорити її вирішення. Наприклад, при виконанні задачі відбувається звернення до зовнішнього пристрою, і на час цієї операції можна не блокувати повністю виконання процеса, а виконувати деякі інші операції.

Для цього в сучасних ОС використовується механізм багатониткової обробки (англ. Multithreading). При цьому вводиться нове поняття – нитка (англ. Thread), яка є послідовністю команд, що обробляються процесором. У рамках одного процеса можуть знаходитися одна або декілька ниток. Мультипрограмування тепер реалізується на рівні ниток, і задачу, яку оформлено у вигляді кількох ниток в рамках одного процеса, можна виконати швидше за рахунок паралельного виконання її окремих частин. Найбільшу ефективність багатониткова обробка дозволяє отримати для виконання розподілених застосунків, наприклад, багатонитковий сервер може паралельно обробляти запити від кількох клієнтів.

У традиційних ОС поняття «нитка» тотожне поняттю «процес»: кожним процесом є єдина нитка виконання. Наприклад, ОС MS-DOS підтримує один призначений для користувача процес. Деякі ОС сімейства UNIX підтримують процеси безлічі користувачів, але при цьому кожен з процесів містить одну нитку. Прикладами багатониткових систем є середовище виконання Java та ОС Windows (на базі ядра NT), Linux, Solaris та ін.

Відмінності між процесом з одною ниткою, та багатонитковим процесом наведені на рис. 2.21.

Нитки, що належать до одного процеса, не настільки ізольовані друг від друга, як процеси в традиційних багатозадачних ОС, між ними легко організувати взаємодію: на відміну від процесів, які належать різним програмам, всі нитки одного процеса завжди належать одній програмі, тому

розробник програми може заздалегідь продумати роботу множини ниток процеса таким чином, щоб вони могли взаємодіяти, а не боротися за ресурси системи.

Всі нитки одного процеса мають один і той же адресний простір, тобто вони поділяють одні й ті самі глобальні змінні.

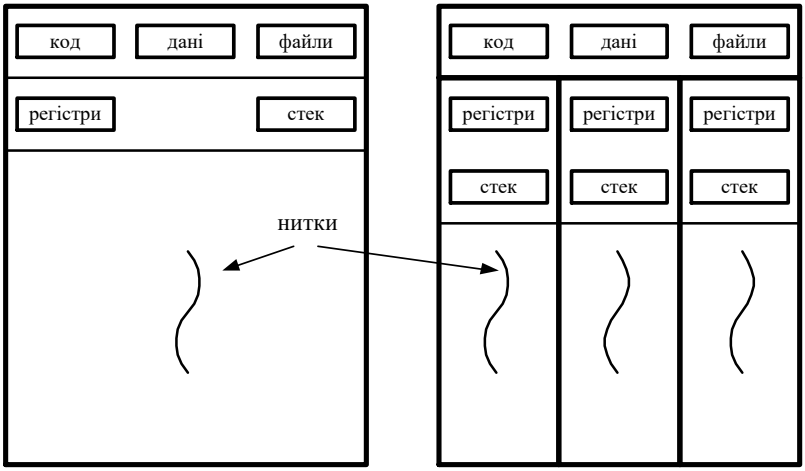


Рисунок 2.21 – Процес з однією ниткою та багатонитковий процес

Оскільки кожна нитка може мати доступ до кожної віртуальної адреси, одна нитка може використовувати стек іншої нитки: між нитками немає повного захисту – всі нитки одного процеса завжди вирішують спільну задачу, і механізм ниток використовується для більш швидкого вирішення задачі. Крім поділу адресного простору, всі нитки поділяють також:

- ♦ глобальні змінні;
- ♦ відкриті файли;
- ♦ дочірні процеси;
- ♦ сигнали та їх обробники;
- ♦ необроблені аварійні сигнали;
- ♦ статистичну інформацію про використання ресурсів.

Нитки мають власні:

- ♦ програмний лічильник;
- ♦ стек;
- ♦ регістри;
- ♦ стан.

Основні переваги використання багатониткової обробки можуть бути згруповані в наступні категорії:

1. Чутливість. Багатониткові інтерактивні системи в певних випадках дозволяють програмі продовжувати своє виконання, навіть якщо частка її блокована або виконує довгу дію, таким чином покращуючи відповідну реакцію від користувача. Наприклад, багатонитковий інтернет-браузер може дозволяти призначену для користувача взаємодію в одній нитці, поки зображення завантажується в іншій його нитці.

2. Спільне використання ресурсів. Окремі процеси можуть розділяти ресурси через використання спільної пам'яті або механізму передачі повідомлень. Реалізація таких методів потребує чітких дій програміста. Нитки ж розділяють пам'ять і ресурси процеса, до якого вони належать, за замовчуванням. Вигода від спільного використання коду і даних дозволяє програмі мати декілька ниток, що виконують різні дії в межах одного й того ж адресного простору. Також, у більшості ОС обмін інформацією між процесами відбувається за участю ядра, яке забезпечує механізм здійснення обміну і захист. Обмін інформацією між нитками одного процеса може проводитися без участі ядра, і, відповідно, швидше.

3. Економія. Виділення пам'яті і ресурсів для створення процеса недешеве. Оскільки нитки розділяють ресурси процеса, до якого вони належать, більш економічно створювати нитки і виконувати перемикання контексту між ними, а не між процесами. На практиці виміряти різницю в накладних витратах може бути важко, але взагалі на створення, завершення та управління процесами витрачається набагато більше часу, ніж на подібні дії з нитками. Для ОС Solaris, наприклад, створення процеса приблизно в тридцять разів повільніше, ніж створення нитки, а перемикання контексту – приблизно в п'ять разів.

4. Масштабованість. Переваги від багатонитковості можуть бути значно збільшені в мультипроцесорній архітектурі, де нитки можуть

виконуватись паралельно на різних процесорах. Процес з однією ниткою може бути запущений на виконання тільки на одному процесорі, незалежно від їх кількості. Багатонитковість на багатопроцесорній машині значно підвищує рівень мультипрограмування.

2.5.1. Багатониткова модель процеса

Процес в однонитковій моделі містить такі головні елементи: БУП, адресний простір (код і дані), стек (як вказано на рис. 2.1). Багатониткова модель процеса, окрім БУП і адресного простору додатково містить для кожної нитки програмний лічильник, регістри, стек, локальну пам'ять, стан. У багатонитковому режимі процеси зазвичай запускаються з однією головною ниткою. Ця нитка може створювати нові нитки зі своїм власним контекстом і стеком, після чого вони розміщуються в черзі готових до виконання ниток.

Багатониткова модель процеса поділяється на дві категорії: з організацією ниток на рівні користувача (англ. ULT – User-Level Thread), їх називають ще волокнами (англ. Fiber), і організацією ниток на рівні ядра ОС (англ. KLT – Kernel-Level Thread), їх також можуть називати ще полегшеними процесами (англ. LWP – Light-Weight Process), хоча не для кожної KLT існує LWP.

Нитки на рівні користувача (волокна) управляються самою програмою. Все управління ULT здійснюється в призначеному для користувача просторі в рамках одного процеса. Зв'язок з ядром ОС у плані управління – відсутній. Ядро продовжує здійснювати планування процеса як єдиного цілого. Використання волокон має певні переваги в порівнянні з використанням ниток на рівні ядра:

♦ управління волокнами виконує сам процес, тому не потрібно перемикатися в режим ядра, що дозволяє уникнути додаткових витрат (перемикання в режим ядра та назад);

♦ планування волокон може проводитися залежно від специфіки програми – алгоритм планування розробляє програміст відносно до своїх потреб;

♦ використання волокон можливо організувати в будь-якій ОС за допомогою бібліотеки процедур, що працюють на рівні користувача і спільно використовується всіма програмами.

До недоліків використання ULT належать:

♦ при виконанні волокном системного виклику блокується не лише дане волокно, але і всі інші волокна даного процеса;

♦ оскільки ядро закріплює за кожним процесом тільки один процесор, декілька волокон одного процеса не можуть виконуватися паралельно в разі використання багатопроцесорної системи (але для однопроцесорної системи виконання декількох волокон у процесі може бути швидшим ніж виконання декількох ниток за рахунок меншої кількості перемикань контексту).

Модель процеса з організацією ниток на рівні користувача також називається моделлю відображення ниток на процесори «багато до одного» (M:1), схематичне зображення такої моделі наведено на рис. 2.22. За такою моделлю побудовані fibers в ОС Microsoft Windows (механізм волокон доступний, починаючи з Windows 2000), а також бібліотека Green Threads в ОС Solaris.

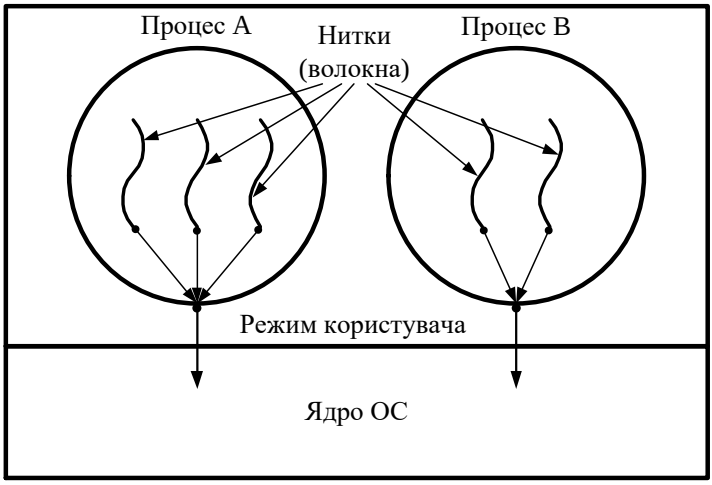


Рисунок 2.22 – Модель процесів з організацією ниток на рівні користувача

Нитки на рівні ядра управляються самим ядром ОС. Ядро підтримує контекст процеса, а також контексти кожної окремої нитки процеса. Планування виконується ядром, виходячи зі стану ниток. Саме завдяки цьому, ядро може здійснювати планування роботи кількох ниток одного процеса на декількох процесорах одночасно. Крім того, при блокуванні однієї з ниток процеса, ядро може вибрати для виконання іншу нитку даного процеса.

Основним недоліком використання ниток на рівні ядра є додаткові накладні витрати на перемикання між нитками всередині одного процеса за рахунок обов'язкових при цьому переходів у режим ядра та назад. Але, якщо нитки часто звертаються до механізмів ядра, така реалізація більш доцільна, ніж організація ниток на рівні користувача. Прикладами ОС, де використовуються нитки на рівні ядра є ОС Microsoft Windows, Linux та ін. Модель процеса з організацією ниток на рівні ядра також називається моделлю відображення ниток на процесори «один до одного» (1:1), у ній реалізовано більш жорсткий контроль над нитками зі сторони ОС. Схематичне зображення моделі наведено на рис. 2.23.

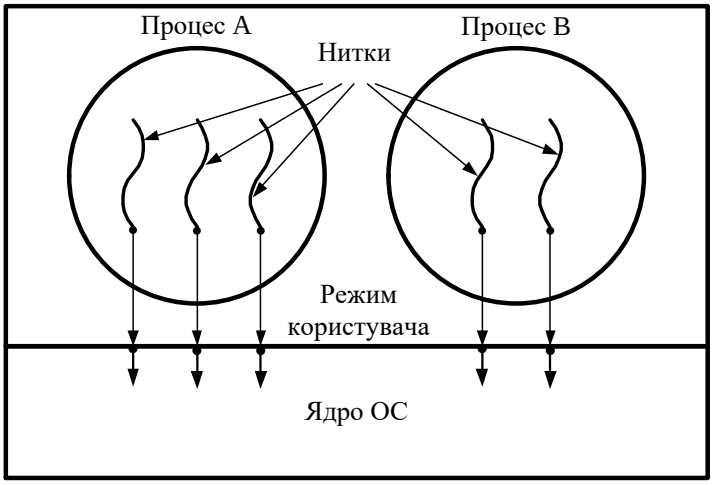


Рисунок 2.23 – Модель процесів з організацією ниток на рівні ядра

Наступний приклад демонструє відмінності при управлінні нитками в моделях (M:1) та (1:1). Припустимо, що в однопроцесорній системі процесорний час виділяється процесам частинами (квантами) по 20 мс кожному об'єкту планування, у системі виконуються тільки 2 процеси: A – поділений на нитки A1, A2, A3 та B, поділений на нитки B1, B2. Кожна нитка в процесі виконується 5 мс і переривається. Процеси A і B рівноправні. Тоді протягом 60 мс порядок виконання ниток для моделі з організацією ниток на рівні користувача буде таким (символ → позначає швидке перемикання між нитками в режимі користувача, символ ➔ – повільніше перемикання між нитками з переходом у режим ядра):

A1 → A2 → A3 → A1 ➔ B1 → B2 → B1→ B2 ➔ A2 → A3 → A1 → A2.

Для моделі процесів з організацією ниток на рівні ядра виконання ниток буде наступним:

A1 ➔ B1 ➔ A2 ➔ B2 ➔ A3 ➔ B1 ➔ A1 ➔ B2 ➔ A2 ➔ B1 ➔ A3 ➔ B2.

Як видно, для моделі (M:1) знадобилось лише 2 перемикання в режим ядра та назад, а для моделі (1:1) – 11.

У наступному прикладі наведено порівняння витрат часу на перемикання контексту для моделей (M:1) та (1:1): в однопроцесорній системі виконується процес, який поділено на 3 нитки; тривалість виконання кожної нитки складає 2 умовні одиниці (у.о.). Процесорний час виділяється процесам для виконання частинами по 3 у.о. Крім вказаного процеса в системі виконуються також і інші процеси, всі процеси рівноправні. На рис. 2.24 наведені діаграми виконання ниток процесів для наведених двох моделей.

При організації ниток на рівні ядра (1) при отриманні процесорного часу нитка встигає виконатись до завершення виділеного часу (використовується тільки дві у.о. з трьох), тому в системі починається виконання іншої нитки що, належить іншому процесу. Через деякий час (вважаємо його фіксованим і однаковим для всіх проміжків – $t_{очікування}$)

процесор буде надано наступній нитці процеса. Для завершення виконання всіх трьох ниток процеса знадобиться $t_{процеса1}=6+2* t_{очікування}$ (у.о.).

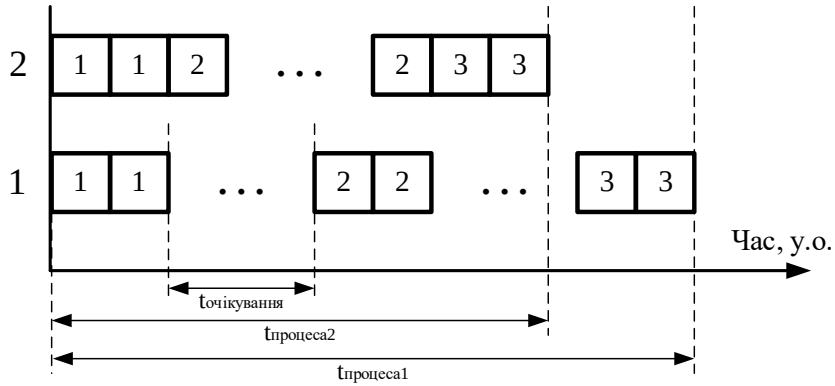


Рисунок 2.24 – Діаграма виконання ниток (M:1) та (1:1)

При організації ниток на рівні користувача (2) період часу, що виділяється для виконання, використовується повністю – нитка 1 завершує своє виконання, але, оскільки виділений час ще залишився, тому починає своє виконання нитка 2. При досягненні обмеження, нитка переривається і передається управління іншим процесам. Через $t_{очікування}$ процес знов отримає управління, і нитка 2 продовжить своє виконання. Після завершення нитки 2 одразу почне виконання нитка 3 і встигне завершитись до вичерпання виділеного проміжку часу. Таким чином, час виконання всіх трьох ниток процеса становитиме $t_{процеса2}=6+t_{очікування}$ (у.о.), що менше, ніж для моделі з організацією ниток на рівні ядра.

Відмінність організації ниток на рівні користувача від використання звичайних функцій у програмі наведена на рис. 2.25 (використання звичайних функцій) та рис. 2.26 (використання волокон).

Виклики функцій майже завжди реалізуються через інструкцію *call*. Ця інструкція виконує два основні кроки: заноситься адреса точки повернення в пам'яті в стек (наступна команда за інструкцію *call*); адреса точки входу у функцію заноситься в регістр IP. Це означає, що коли запускається нова функція, значення на вершині стека є адресою повернення (адресою, до якої

потрібно перейти, щоб продовжити виконання попередньої функції). Для повернення з функції зазвичай використовується інструкція *ret*, яка витягує з вершини стека дані і використовує їх як адресу пам'яті для переходу.

Припустимо, що в програмі є три функції *f1()*, *f2()*, *f3()*, які викликаються так, як наведено на рис. 2.25. Цифри в колі біля стрілок позначають послідовність, в якій виконуються переходи між функціями. Під час виконання цих функцій виконується виведення: ABCABC2BC2BC…2BC…

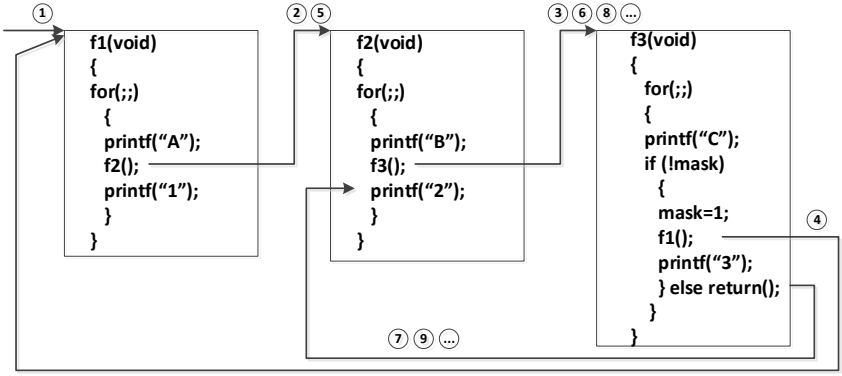


Рисунок 2.25 – Приклад використання функцій

Волокна в ОС Windows є симетричними (всередині процеса всі волокна еквівалентні – неможливо виділити головне волокно). Перемикатись можна тільки між волокнами (для цього головну нитку процеса необхідно явно конвертувати у волокно). Якщо виконується вихід з будь-якого волокна – процес завершується. На відміну від функцій, кожне волокно має свій власний стек (так само, як і нитка). Виклики для перемикання між волокнами працюють майже так само, як і виклики *call* та *ret* (власне вони і виконуються парами при перемиканнях): для поточного волокна виконується занесення адреси точки повернення в пам'яті в стек поточного волокна (адреса наступної інструкції після виклику перемикання), далі виконується перемикання стека на нове волокно, та за допомогою інструкції *ret* витягується з вершини стека адреса пам'яті для переходу (це завжди буде

наступна інструкція кода, яка розташована одразу після інструкції перемикання між волокнами). На рис. 2.26 наведено поведінку програми з трьома волокнами *f1()*, *f2()*, *f3()*, які виконують такі самі дії, як і в попередньому випадку. У результаті виконання буде виведено наступну послідовність символів: ABC1A2B3C, після чого роботу буде завершено (перехід 7), тому що вихід з будь-якого волокна (в нашому випадку – використовується *return*) приводить до завершення всього процеса.

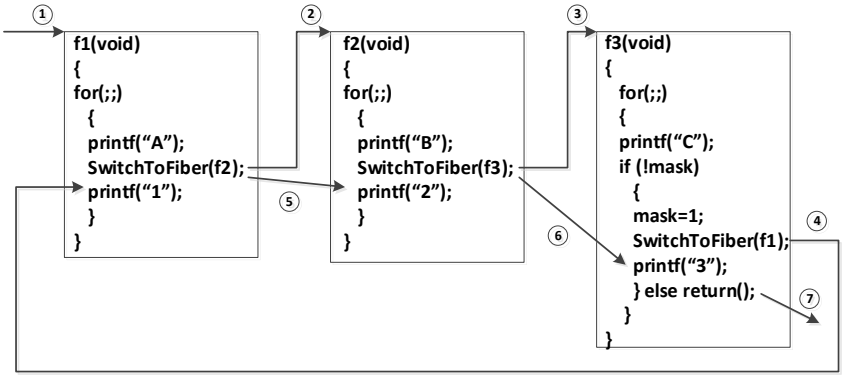


Рисунок 2.26 – Приклад використання волокон

У той час, як модель «багато до одного» (M:1) дозволяє розробнику створювати скільки завгодно ниток користувача, вона не призводить до збільшення паралелізму, оскільки ядро може планувати лише одну нитку ядра за раз. Крім того, якщо одна нитка з процеса виконує блокуючий виклик, то буде заблоковано весь процес. Модель «один до одного» (1:1) забезпечує більший паралелізм, але розробник має бути обережним, щоб не створювати занадто багато ниток у програмі, в ОС існують обмеження на максимальну кількість підтримуваних ниток у режимі ядра.

Модель «багато до багатьох» (M:N) (або ще має назву – «дворівнева модель») є комбінованим підходом, вона прибирає негативні ефекти двох інших моделей: немає необхідності в ОС вводити обмеження на загальну кількість ниток (розробники можуть створювати скільки завгодно ниток користувача), а відповідні нитки ядра можуть виконуватися паралельно

(число ниток ядра може бути специфічним для конкретної програми або конкретної архітектури обчислювальної системи: програмі може бути виділено кілька ниток ядра на багатопроцесорній системі і одна на однопроцесорній). Крім того, коли нитка виконує блокуючий системний виклик, ядро може запланувати виконання іншої нитки. Схематичне зображення моделі наведено на рис. 2.27.

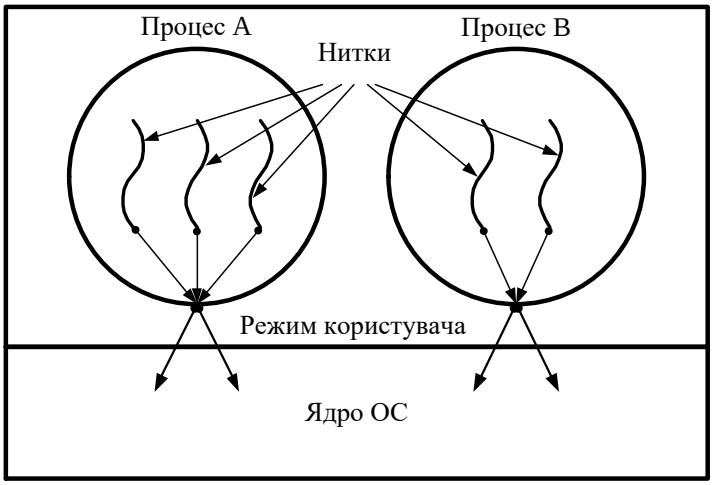


Рисунок 2.27 – Модель процесів з комбінованою організацією ниток

В ОС Solaris (до версії 9) реалізовано модель відображення ниток на процесори «багато до багатьох» (M:N). Нитки на рівні користувача за допомогою бібліотеки ниток *T1* (починаючи з 9 версії використовується бібліотека *T2*) відображуються в таке ж (M) або менше число (N) ниток на рівні ядра. При цьому число ниток на рівні ядра є змінним параметром, що дозволяє оптимізувати роботу програми. Середовище Java на Solaris є першою комерційною реалізацією Java у форматі «багато до багатьох» в ОС Solaris з багатонитковим ядром MT (англ. Multithreading Kernel).

На рис. 2.28 наведено реалізацію моделі «багато до багатьох» (M:N) в ОС Solaris. Нитки існують на двох рівнях: користувача (нитки ULT) та на рівні ядра (нитки KLT) – на рис. 2.28 позначені як KT1-KT10. Коло, в яке

сходяться лінії від ниток KLT – планувальник ниток ядра, він розподіляє, на яких процесорах (P1-P4) будуть вони виконуватись. На рівні користувача для кожного процеса (Процес1-Процес5) за допомогою бібліотеки ниток користувача T1 виконується планування ниток ULT. У результаті планування формуються полегшені процеси LWP (L1-L9). Нитки ULT не обов'язково видимі для ядра ОС: ядро розглядає полегшені процеси LWP як сутності, які можна планувати. Кожен процес LWP пов'язаний з ниткою ядра KLT, але не кожна нитка ядра пов'язана з LWP. Наприклад, це може бути служба ядра – «демон» (нитка ядра KT2).

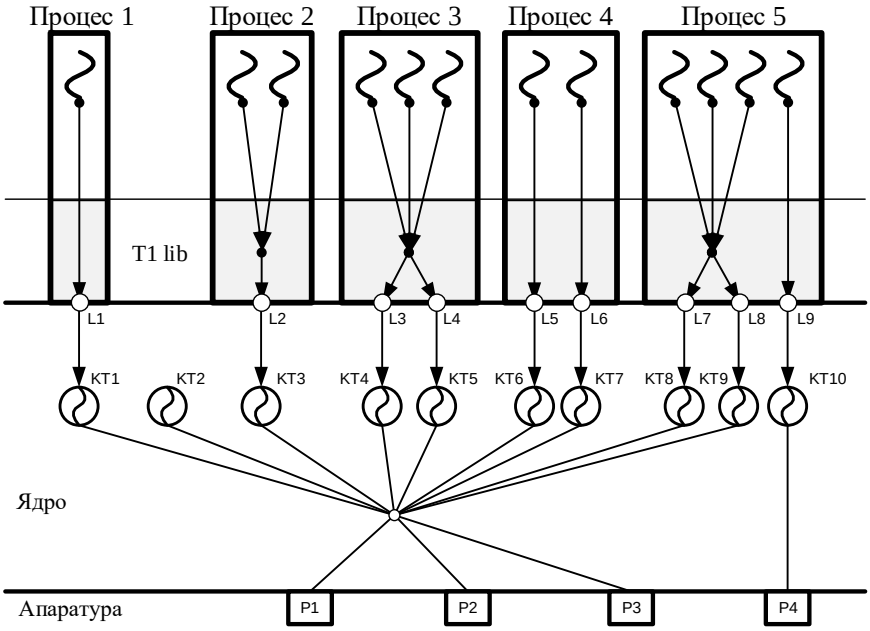


Рисунок 2.28 – Організація ниток в ОС Solaris

Процес може паралельно виконуватись лише на такій кількості процесорів, скільки він має LWP, відповідно в ядрі він представлений саме такою кількістю ниток KLT. Процеси 1 і 4 демонструють класичну модель (1:1), процеси 2 і 3 – модель (M:N), процес 5 є гібридним: він містить кілька

ниток, запланованих відповідно до моделі (M:N), і одну нитку відповідно до моделі (1:1), яка, до речі, також має жорстке закріплення до процесора (P4).

Хоча модель «багато до багатьох» виглядає найбільш гнучкою з трьох варіантів, на практиці її важко реалізувати. Крім того, зі збільшенням кількості процесорних ядер у більшості систем обмеження кількості ниток ядра стало менш важливим. У результаті більшість ОС зараз використовують модель «один до одного».

2.5.2. Програмування ниток

При розробці програм слід прийняти рішення, чи виконувати розбиття програми на нитки, чи ні. Використання ниток обґрунтовано в таких випадках:

♦ процес виконує тривалі обчислення у фоновому режимі. Головна нитка повинна продовжувати виконання, у той час як інша нитка виконує фонову задачу. У віконних інтерактивних застосунках, якщо головна нитка зайнята тривалими обчисленнями, вона не може обробляти повідомлення клавіатури і миші, і застосунок перестає відгукуватися на дії користувача. З цієї причини слід програмувати довготривалі обчислення в окремих нитках, навіть якщо головна нитка в цей час тільки очікує певних подій (наприклад, завершення тих самих обчислень) і не виконується. Таке рішення гарантує, що процес не буде позначений ОС таким, що не відповідає на зовнішні події;

♦ якщо в процесі є очікування відповіді від іншого процеса на локальному або віддаленому комп'ютері (сервера застосунків, сервера баз даних або клієнта). Запуск процедури очікування в окремій нитці означає, що головна нитка негайно звільняється для виконання інших дій, не пов'язаних з очікуванням;

♦ використання інтенсивних обчислень. Такі методи можуть виконуватися швидше на багатопроцесорних комп'ютерах, якщо робоче навантаження рознесено по декількох нитках.

Однак використання ниток не завжди є доцільним:

♦ використання ниток призводить до значного підвищення складності програм. Складність збільшують не додаткові нитки самі по собі, а необхідність організації їх взаємодії. Від цього залежить як тривалість циклу

розробки програм, так і складність їх налагодження (зростає кількість важковловимих помилок);

♦ надмірне використання багатонитковості забирає ресурси і процесорний час на створення ниток і перемикання між ними. Зокрема, коли використовуються операції читання або запису на диск, більш швидким може виявитися послідовне виконання операцій в одній або двох нитках, ніж одночасне їх виконання в багатьох нитках.

Ще одне обмеження використання великої кількості ниток пов'язане з тим, що реальну паралельність можна досягти тільки при наявності кількох процесорів. Але збільшення кількості процесорів у кілька разів не приводить до збільшення швидкості виконання процесів при їх (або множини ниток) паралельному виконанні в таку ж кількість разів. Ця особливість має назву – Закон Амдала, який визначає потенційне збільшення продуктивності від додавання додаткових обчислювальних ядер процеса, який має як послідовні, так і паралельні компоненти. Якщо S є частиною програми, яка повинна виконуватися послідовно в системі з N процесорними ядрами, формула виглядає так:

$$\text{Прискорення} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Згідно цієї формули можна розрахувати прискорення при збільшенні кількості процесорних ядер. Наприклад, є програма, яка на 75 % є паралельною та на 25 % послідовною. При запуску програми в системі з двома процесорними ядрами можна отримати прискорення в 1,6 рази. Якщо додати ще два ядра, прискорення буде лише в 2,28 рази.

Основний принцип закону Амдала: послідовна частина програми може мати непропорційний вплив на продуктивність при додаванні додаткових обчислювальних ядер. Так, якщо S наближається до нескінченності, прискорення збігається до $1/S$. Наприклад, якщо 50 % програми виконується

послідовно, максимальне прискорення становить два рази, незалежно від кількості процесорних ядер.

При програмуванні програми з множинними нитками також необхідно забезпечити багатониткову безпеку функцій (англ. Thread Safety). Програми, що виконуються через множину процесів, не мають таких вимог. Одна нитка може пошкодити інші, оскільки нитки ділять загальний адресний простір. Процеси більш ізольовані друг від друга. Відповідно, функція називається безпечною (англ. Thread-safe), якщо може безпечно викликатися кількома нитками одночасно. Існують також реентерабельні (англ. Reentrant) функції, розроблені таким чином, що кілька її викликів можуть безпечно виконуватися одночасно. Це важливо навіть у однониткових програмах, оскільки виконання в будь-який момент може бути перервано обробкою сигналу.

Для ОС, які підтримують нитки, при розробці програм кожен процес має основну, або первинну, нитку, яка запускається першою. У мові С функція основної нитки завжди носить ім'я *main()* (якщо не перевизначена точка входу в програму). Кожна нитка має свою послідовність інструкцій, що виконується незалежно від інших ниток і, можливо, паралельно з ними.

Нитки, породжені всередині одного процеса, рівноправні і існують в єдиному адресному просторі, розділяючи ресурси (дескриптори файлів, глобальні змінні тощо). Вони можуть створювати, припиняти і завершувати інші нитки, при цьому витрати обчислювальних ресурсів ОС, пов'язаних із створенням нитки, її підтримкою і управлінням, значно нижче порівняно з аналогічними витратами для створення, підтримки та управління процесом. Для управління нитками використовуються спеціальні засоби, які доступні програмісту через відповідні мовні конструкції, бібліотеки або системні виклики ОС. Наприклад, бібліотека ниток *Pthread*[75] визначає об'єкт атрибутів нитки, інкапсулює властивості нитки, до яких творець об'єкту

[75] Стандарт IEEE Std 1003.1c™-1995 IEEE Standard for Information Technology--Portable Operating System Interface (POSIX(R)) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)

може отримати доступ і модифікувати їх. Об'єкт атрибутів нитки визначає наступні компоненти: область видимості, розмір стека, адресу стека, пріоритет, стан, стратегію планування, параметр «відкріплення». Область видимості визначає, з якими нитками конкретна нитка конкурує за володіння системними ресурсами – з нитками тільки свого процеса або всіма нитками системи. Об'єкт атрибутів нитки може бути пов'язаний з однією або кількома нитками, тобто всі нитки, що використовують один об'єкт атрибутів, набувають всі властивості, визначені цим об'єктом. Після того як об'єкт атрибутів створено і ініціалізовано, його можна використовувати в будь-яких зверненнях функції створення ниток. Для встановлення і зчитування значень атрибутів передбачені спеціальні методи об'єкта атрибутів.

Pthread визначає набір типів та функцій мовою програмування С, серед типів основні такі:

- *pthread_t* – ідентифікатор нитки;
- *pthread_mutex_t* – м'ютекс;
- *pthread_mutexattr_t* – об'єкт атрибутів м'ютексу;
- *pthread_cond_t* – умовна змінна;
- *pthread_condattr_t* – об'єкт атрибуту умовної змінної;
- *pthread_key_t* – дані, специфічні для нитки;
- *pthread_once_t* – контекст контролю динамічної ініціалізації;
- *pthread_attr_t* – список атрибутів нитки та ін.

Для створення нитки використовується функція *pthread_create()*. Виклик цієї бібліотечної функції приводить до генерування системного виклику *clone*. Перед її викликом необхідно створити функцію, з якої буде створюватись нитка або нитки. Після створення нитки, функція, що її створила, продовжує виконувати свої наступні дії паралельно нитці.

Завершення ниток виконується в таких випадках:

- функція нитки виконує *return* та повертає результат обчислень;
- у результаті виклику завершення виконання нитки *pthread_exit()* (системний виклик *exit*);
- внаслідок виклику скасування нитки *pthread_cancel()*;

♦ одна з ниток здійснює виклик *exit()*;

♦ основна нитка функції *main()* виконує *return*, і в такому випадку всі нитки процеса також завершуються.

Функція *pthread_join()* (реалізована через системний виклик *wait4*) очікує завершення нитки. Якщо нитку вже завершено, то функція негайно повертає значення. Основне призначення функції – синхронізація ниток. Функція *pthread_join()* схожа на виклик *waitpid()*, але з деякими відмінностями. По-перше, всі нитки однорангові, у них відсутній ієрархічний порядок, тоді як процеси утворюють дерево і підпорядковані ієрархії «батько-нащадок». По-друге, не можна дати вказівку одній нитці очікувати завершення будь-якої іншої нитки – обов'язково треба вказати, яку саме нитку чекати. Також неможливо здійснити виклик *pthread_join()* без блокування поточної нитки.

На рис. 2.29 наведено схему використання функцій *pthread_create()*, *pthread_exit()* та *pthread_join()*.

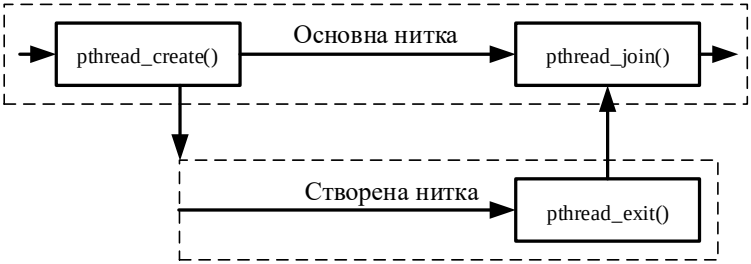


Рисунок 2.29 – Створення, завершення та очікування нитки в бібліотеці Pthread

Функція *pthread_cancel()* використовується для дострокового завершення нитки. Але, незважаючи на те, що *pthread_cancel()* повертається одразу і може завершити нитку достроково, її не можна назвати засобом примусового завершення ниток. Справа в тому, що нитка не тільки може самостійно вибрати момент завершення у відповідь на виклик *pthread_cancel()*, а й зовсім його ігнорувати. Виклик функції *pthread_cancel()* слід розглядати як запит виконання дострокового завершення нитки. Тому,

якщо важливо, щоб нитка була видалена, потрібно дочекатися її завершення функцією *pthread_join()*.

До будь-якої нитки за замовчуванням можна приєднатися викликом *pthread_join()* і очікувати на її завершення. Однак, у деяких випадках все, що потрібно – це завершити нитку та автоматично вивантажити ресурси назад у розпорядження ОС. У таких випадках використовується функція від'єднання *pthread_detach()* (якщо завершення нитки не перехопити викликом *pthread_join()*, то нитка після завершення перейде в стан «зомбі», від'єднання нитки не приводить до такої ситуації). Від'єднану нитку вже не перехопити за допомогою виклику *pthread_join()*, також не можна скасувати від'єднаний стан.

Текст прикладу демонструє роботу з бібліотекою ниток *Pthread* в ОС Linux згідно з рис. 2.29:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* func(void* arg)
{
  char* str = (char*)arg;
  printf("%s \n", str);
  // вихід з нитки
  pthread_exit(NULL);
}

int main()
{
  pthread_t ptid;
  // створення нитки
  pthread_create(&ptid, NULL, &func, "New thread");
  // очікування завершення створеної нитки
  pthread_join(ptid, NULL);
  return 0;
}
```

Детальну інформацію щодо використання бібліотеки *Pthread* у програмуванні ниток для ОС Linux можна знайти в темі «Нитки і семафори ниток в ОС Linux» навчального посібника «Системне програмне забезпечення. Програмування системних механізмів ОС» [46].

В ОС сімейства Windows нитка є об'єктом ядра. Кожен об'єкт ядра – це блок пам'яті, виділений ядром і доступний тільки ядру. Цей блок є структурою даних, в елементах якої міститься інформація про об'єкт (атрибути нитки). Об'єкт ядра «нитка» містить атрибути, які присутні у всіх об'єктах ядра (ім'я об'єкта, дескриптор захисту, лічильник числа користувачів), і атрибути, характерні тільки для об'єкта «нитка» (ідентифікатор процеса, якому вона належить, базовий пріоритет, код завершення).

Будь-яка проста багатониткова програма повинна складатися з основної нитки і функцій, в яких міститься код інших ниток. Принцип, за яким створюються нитки, визначає модель створення і управління ниток. У той же час нитки – це виконувані частини програми, які змагаються між собою за використання процесора, пам'яті та інших ресурсів. Для передачі управління іншій нитці на рівні системи використовується процедура перемикання контексту (по аналогії з процесами).

Операції для роботи з нитками подібні до операцій для роботи з процесами, але є деякі відмінності. Розглянемо операції створення та завершення ниток (для моделі з реалізацією ниток на рівні ядра ОС).

При створенні нитки ядро має виконати наступні дії:

♦ створити дескриптор нитки і розмістити його в таблиці ниток;

♦ створити інформаційні структури, необхідні для функціонування нитки на даній апаратній архітектурі (наприклад, стек нитки);

♦ ініціалізувати значення полів загального призначення дескриптора нитки, виходячи з параметрів виклику або прийнятих значень за замовчуванням;

♦ ініціалізувати поле дескриптора «апаратний контекст виконання нитки» на підставі параметрів виклику або у відповідності до значень за замовчуванням (наприклад, встановити покажчик стека на верхню межу стека, а покажчик команд – на вхідну точку нитки);

♦ сповістити підсистеми, які беруть участь в управлінні нитками, про створення нової нитки з метою завершення ініціалізації її дескриптора;

♦ перевести нитку в стан «готова до виконання».

При завершенні нитки ядро ОС має виконати наступні дії:

♦ зберегти статистичні дані нитки і код повернення в її дескрипторі;

♦ перевести всі ресурси, що належать нитці в несуперечливий і стабільний стан;

♦ звільнити всі ресурси, що належали або використовувалися ниткою;

♦ сповістити підсистеми, які беруть участь в управлінні нитками, про завершення нитки з метою коригування ними її дескриптора;

♦ стан нитки встановити в значення «завершена»;

♦ якщо дана нитка є останньою активною ниткою в процесі – завершити процес.

У класичних ОС класу UNIX (наприклад, в ОС Solaris) для створення нитки може використовуватися системний виклик *thr_create()*; у Windows – функції API *CreateThread(), CreateThreadEx()* (відповідні системні виклики *NtCreateThread, NtCreateThreadEx*). В ОС UNIX для завершення нитки використовується системний виклик *thr_exit()*; у Windows – функція API *ExitThread()* (системний виклик *NtTerminateThread*). Для того, щоб завершити нитку ззовні, в UNIX використовується виклик *thr_kill()*, у Windows – *TerminateThread()* (також системний виклик *NtTerminateThread*).

Наступний приклад демонструє створення, вихід та приєднання ниток для ОС Solaris:

```
#include <stdio.h>
#include <thread.h>

void *func(void * Junk)
{
  int *valuePTR = (int *) Junk; /* перетворення в int ptr */
  int value = *valuePTR;        /* отримання значення */
  printf("Thread: value = %d\n", value);
  thr_exit(NULL);
}
```

```
void main(void)
{
  thread_t  ID_t;
  size_t    Status;
  int       v1 = 1;
  thr_create(NULL, 0, func, (void *) (&v1), 0, &ID_t);
  thr_join(ID_t, 0, (void *) &Status);
}
```

Наступний приклад демонструє роботу з нитками в ОС Windows (через функції Windows API):

```
#include <windows.h>

unsigned ThreadFunc(void* arg)
{
        char** str = (char**)arg;
        MessageBox(0, str[0], str[1], 0);
        ExitThread(0);
}
int main(void)
{
        const char* InitStr[2] = { "First","Thread" };
        unsigned long uThreadID;
        HANDLE hThread;
        hThread = CreateThread(NULL, 0,
                (LPTHREAD_START_ROUTINE)ThreadFunc,
                InitStr, 0, &uThreadID);
        // Очікування завершення роботи нитки
        WaitForSingleObject(hThread, INFINITE);
        // Закриття дескриптору
        CloseHandle(hThread);
        return 0;
}
```

Детальну інформацію щодо роботи з нитками в ОС Windows можна знайти в темі «Процеси, нитки та волокна в ОС Windows» навчального посібника «Системне програмне забезпечення. Програмування системних механізмів ОС» [46].

2.6. Стани процесів (ниток) у деяких ОС

Враховуючи надану вище загальну інформацію про стани процесів, системні виклики та організацію ниток в ОС, слід навести уточнені стани, в яких можуть перебувати процеси або нитки, для конкретних ОС.

2.6.1. Стани ниток в ОС Windows

У сучасних версіях ОС Windows управління виконується на рівні ниток.

На рис. 2.30 наведено стани ниток та переходи між ними (номери позначають основні переходи між станами).

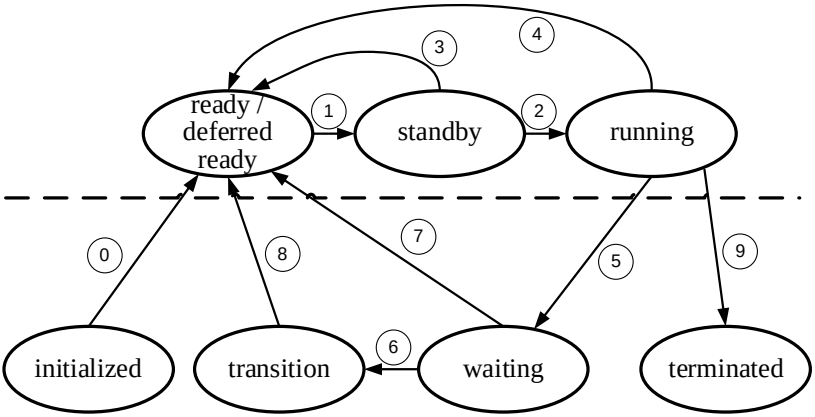


Рисунок 2.30 – Стани ниток в ОС Windows

Нитка в ОС Windows може знаходитись в одному з таких станів (у дужках вказані внутрішні номери станів):

♦ initialized (0) – початковий стан нитки. У системі створюється та ініціалізується об’єкт «нитка», після чого розміщується в черзі готових до виконання ниток (перехід 0);

♦ ready (1) – нитка в стані готовності очікує на виконання. Під час пошуку нитки для виконання диспетчер процесорного часу розглядає лише нитки в стані готовності, обрані нитки переводяться в стан «standby» (перехід 1);

♦ deferred ready (7) – цей стан використовується для ниток, які було обрано для виконання на певному процесорі, але вони фактично ще не запускалися на ньому. Зазвичай планувальник розміщує нитку на будь-який доступний процесор. Програміст може вказати маску спорідненості процесорів (англ. Processor Affinity Mask) для нитки, щоб вказати, на якому процесорі нитка може виконуватися. Використовуючи цей метод, програміст може розподіляти процесори між нитками. Відповідними функціями Windows API є *SetProcessorAffinityMask()* та *GetProcessorAffinityMask()*. Функція *SetThreadIdealProcessor()* дозволяє вказати кращий процесор, що підлягає використанню планувальником за першої ж можливості. Стани «ready» та «deferred ready» на рис. 2.30 зображені як один стан, оскільки стан «deferred ready» використовується як тимчасовий стан для підпрограм планування процесорного часу;

♦ standby (3) – нитку в цьому стані було обрано для наступного виконання на певному процесорі та диспетчер готовий до виконання перемикання контексту на цю нитку. Для кожного процесора в системі в стані «standby» може бути тільки одна нитка. Нитка може бути витіснена зі стану «standby» до стану «ready» (перехід 3), якщо, наприклад, нитка з вищим пріоритетом стає готовою до виконання до того, як почнеться виконання раніше обраної нитки. Перебування ниток у цьому стані дуже короткочасне;

♦ running (2) – після того, як диспетчер виконує перемикання контексту на нитку, вона переходить у стан виконання (перехід 2). Виконання нитки продовжується, доки виділений їй квант не закінчиться (перехід 4), і інша нитка з таким самим пріоритетом не буде обрана до виконання. Нитка може бути витіснена іншою ниткою з вищим пріоритетом (перехід 4) або добровільно залишити стан «running» за допомогою виклика функції *Sleep(0)* – перехід 4; вона може завершитись (перехід 9) або перейти в стан очікування «waiting» (перехід 5);

♦ waiting (5) – нитка може перейти в стан очікування двома способами: може добровільно чекати об’єкт синхронізації або підсистема ОС може призупинити нитку. Коли очікування нитки закінчується, залежно від її

пріоритету, нитка або починає працювати негайно (низка переходів 7-1-2), або повертається в стан готовності (перехід 7);

♦ transition (6) – нитка переходить у цей стан, якщо вона готова до виконання, але її стек ядра вивантажено з пам'яті (перехід 6). Після повернення стека ядра в пам'ять нитка переходить у стан готовності (перехід 8);

♦ terminated (4) – коли нитка завершує виконання, вона переходить у цей стан. Після завершення об'єкт нитки (структура даних у системній пам'яті, яка описує нитку) може бути звільнена або ні – менеджер об'єктів встановлює політику щодо часу видалення об'єкта. Нитка також може перейти в стан завершення з інших станів, якщо її явно закрито іншою ниткою, наприклад, викликом функції Windows API *TerminateThread()*.

Горизонтальна лінія на рис. 2.30 поділяє стани на придатні для виконання (англ. Runnable) – зверху, та непридатні для запуску (англ. Not Runnable) стани.

Не існує способу, який би дозволив застосунку визначити стан іншої нитки. Навіть якби такий спосіб і існував, стан нитки може змінитися ще до того, як нитка, яка отримає ці дані, встигне виконати будь-які дії у відповідь на отриману інформацію.

2.6.2. Стани процесів в ОС UNIX

На рис. 2.31 наведено діаграму переходів процесів між станами:

♦ created – процес створюється батьківським процесом за допомогою системного виклику *fork* (перехід 0), при виконанні цього виклику процес переводиться в початковий стан – «created». Процес існує, але не готовий до виконання, хоч і не припинений. Цей стан є початковим для всіх процесів за винятком процеса з ідентифікатором 0 (процес *swapper* – підкачування в пам'ять – частина ядра ОС, що постійно виконує процедуру *sched()* – єдиний процес у системі, який створюється без використання виклику *fork*; наступний процес – це процес ініціалізації *init*, який має ідентифікатор 1);

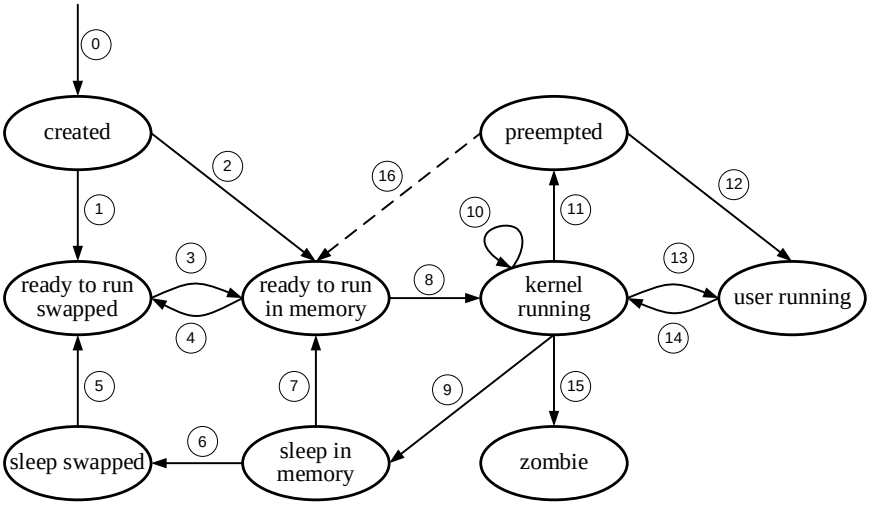


Рисунок 2.31 – Стани процесів в ОС UNIX

♦ ready to run, swapped – процес готовий до запуску, але swapper (процес 0) має завантажити процес в оперативну пам'ять, перш ніж він буде запущений під управлінням ядра. У цей стан процес попадає, якщо для його розміщення недостатньо місця в оперативній пам'яті (перехід 1). Також процес може бути вивантажено з оперативної пам'яті (переходи 4 та 5);

♦ ready to run in memory – процес не виконується, але готовий до запуску під керуванням ядра. У цей стан процес потрапляє, коли він завантажений в оперативну пам'ять (переходи 2, 3, 7). Планувальник обирає процеси з цього стану для виконання (перехід 8);

♦ sleep, swapped – процес призупинено і swapper вивантажив його в зовнішню пам'ять, щоб в оперативній пам'яті звільнити місце для інших процесів (перехід 6);

♦ sleep in memory – процес призупинено і він перебуває в оперативній пам'яті. У цей стан процес потрапляє (перехід 9), коли йому необхідно дочекатися виконання певної зовнішньої операції (наприклад, введення-виведення). У цьому стані процес перебуватиме доти, доки не отримає повідомлення про закінчення операції. Коли зовнішня операція завершиться,

відбудеться апаратне переривання роботи ЦП і процес буде переведено до стану «ready to run in memory» (перехід 7);

♦ kernel running – процес виконується в режимі ядра. У цей стан процес завжди потрапляє на початку своєї роботи (перехід 8). Після цього процес переходить у режим користувача (перехід 13). При перериванні роботи процесора та подальшому поверненню з переривання процес може здійснити перехід 10 – продовжити роботу в режимі ядра;

♦ preempted – коли процес повертається з привілейованого режиму (режиму ядра) до непривілейованого (режим користувача), ядро резервує його та перемикає контекст на інший процес (перехід 11). Стан «preempted» насправді не відрізняється від стану «ready to run in memory», але вони виділяються в окремі стани, щоб підкреслити, що процес, що виконується в режимі ядра, може бути зарезервований тільки в тому випадку, якщо він збирається повернутися в режим користувача. За певних умов планувальник обирає процес зі стану «preempted» для виконання і той знову повернеться в стан «kernel running» (переходи 16 та 8), в іншому випадку процес буде переведено в стан «user running» (перехід 12);

♦ user running – процес виконується в режимі користувача. У разі звернення до механізмів ядра (наприклад – системний виклик), процес може бути знову повернений з режиму користувача в режим ядра (перехід 14);

♦ zombie – процес викликає системну функцію *exit* та припиняє існування (перехід 15). Якщо процес знаходився в стані «user running», то він спочатку переводиться в стан «kernel running» (перехід 14). Однак після цього залишається деяка інформація (код виходу і статистичні дані) у таблиці процесів, яка може бути отримана батьківським процесом. Цей стан є останнім станом процеса.

Процес може керувати деякими з переходів на рівні користувача. По-перше, один процес може створити інший процес. Але, в який саме стан створений процес перейде («ready to run, swapped» або «ready to run in memory») залежить від ядра. По-друге, процес може звернутися до різних системних функцій, щоб перейти зі стану «user running» у стан «kernel running». Проте момент повернення з режиму ядра від процеса також не залежить. Нарешті, процес може завершитися за допомогою виклику *exit* за

власним бажанням, але зовнішні події можуть вимагати завершення процеса без явного звернення до виклику *exit*. Усі інші переходи відносяться до жорстко закріпленої частини моделі, закодованої в ядрі.

2.6.3. Стани процесів в ОС Linux

На рис. 2.32 наведено діаграму станів процесів в ОС Linux (у дужках вказано внутрішнє позначення станів).

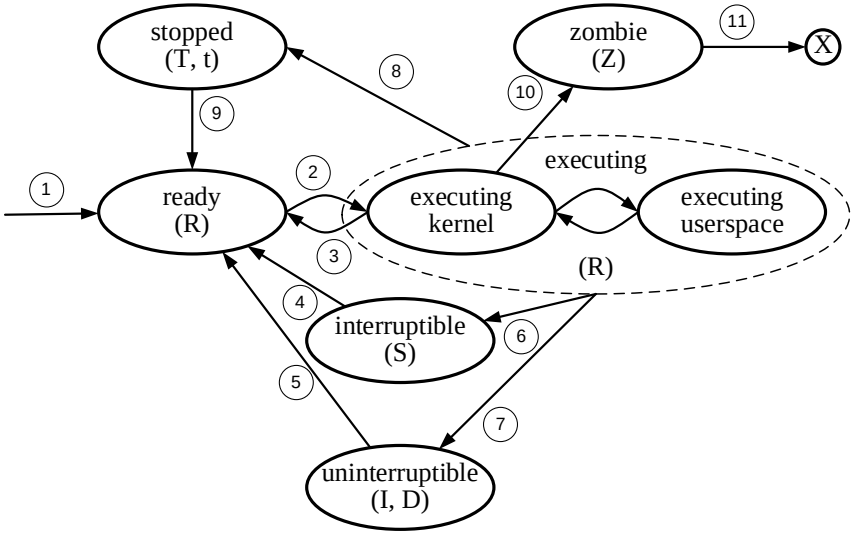


Рисунок 2.32 – Стани процесів в ОС Linux

Для будь-якого процеса Linux початковою точкою є момент його створення. Наприклад, батьківський процес може ініціювати дочірній процес за допомогою системного виклику *fork*. Після запуску процес переходить у стан «Running (Runnable)» (перехід 1).

Видимі ззовні коди станів процеса в Linux згорнуті в один символ. Так, різниця між «очікуванням» і «запущеним» процесом розмита, оскільки процеси виконуються в короткі проміжки часу, так що для користувача немає великої різниці між процесом, готовим до запуску, і процесом, що виконується. Крім того, вивантажуються в зовнішню пам’ять не цілі

процеси, а окремі сторінки пам'яті, тому немає станів, що відображаються як «Swapped out and waiting» або «Swapped out and blocked».

Існують наступні головні стани процесів:

♦ running або runnable (R) – TASK_RUNNING. Процес виконується або очікує на виконання. У запущеному стані процес займає ядро ЦП для виконання свого коду. Внутрішньо ядро розрізняє запущені процеси та процеси, що очікують виконання, але це не відображається через код стану процеса. Хоча стан очікування на виконання і запущений стани відрізняються, вони разом згруповані в один стан, позначений символом R. Переходи 2 і 3 реалізуються планувальником на доступних процесорах. Процес може працювати в режимі користувача або в режимі ядра;

♦ interruptible (S) – TASK_INTERRUPTIBLE. Процес очікує на виконання умови, наприклад, доступ до ресурсу або сигналу (таким сигналом є, наприклад, завершення дочірнього процеса). Цей стан можна перервати за допомогою виклику *kill*. У цей стан приводить (перехід 6) більшість системних викликів, які пов'язані з очікуванням певної події (*nanosleep, select, poll, wait4*, та ін.). Після виконання умови процес переходить до стану R (перехід 4);

♦ uninterruptible (D) – TASK_UNINTERRUPTIBLE. Процес очікує виконання умови, але не реагує на сигнали. Стан використовується, коли переривання або завершення процеса може перевести пристрої або процес у невизначений стан. Зазвичай використовується під час операцій введення/виведення (перехід 7). Процес переходить у цей стан також, коли його сторінки пам'яті вивантажуються в зовнішню пам'ять. Після виконання умови процес переходить до стану R (перехід 4). Також у цьому стані знаходиться нитка ядра, яка не має нічого робити та заблокована в очікуванні нової роботи. Цей стан – idle (I), визначено як TASK_IDLE = TASK_UNINTERRUPTIBLE | TASK_NOLOAD – такий самий, як D (оскільки зазвичай нитки ядра не перериваються), він був введений з облікових причин, оскільки процеси в стані D вважаються такими, що навантажують систему;

♦ stopped (T) – TASK_STOPPED. Процес був зупинений (перехід 8) сигналом на зразок SIGSTOP (процес не може ігнорувати) або SIGTSTP

(процес може ігнорувати цей сигнал і продовжувати виконуватися після його отримання). Процес можна відновити іншим сигналом – SIGCONT (перехід 9). Цей стан не зовсім відповідає теоретичному стану «заблокований», оскільки процес не чекає на подію сам по собі, а зазвичай блокується від подальшого виконання втручанням іншого процеса або користувача (наприклад, натискання комбінації клавіш Ctrl+Z). Також да цього стану відноситься аналогічний стан «traced (t)» – TASK_TRACED. Процес блокується від планування налагоджувачем або процесом трасування, а не очікуванням події – його тимчасово призупинено, щоб перевірити його статус;

♦ zombie (Z) – EXIT_ZOMBIE. Процес завершується (перехід 10), а запис у списку процесів існує лише для того, щоб батьківський процес міг збирати інформацію про статус завершення. Коли процес завершує своє виконання, він надсилає сигнал SIGCHLD батьківському процесу та переходить у стан «зомбі». Процес залишатиметься в цьому стані, доки батьківський процес не видалить його з таблиці процесів. Щоб видалити завершений дочірній процес із таблиці процесів, батьківський процес повинен прочитати вихідне значення дочірнього процеса за допомогою системних викликів очікування (*wait4, waitpid*);

♦ dead (X) – EXIT_DEAD. Процес завершено. Вивільняються всі ресурси процеса (перехід 11). За звичайних обставин цей стан не повинен бути видимим. Цей стан з'являється лише тоді, коли ядро чистить запис процеса. На багатопроцесорній системі цей стан інколи можна побачити за допомогою утиліти *ps*, якщо вона виконується на одному процесорі, а ядро в цей час видаляє інформацію про процес на іншому.

У версіях Linux до 1.1.30 процес міг також знаходитись у стані W – TASK_SWAPPING. У цьому стані процес заблоковано в очікуванні зчитування сторінки пам'яті з зовнішньої пам'яті в оперативну пам'ять. Починаючи з версії 2.3.43 більше немає можливості перевести процес у цей стан, а з версії 2.5.50 усі посилання на цей стан було видалено з системи.

Опис станів процесів знаходиться у файлі */include/linux/sched.h*. Фрагмент змісту для версії ядра 6.9.3:

```
/* Used in tsk->__state: */
#define TASK_RUNNING                    0x00000000
#define TASK_INTERRUPTIBLE              0x00000001
#define TASK_UNINTERRUPTIBLE            0x00000002
#define __TASK_STOPPED                  0x00000004
#define __TASK_TRACED                   0x00000008
/* Used in tsk->exit_state: */
#define EXIT_DEAD                       0x00000010
#define EXIT_ZOMBIE                     0x00000020
#define EXIT_TRACE                      (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->__state again: */
#define TASK_PARKED                     0x00000040
#define TASK_DEAD                       0x00000080
#define TASK_WAKEKILL                   0x00000100
#define TASK_WAKING                     0x00000200
#define TASK_NOLOAD                     0x00000400
#define TASK_NEW                        0x00000800
...
#define TASK_FROZEN                     0x00008000
#define TASK_STATE_MAX                  0x00010000
#define TASK_ANY                        (TASK_STATE_MAX-1)
/* Convenience macros for the sake of set_current_state: */
#define TASK_KILLABLE           (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED            (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED             __TASK_TRACED
#define TASK_IDLE               (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)
/* Convenience macros for the sake of wake_up(): */
#define TASK_NORMAL    (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
```

Для порівняння – зміст для версії ядра 1.1:

```
#define TASK_RUNNING         0
#define TASK_INTERRUPTIBLE   1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_ZOMBIE          3
#define TASK_STOPPED         4
#define TASK_SWAPPING        5
```

2.6.4. Стани ниток в ОС Solaris

Нитка ядра в Solaris – це об’єкт, який фактично ставиться в чергу для планування та диспетчеризації. Це є найбільш помітною відмінністю від традиційних реалізацій UNIX, де планування та диспетчеризація виконується над процесами. Саме нитці ядра KLT, а не процесу, призначається клас планування та пріоритет (хоча в Solaris 10 KLT та LWP мають однакове значення ідентифікатора). На рис. 2.33 наведено діаграму переходів між станами для ниток ядра KLT в ОС Solaris.

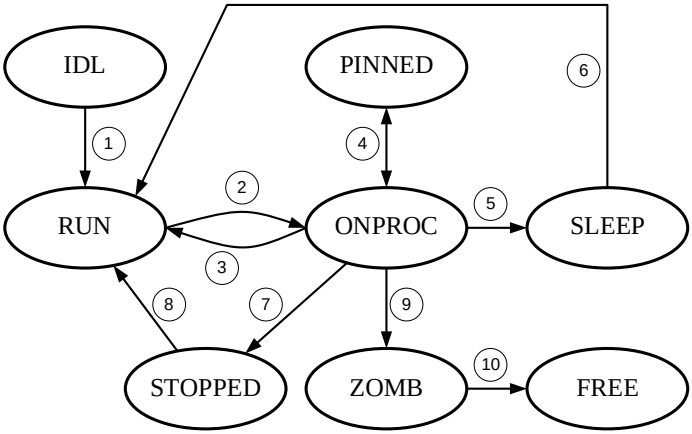


Рисунок 2.33 – Стани ниток ядра в ОС Solaris

Структура, яка описує нитку ядра, знаходиться у файлі *usr/src/uts/common/sys/thread.h* та містить поле *t_state*, в якому вказується поточний стан нитки. У системі підтримуються наступні стани:

♦ IDL – стан, в якому фактично виконується створення нитки ядра під час системного виклику *fork*. Після створення нитка потрапляє в стан RUN (перехід 1);

♦ RUN (стан нитки TS_RUN) – готовність до виконання. Нитка готова для виконання, чекає на виділення процесора. Нитка переходить від ONPROC до RUN (перехід 3), якщо її витісняє нитка з вищим пріоритетом або через квант часу;

♦ ONPROC (стан нитки TS_ONPROC) – виконання. Нитка виконується на процесорі в результаті диспетчеризацій (перехід 2);

♦ SLEEP (стан нитки TS_SLEEP) – очікування події. Нитка переходить від ONPROC (перехід 5) до SLEEP, якщо вона заблокована і повинна очікувати події для повернення в стан RUN (перехід 6). Блокування відбувається, якщо нитка робить системний виклик і повинна чекати виконання системної служби;

♦ STOPPED (стан нитки TS_STOPPED) – нитку зупинено. Нитка переходить у стан STOPPED (перехід 7), якщо її процес зупинено; це може бути зроблено з метою налагодження. Повернення зі стану STOPPED відбувається в стан RUN (перехід 8);

♦ ZOMB (стан нитки TS_ZOMB). Нитка завершила виконання (перехід 9), але ще не видалена з системи;

♦ FREE (стан нитки TS_FREE). Структура опису нитки видалена (перехід 10);

♦ PINNED. Більшість ОС містять дві основні форми одночасної діяльності: процеси (або нитки) та переривання. Процеси співпрацюють один з одним і керують використанням спільної інформації за допомогою механізмів взаємного виключення і синхронізації. Переривання синхронізуються шляхом запобігання їх обробці протягом певного періоду часу. Solaris об’єднує ці дві концепції в одну модель, а саме – нитки ядра. Для цього процедури обробки переривань перетворюються в нитки ядра. Мотивація для перетворення переривань у нитки полягає в зменшенні накладних витрат. Коли виникає переривання, воно доставляється до певного процесора, а нитка, яка виконувалась на цьому процесорі, закріплюється – переводиться в стан PINNED (перехід 4). Закріплена нитка не може бути переміщена на інший процесор, її контекст зберігається і нитка призупиняється, доки не буде оброблено переривання. Потім процесор починає виконувати нитку переривання. Після завершення обробки переривання нитка повертається до виконання (в стан ONPROC).

При розгляді таблиці процесів можна побачити, що поле стану процеса підтримується в структурі процеса разом з ниткою ядра. Але під час виконання змінюється нитка ядра, а не процес. Здебільшого існує певний

зв’язок між станами, визначеними для процеса, і станами нитки ядра, як наведено в таблиці 2.3.

Таблиця 2.3 – Зв’язок між станами процеса та нитки ядра в ОС Solaris

Стани процеса	Стани нитки ядра	Опис
SIDL		Стан під час виконання виклику *fork*
SRUN	TS_RUN	Готовність до виконання
SONPROC	TS_ONPROC	Виконується на процесорі
SSLEEP	TS_SLEEP	Блокування
SSTOP	TS_STOPPED	Зупинка
SZOMB	TS_ZOMB	KLT або LWP завершено
	TS_FREE	Нитка чекає на видалення структур

Контрольні запитання

1. Поясніть відмінності в поняттях «програма» і «процес».
2. Які процеси відносяться до системних?
3. Які процеси називаються еквівалентними?
4. Наведіть приклади фонових процесів.
5. У чому сутність інтерактивних процесів?
6. Поясніть існуючі схеми породження процесів у сучасних ОС.
7. Наведіть приклади взаємозалежних (взаємопов’язаних) процесів.
8. Навіщо в сучасних ОС використовується пріоритет процеса?
9. Яким чином процес описується всередині ОС?
10. Надайте характеристику системам з використанням моделі процеса з двома станами.
11. Чим відрізняється системний виклик від функції API?
12. Які події в ОС призводять до створення процесів?
13. Яка інформація зберігається в контексті процеса?
14. Чим відрізняється перемикання контексту від перемикання процеса?
15. Назвіть переваги використання ниток на рівні користувача перед використанням ниток на рівні ядра.
16. Чому взаємодія між нитками одного процеса між собою ефективніше, ніж взаємодія окремих процесів?
17. Яка основна інформація міститься в БУП?

18. Чим відрізняються призупинені стани процеса від блокованого та готового стану?
19. Чим відрізняються моделі процесів з 2, 5 та 7 станами?
20. Що таке черга процесів? Які бувають черги процесів?
21. З чого складається життєвий цикл процеса?
22. Які механізми використовуються в системних викликах для переходу в режим ядра ОС?
23. Чим відрізняється API від ABI?
24. Чим відрізняється інструкція *sysenter* від *syscall*?
25. Що таке нативні функції в ОС Windows?
26. Чим відрізняються нативні функції в ОС Windows з префіксами *Zw* та *Nt*? Чим відрізняються процедури ядра в ОС Windows з такими ж префіксами?
27. Чим відрізняються виклики *fork* та *vfork*?
28. Для чого існує безліч варіантів функцій виконання програми в процесі, якщо всі вони зводяться до одного системного виклику *execve*?
29. Для чого в сучасних версіях ОС Windows залишили системний виклик *NtCreateProcessEx*?
30. Яким чином можна виконати дії з процесами в ОС Windows?
31. Чому не рекомендується використовувати Native API при розробці застосунків для ОС Windows?
32. Які ресурси нитки поділяють між собою?
33. Чим поведінка волокон відрізняється від поведінки функцій?
34. У чому полягає складність реалізації моделі ниток «багато до багатьох» (M:N)?
35. Яким чином реалізована робота з нитками в ОС Linux?
36. Порівняйте між собою діаграми станів процесів та ниток у розглянутих ОС. Які основні відмінності та чим вони викликані?
37. У чому особливість реалізації обробки переривань в ОС Solaris?

3. УПРАВЛІННЯ ПРОЦЕСОРНИМ ЧАСОМ

Планування процесорного часу є основою багатозадачних ОС. Завдяки можливості перемикання процесора між процесами ОС може виконувати обчислення більш ефективно.

Практично у всіх процесів чергуються періоди обчислень з операціями введення-виведення. Зазвичай процесор деякий час працює без зупинки, потім, наприклад, відбувається системний виклик на виконання операції введення-виведення. Після виконання системного виклику процесор знову виконує обчислення, поки йому, наприклад, не знадобляться нові дані або не буде потрібно записати отримані дані, і т.д.

Важливо відзначити, що одні процеси більшу частину часу зайняті обчисленнями, а інші більшу частину часу очікують завершення операцій введення-виведення. Перші називаються процесами, що обмежені можливостями процесора (англ. CPU-bound process), інші – процесами, що обмежені можливостями пристроїв введення-виведення (англ. I/O-bound process). Варто зазначити, що зі збільшенням швидкості процесорів процеси стають все більш обмеженими можливостями пристроїв введення-виведення.

В однопроцесорній системі тільки один процес може виконуватися (знаходитися в активному стані), будь-які інші процеси повинні чекати, поки процесор не звільниться і можна буде його зайняти. Метою мультипрограмування є максимальне завантаження процесора: процеси постійно повинні виконуватись і процесор не повинен простоювати. В однозадачній системі процес виконується, поки не перейде в стан очікування певної події (як правило, для виконання деяких запитів введення-виведення). У цей час процесор не виконує корисну роботу. У багатозадачній ОС кілька процесів знаходяться в оперативній пам’яті одночасно. Коли один процес переходить у стан очікування, ОС забирає в нього ресурс ЦП і виділяє процесор іншому процесу. Кожен раз, коли один процес повинен чекати, іншому процесу надається ЦП.

Слід також зазначити, що в багатозадачній ОС процеси практично завжди «змагаються» за використання того або іншого комп’ютерного

ресурсу. Планування використання цих ресурсів є основною функцією ОС. Процесор (або процесорний час) є одним з основних первинних ресурсів комп'ютера. Таким чином, планування його використання займає центральне місце при розробці ОС.

3.1. Планувальник та диспетчер процесорного часу

Кожного разу, коли процесор простоює, ОС повинна обрати один з процесів з черги готових до виконання процесів. Процедура вибору реалізується планувальником процесорного часу (реалізується короткострокове планування процесів) за певним алгоритмом (дисципліною) планування.

Планувальник процесорного часу або планувальник ЦП (англ. CPU Scheduler) вирішує, який процес із черги готових до виконання процесів може бути переведений в активний стан. Крім цього, планувальник повинен вирішувати задачу ефективного використання процесора, оскільки перемикання контексту процеса потребує системних витрат через необхідність виконання таких операцій як: перемикання з режиму користувача в режим ядра, збереження стану поточного процеса і карти пам'яті (ознак звернень до ділянок пам'яті), запуск наступного процеса, а також у більшості випадків перезавантаження кеш-пам'яті.

Диспетчер процесорного часу або диспетчер ЦП (англ. CPU Dispatcher) реалізує рішення, яке було прийнято планувальником. Виконується перемикання контексту процеса відповідно до дисципліни планування. Наприклад, у системах розподілу часу кожен процес виконується протягом виділеного інтервалу часу, після закінчення якого (або після настання деяких інших подій) поточний процес знімається з процесора, навіть якщо він не закінчив виконання, і процесор призначається іншому процесу, який обирається відповідно до прийнятої в ОС дисципліни планування. У будь-якому випадку при перемиканні контексту між нитками одного процеса потрібно менше ресурсів, ніж при перемиканні процесів. Для спрощення далі будемо вважати, що виконується планування виконання процесів, а не ниток, хоча дані, подані нижче, можна поширити і на багатониткові процеси.

Диспетчер процесорного часу виконує наступні дії:

♦ перемикання контексту;

♦ перемикання в режим користувача;

♦ передача управління в потрібне місце в процесі для відновлення його виконання.

При перемиканні контексту забороняються переривання; зберігаються такі дані поточного процеса як: лічильник команд, значення регістрів процесора з відповідних полів БУП; стан процеса змінюється з активного на заблокований або на готовий до виконання. Далі, для нового процеса, який був обраний на виконання, значення лічильника команд і контекстні дані завантажуються у відповідні регістри процесора, знову дозволяються переривання, і лише після цього управління передається новому процесу. Час, витрачений на виконання зупинки одного процеса і старту іншого, називається прихованою активністю (латентністю) диспетчера процесорного часу (англ. Dispatch latency). Зрозуміло, що система повинна прагнути мінімізувати цей час, однак набір критеріїв ефективності диспетчеризації більш складний.

Процес диспетчеризації для конкретного процеса може виконуватися багаторазово, тобто процес може декілька разів переходити зі стану готовності до виконання в стан виконання і назад під час свого життєвого циклу. Під час роботи планувальника процесорного часу виконується не тільки вибір процеса на виконання але й модифікація черги готових до виконання процесів та інших структур даних ОС.

Активація роботи планувальника процесорного часу виконується періодично при настанні певних подій у системі, а саме:

♦ створення нового процеса. Необхідно вирішити, який процес буде виконуватись;

♦ завершення процеса. Процес переходить у стан «завершений» і перестає конкурувати за ресурс ЦП. Необхідно обрати інший процес серед готових до виконання;

♦ блокування процеса. Процес переводиться в стан очікування. Якщо виконання процеса блокується запитом на виконання операції введення-

виведення, запитом додаткових ресурсів або по іншій причині, необхідно обрати інший процес на виконання;

♦ розблокування процеса. Процес переводиться в стан готових до виконання процесів. При розблокуванні процеса (наприклад, при завершенні операції введення-виведення або виділенні потрібного ресурсу) планувальник повинен обрати процес на виконання – той, що виконувався в момент розблокування, або той, що було розблоковано;

♦ апаратна подія. При виникненні апаратної події може відбутися переривання ЦП і виконатися запуск обробника переривань. Після завершення обробки переривання планувальник може обрати на виконання інший, а не перерваний процес (наприклад, якщо апаратний таймер виконує періодичні переривання з деякою частотою, рішення про планування можуть прийматися при перериваннях від таймера).

Від того, як приймається рішення про звільнення процесора, залежить клас алгоритмів планування [2, 9, 13, 18, 27, 31, 32, 34-37].

Алгоритми планування без витіснення (англ. Nonpreemptive або Cooperative Scheduling Scheme). Характерною рисою алгоритмів цього класу є те, що обраному процесу дозволяється виконуватись до його блокування або до добровільного звільнення ним процесора. При організації багатозадачності без витіснення управління системою може бути загублено на деякий час (добровільне звільнення процесора потрібно передбачати при розробці програм). Довжина некерованості залежить від ступеню ефективності передачі управління іншим процесам та може бути достатньо великою (може навіть призводити до повної зупинки роботи ОС). Прикладом такої системи є ОС Windows 3.x, в якій активація планувальника процесорного часу відбувається тільки в випадках, коли процес передає управління до ОС (виконує звернення до функцій API, які в свою чергу приводять до системних викликів). Якщо в програмі є нескінченний цикл без звернень до функцій API, перехід до процедур ядра не відбудеться ніколи і система перестане функціонувати.

Алгоритми планування з витісненням (англ. Preemptive Scheduling Scheme). Характерною рисою алгоритмів цього класу є те, що процесу дозволяється виконуватись деякий фіксований час. Якщо до завершення

інтервалу часу процес не встигає завершитись, він призупиняється (переводиться в чергу готових процесів), і управління переходить до іншого процеса (якщо такий є). Більшість алгоритмів планування з витісненням реалізується планувальником, який активується по перериванню від таймера. Механізм диспетчеризації при організації багатозадачності з витісненням цілком реалізується ядром ОС: при розробці програм немає необхідності враховувати можливості передачі управління іншим процесам: програма створюється так нібито вона виконується в однозадачній системі.

Оскільки зовнішні події, які є чинником перемикань, в алгоритмах планування з витісненням виникають набагато частіше, ніж добровільна передача управління, то і перемикання контексту процесів виконується набагато частіше при реалізації алгоритмів планування з витісненням, ніж без витіснення.

3.2. Критерії ефективності дисциплін планування

Для ефективного планування процесорного часу необхідно зважати на тип ОС – від області використання залежать вимоги до алгоритмів планування.

У системах пакетної обробки даних немає користувачів, які взаємодіють з процесами під час їх виконання. У таких системах найбільш ефективні алгоритми без витіснення, або з не частим виконанням перемикань. Такий підхід зменшує кількість перемикань контексту процесів і за рахунок цього збільшує ефективність роботи системи.

В інтерактивних системах необхідні алгоритми з витісненням, щоб запобігти захват процесора одним процесом – навмисно або внаслідок помилок програмування. Необхідно підтримувати певну ступінь мультипрограмування за рахунок обмеження кількості процесів, що знаходяться в системі одночасно, і орієнтуватись на потреби користувача (наприклад, ОС не відповідає вимогам, якщо після натискання на клавішу реакція системи відбувається лише через декілька секунд).

У системах реального часу основною задачею є гарантована обробка сигналів, що надходять ззовні, на протязі заданих часових інтервалів.

Для порівняння алгоритмів планування процесорного часу використовуються наступні критерії, які так або інакше повинні враховуватися системою:

1) пропускна здатність (англ. Throughput) – вимірюється кількістю виконаних процесів за одиницю часу. Критерій оптимізації – максимізація;

2) середній час оберту (англ. Turnaround Time) – час від моменту появи процеса в системі до його завершення, складається з часу очікування у вхідній черзі, часу очікування в черзі готових до виконання процесів, часу очікування пристроїв введення-виведення, часу виконання. Слід зазначити, що час реакції процеса або загальний час перебування процеса в системі T (критерій ефективності виконання процесів) і середній час оберту – це взаємозалежні критерії. Критерій оптимізації – мінімізація;

3) навантаження ЦП (англ. CPU Utilization) – у реальних системах може коливатися від 40 % (для системи, яка незавантажена) до 90 % (для сильно завантаженої системи). Отримати значення навантаження ЦП, наприклад, можна отримати за допомогою команди *top* в ОС Linux, mac OS і UNIX. Кращою моделлю для розрахунку навантаження ЦП є розгляд використання процесора з імовірнісної точки зору. Припустимо, що процес витрачає частку p свого часу на очікування завершення операції введення-виведення. Якщо є n процесів у пам'яті одночасно, ймовірність того, що всі n процесів очікують введення-виведення (ЦП буде простоювати) дорівнює p^n, а навантаження ЦП визначається відповідно як $1 - p^n$. Для процесів існує пов'язаний критерій – відношення реактивності $R = t/T$, де t – довжина процеса або процесорний час, необхідний для виконання процеса. Критерій оптимізації – максимізація даного показника;

4) час очікування (англ. Waiting Time) – загальний час знаходження процеса в черзі готових до виконання процесів. Алгоритм планування ЦП не впливає на кількість часу, протягом якого процес виконується або здійснюється введення/виведення. Це впливає лише на кількість часу, який процес витрачає на очікування в черзі готовності. Час очікування – це сума всіх періодів очікування в готовій черзі. Для процесів пов'язаний критерій – загублений час $M = T - t$. Критерій оптимізації – мінімізація;

5) час відгуку (англ. Response Time) – час, що сплинув від моменту надходження запиту до отримання першої відповіді на нього (час, необхідний лише для початку відповіді, а не повної відповіді). Це важливий показник для інтерактивних систем. В інтерактивній системі час виконання може бути не найкращим критерієм. Часто процес може видавати певні результати досить рано і може продовжувати обчислювати нові результати, поки попередні результати виводяться користувачеві. Критерій оптимізації – мінімізація.

Серед кількісних характеристик також використовується зворотне до відношення реактивності значення – штрафне відношення (англ. Penalty ratio) $P = 1/R = T/t$ (кількість разів, в які час оберту більше довжини процеса). Критерій оптимізації – мінімізація.

Серед інших критеріїв виділяють:

♦ справедливість (англ. Fairness) – гарантоване виділення кожному процесу справедливої частки процесорного часу («справедливої» не означає «рівної», однак схожі процеси повинні отримувати однакове обслуговування);

♦ баланс (англ. Balance) – підтримка зайнятості всіх частин обчислювальної системи. Необхідно рівномірно навантажувати ресурси обчислювальної системи, надаючи перевагу тим процесам, які будуть займати ресурси, що використовуються нечасто. Також навантаження можна більш-менш рівномірно розподілити між процесором та пристроями введення-виведення, якщо обирати для виконання по черзі процеси, що обмежені можливостями процесора, і процеси, що обмежені можливостями пристроїв введення-виведення;

♦ накладні витрати (англ. Overhead) – відношення часу центрального процесора, що використано на реалізацію алгоритму планування і перемикання контекстів, до часу роботи процесів. Наприклад, якщо на кожні 100 мілісекунд виконання процеса буде припадати 200 мілісекунд на визначення наступного процеса для отримання процесора і на перемикання контексту, то такий алгоритм є дуже неефективним;

♦ масштабованість (англ. Scalability) – зростання накладних витрат при збільшенні кількості процесів у системі повинно відбуватися повільно.

Працездатність при поступовому збільшенні навантаження не повинна зникати, а поступово знижуватись.

Як і при будь-якій оптимізації, незалежно від алгоритму, задовольнити всім критеріям одночасно неможливо. Покращуючи роботу алгоритму з точки зору одного критерію, вона погіршується з точки зору іншого. Залежно від типу ОС віддають перевагу тільки деяким критеріям: надається перевага одному класу задач за рахунок дискримінації інших.

Для систем пакетної обробки даних найбільш важливим є задовольняння таким вимогам:

♦ максимальна пропускна здатність;

♦ мінімальний середній час оберту;

♦ максимальне навантаження ЦП (цей показник насправді не може дати вірну інформацію окремо без аналізу інших показників – високий рівень навантаження ЦП може сигналізувати лише про необхідність нарощування обчислювальних потужностей системи).

Для інтерактивних систем найбільш важливим є задовольняння таким критеріям:

♦ мінімальний час відгуку;

♦ мінімальний час очікування;

♦ максимальна відповідність або пропорційність (англ. Proportionality) – суб'єктивний параметр, що базується на часі відгуку. Необхідно дотримувати відповідність реального часу виконання операцій з уявленням користувача про можливі для цього витрати часу.

Для систем реального часу найбільш важливим є задовольняння таким критеріям:

♦ закінчення обслуговування до строку (англ. Meeting deadlines) – запобігання втраті даних (якщо даний критерій не виконується – ОС не може бути віднесена до класу систем реального часу);

♦ передбачуваність (англ. Predictability) – запобігання погіршення якості в мультимедійних системах. Одна і та ж програма при кількох запусках на виконання повинна виконуватись приблизно за один і той же час – одні й ті ж дії повинні виконуватись з однаковими витратами часу.

Алгоритми оцінюються також за набором статичних та динамічних параметрів процесів. До статичних параметрів відносяться граничні значення ресурсів: користувач, який запустив процес; важливість процеса (пріоритет); обсяг запрошеного процесорного часу; види і кількість запрошених ресурсів. До динамічних параметрів відносяться: час, пройдений з моменту вивантаження процеса на диск або з моменту завантаження в оперативну пам'ять; обсяг процесорного часу, який було вже виділено процесу; інтервал часу неперервного використання процесора (англ. CPU Burst); інтервал часу неперервного очікування введення-виведення (англ. I/O Burst). Сумарне значення всіх інтервалів часу неперервного використання процесора складає довжину процеса t.

Власне, виконання процесів складається з почергового використання процесора – CPU Burst і очікування введення-виведення – I/O Burst. Інтерактивні процеси (обмежені можливостями пристроїв введення-виведення (англ. I/O-bound)) мають великі інтервали I/O Burst та малі інтервали CPU Burst, трудомісткі обчислювальні процеси (обмежені можливостями процесора (англ. processor-bound)) – навпаки малі інтервали I/O Burst та великі CPU Burst. При цьому приблизно 90 % всіх процесів мають CPU Burst тривалістю менше 8 мс.

3.3. Дисципліни планування процесорного часу

Більшість сучасних ОС використовують дуже схожі алгоритми планування процесорного часу, усі вони базуються на однакових основних ідеях, але з деякими модифікаціями, специфічними для конкретної ОС.

Хоча більшість сучасних архітектур ЦП мають кілька процесорних ядер, опис базових дисциплін планування виконується в контексті лише одного доступного процесорного ядра: у системі в один момент часу може бути активним лише один процес.

При розгляді базових дисциплін вважається, що планування виконується для процесів, які обмежені можливостями процесора (якщо не буде вказано окремо). Для дисциплін планування конкретних ОС опис наводиться в більш повному обсязі, достатньому для опанування матеріалу.

Час виконання коду планувальника і диспетчера процесорного часу вважається мізерно малим по відношенню до прийнятих умовних одиниць часу і не враховується. При надходженні процеса в систему, він одразу потрапляє в кінець черги готових процесів. Таким чином, процеси, для яких наводяться приклади роботи планувальника процесорного часу, відповідають моделі процеса з двома станами (рис. 2.4) і схемі обслуговування черги процесів (рис. 2.5).

Для графічної демонстрації прикладів роботи алгоритмів використовуються діаграми Ганта (англ. Gantt chart). Діаграма Ганта (відповідно до планування процесорного часу) складається з полос, що орієнтовані вздовж осі часу, який вимірюється в умовних одиницях. Кожна полоса на діаграмі зображує хід виконання окремого процеса, її кінцівки позначають моменти його надходження в систему і завершення виконання, протяжність полоси – час перебування процеса в системі. Крім того, кожна полоса поділена на інтервали часу: заштрихована область полоси – процес у стані готового до виконання, не заштрихована – процес в активному стані. Вертикальна вісь діаграми містить перелік процесів. Під віссю часу для кожного інтервалу часу вказується, який саме процес на даному інтервалі отримав ресурс ЦП.

3.3.1. Алгоритм FCFS

Алгоритм FCFS («Першим прийшов – першим обслуговується»; англ. First Come – First Served) – самий простий алгоритм класу алгоритмів без витіснення: процеси обслуговуються в порядку їх надходження в систему. Інколи цей алгоритм називають FIFO (англ. First In – First Out) – за принципом побудови і обслуговування черги готових процесів. Процеси в стані готових до виконання найчастіше формують одну спільну чергу, БУП нових процесів розміщуються в кінці черги, для виконання обирається процес з початку черги. БУП процесів, що вже виконувались, але були переведені в стан очікування (були заблоковані або ініціювали операцію введення-виведення), після завершення очікування (або блокування) розміщуються в кінці черги, або перед БУП процесів, що ще жодного разу не отримували процесорний час (в цьому випадку для них може бути

організована окрема черга, яка обслуговується в першу чергу, і тільки коли вона порожня, тоді обслуговується черга нових процесів). Схема обслуговування процесів за алгоритмом FCFS наведена на рис. 3.1.

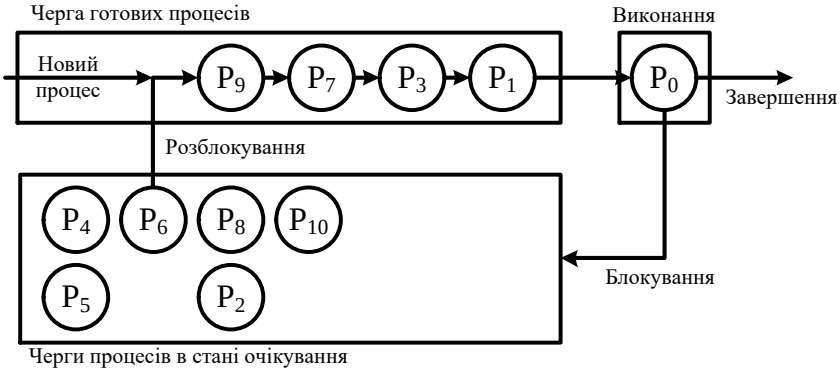


Рисунок 3.1 – Схема обслуговування процесів за алгоритмом FCFS

Перевагою даного алгоритму є простота його реалізації: використовується тільки одна структура даних – черга, немає зовнішнього втручання в хід обчислень. Також даний алгоритм не призводить до відкладання виконання процеса – якщо процес поступив до системи, він обов'язково через деякий час почне виконуватись. Проблемою може бути монополізація одним з процесів використання ЦП. Якщо починає виконуватись довгий процес, інші процеси, навіть зовсім короткі, повинні чекати, доки цей процес не звільнить ресурс ЦП. Причому, для коротких процесів час очікування відносно довжини процесів буде занадто великим, а для довгих – малим.

Наприклад, процес довжиною t_1=1000 у.о. (умовних одиниць часу) отримав процесорний час для свого виконання, у черзі за ним знаходиться процес довжиною t_2=10 у.о. Тоді час перебування процеса в системі для першого процеса буде T_1=1000 у.о., для другого процеса T_2=1010 у.о. У випадку, якщо процеси поміняти місцями в черзі готових до виконання процесів час перебування зміниться: T_2=10 у.о., T_1=1010 у.о. При збільшенні навантаження на систему, загальний час перебування процесів у системі T

може значно зрости. Середній же час оберту не залежить від довжини процесів.

Для алгоритму FCFS притаманний ефект супроводження (англ. Convoy Effect) – якщо короткі процеси знаходяться в черзі готових до виконання процесів позаду довгих процесів, то середній час очікування (а також і середній час перебування процеса в системі і середній час оберту) буде значно довшим у порівнянні з ситуацією, коли короткі процеси знаходяться в черзі попереду довгих. Ефект супроводження впливає на весь цикл виконання процесів: короткі процеси будуть постійно чекати, поки система закінчить обслуговувати довгий процес: на процесорі, на пристроях введення-виведення тощо.

У наведеному вище прикладі в першій ситуації (короткий процес позаду довгого) середній час оберту буде 1005 у.о., у другій (довгий процес позаду короткого) – 510 у.о.

Для демонстрації роботи планувальника процесорного часу за алгоритмом FCFS наведемо приклад роботи для наступного випадку: у систему надходить 5 процесів (A, B, C, D, E) у різний час і з різною довжиною, згідно даних таблиці 3.1.

Таблиця 3.1 – Вихідні дані для планування

Процес	Час надходження, у.о.	Довжина процеса, у.о.
A	0	3
B	1	5
C	3	2
D	9	5
E	12	2

На рис. 3.2 для даних з табл. 3.1 наведено діаграму Ганта виконання процесів за алгоритмом планування FCFS. Процес A надходить у систему в момент часу 0, його БУП додається в чергу готових процесів. Оскільки черга готових до виконання процесів складається тільки з БУП процеса A і немає жодного активного процеса, то він обирається для виконання. У момент часу

1 надходить процес B і його БУП розміщується в черзі готових процесів. Оскільки процесор зайнятий виконанням процеса A, процес B чекає в черзі.

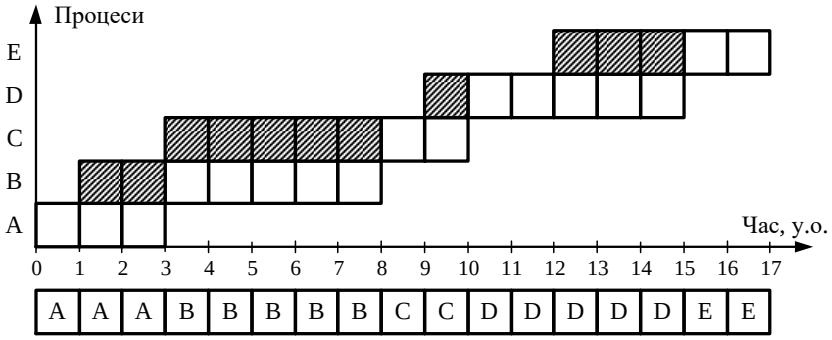


Рисунок 3.2 – Діаграма Ганта планування за алгоритмом FCFS

У момент часу 3 надходить процес C і його БУП розміщується в черзі готових процесів. У цей же момент часу процес A завершує своє виконання і з початку черги готових процесів обирається наступний процес для виконання – B. У момент часу 8 процес B завершує своє виконання і з черги готових процесів обирається наступний процес для виконання – C. У момент часу 9 надходить процес D і його БУП розміщується в черзі готових процесів. У момент часу 10 процес C завершує своє виконання і з черги готових процесів обирається процес D для виконання. У момент часу 12 надходить процес E і його БУП розміщується в черзі готових процесів. У момент часу 15 процес D завершує своє виконання і з черги готових процесів обирається процес E для виконання, який завершує своє виконання в момент часу 17.

За даними табл. 3.1 і по діаграмі на рис. 3.2 визначаються основні кількісні показники ефективності алгоритму FCFS, результати розрахунків яких зведені в табл. 3.2 – для коротких процесів (C і E) штрафне відношення і загублений час мають більші значення, ніж для довгих.

Таблиця 3.2 – Кількісні показники ефективності алгоритму FCFS

Показник	Процес					Середнє значення
	A	B	C	D	E	
T	3	7	7	6	5	5.6
M	0	2	5	1	3	2.2
P	1.0	1.4	3.5	1.2	2.5	1.92

Кількість перемикань контексту процесів дорівнює 6 (перемикання на процес A в момент часу 0; між процесами A і B, B і C, C і D, D і E в моменти часу 3, 8, 10, 15; і з процеса E в момент часу 17 при його завершенні).

Оскільки час відгуку в алгоритмі залежить від довжини активного процеса, то його використання стає неможливим в інтерактивних системах і багатозадачних системах реального часу.

Короткі висновки:

♦ переваги алгоритму – FCFS простий у реалізації та не потребує великих витрат на обробку; він підходить для систем, які мають невелику кількість процесів і не потребують пріоритезації процесів;

♦ недоліки алгоритму – FCFS не підходить для систем, які мають багато процесів, оскільки це може призвести до збільшення середнього часу очікування. Крім того, він не підходить для систем, яким потрібно визначати пріоритетність процесів на основі їх важливості чи терміновості.

3.3.2. Алгоритм RR

Алгоритм RR («Циклічне планування» або «Кругова карусель»; англ. Round Robin) – простий і справедливий алгоритм планування з витісненням. Процесорний час виділяється процесам порціями – квантами (англ. Time slice або Quantum). Якщо при закінченні кванта часу процес ще виконується, він переривається і розміщується в кінці черги готових до виконання процесів, а управління передається наступному процесу з початку черги. Якщо процес блокується або завершує своє виконання до вичерпання кванта часу, перехід управління до іншого процеса виконується негайно. Алгоритм RR є версією алгоритму FCFS, але з витісненням і використовується в якості

основи в інтерактивних системах. На рис. 3.3 наведено схему обслуговування процесів за алгоритмом RR.

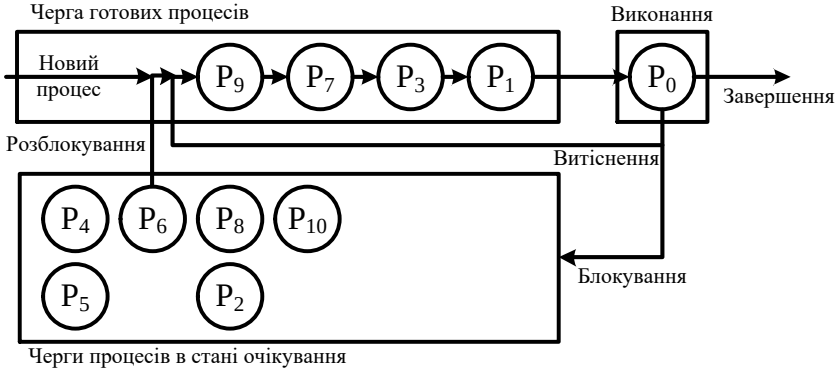


Рисунок 3.3 – Схема обслуговування процесів за алгоритмом RR

В алгоритмі RR необхідно правильно обрати розмір кванта часу: якщо квант дуже малий, тоді більша частина часу буде витрачатися на організацію перемикання контекстів процесів, а при збільшенні розміру кванта значно збільшується час відгуку і алгоритм за своїми характеристиками буде поступово наближатися до алгоритму FCFS (більшість процесів встигатиме виконуватися за один квант), а при значенні кванта, що прагне до нескінченності, алгоритм RR повністю перероджується в алгоритм FCFS.

Значення кванта необхідно обирати як компроміс між реакцією системи (часом відгуку) на дії користувача і накладними витратами на організацію перемикання контекстів процесів. Накладні витрати на перемикання контекстів процесів визначаються як $C/(Q + C)$, де C – тривалість операції перемикання контекстів, Q – тривалість кванта часу.

Наприклад, у черзі готових до виконання процесів є 10 процесів. У першому випадку Q = 100 мс, C = 5 мс, у другому – Q = 10 мс, C = 5 мс. Процес 0 починає виконуватись негайно. Після того, як він вичерпає квант часу Q, відбудеться перемикання (тривалістю C) на процес 1, і так далі. У табл. 3.3 наведені результати обчислення часу відгуку для обох випадків.

Таблиця 3.3 – Порівняння часу відгуку для різних розмірів Q

Процес	Час відгуку для Q = 100 мс	Час відгуку для Q = 10 мс
0	0	0
1	105	15
2	210	30
3	315	45
4	420	60
5	525	75
6	630	90
7	735	105
8	840	120
9	945	135

Для першого випадку останній процес повинен чекати майже 1 сек, перед тим як він почне виконуватись. Це занадто повільно для інтерактивних процесів. Якщо розмір кванта Q = 10 мс, останній процес повинен чекати процесор менше, ніж 1/7 секунди. Зворотною стороною цього є те, що з невеликим квантом накладні витрати на перемикання контексту будуть дорівнювати (5 / (10 + 5)) = 33.3(3) %. Це означає, що третина процесорного часу витрачається на перемикання контекстів процесів. Для випадку з Q = 100 мс накладні витрати перемикання контекстів складають тільки (5 / (100 + 5)) = 4.76 %.

Для випадку невеликих черг готових процесів розмір кванта можна збільшувати, для довгих черг в інтерактивних системах розмір кванта повинен бути зменшений. У такому випадку розмір кванта може залежати від розміру черги: $Q = q/n$, де q – базовий розмір кванта, n – розмір черги. Іншим варіантом може бути визначення розміру кванта залежно від пріоритету процеса: функціональна залежність кванта може мати будь-який припустимий вигляд, але повинна мати властивості монотонності, позитивної визначеності і обмеженості.

Може враховуватись зовнішній і внутрішній пріоритет (який залежить від того, як довго процес очікував обслуговування, скільки разів переривався і які ресурси йому були необхідні). Такий різновид алгоритму має назву WSN («Наступним обслуговується найгірший процес»; англ. Worst Service Next).

Ще одним варіантом є алгоритм квантування з мінімізацією кількості перемикань: процес, що активізується, займає не тільки свій квант, але й залишки квантів процесів, які перейшли до цього моменту часу в стан очікування. Наприклад, в ОС OS/2 змінна TIMESLICE в системному файлі конфігурації дозволяла вказувати мінімальну і максимальну величину кванта. Якщо процес переривався при закінченні кванта часу, то новий квант для нього буде збільшено, і так до тих пір, поки тривалість кванта не досягне максимального значення.

У реальних ОС тривалість кванта часу зазвичай приймається в розмірі 10-100 мс. Для більшості процесів штрафне відношення в середньому постійне, для коротких процесів (довжина яких менше розміру кванта) – штрафне відношення велике.

У чистому вигляді алгоритм RR використовується лише інколи, у більшості сучасних ОС він входить до складу більш складних алгоритмів на основі пріоритетного планування.

На рис. 3.4 наведено приклад роботи планувальника процесорного часу за алгоритмом RR для вихідних даних з табл. 3.1. і розміром кванта $Q = 1$ у.о.

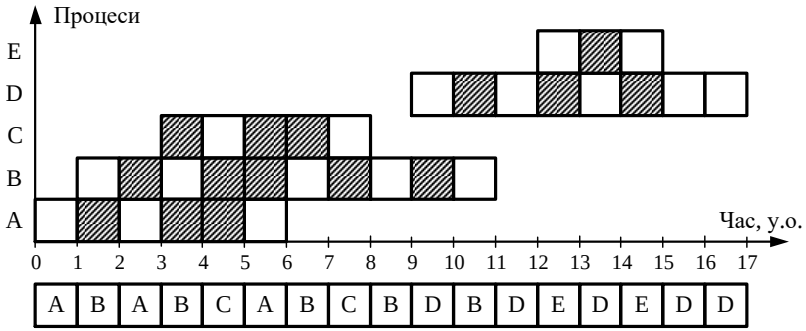


Рисунок 3.4 – Діаграма Ганта планування за алгоритмом RR ($Q = 1$)

Кількісні показники ефективності для прикладу планування за алгоритмом RR з розміром кванта $Q = 1$ наведені в табл. 3.4.

Таблиця 3.4 – Кількісні показники ефективності алгоритму RR ($Q = 1$)

Показник	Процес					Середнє значення
	A	B	C	D	E	
T	6	10	5	8	3	6.4
M	3	5	3	3	1	3.0
P	2.0	2.0	2.5	1.6	1.5	1.92

Для наглядного порівняння поведінки планувальника процесорного часу за алгоритмом RR та залежності кількісних характеристик від розміру кванта часу на рис. 3.5 наведено приклад роботи планувальника на тих самих вихідних даних, але з розміром кванта $Q = 4$ у.о. (якщо процес завершується до того, як вичерпано квант часу – одразу починає виконання новий процес, але залишок кванта йому не додається).

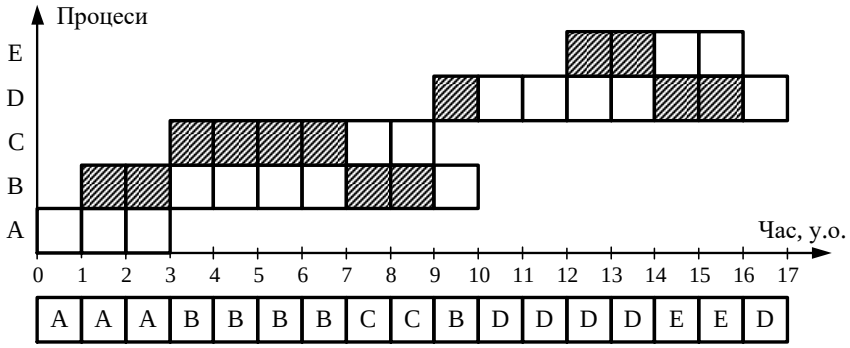


Рисунок 3.5 – Діаграма Ганта планування за алгоритмом RR ($Q = 4$)

Результати розрахунків характеристик наведені в табл. 3.5. Для даного прикладу збільшення розміру кванта привело до покращення кількісних показників, крім того, число перемикань контексту процесів значно зменшилось (8 проти 18 у випадку з квантом $Q = 1$). Середні показники

загального часу перебування процеса в системі T, загубленого часу M та штрафного відношення P також покращилися.

Але не слід забувати, що в другому випадку збільшився час відгуку, і це негативно впливає на реакцію системи на зовнішні події.

Таблиця 3.5 – Кількісні показники ефективності алгоритму RR (Q =4)

Показник	Процес					Середнє значення
	A	B	C	D	E	
T	3	9	6	8	4	6.0
M	0	4	4	3	2	2.6
P	1.0	1.8	3.0	1.6	2.0	1.88

Короткі висновки:

♦ переваги алгоритму – RR (як і FCFS) простий у реалізації та не потребує великих витрат на обробку; він лежить в основі планування в інтерактивних системах. Не має нескінченного відкладання виконання процесів та на відміну від FCFS не має ефекту супроводження;

♦ недоліки алгоритму – в алгоритмі RR пропускна здатність сильно залежить від розміру кванта. Накладні витрати на організацію перемикань контекстів процесів значно вищі, ніж в алгоритмі FCFS, тому що кількість перемикань на кілька порядків більша.

3.3.3. Алгоритми пріоритетного планування

При організації планування без пріоритетів вибір процесів для виконання відбувається без врахування їх відносної важливості і часу обслуговування. В ОС зазвичай виконуються не тільки процеси користувачів, але й системні процеси, важливість яких може бути як вище так і нижче, ніж для процесів користувача. Крім цього, процеси користувачів також можуть бути нерівнозначними. Ці факти приводять до необхідності використання пріоритетного планування (англ. Priority scheduling).

Для врахування важливості процесів їм присвоюються пріоритети (які виражаються числами), що впливають на частоту отримання процесорного часу та в деяких планувальниках й на розмір кванта. Якщо пріоритет не

змінюється з часом, то реалізовано диспетчеризацію з фіксованими пріоритетами, якщо пріоритет змінюється під час виконання процеса, то реалізовано диспетчеризацію з динамічними пріоритетами. При реалізації планувальника, що працює по алгоритму планування з динамічними пріоритетами, потребуються додаткові часові витрати на розрахунок пріоритетів під час виконання процесів, але це дає більш ефективне планування. Фіксовані пріоритети часто використовуються в ОС реального часу.

При використанні пріоритетного планування, якщо в системі є готовий до виконання хоча б один з високопріоритетних процесів, то процеси з низькими пріоритетами будуть очікувати. При використанні планування з витісненням при появі в системі процеса з пріоритетом вище, ніж в активного процеса, процесорний час віддається новому процесу, а процес з низьким пріоритетом витісняється в чергу готових процесів, навіть якщо його квант ще не вичерпаний. Для запобігання ситуації, коли при наявності процесів з високими пріоритетами процеси з низькими пріоритетами ніколи не будуть виконуватись (будуть «голодувати»), в алгоритми вносяться деякі модифікації. Ситуація «голодування» (англ. Starvation) – це ситуація, коли процес не може приступити до виконання протягом невизначеного періоду часу за причиною відсутності необхідних ресурсів системи (у даному випадку – процесорного часу). Для боротьби з «голодуванням» можна використовувати базовий і ефективний пріоритети процеса. Базовий пріоритет задається процесу на початку виконання і не змінюється планувальником. Ефективний пріоритет використовується планувальником при виборі процесів на виконання і може їм змінюватись згідно з реалізованим алгоритмом планування:

♦ кожний такт таймеру пріоритет процеса, що виконується, зменшується; рано чи пізно готові до виконання процеси з низькими пріоритетами зможуть почати виконуватись, тому що їх пріоритети стануть більшими відносно процесів з початковими високими пріоритетами. Через деякий час ефективний пріоритет перерваного процеса відновлюється до початкового значення;

♦ періодично планувальник перевіряє час, який процеси з низькими пріоритетами знаходяться в черзі готових до виконання процесів. У разі досягнення в деякого процеса критичного значення часу очікування, йому тимчасово призначається найвищий ефективний пріоритет, який поступово, з отриманням цим процесом процесорного часу, зменшується до початкового значення;

♦ система для вирішення задач ефективного планування може динамічно призначати високі пріоритети для процесів, які проводять більшу частину часу в очікуванні завершення операцій введення-виведення, щоб після завершення таких операцій процес одразу отримував процесорний час і міг починати нову операцію введення-виведення. При динамічному призначенні пріоритетів процесам, що активно використовують введення-виведення, пріоритет також може обиратися як $1/f$, де f – частка використаного в останній раз кванта часу, тобто, якщо процес використав тільки десяту частину кванта, то пріоритет відповідно призначався рівним 10.

Всі наступні алгоритми також є алгоритмами пріоритетного планування.

3.3.4. Алгоритм SPN

Алгоритм SPN («Найкоротший процес – наступний»; англ. Shortest Process Next) має ще назви SJF («Найкоротша задача – перша»; англ. Shortest Job First) та SJN («Найкоротша задача – наступна»; англ. Shortest Job Next). Алгоритм відноситься до класу алгоритмів без витіснення та передбачає, що довжини процесів, що поступають до системи, відомі наперед. Черга готових до виконання процесів сортується за збільшенням часу виконання процесів, новий процес додається до черги в позицію відповідно до своєї довжини.

Пріоритетами процесів виступають їх довжини: при плануванні перевага віддається самому короткому процесу. Відповідно, для роботи планувальника потребується інформація про процес – час його виконання t або час наступного перебування в активному стані CPU Burst. Отримати значення часу можна кількома способами:

♦ оцінка програмістом за допомогою мови управління завданнями JCL, в якій одним з параметрів вказувався користувачем час, необхідний для виконання процеса;

♦ по історичним міркуванням (береться середній час, необхідний для виконання процеса, по попереднім його запускам);

♦ по часу виконання подібних процесів (процеси одного типу можуть виконуватись приблизно один і той же час);

♦ передбачити по попередній поведінці в поточному запуску (у разі, коли процес перебуває в активному стані декілька разів).

Перші три варіанти часто не можуть показати достовірні результати: при оцінці програміст може випадково або навмисно вказати невірні дані (і довгий процес буде оцінено як короткий або навпаки); історія запусків має значення тільки при багатократному запуску процеса (при перших кількох запусках інформації для оцінювання довжини процеса недостатньо); віднесення процеса до якогось класу подібних процесів часто буває неможливо виконати (ґрунтуючись, наприклад, на обсязі процесів неможливо передбачити тривалість їх виконання, а аналіз коду для визначення тривалості виконання – складна і довга операція).

Більшість процесів виконують не тільки обчислення, але й операції введення-виведення, очікування певних подій тощо. Один інтервал CPU Burst може розглядатися як окрема частина процеса, час оберту якої необхідно мінімізувати. Тому найбільш часто використовується передбачення тривалості виконання частини процеса по попередній поведінці:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n ,$$

де τ_{n+1} – передвіщена довжина наступного інтервалу часу неперервного використання процесора (CPU Burst); α – фактор, що впливає на механізм передбачення ($0 \le \alpha \le 1$); t_n – актуальна довжина попереднього неперервного використання процесора; τ_n – передвіщена довжина попереднього інтервалу часу неперервного використання процесора. Перший доданок враховує

останню поведінку процеса, другий доданок враховує його передісторію. При значенні $\alpha = 0$ стеження за поведінкою процеса не відбувається: $\tau_{n+1} = \tau_n = \ldots = \tau_0$, тобто всі інтервали неперервного використання процесора вважаються однаковими. При значенні $\alpha = 1$ не враховується вся передісторія виконання процеса: інтервал наступного часу неперервного використання процесора передвіщається в розмірі, рівному актуальній довжині попереднього неперервного використання процесора: $\tau_{n+1} = t_n$. При значенні $\alpha = 0.5$ рівною мірою враховується і остання поведінка і передісторія. На рис. 3.6 наведено приклад передбачення тривалості виконання з параметром $\alpha = 0.5$ ($\tau_0 = 10$).

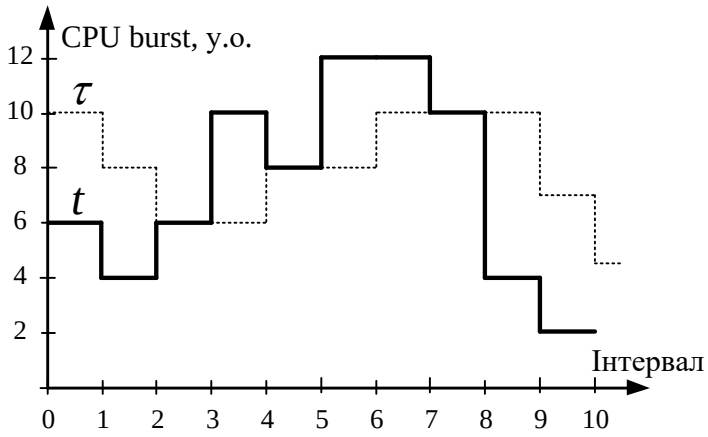


Рисунок 3.6 – Прогнозування тривалості інтервалів CPU Burst

На рис. 3.7 наведено приклад роботи планувальника процесорного часу за алгоритмом SPN для вихідних даних з табл. 3.1.

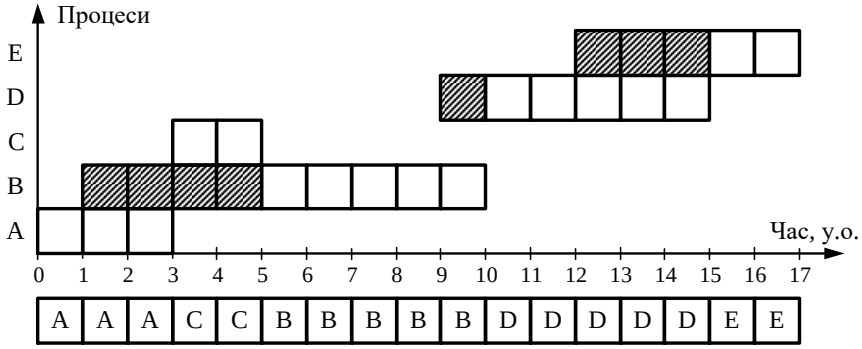


Рисунок 3.7 – Діаграма Ганта планування за алгоритмом SPN

Результати розрахунку характеристик наведені в табл. 3.6.

Таблиця 3.6 – Кількісні показники ефективності алгоритму SPN

Показник	Процес					Середнє значення
	A	B	C	D	E	
T	3	9	2	6	5	5.0
M	0	4	0	1	3	1.6
P	1.0	1.8	1.0	1.2	2.5	1.5

Короткі висновки:

♦ переваги алгоритму – SPN використовується в системах пакетної обробки даних (також цей алгоритм часто використовується при довгостроковому плануванні процесів під час їх створення). Він дозволяє отримати максимальну пропускну здатність і невеликий середній час оберту;

♦ недоліки алгоритму – для найбільш довгих процесів використання даного алгоритму може викликати нескінченно довге відкладання виконання при умові постійного надходження до системи більш коротких процесів. Крім того, невірно розрахована тривалість наступного часу неперервного використання процесора може призвести до того, що процес, якому залишилось виконуватись зовсім небагато часу, буде змушений

чекати процесор разом з довгими процесами, які тільки потрапили до черги готових до виконання процесів.

3.3.5. Алгоритм SRTF

Алгоритм SRTF («З найменшим часом, що залишився, перший»; англ. Shortest Remaining Time First) також має назви PSJF (англ. Preemptive SJF) або PSPN («Найкоротший процес – наступний, з витісненням»; англ. Preemptive Shortest Process Next). Застосовується в мультизадачних системах пакетної обробки даних. Для його роботи також потребується попередня інформація про процес – час його виконання. При надходженні в чергу готових процесів нового процеса, більш короткого, ніж залишок часу для виконання поточного активного процеса, відбувається витіснення активного процеса та починається виконання нового процеса. Крім того, при кожному перемиканні контексту для усіх готових до виконання процесів необхідно обчислювати час до їх завершення. У тому випадку, коли процес повертається в чергу готових процесів після завершення блокування, і очікуваний час до його завершення менше, ніж в активного процеса, також відбувається витіснення активного процеса на користь повернутого процеса.

Пропускна здатність при використанні цього алгоритму вище, ніж для SPN, але кількість перемикань контекстів процесів також більше. До недоліків роботи алгоритму (так само як і для SPN) слід віднести необхідність підтримувати чергу готових процесів у стані сортування за зменшенням часу, що залишився до їх завершення, та ще більшу дискримінацію довгих процесів (навіть якщо довгий процес почне своє виконання – він може бути витіснений новими більш короткими процесами).

Приклад роботи планувальника процесорного часу за алгоритмом SRTF для вихідних даних з табл. 3.1 наведено на рис. 3.8. На відміну від прикладу для алгоритму SPN (рис. 3.7), при надходженні процеса E (момент часу 12) процес D буде витіснений, оскільки час, необхідний для його завершення (3 у.о.) більше часу, необхідного для завершення процеса E (2 у.о.).

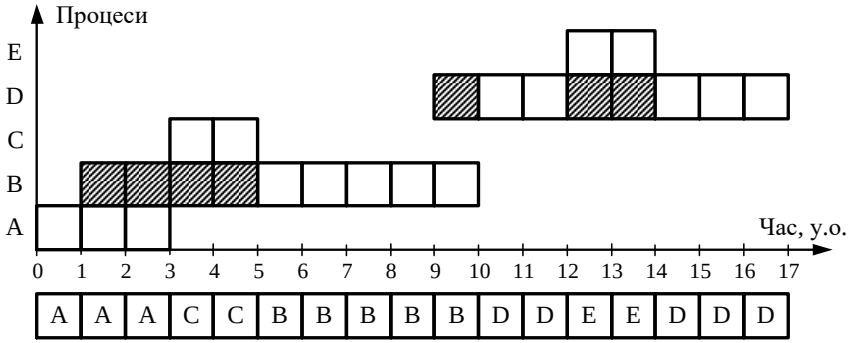


Рисунок 3.8 – Діаграма Ганта планування за алгоритмом SRTF

Результати обчислення характеристик ефективності наведені в табл. 3.7. Як видно на прикладі, цей алгоритм має найкращі характеристики з усіх раніше розглянутих (час перебування в системі, загублений час, штрафне відношення).

Таблиця 3.7 – Кількісні показники ефективності алгоритму SRTF

Показник	Процес					Середнє значення
	A	B	C	D	E	
T	3	9	2	8	2	4.8
M	0	4	0	3	0	1.4
P	1.0	1.8	1.0	1.6	1.0	1.28

Короткі висновки:

♦ переваги алгоритму – короткі процеси обробляються дуже швидко. Система вимагає дуже невеликих накладних витрат (хоча і більше, ніж в алгоритмах FCFS та RR), оскільки вона приймає рішення лише після завершення процеса або після додавання нового процеса (або процеса після завершення блокування) в чергу готових процесів;

♦ недоліки алгоритму – подібно до SPN також може призвести до зупинки процеса – довгі процеси можуть бути відкладені на невизначений час, якщо короткі процеси постійно додаються до черги готових процесів.

3.3.6. Алгоритм HRRN

Алгоритм HRRN («З найбільшим відносним часом відгуку – наступний»; англ. Highest Response Ratio Next) також має назву HPRN («З найбільшим штрафним відношенням – наступний»; англ. Highest Penalty Ratio Next) і відноситься до класу алгоритмів без витіснення.

Основним недоліком алгоритмів SPN і SRTF є ігнорування процесів з великим часом виконання. В алгоритмі HRRN прийняті міри для усунення цього недоліку. Згідно цього алгоритму, час перебування процеса в черзі готових до виконання процесів безпосередньо впливає на порядок обслуговування процесів. Відносний час відгуку (штрафне відношення) для кожного процеса визначається як:

$$P = \frac{M + t}{t},$$

де M – час очікування початку виконання в черзі готових процесів (загублений час); t – час виконання процеса (очікуваний).

Штрафне відношення для процеса, що тільки поступив до системи, завжди дорівнює 1. Оскільки загублений час при очікуванні в черзі зростає, то для процеса, який довго не отримує процесорний час, штрафне відношення збільшується. З черги готових процесів за алгоритмом HRRN обирається процес, в якого штрафне відношення буде максимальним (в разі кількох однакових максимальних значень вибір виконується за дисципліною FCFS). Значення штрафного відношення визначає пріоритет процеса. Таким чином, процес з великим часом очікування досягне миті, коли він отримає процесорний час. Коли в черзі готових процесів є процеси, що довгий час не отримували ресурс процесора, нові процеси не почнуть одразу виконуватись, тому що в будь-якого процеса з черги відносний час відгуку буде більше 1 (дається невелика перевага існуючим процесам).

Відносний час відгуку динамічно змінюється в часі, тому його необхідно періодично обчислювати. Всякий раз, коли звільнюється процесор перед вибором нового процеса з черги готових до виконання, для всіх процесів з черги виконується перерахунок значення штрафного

відношення. Даний алгоритм може бути реалізований і для планування процесорного часу з витісненням на основі алгоритму RR: при закінченні кванта часу, виділеного для виконання процеса, виконується перерахунок штрафного відношення і обирається процес не з початку черги, а з максимальним значенням цього відношення.

Алгоритм забезпечує невеликий середній час оберту і високу пропускну здатність, що дозволяє його використовувати в системах пакетної обробки даних (без витіснення) та в інтерактивних системах (з витісненням). Але планувальник, який працює по даному алгоритму, потребує виконання постійних розрахунків штрафного відношення, тому моменти перепланування будуть довшими, ніж при використанні будь-якого з раніше розглянутих алгоритмів. Крім того, розрахунок штрафного відношення потребує знань про тривалості виконання процесів.

Приклад роботи планувальника процесорного часу за алгоритмом HRRN (варіант з витісненням – розмір кванта часу $Q = 1$) для вихідних даних з табл. 3.1 наведено на рис. 3.9.

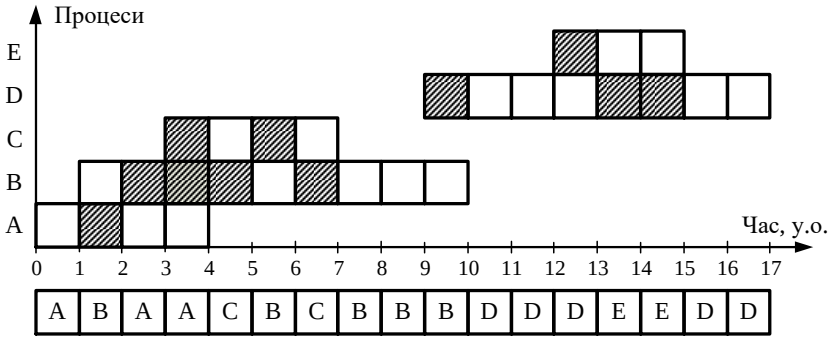


Рисунок 3.9 – Діаграма Ганта планування за алгоритмом HRRN з витісненням ($Q = 1$)

На відміну від алгоритму RR процес A в момент часу 3 отримає ресурс процесора, тому що штрафне відношення для нього буде максимальним – 1.33(3) (значення для процеса B – 1.2, для процеса C – 1). Так само отримає процесорний час процес E в моменти часу 13 і 14 (значення штрафного

відношення – 1.5 проти значень штрафного відношення процеса D – 1.2 і 1.4 відповідно).

На рис. 3.9 також можна помітити, що для коротких процесів штрафне відношення зростає швидше, ніж для довгих процесів, тому час очікування в черзі готових до виконання процесів буде для них невеликим.

Результати обчислення характеристик ефективності наведені в табл. 3.8. Як видно з даних, цей алгоритм показує кращі результати, ніж алгоритм RR.

Приклад для варіанта алгоритму HRRN без витіснення для вихідних даних з табл. 3.1 не наводиться, тому що всі результати співпадають з прикладом для алгоритму FCFS (рис. 3.2 і табл. 3.2).

Таблиця 3.8 – Кількісні показники ефективності алгоритму HRRN, який реалізований з витісненням (Q = 1)

Показник	Процес					Середнє значення
	A	B	C	D	E	
T	4	9	4	8	3	5.6
M	1	4	2	3	1	2.2
P	0.86	1.8	2.0	1.6	1.5	1.55

Короткі висновки:

♦ переваги алгоритму – HRRN зазвичай дає кращу продуктивність, ніж планування за алгоритмом SPN. Зменшується час очікування для довготривалих процесів, а також зменшується час перебування в системі коротких процесів;

♦ недоліки алгоритму – реалізація планування HRRN майже неможлива, оскільки неможливо заздалегідь знати точний час виконання кожного процеса. Якщо все ж таки використовувати якимось чином отримані довжини процесів, виникає перенавантаження ЦП додатковими розрахунками (великі накладні витрати).

3.3.7. Алгоритм SRR

Алгоритм SRR («Егоїстична кругова карусель»; англ. Selfish Round Robin) базується на алгоритмі RR.

При використанні алгоритму RR всі процеси в черзі готових до виконання вважаються рівнозначними. На відміну від RR, в алгоритмі SRR використовується динамічне збільшення пріоритетів процесів з плином часу. Всі процеси поділяються на два класи: нові і прийняті (нові процеси розміщуються в черзі затримки, прийняті процеси знаходяться в активній черзі). Процеси з черги затримки не обслуговуються. Процеси з активної черги обслуговуються за алгоритмом RR. На рис. 3.10 наведено схему обслуговування процесів за алгоритмом SRR.

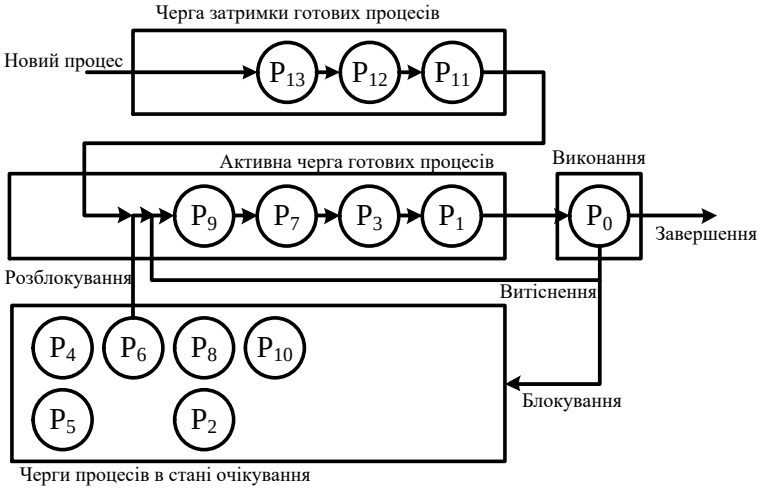


Рисунок 3.10 – Схема обслуговування процесів за алгоритмом SRR

У всіх процесів кожний квант часу збільшується пріоритет. Новий процес переміщується з черги затримки до активної черги, коли його пріоритет досягне значення мінімального пріоритету одного з процесів в активній черзі (або коли в активній черзі не залишиться жодного процеса). У всіх прийнятих процесів пріоритет, незважаючи на постійне його збільшення, ігнорується при плануванні, і вважається, що у всіх прийнятих

процесів пріоритет однаковий (обслуговування ведеться по дисципліні RR без врахування пріоритетів).

Пріоритет нових процесів збільшується кожен квант часу на величину $a \geq 0$, пріоритет прийнятих процесів – на величину $b \geq 0$. За допомогою значень a і b алгоритм може бути модифікований:

♦ якщо $b = 0$, алгоритм SRR перетворюється в алгоритм RR (нові процеси одразу становляться прийнятими);

♦ якщо $b \geq a > 0$, алгоритм SRR перетворюється в алгоритм FCFS (в активній черзі завжди знаходиться не більше одного процеса);

♦ якщо $a > b > 0$, працює алгоритм SRR;

♦ якщо $b > a = 0$, алгоритм SRR перетворюється в алгоритм RR з прийняттям процесів порціями (всі нові процеси з черги затримки приймаються в активну чергу тільки після того, як у ній не залишиться жодного прийнятого процеса).

Загалом, SRR віддає перевагу процесам, що давно надійшли в систему, над процесами, які надійшли в систему недавно.

На рис. 3.11 наведено приклад роботи планувальника процесорного часу за алгоритмом SRR для вихідних даних з табл. 3.1, розміром кванта $Q = 1$ у.о. і значеннями $a = 2$ і $b = 1$.

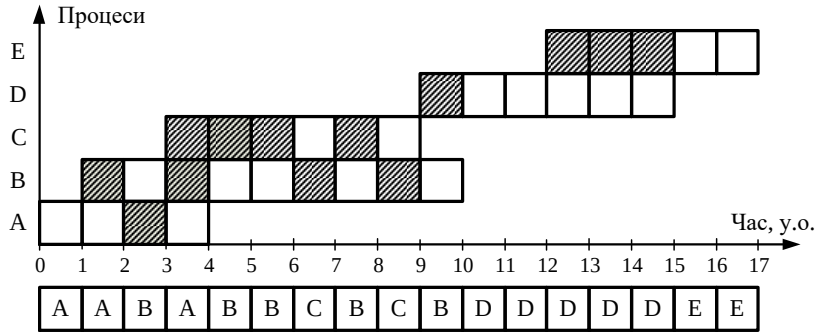


Рисунок 3.11 – Діаграма Ганта планування за алгоритмом SRR

У момент часу 0 поступає процес A, оскільки активна черга порожня, він одразу стає прийнятим і починає виконуватись. У момент часу 1 процес A має пріоритет 1 і поступає процес B, його пріоритет дорівнює 0, процес B розміщується в черзі затримки. У момент часу 2 пріоритет процеса A дорівнює 2, і пріоритет процеса B також дорівнює 2. Процес B стає прийнятим і отримує квант процесорного часу згідно з алгоритмом обслуговування RR активної черги. У момент часу 3 надходить у чергу затримки процес C з пріоритетом 0. У прийнятих процесів A і B пріоритет дорівнює 3. У момент часу 4 у нового процеса C пріоритет дорівнює 2, процес A завершує виконання, у прийнятого процеса B пріоритет дорівнює 4. Процес C стає прийнятим у момент часу 6 з пріоритетом 6. З цього моменту процеси B і C обслуговуються за алгоритмом RR. Подальша робота планувальника виконується аналогічним чином.

Кількісні показники ефективності для прикладу планування за алгоритмом SRR наведені в табл. 3.9.

Таблиця 3.9 – Кількісні показники ефективності алгоритму SRR

Показник	Процес					Середнє значення
	A	B	C	D	E	
T	4	9	6	6	5	6.0
M	1	4	4	1	3	2.6
P	1.33	1.8	3.0	1.2	2.5	1.97

Серед проблем, що виникають при реалізації алгоритму SRR, є питання обчислення пріоритетів процесів, що повертаються в активну чергу готових процесів після блокування: якщо під час блокування пріоритет не зростає – через деякий час він буде значно нижчим від пріоритетів інших процесів в активній черзі і тим самим дозволить багатьом новим процесам стати прийнятими раніше строку. Інша проблема – при постійному завантаженні системи пріоритет прийнятих процесів може зростати нескінченно, що може викликати переповнення структур даних, в яких зберігаються його значення.

Існує спрощений варіант алгоритму SRR – без використання динамічних пріоритетів. Це вирішує вказані вище проблеми. Вводиться

максимальне обмеження на кількість прийнятих процесів, які можуть бути розміщені в активній черзі. Як тільки досягнуто максимальний обсяг активної черги, нові процеси починають розміщуватись у черзі затримки. При завершенні одного процеса з активної черги один процес з черги затримки переводиться в активну чергу (стає прийнятим) і бере участь у плануванні за алгоритмом RR. У такому спрощеному варіанті є свої недоліки: якщо в активній черзі знаходяться дуже довгі процеси, виконання нових процесів відкладається на дуже великий час. Також викликає проблему реалізація поведінки при блокуванні процеса: якщо в активну чергу під час блокування будь-якого процеса додається новий процес, а старий процес буде розблоковано – розмір активної черги перевищить встановлений для неї максимум. Якщо при блокуванні прийнятого процеса в активну чергу не додавати новий процес, може статися ситуація, коли всі процеси з активної черги будуть заблоковані, і продуктивність системи значно знизиться.

Короткі висновки:

♦ переваги алгоритму – SRR забезпечує справедливість між усіма процесами, надаючи кожному процесу однакову кількість процесорного часу;

♦ недоліки алгоритму – SRR може призвести до низького часу відгуку для інтерактивних процесів. SRR не підходить для систем реального часу, які вимагають суворих обмежень за часом.

3.3.8. Алгоритм VRR

Алгоритм VRR («Віртуальна кругова карусель»; англ. Virtual Round Robin) [20] є розширенням класичного алгоритму RR. На відміну від алгоритму RR в алгоритмі VRR компенсується деяке обмеження у виділенні процесорного часу для процесів, що виконують операції введення-виведення. Коли процес виконує операцію введення-виведення, він не використовує свій квант часу в повному обсязі, а переривається і ставиться в чергу в стані очікування. Після завершення операції введення-виведення в класичному алгоритмі процес переводиться із стану очікування в стан готових процесів і розташовується в кінці черги. Таким чином, процеси,

обмежені можливостями пристроїв введення-виведення, отримують менше процесорного часу ніж процеси, обмежені можливостями процесора.

Планувальник, що використовує алгоритм VRR, надає можливість компенсувати не до кінця витрачені кванти часу. Черга готових процесів розбивається на дві черги: основна черга і додаткова (пріоритетна). Додаткова черга використовується для підвищення пріоритету для процесів, обмежених можливостями пристроїв введення-виведення: якщо ця черга не порожня, то обслуговуються процеси з неї, якщо порожня – тоді з основної черги. Відповідно, у додаткову чергу можуть потрапити тільки ті процеси, що перейшли зі стану очікування в стан готових до виконання процесів.

Якщо процес витісняється з процесора після повного використання кванта часу, він розміщується в кінці основної черги. Основна черга обслуговується за класичним алгоритмом RR. Додаткова черга обслуговується також за класичним алгоритмом RR, але після отримання процесорного часу процес потрапляє або в основну чергу (вичерпав квант часу), або буде переведений у стан очікування (ініціював операцію введення-виведення). Якщо прийняти тривалість кванта часу Q, а використану частку кванта Q_0, то тривалість кванта часу, що виділяється для процесів у додатковій черзі дорівнює $(Q - Q_0)$, тобто процеси отримують тільки невикористану частку кванта до переведення в стан очікування. Наприклад, Q = 10 мс, процес через 6 мс видав команду ініціювання операції введення-виведення, він буде блокований, а процесор буде відданий іншому процесу. Після завершення операції введення-виведення і розблокування, процес буде розташований у додатковій черзі, і коли він отримає процесорний час, йому буде виділений квант лише (10 – 6) = 4 мс.

Схема обслуговування процесів VRR наведена на рис. 3.12.

У ситуації, зображеній на рис. 3.12, після того, як процес P_0 звільнить процесор (або буде витіснений, або блокований), процесорний час буде надано процесу P_1, що знаходиться на початку додаткової пріоритетної черги; далі по порядку процесорний час буде надано процесам P_3, P_7 P_9 та P_6 (що буде розблоковано). Якщо після звільнення процесора процесом P_6 додаткова черга буде порожня, то тільки тоді процесорний час буде виділено процесу P_{11} з основної черги готових процесів.

Дослідження показали, що цей алгоритм краще, ніж класичний алгоритм RR, з точки зору справедливого виділення процесорного часу процесам. Крім цього, планувальник, що реалізує цей алгоритм, мінімізує дисперсію часу відгуку для процесів, обмежених можливостями пристроїв введення-виведення і процесів, обмежених можливостями процесора.

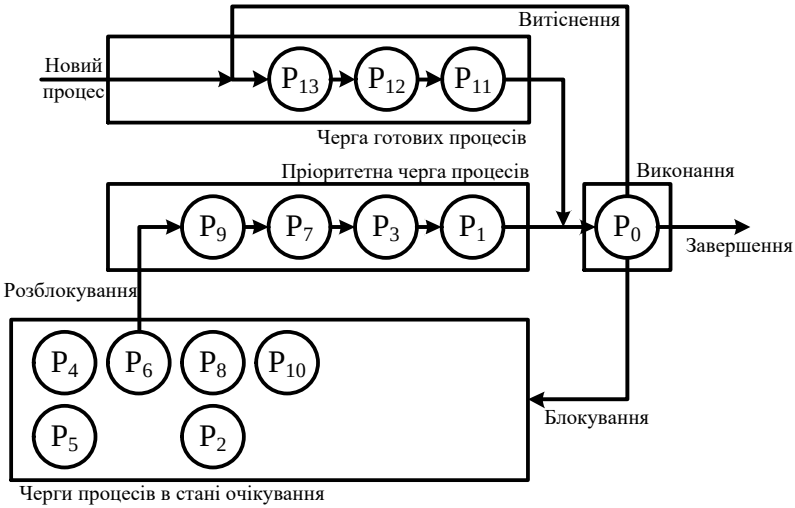


Рисунок 3.12 – Схема обслуговування процесів за алгоритмом VRR

Короткі висновки:

♦ переваги алгоритму – VRR зберігає всі основні властивості алгоритму RR, крім того, він поводиться рівномірно та справедливо з усіма типами процесів. Наявність фіктивних операцій введення-виведення (які інколи користувач спеціально може додавати в процеси для підвищення пріоритету) в процесах не приводить до збільшення привілейованості процесів;

♦ недоліки алгоритму – VRR має більшу частоту перемикань контексту процесів по зрівнянню з алгоритмом RR. Також не враховуються пріоритети процесів.

3.3.9. Алгоритм MLQ

Алгоритм MLQ («Багаторівнева черга»; англ. Multilevel Queue) – змішаний, побудований на основі кількох базових алгоритмів. Готові процеси поділяються на кілька класів і для кожного з класів призначається своя черга. Це дозволяє класифікувати і розділити різні типи процесів: процеси реального часу, системні процеси, інтерактивні процеси, інтерактивні процеси з низьким пріоритетом і фонові (пакетні) процеси. Кожна черга обслуговується за своїм алгоритмом. Планувальник обирає для обробки найбільш пріоритетну чергу (клас), в якій є принаймні один процес. Після вибору черги згідно з обраним алгоритмом (в більшості випадків побудованим на основі алгоритму RR) обирається процес з цієї черги. Наприклад, всі черги обслуговуються за алгоритмом RR, але з різним розміром кванта часу: для низькопріоритетних черг розмір кванта часу збільшується. Для черги з мінімальним пріоритетом може застосовуватись алгоритм FCFS.

На рис. 3.13 наведено загальну схему обслуговування процесів за алгоритмом MLQ.

Вибір черги, яка обслуговується планувальником, може виконуватись двома способами. Перший спосіб оснований на фіксованому пріоритетному плануванні: кожна черга має фіксований пріоритет. Спочатку повністю обслуговується більш пріоритетна черга, і лише коли ця черга порожня, обслуговується другорядна черга. Ніякий процес у черзі для фонових процесів не може працювати, якщо в чергах для системних процесів та інтерактивних процесів є хоча б один процес. Якщо в черзі з більшим пріоритетом з’являється новий процес, то менш пріоритетний процес переривається (якщо він був в активному стані) і починає виконуватися процес з більш пріоритетної черги. Пріоритети призначаються процесам на основі їх типу, характеристик і важливості. Наприклад, інтерактивні процеси для виконання введення/виведення можуть мати вищий пріоритет, ніж фонові процеси, що виконують резервне копіювання файлів (також операції введення/виведення).

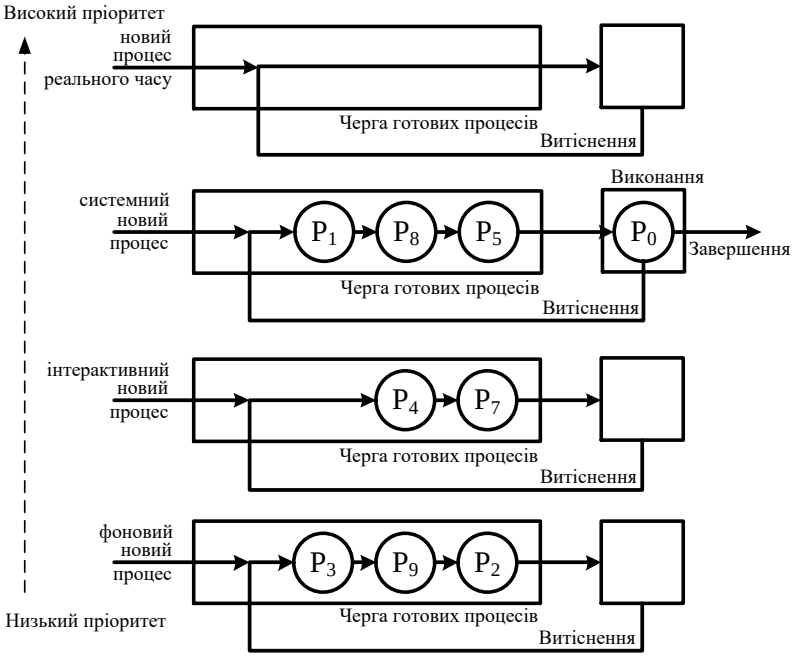


Рисунок 3.13 – Схема обслуговування процесів за алгоритмом MLQ

Другий спосіб базується на розподілі часу: кожній черзі на обслуговування виділяється певний період процесорного часу, наприклад, 80 % для пріоритетної черги, що обслуговується за алгоритмом RR, і 20 % для другорядної, що обслуговується за алгоритмом FCFS.

У першому випадку можливе виникнення ситуації нескінченного відкладання, у другому – процедура обслуговування більш справедлива.

На рис. 3.13 процеси поділені на 4 класи (наведено за збільшенням пріоритету черги): фонові, інтерактивні, системні, реального часу. Нові процеси при переході в стан готових до виконання процесів розташовуються у відповідних чергах. Використовується фіксоване пріоритетне планування. У прикладі на рис. 3.13 виконуються процеси з черги системних процесів, тому що черга процесів реального часу порожня. Процеси з черг інтерактивних і фонових процесів чекають.

Даний алгоритм має низькі накладні витрати на планування: оскільки процеси назавжди закріплено за відповідними чергами, накладні витрати на планування низькі, оскільки планувальнику потрібно лише обрати відповідну чергу для виконання. Також у ньому ефективно розподіляється ресурс ЦП: гарантується, що процеси з вищим рівнем пріоритету виконуються своєчасно, водночас дозволяючи процесам з нижчим пріоритетом виконуватися, коли ЦП не зайнятий високопріоритетною роботою. Наявність можливості налаштування дозволяє адаптувати алгоритм відповідно до конкретних вимог різних типів процесів. Для кожної черги можна використовувати різні алгоритми планування залежно від вимог процесів у черзі. Використання витіснення дає можливість процесам з вищим пріоритетом випереджати процеси з нижчим пріоритетом, що гарантує своєчасне виконання процесів з високим пріоритетом.

Короткі висновки:

♦ переваги алгоритму – алгоритм MLQ забезпечує справедливий розподіл процесорного часу для різних типів процесів на основі їх пріоритету та вимог;

♦ недоліки алгоритму – деяким процесам може бракувати ЦП, якщо черги з вищим пріоритетом ніколи не порожніють. Також може виникнути додаткова складність у реалізації та підтримці кількох черг і різних для них алгоритмів планування.

3.3.10. Алгоритм MLFQ

Алгоритм MLFQ («Багаторівнева черга зі зворотнім зв’язком»; англ. Multilevel Feedback Queue) побудований на основі алгоритму MLQ. Вперше алгоритм був описаний у 1962 році і використовувався в першій ОС з використанням розподілу часу CTSS (Compatible Time-Sharing System)[76] і пізніше в ОС Multics. При використанні алгоритму MLQ процеси жорстко

[76] Перша версія випущена в 1961 р., остання версія – 1973 р., апаратна платформа IBM 7094, ресурс у мережі Інтернет https://www.cozx.com/dpitts/ibm7090.html

закріплені в своїх чергах. Це робить даний алгоритм негнучким. MLFQ, навпаки, дозволяє процесу переміщення між чергами. Ідея полягає в тому, щоб розділяти процеси під час виконання у відповідності з їх особливостями використання процесора. Наприклад, якщо процес використовує занадто багато процесорного часу, він буде переміщений у чергу з меншим пріоритетом. Це дозволить процесам, обмеженим можливостями пристроїв введення-виведення, бути розташованим у черзі з більш високим пріоритетом.

Основою алгоритму є те, що для черг з високими пріоритетами виділяються короткі кванти. Довжина кванта збільшується з кожною чергою в напрямку зменшення пріоритету. Процес починає своє виконання в черзі з найвищим пріоритетом. Після того, як процес звільнить процесор, перевіряється причина його переривання: або витіснення по причині завершення виділеного кванта часу, або блокування для виконання операції введення-виведення. Якщо процес блокується, то після розблокування він розміщується в тій самій черзі. Однак, якщо процес вичерпав виділений йому квант часу, то планувальник витісняє цей процес і розміщує його в черзі з наступним нижчим рівнем пріоритету.

Для кожної черги може використовуватись свій власний алгоритм планування. Поки процес повною мірою використовує виділений квант у чергах на кожному рівні пріоритету, він продовжує переміщуватись по чергах зі зниженням пріоритету (наприкінці досягнувши черги найнижчого рівня, де він буде отримувати процесорний час, наприклад, за алгоритмом RR). У чергах на кожному рівні з більш низьким пріоритетом, розмір кванта збільшується (так що процес не часто отримуватиме процесор, але на більший проміжок часу). Підставою для цього є те, що процеси, які знаходяться на більш низьких рівнях, показали, що вони обмежені можливостями процесора, а не пристроїв введення-виведення. Збільшення кванта часу дозволяє мінімізувати накладні витрати, пов’язані з організацією перемикання контексту, і отримати більш ефективне використання процесора. Але процеси в чергах з меншим пріоритетом можуть отримати процесорний час тільки за умови відсутності процесів у чергах з більшим пріоритетом.

На рис. 3.14 наведено схему обслуговування процесів за алгоритмом MLFQ.

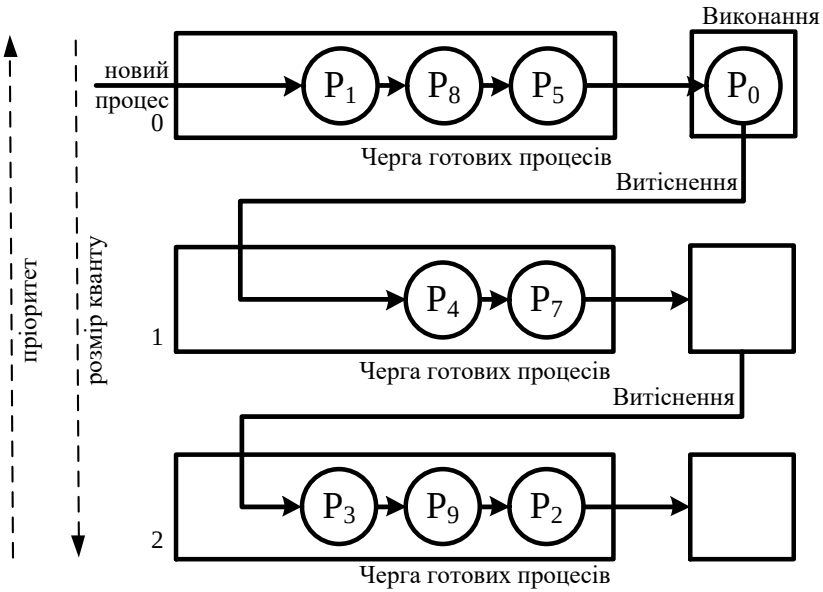


Рисунок 3.14 – Схема обслуговування процесів за алгоритмом MLFQ

Крім переведення в менш пріоритетні черги, процеси, які чекають виділення процесорного часу занадто довго в черзі з низьким пріоритетом, можуть бути переміщені в черги з більшими пріоритетами. Такі дії запобігають «голодуванню» процесів. Переміщення процесів у черги з більшими пріоритетами може виконуватись за різними принципами. Наприклад, при завершенні блокування після очікування введення з клавіатури процес може бути переміщений у чергу з самим високим пріоритетом; після завершення дискових операцій введення-виведення процес може бути переміщений у чергу з наступним за найбільшим пріоритетом (але якщо процес був у черзі з найвищим пріоритетом, він там і залишиться); а після завершення всіх інших подій, що викликали блокування процеса, з черги з найнижчим пріоритетом – у чергу з наступним по

зростанню пріоритетом. Переміщення процесів з черг з низькими пріоритетами в черги з більшими пріоритетами дозволяє більш повно враховувати зміни поведінки процесів з плином часу. Але такий механізм підвищення ефективності має лазівку для шахраїв: програміст може навмисно додавати в програму фіктивні операції введення-виведення для підвищення пріоритету свого процеса.

Алгоритм MLFQ реалізує найбільш загальний підхід до планування процесорного часу. Він найбільш трудомісткий при реалізації, але в той же час надає найбільшу гнучкість. Існує багато різновидів реалізацій алгоритму MLFQ. Для повного опису конкретної реалізації алгоритму необхідно визначити наступні параметри:

- кількість черг готових до виконання процесів;
- алгоритм планування вибору черги для обслуговування;
- алгоритми планування вибору процесів для виконання з кожної черги;
- правила розміщення нового процеса в одній з черг готових до виконання процесів;
- правила переміщення процесів між чергами.

Припустимо, для схеми обслуговування процесів на рис. 3.14, що черга 0 обслуговується за алгоритмом RR з розміром кванта часу Q = 1 у.о., черга 1 обслуговується за алгоритмом RR з розміром кванта часу Q = 2 у.о., черга 2 обслуговується за алгоритмом RR з розміром кванта часу Q = 4 у.о. Зворотні переходи процесів у черги з меншими номерами відсутні. У разі появи нового процеса в черзі 0, процес що виконується, не переривається, а повністю використовує виділений квант.

На рис. 3.15 наведено приклад роботи планувальника процесорного часу за алгоритмом MLFQ для вихідних даних з табл. 3.1. У клітинках полос вказано додатково номер черги, в яких процеси знаходились при виділенні їм процесорного часу.

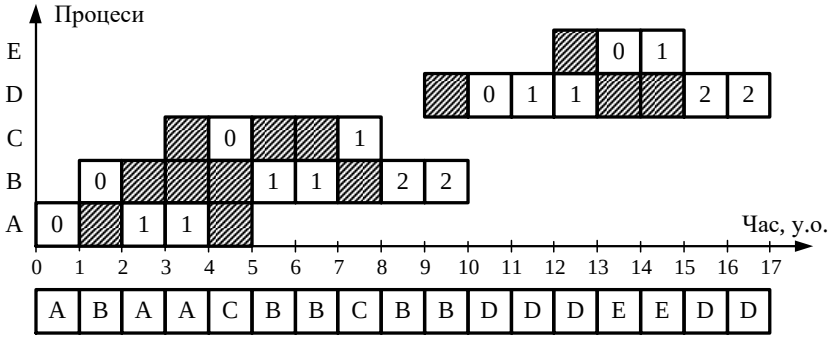


Рисунок 3.15 – Діаграма Ганта планування за алгоритмом MLFQ

Кількісні показники ефективності для прикладу планування за алгоритмом MLFQ наведені в табл. 3.10.

Таблиця 3.10 – Кількісні показники ефективності алгоритму MLFQ

Показник	Процес					Середнє значення
	A	B	C	D	E	
T	4	9	5	8	3	5.8
M	1	4	3	3	1	2.4
P	1.33	1.8	2.5	1.6	1.5	1.75

Припустимо, що черги 0-2 для схеми на рис. 3.14 обслуговуються так само, як у прикладі вище. Але в разі появи нового процеса в черзі 0, процес, що виконується (якщо він з черги 1 або 2), переривається і розміщується в кінці своєї черги, а замість нього починає своє виконання новий процес з більш пріоритетної черги. Зворотні переходи процесів у черги з меншими номерами також відсутні. Приклад роботи планувальника за такою модифікацією алгоритму MLFQ наведений на рис. 3.16.

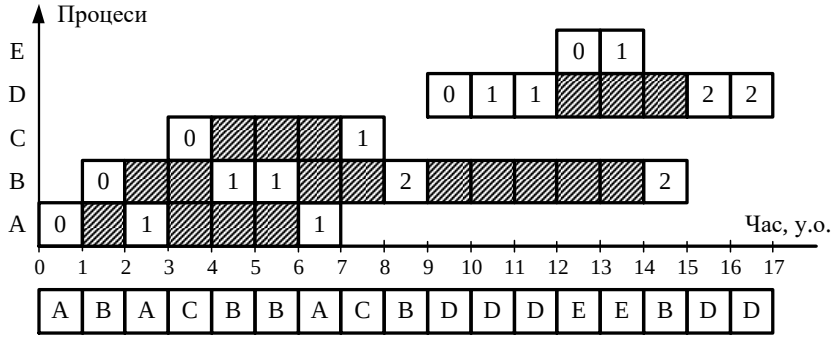


Рисунок 3.16 – Діаграма Ганта планування за алгоритмом MLFQ (варіант 2)

Як видно, у момент часу 3 при появі в системі процеса С, процес А переривається, і процесорний час віддається процесу С. Так само відбувається в момент часу 9, коли поступає в систему процес D. Для даного прикладу кількісні характеристики ефективності не наводяться, але помітно, що процеси, які потрапляють у чергу з найменшим пріоритетом можуть «голодувати». У такому випадку в алгоритм бажано додати правила переміщення процесів з найменш пріоритетної черги до більш пріоритетних, щоб надати їм можливість завершити виконання.

Короткі висновки:

♦ переваги алгоритму – можливість легкого поділу процесів на категорії в залежності від їх потреб у процесорному часі. Алгоритм надає перевагу процесам, обмеженим можливостями пристроїв введення-виведення. Варіанти реалізації алгоритму MLFQ зі збільшенням розміру кванта при зменшенні пріоритету черги також покращують ефективність виконання процесів, обмежених можливостями процесора, надаючи їм більшу частку процесорного часу і зменшуючи кількість перемикань контексту;

♦ недоліки алгоритму – реалізація схеми призначення пріоритетів: пріоритети контролюються системою, а не адміністратором або користувачами. Процес вважається важливим не тому, що він насправді такий, а по великій кількості операцій введення-виведення. Планувальник,

який працює за алгоритмом MLFQ також має той недолік, що при зміні поведінки процеса (наприклад, процес, що часто виконував операції введення-виведення, став виконувати багато обчислювальної роботи та навпаки), ефективність роботи системи знижується. Ще один недолік – складність алгоритму та високі накладні витрати при його реалізації.

На основі алгоритму MLFQ побудовані планувальники процесорного часу в ОС Solaris, FreeBSD, Windows NT.

3.3.11. Алгоритм лотерейного планування

Алгоритм лотерейного планування (англ. Lottery Scheduling) оснований на імовірнісному підході: в основі алгоритму лежить роздача процесам «лотерейних квитків» на виділення процесорного часу. Розподіл «квитків» може бути нерівномірним: для процесів з більшим пріоритетом видається більша кількість «квитків». Планувальник випадковим чином обирає номер «квитка» і надає процесорний час (в розмірі одного кванта) процесу, якому цей «квиток» виділено. Таким чином, важливі процеси частіше будуть отримувати процесорний час. Кожен процес у середньому отримує таку долю процесорного часу, яка доля «квитків» йому виділена. Наприклад, якщо в порції 100 «квитків» і 10 з них виділені одному процесу, то в середньому процесорний час цьому процесу буде надаватися в розмірі 10 % від загального часу.

Алгоритм лотерейного планування дуже гнучкий та чутливий до змін. Цей алгоритм може бути наближеним до інших алгоритмів планування. Наприклад, для наближення до алгоритму SPN (але заснованому на квантуванні часу) достатньо виділяти більшу кількість «квитків» коротким процесам, ніж довгим. Лотерейне планування вирішує проблему можливого «голодування» процесів. Якщо видавати кожному процесу принаймні один «лотерейний квиток», то це гарантує, що процеси матимуть ненульову імовірність бути відібраними при розподілу процесорного часу. «Квитки» виділяються порціями, коли вся порція використана (не залишилось процесів з квитками і, відповідно, немає «голодування»), розподіляється нова порція «квитків» між процесами.

Табл. 3.11 демонструє приклад впливу кількості «лотерейних квитків» на обсяги отримуваного процесорного часу, а саме – значення часток процесорного часу, що виділяються довгим та коротким процесам, за умови їх різної кількості. Передбачається, що короткому процесу надається 10 «квитків», довгому – 1 «квиток».

Таблиця 3.11 – Параметри алгоритму лотерейного планування

Кількість коротких процесів	Кількість довгих процесів	Процесорний час на кожний короткий процес, %	Процесорний час на кожний довгий процес, %
1	1	91	9
0	2	-	50
2	0	50	-
10	1	10	1
1	10	50	5

Лотерейне планування має властивості швидкого реагування на зміну ситуації в системі: при підвищенні пріоритету процеса, йому буде надано більшу кількість «квитків» у новій порції, і після закінчення розподілу процесорного часу за «квитками» в поточній порції ці зміни вже будуть враховані. Якщо в систему надходить новий процес і йому буде виділено кілька «квитків», то вже при обробці наступної порції він отримає шанс на отримання процесорного часу, пропорційний кількості отриманих «квитків». Таким же чином при взаємодії процеси можуть обмінюватись «квитками» залежно від того, якому процесу процесорний час потрібний більше. Також за допомогою лотерейного планування можна легко організувати завантаження ЦП кількома постійно готовими до виконання процесами в певній пропорції, наприклад: 50:25:10:5.

Але при реалізації алгоритму лотерейного планування слід брати до уваги, що при значній кількості процесів можуть виділятись мільярди «квитків». Організація підтримки масиву такої кількості «квитків», в якому кожен індекс відповідає «квитку» і кожен елемент пов’язаний з процесом,

якому видано «квиток», може привести як до великих витрат пам'яті, так і до значних додаткових обчислень.

Короткі висновки:

♦ переваги алгоритму – лотерейний алгоритм гнучкий та чутливий до змін. Проста реалізація, яку можна поєднувати з іншими алгоритмами;

♦ недоліки алгоритму – наявність додаткових накладних витрат для генерації «лотерейних квитків». Лотерейне планування не є безпечним алгоритмом, оскільки засноване на випадковості, якою можна маніпулювати: зловмисник потенційно може отримати несправедливу перевагу, маніпулюючи створенням «лотерейних квитків».

3.3.12. Алгоритм гарантованого планування

Алгоритм гарантованого планування (англ. Guaranteed Scheduling) полягає в тому, щоб дати та виконати реальну (гарантовану) обіцянку користувачам щодо продуктивності: якщо N користувачів увійшли в систему, кожен з них отримає приблизно $1/N$ потужності ЦП (аналогічно, в однокористувацькій системі з N запущеними процесами, за всіх рівних умов, кожен процес повинен отримати $1/N$ ресурса процесора).

Припустимо, що кожен користувач має свій номер від 1 до N. Для кожного користувача відстежується час його перебування в системі T_i і загальний процесорний час, що вже було виділено всім процесам користувача τ_i. Величина T_i/N – справедливе виділення процесорного часу для і-го користувача.

Якщо $\tau_i \ll T_i/N$, то і-й користувач несправедливо обділений процесорним часом. Якщо $\tau_i \gg T_i/N$, то система надає перевагу користувачу з номером і.

Для процесів кожного користувача обчислюється коефіцієнт справедливості $\tau_i \cdot N/T_i$. Коефіцієнт 0.5 означає, що процес отримав лише половину того, що мав отримати, а коефіцієнт 2.0 означає, що процес отримав у два рази більше, ніж мав на це право. Черговий квант часу виділяється процесу з найменшим значенням коефіцієнту справедливості.

Наприклад, є три процеси A, B і C, які використали 2, 3 і 1 секунду процесорного часу відповідно. Загальний використаний час ЦП на даний

момент становить 6 с. Значення коефіцієнтів справедливості для кожного процеса буде 1, 1.5, 0.5 відповідно. За алгоритмом гарантованого планування наступним повинен отримати процесорний час процес С.

Короткі висновки:

♦ переваги алгоритму – гарантований алгоритм забезпечує справедливий розподіл процесорного часу між процесами для багатокористувацьких ОС;

♦ недоліки алгоритму – додаткові накладні витрати для перерахунку коефіцієнтів справедливості. Також неможливо передбачити поведінку користувачів: якщо користувач тривалий час у сеансі роботи не запускає ніяких процесів, то після того, як все ж таки виконає запуск великої кількості процесів, то вони отримають невиправдано велику частку процесорного часу.

На основі алгоритму гарантованого планування побудовано алгоритм планування процесорного часу в ОС Linux.

3.3.13. Алгоритм справедливого планування FSS

На відміну від гарантованого планування в алгоритмі справедливого планування FSS (англ. Fair-Share Scheduling) фокус зміщується на користувача. Планування FSS зазвичай використовується в багатокористувацьких системах і віртуалізованих середовищах, де кілька користувачів або віртуальних машин спільно використовують один фізичний процесор. Зокрема, у багатокористувацьких системах основна ідея полягає в тому, щоб кожному користувачеві (та/або групі користувачів) виділяти достатню кількість процесорного часу, щоб гарантувати, що жоден користувач не втратить можливості використовувати систему.

Якщо припустити, що кожен процес планується окремо, незалежно від того, хто є його власником, то може виникнути така ситуація: якщо користувач User1 запускає 4 процеси (A, B, C, D), а користувач User2 запускає 1 процес (E) з однаковими пріоритетами, то User1 отримає 80 % процесорного часу, а User2 лише 20 %. Для того щоб запобігти такій ситуації можна враховувати власників процесів, перш ніж їх планувати. В алгоритмі FSS кожному користувачеві (або групі процесів, якими володіє цей

користувач) виділяється певна «справедлива» частка (англ. Fair Share) процесорного часу.

Наприклад, при рівному поділу користувачі User1 і User2 отримають по 50 % процесорного часу незалежно від кількості процесів у кожного з них, можлива послідовність планування, яка відповідає такому обмеженню, буде такою:

```
A E B E C E D E A E B E C E D E...
```

У тому випадку, коли користувач User1 має право на вдвічі більшу частку процесорного часу, ніж User2, отримаємо таку послідовність:

```
A B E C D E A B E C D E A B E C...
```

Другий приклад демонструє динаміку змін: якщо чотири користувачі (User1, User2, User3, User4) одночасно виконують по одному процесу кожен, планувальник розподілить процесорний час таким чином, щоб кожен користувач отримував 25 % від загального часу. Якщо користувач User2 починає виконувати другий процес, кожен користувач все одно отримає 25 % від загальної кількості циклів, але кожен із процесів користувача User2 тепер матиме 12.5 % процесорного часу. З іншого боку, якщо новий користувач User5 запускає процес у системі, планувальник перерозподілить доступний процесорний час таким чином, щоб кожен користувач отримував 20 % від загального часу (відповідно 2 процеси користувача User2 матимуть по 10 % процесорного часу).

Справедливе планування – це стратегія планування, в якій використання ЦП рівномірно розподіляється не тільки між користувачами, але й між групами користувачів.

У цьому випадку доступний процесорний час розподіляється спочатку між групами, потім між користувачами в групах, а потім між процесами для користувачів. Наприклад, якщо існує три групи (Group1, Group2, Group3), що містять відповідно чотири, два та три користувача, процесорний час буде розподілений наступним чином:

```
100 % / 3 групи = 33,3 % на групу
Group1: (33,3 % / 4 користувача) = 8,3 % на користувача
Group2: (33,3 % / 2 користувача) = 16,7 % на користувача
Group3: (33,3 % / 3 користувача) = 11,1 % на користувача
```

Одним із поширених методів логічної реалізації стратегії планування справедливого розподілу є рекурсивне застосування стратегії циклічного планування на кожному рівні абстракції (процеси, користувачі, групи тощо). Рівний розподіл часу дасть однакові результати.

Більш складний приклад демонструє використання кількох рівнів абстракції. Припустимо, що є 3 групи процесів (Group1, Group2, Group3) та 3 процеси (A, B, C), які входять до різних груп (один процес може входити до кількох груп одразу).

Перша група Group1 складається з 3 процесів (A використовує 1/6 процесорного часу в групі, B використовує 2/6 процесорного часу в групі, C використовує 3/6 процесорного часу в групі) та використовує 1/4 загального процесорного часу.

Друга група Group2 складається з 2 процесів (B використовує 2/5 процесорного часу в групі, C використовує 3/5 процесорного часу в групі) та використовує 2/4 загального процесорного часу.

Третя група Group3 складається з 1 процеса (C використовує 100 % процесорного часу в групі) та використовує 1/4 загального процесорного часу.

Кожен процес тоді повинен отримати такі частки загального процесорного часу:

```
Процес A  (1/6 * 1/4) = 4.2 %,
Процес B  (2/6 * 1/4) + (2/5 * 2/4) = 28.3 %,
Процес C  (3/6 * 1/4) + (3/5 * 2/4) + (1 * 1/4) = 67.5 %.
```

Для встановлення переважного права на отримання процесорного часу процесом у разі використання груп процесів застосовуються пріоритети. Планування виконується на основі пріоритету, який враховує базовий

пріоритет процеса, його останнє використання процесора та останнє використання процесора групою, до якої належить процес. Чим вище числове значення пріоритету, тим нижчий пріоритет.

Для процеса j у групі k для початку інтервалу часу i пріоритет $P_j(i)$ розраховується за наступною формулою:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2},$$
$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2},$$
$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k},$$

де $CPU_j(i)$ – показник навантаження (ціле число) процесора процесом j впродовж інтервалу i; $GCPU_k(i)$ – показник навантаження (ціле число) процесора групою k впродовж інтервалу i; $Base_j$ – базовий пріоритет (ціле число) процеса j; W_k – ваговий коефіцієнт, призначений групі k з обмеженнями $0 < W_k \leq 1$ та $\sum_k W_k = 1$.

Кожному процесу j присвоюється базовий пріоритет $Base_j$. Пріоритет процеса j падає, оскільки процес використовує процесор і група k, до якої належить процес, також відповідно використовує процесор. У випадку навантаження процесора групою k середнє значення нормалізується шляхом ділення на вагу W_k цієї групи. Чим більша вага W_k , яка призначається групі k, тим менше навантаження групи на процесор впливатиме на її пріоритет.

Наприклад, є один процес (A), що належить групі Group1 ($k = 1$), та два процеси (B, C), що належать групі Group2 ($k = 2$). Кожна група має вагу $W_k = 0.5$. Всі процеси обмежені можливостями процесора (CPU-bound process) і готові до виконання. Всі процеси мають базовий пріоритет $Base_j = 60$. Навантаження процесора вимірюється таким чином: ЦП переривається 60 разів на секунду; під час кожного переривання час використання процесора для поточного процеса та групи, в яку входить цей процес, збільшується. Один раз на секунду пріоритети перераховуються.

Табл. 3.12 демонструє результати обчислень пріоритетів під час перших 6 секунд виконання процесів.

Таблиця 3.12 – Приклад роботи алгоритму FSS з пріоритетами

i	Група Group1			Група Group2				
	Процес А		$GCPU_1(i)$	Процес В		Процес С		$GCPU_2(i)$
	$P_A(i)$	$CPU_A(i)$		$P_B(i)$	$CPU_B(i)$	$P_C(i)$	$CPU_C(i)$	
0	60	0	0	60	0	60	0	0
1	90	30	30	60	0	60	0	0
2	74	15	15	90	30	75	0	30
3	96	37	37	74	15	67	0	15
4	78	18	18	81	7	93	30	37
5	98	39	39	70	3	76	15	18

На початку роботи у всіх процесів пріоритети $P_j(0)$ будуть дорівнювати базовому пріоритету $Base_j = 60$. При однакових пріоритетах було обрано для виконання перший процес А. Через 1 секунду його виконання $CPU_A(1) = \frac{60}{2}$ та $GCPU_A(1) = \frac{60}{2}$, відповідно пріоритет процеса А

$$P_A(1) = 60 + \frac{30}{2} + \frac{30}{4\times0.5} = 90.$$

На наступному кроці обирається процес В, як перший процес з найменшим пріоритетом. Перерахунок пріоритетів:

$$CPU_A(2) = \frac{30}{2},\ GCPU_A(2) = \frac{30}{2},\ P_A(2) = 60 + \frac{15}{2} + \frac{15}{4\times0.5} = 74.$$
$$CPU_B(2) = \frac{60}{2},\ GCPU_B(2) = \frac{60}{2},\ P_B(2) = 60 + \frac{30}{2} + \frac{30}{4\times0.5} = 90.$$
$$CPU_C(2) = \frac{0}{2},\ GCPU_C(2) = \frac{60}{2},\ P_C(2) = 60 + \frac{0}{2} + \frac{30}{4\times0.5} = 75.$$

На наступному кроці буде обрано процес А (найменший пріоритет). У результаті після 6 секунд роботи послідовність отримання процесорного часу буде такою:

A B A C A B.

Переваги справедливого планування FSS:

♦ справедливість – забезпечує справедливий розподіл ресурсів ЦП між різними групами процесів, незалежно від їхніх пріоритетів або вимог до ресурсів;

♦ використання ресурсів – оптимізує використання ресурсів шляхом розподілу ресурсів ЦП на основі минулого використання та прав, а не на основі схеми фіксованого пріоритету;

♦ запобігання «голоду» – запобігає браку ресурсів шляхом розподілу ресурсів ЦП на основі прав групи процесів і невикористаних ресурсів;

♦ передбачуваність – забезпечує передбачуваний розподіл ресурсів ЦП для кожної групи процесів, що може покращити стабільність і продуктивність системи;

♦ масштабованість – може застосовуватися до систем із великою кількістю процесів або груп процесів, гарантуючи, що кожна група отримує свою справедливу частку ресурсів ЦП;

♦ гнучкість – дозволяє користувачам регулювати вагу кожної групи процесів на основі їхніх вимог до ресурсів і шаблонів використання, забезпечуючи гнучкість у розподілі ресурсів;

♦ покращена швидкість реагування − може покращити швидкість реагування системи, забезпечуючи справедливий розподіл ресурсів ЦП між різними групами процесів, зменшуючи імовірність монополізації ресурсів процесами, які інтенсивно використовують ЦП.

Недоліки справедливого планування FSS:

♦ накладні витрати − можуть бути великими в обчислювальному плані та вимагати додаткових витрат для відстеження використання ЦП і розрахунку ваг «справедливої» частки;

♦ складність – FSS складніше, ніж інші алгоритми планування, і вимагає додаткової конфігурації та керування для забезпечення належної роботи;

♦ може бути непридатним для систем із жорсткими вимогами до реального часу або для процесів, які вимагають фіксованого розподілу ресурсів ЦП;

♦ труднощі в налаштуванні − налаштування ваг «справедливої» частки може бути складним, оскільки вимагає розуміння вимог до ресурсів і моделей використання кожної групи процесів;

♦ обмежена ефективність − може бути неефективним у системах з великою кількістю короткотривалих процесів або в системах з дуже різними вимогами до ресурсів.

Існує кілька варіацій алгоритму справедливого планування, які використовується не тільки для планування процесорного часу, але й інших ресурсів (наприклад – обробки мережевих пакетів даних):

♦ зважена справедлива черга WFQ (англ. Weighted Fair Queuing) – призначає вагу кожному процесу на основі його характеристик трафіку та гарантує, що кожен процес отримує належну частку часу ЦП;

♦ узагальнене спільне використання процесора GPS (англ. Generalized Processor Sharing) − більш загальна версія WFQ, що забезпечує точніший контроль над розподілом ресурсів ЦП. Алгоритм заснований на концепції моделі рідини та призначає віртуальний процесор для кожного процеса;

♦ стохастична черга справедливості SFQ (англ. Stochastic Fairness Queuing) – цей алгоритм розділяє доступну пропускну здатність на часові слоти однакового розміру та призначає кожному процесу певний часовий слот на основі його хеш-значення. Це гарантує, що кожен процес отримує справедливу частку процесорного часу;

♦ чесна черга FQ (англ. Fair Queueing) – призначає чергу для кожного процеса та обслуговує черги циклічним способом, гарантуючи, що кожна черга отримує належну частку процесорного часу.

Алгоритм FSS використовується в ОС Solaris для процесів з пріоритетами 0-59. В ОС Solaris підтримується 170 рівнів пріоритетів PRI:

0-59 – класи планувальників TS (Timesharing), який планується за алгоритмом MLFQ; IA (Interactive), який відрізняється від TS збільшеним на 10 пріоритетом (встановлюється командою «nice -3.3») та перевернутою чергою; FSS (Fair-Share Scheduler) та FX (Fixed-Priority);

80-99 – клас планувальника SYS (System);

100-159 – клас планувальника RT (Real-Time);

160-169 – обробники переривань.

При плануванні в класі TS враховується значення «user priority» (який встановлюється командою «*nice* -3*значення»). Коли нитка використовує свій квант (розмір якого також залежить від пріоритету PRI) на процесорі, від значення PRI віднімається 10 і додається «user priority». Один раз на секунду і при поверненні з черги очікування «sleep queue» значення PRI встановлюється в практично максимальне значення діапазону: 50-59 (додається значення «user priority» з урахуванням меж діапазону пріоритетів цього класу).

3.3.14. Традиційне планування UNIX

Алгоритм традиційного планування UNIX [24, 28] використовується в SVR3 і в BSD UNIX 4.3 (в сучасних версіях він змінений). Алгоритм планування розроблений таким чином, щоб забезпечити прийнятний час відгуку для інтерактивних користувачів, водночас гарантуючи відсутність «голодування» низькопріоритетних завдань. В алгоритмі використовуються багаторівневі черги зі зворотним зв'язком із застосуванням планування RR у межах кожної черги (для кожної групи рівнів пріоритетів), розмір кванта дорівнює 1 секунді: якщо поточний процес не блокується або не завершується в межах однієї секунди, він витісняється. Пріоритет розраховується подібно до розрахунку в алгоритмі FSS:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} \text{ (для SVR3)},$$

$$CPU_j(i) = CPU_j(i-1) \times \frac{\text{load\_avg}}{2 \times \text{load\_avg} + 1} \text{ (для BSD 4.3)},$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j \ ,$$

де $CPU_j(i)$ – показник навантаження (ціле число) процесора процесом j впродовж інтервалу i; load_avg – середня кількість процесів у стані готовності до запуску (стан «ready to run in memory»); $Base_j$ – базовий пріоритет (ціле число) процеса j; $nice_j$ – коефіцієнт, що вказується користувачем.

Пріоритет кожного процеса перераховується один раз на секунду, під час завершення кванта часу (момент перепланування). Призначення базового пріоритету полягає в поділі процесів на фіксовані групи рівнів пріоритетів. Значення компонентів $CPU_j(i)$ та $nice_j$ обмежені вимогою того, щоб процес не міг вийти з призначеної йому на підставі базового пріоритету групи. Ці групи використовуються для оптимізації доступу до пристроїв і забезпечення швидкого відгуку ОС на системні виклики. У порядку зменшення пріоритету групи є такими:

- swapper (процес підкачування в пам’ять з ідентифікатором 0);
- процеси управління блочними пристроями введення/виведення;
- процеси управління файлами;
- процеси управління символьними пристроями введення/виведення;
- процеси користувача.

Коефіцієнт $nice_j$ дозволяє користувачеві добровільно зменшити пріоритет свого процеса (в межах групи), щоб надати перевагу іншим користувачам.

Наприклад, є три процеси (A, B, C), які обмежені можливостями процесора (CPU-bound process) і готові до виконання. Всі процеси мають базовий пріоритет $Base_j = 60$, добавка $nice_j$ – відсутня. Навантаження процесора вимірюється таким чином: ЦП переривається 60 разів на секунду; під час кожного переривання час використання процесора для поточного процеса збільшується. Один раз на секунду пріоритети перераховуються, використовується планування для SVR3.

У табл. 3.13 наведено результати обчислень пріоритетів під час перших 6 секунд виконання процесів.

Таблиця 3.13 – Приклад традиційного планування UNIX

i	Процес A		Процес B		Процес C	
	$P_A(i)$	$CPU_A(i)$	$P_B(i)$	$CPU_B(i)$	$P_C(i)$	$CPU_C(i)$
0	60	0	60	0	60	0
1	75	30	60	0	60	0
2	67	15	75	30	60	0
3	63	7	67	15	75	30
4	76	33	63	7	67	15
5	68	16	76	33	63	7

На початку роботи у всіх процесів пріоритети $P_j(0)$ будуть дорівнювати базовому пріоритету $Base_j = 60$. При однакових пріоритетах було обрано для виконання перший процес A. Через секунду його виконання $CPU_A(1) = \frac{60}{2}$, відповідно пріоритет $P_A(1) = 60 + \frac{30}{2} = 75$.

На наступному кроці обирається процес B, як перший процес з найменшим пріоритетом. Перерахунок пріоритетів:

$$CPU_A(2) = \frac{30}{2},\ P_A(2) = 60 + \frac{15}{2} = 67.$$
$$CPU_B(2) = \frac{60}{2},\ P_B(2) = 60 + \frac{30}{2} = 75.$$
$$CPU_C(2) = \frac{0}{2},\ P_C(2) = 60 + \frac{0}{2} = 60.$$

На наступному кроці буде обрано процес C (найменший пріоритет). У результаті після 6 секунд роботи послідовність отримання процесорного часу буде такою:

A B C A B C.

Короткі висновки:

♦ переваги алгоритму – порівняно проста реалізація; підходить для систем з розподілом часу та систем загального призначення. Дозволяє

уникати нескінченне відкладання виконання процесів та дає перевагу процесам, які часто виконують операції введення/виведення;

♦ недоліки алгоритму – погано масштабується, відсутність гарантій процесам. Користувачі не можуть налаштувати планування (за винятком коефіцієнта $nice_j$). Процес, що працює в режимі ядра протягом тривалого часу, може затримати всю систему (інверсія пріоритетів). Також відсутня підтримка багатопроцесорних та багатоядерних систем.

3.3.15. Статичний алгоритм планування RMS

Статичний алгоритм планування RMS (англ. Rate Monotonic Scheduling) використовується в ОС реального часу. У статичному алгоритмі пріоритети процесів визначаються на етапі їх створення, тобто перед запуском планувальника процесорного часу і на протязі всієї роботи не змінюються.

У теорії ОС реального часу виділяють періодичні та аперіодичні завдання (англ. Periodic and Aperiodic Tasks) [11]. Періодичне завдання це те, що виконується відповідно до заздалегідь чітко відомого розкладу і накладає жорсткі обмеження на час виконання – граничні строки виконання (англ. Hard Deadlines). На відміну від цього, аперіодичні завдання виконуються заздалегідь невизначений час і можуть накладати як жорсткі, так і м'які вимоги до часу виконання (англ. Soft Deadlines). На рис. 3.17 наведено основні часові характеристики для періодичних завдань. Відносний граничний строк виконання обчислюється від початку періоду, та зазвичай співпадає з розміром періоду; c' – час, що залишився до завершення виконання завдання в поточний момент часу t.

Особливості реалізації періодичних та аперіодичних завдань на процесах залежить від побудови ОС реального часу – процес може перезапускатись або отримувати управління в певні моменти часу. У більшості ОС процес після виконання необхідних дій не завершується, а відкладається на час, через який буде необхідна його наступна активація.

Рисунок 3.17 – Часові характеристики періодичних завдань в ОС реального часу

Завдання в ОС реального часу – одиниця обробки, що виконується конкурентно з іншими завданнями. Завдання є основним засобом обробки внутрішніх подій. Завдання має певне значення пріоритету, що визначає його відносні претензії захоплення процесора. Замість пріоритету можна використовувати значення терміну виконання. Завдання має точку входу (англ. Entry Point). Завдання зазвичай є функцією в програмі. Завдання зазвичай виконує виклики ОС для взаємодії з іншими завданнями. Зазвичай завдання вбудованої ОС еквівалентне нитці в звичайній ОС, тобто працює в одному адресному просторі з іншими завданнями.

Далі будемо припускати саме таку реалізацію. Нижче наведено приклад реалізації періодичних дій для ОС FreeRTOS у межах одного процеса:

```
// виконання дій кожні 10 тіків таймера
 void vTaskFunction( void * pvParameters )
 {
 TickType_t xLastWakeTime;
 const TickType_t xFrequency = 10;
   // Ініціалізація змінної xLastWakeTime значенням поточного часу
   xLastWakeTime = xTaskGetTickCount();
   for( ;; )
   {   // Очікування наступного періоду
       vTaskDelayUntil( &xLastWakeTime, xFrequency );
       // Виконання періодичних дій
       ...
   }
 }
```

Сутність алгоритму RMS полягає в призначенні статичних пріоритетів періодичним завданням на основі граничних строків виконання їх періодів.

Алгоритм RMS потребує виконання наступних умов:

♦ всі завдання незалежні між собою;

♦ кожне завдання має бути завершено протягом свого періоду;

♦ кожне завдання може бути призупинено більш пріоритетним завданням;

♦ час виконання кожного завдання постійний;

♦ переривання завдання відбувається миттєво.

Набір завдань можна запланувати за алгоритмом RMS, лише якщо вони задовольняють такому рівнянню:

$$U = \sum_{i=0}^{n} \frac{c_i}{p_i} \leq n \times (2^{\frac{1}{n}} - 1),$$

де U – навантаження процесора; n – кількість завдань у системі; c_i – час виконання завдання i; p_i – період виконання завдання i.

При кількості завдань n, що прагне до нескінченності, максимальне навантаження процесора U дорівнює 69 %. Залишок процесорного часу може бути використаний для низькопріоритетних завдань. При використанні процесора менше ніж на 69 %, алгоритм RMS гарантує успішне виконання завдань у системі, інакше набір завдань може бути спланований, але їх виконання не гарантується.

У табл. 3.14 наведено вихідні дані для прикладу роботи планувальника за алгоритмом RMS (граничний строк співпадає з періодом).

Таблиця 3.14 – Вихідні дані для прикладу роботи RMS

Завдання	Час виконання (c_i)	Період виконання (p_i)
A	3	20
B	2	5
C	2	10

Завдання A виконується по 3 одиниці часу кожні 20 одиниць часу.
Завдання B виконується по 2 одиниці часу кожні 5 одиниць часу.
Завдання C виконується по 2 одиниці часу кожні 10 одиниць часу.
Перевірка можливості планування:

$$\left(\frac{3}{20}+\frac{2}{5}+\frac{2}{10}\right) \leq 3 \times (2^{\frac{1}{3}}-1),$$
$$0.75 \leq 0.7798.$$

Сукупне використання процесорного часу трьома завданнями є меншим за порогове значення, що означає, що наведений вище набір завдань можна запланувати:

♦ час планування – для розрахунку часу планування алгоритму береться найменше спільне кратне для періодів виконання всіх завдань. У наведеному прикладі це значення дорівнює 20. За цей час завдання A повинно отримати 3 одиниці процесорного часу, B – 8, C – 4. Таким чином, можна запланувати час періодами по 20 одиниць часу;

♦ пріоритет буде найвищим для завдання, яке має найменший період виконання. Таким чином, B матиме найвищий пріоритет, далі – C, і нарешті – A.

На рис. 3.18 наведено приклад роботи системи, де є три періодичні завдання згідно табл. 3.14 (блоки сірого кольору – завдання виконується на процесорі).

При плануванні на початку кожного періоду процесорний час виділяється завданню з самим великим пріоритетом, наступні проміжки часу – завданням по зменшенню пріоритетів. Якщо на початку виконується менш пріоритетне завдання – воно переривається на користь більш пріоритетного.

Червоними вертикальними лініями позначено граничні строки виконання завдань згідно їх періодам, номера на блоках вказують номери періодів виконання завдань.

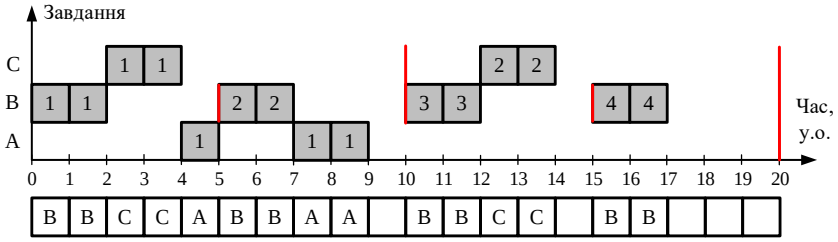


Рисунок 3.18 – Діаграма Ганта планування за алгоритмом RMS

Завдання B виконуватиметься першим протягом 2 одиниць часу, оскільки воно має найвищий пріоритет. Після завершення двох одиниць завдання C отримає процесорний час і працюватиме свої 2 одиниці часу (і повністю завершить своє виконання в своєму періоді – 10 одиниць часу). Далі завдання A, яке має найменший пріоритет, отримає процесорний час, і працюватиме 1 одиницю часу. У завдання B наступить новий пороговий проміжок у 5 одиниць, і, оскільки його пріоритет вище, ніж у A, то воно його витіснить і отримає 2 одиниці процесорного часу. Оскільки завдання C завершило свої 2 одиниці часу за свій інтервал у 10 одиниць часу, то процесорний час отримає A, та використає 2 одиниці часу (повністю завершить своє виконання в свій період у 20 одиниць часу). Інтервал часу 9-10 не використовується завданнями A, B, C, тому може бути використаний для інших непріоритетних завдань. Через 10 одиниць часу завдання B виконуватиметься 2 одиниці часу (його третій період – проміжок 10-15). Далі отримає процесорний час завдання C і завершить своє виконання на своєму другому періоді. Інтервал 14-15 знову не використовується завданнями A, B, C. Через 15 одиниць часу від початку завдання B отримає 2 одиниці процесорного часу та завершить своє виконання на своєму четвертому періоді. Останні 3 одиниці процесорного часу так само можуть бути використані іншими непріоритетними завданнями. З моменту часу 20 послідовність буде повторена.

П’ять не зайнятих інтервалів часу в прикладі, як було вказано вище – можуть використовуватись непріоритетними завданнями. Такий підхід лежить у модифікації алгоритмів планування для систем реального часу (як

статичних, так і динамічних), який має назву – Background Server. В основі модифікації Background Server лежить фонове виконання аперіодичних завдань, тобто періодичні завдання системи обслуговуються на основі будь-якого основного алгоритму (RMS, EDF, LLF), а у вікнах, що залишилися, процесорний час виділяється для аперіодичних завдань, які готові до виконання в даний момент часу. Аперіодичні завдання обслуговуються тільки тоді, коли всі періодичні завдання призупинено. Якщо аперіодичне завдання не встигло завершити виконання, тобто було витіснене, воно повертається в кінець черги готових аперіодичних завдань (обслуговування черги ведеться за алгоритмом FCFS). Основним недоліком модифікації Background Server є те, що вона не дозволяє працювати з аперіодичними завданнями реального часу.

Існує також модифікація Deferrable Server (сервер, що допускає затримку) – у системі створюється сервер періодичних завдань, який призначений для обслуговування аперіодичних завдань і має такий параметр як бюджет. За кількістю бюджету планувальник визначає, чи буде оброблятися в даний період сервера аперіодичне завдання. Якщо значення бюджету дорівнює нулю, то управління отримують готові періодичні завдання до моменту оновлення бюджету сервера, тобто до початку наступного періоду сервера.

Модифікація Sporadic Server (спорадичний сервер) за принципом аналогічна попередньому варіанту: для обслуговування аперіодичних завдань створюється спеціальне періодичне завдання (сервер), що має максимальний пріоритет. Аналогічно до Deferrable Server, у модифікації Sporadic Server у сервера є свій бюджет, після використання якого управління передається наступному завданню зі списку готових до виконання завдань. Єдина відмінність Sporadic Server від Deferrable Server полягає в тому, що бюджетний час поповнюється не в кожен період сервера, а через «період поповнення», тобто через визначену кількість часу після початку чергового періоду сервера.

Другий приклад демонструє поведінку алгоритму RMS у разі порушення умови навантаження процесора.

В табл. 3.15 наведено вихідні дані (відносний граничний строк співпадає з періодом p_i).

Таблиця 3.15 – Вихідні дані для прикладу роботи RMS (приклад 2)

Завдання	Час виконання (c_i)	Період виконання (p_i)
A	1	4
B	2	6
C	3	8

Перевірка можливості планування:

$$\left(\frac{1}{4}+\frac{2}{6}+\frac{3}{8}\right) \le 3 \times (2^{\frac{1}{3}}-1),$$

$0.958 > 0.7798$ (порушення умови!).

Сукупне використання процесорного часу трьома завданнями є більшим за порогове значення, умову порушено. Час планування – найменше спільне кратне для періодів виконання всіх завдань (24 одиниці часу), найвищий пріоритет у завдання з найменшим періодом виконання – A.

На рис. 3.19 наведено другий приклад роботи системи (замальовані сірим кольором блоки – завдання виконується на процесорі, замальований жовтим кольором блок – завдання виконується на процесорі, але перевищений граничний строк його виконання, номери на блоках вказують номери періодів виконання завдань).

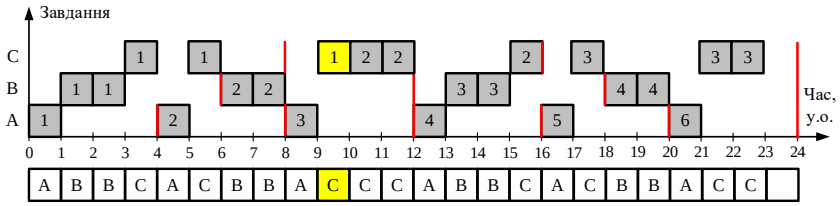


Рисунок 3.19 – Діаграма Ганта планування за алгоритмом RMS (приклад 2)

Як видно, завдання С не встигає виконатись у граничний строк у першому періоді свого виконання.

Короткі висновки:

♦ переваги алгоритму – проста реалізація, як у будь-яких алгоритмів зі статичним призначенням пріоритетів; достатньо спланувати час, який дорівнює найменшому спільному кратному для періодів виконання всіх завдань і потім отриманий результат повторювати;

♦ недоліки алгоритму – дуже складно підтримувати аперіодичні та спорадичні завдання. Існує небезпека тупикових блокувань взаємодіючих завдань.

3.3.16. Динамічний алгоритм планування EDF

У динамічних алгоритмах планування в ОС реального часу пріоритети завданням призначаються під час роботи системи на основі аналізу поточної ситуації. Динамічні алгоритми бувають двох типів: для періодичних та аперіодичних завдань. Алгоритм планування по найближчому терміну завершення (граничному строку виконання) EDF (англ. Earliest Deadline First) – це оптимальний динамічний алгоритм планування пріоритетів, який використовується для вбудованих систем реального часу.

Його можна використовувати як для статичного, так і для динамічного планування в реальному часі. Має переваги над статичним алгоритмом RMS при великих навантаженнях системи. Але цей алгоритм не стабільний, оскільки низькопріоритетні завдання можуть отримувати процесорний час при великому навантаженні системи.

Суть алгоритму EDF полягає в тому, що в момент події планування, планувальником обирається завдання, що має ранній крайній строк виконання дій, тобто час, що залишився до кінця періоду. Основними подіями планування ОС реального часу є готовність завдання до виконання та призупинення виконання завдання. Формальною умовою даного алгоритму є те, що максимальне навантаження процесора має бути менше 100 %.

Пріоритет завдання обернено пропорційний його граничному строку виконання. Оскільки граничний строк виконання залежить від поточного

моменту часу, завдання, яке має вищий пріоритет через найближчий строк виконання в один момент, може мати низький пріоритет у наступний момент через більш ранній строк виконання іншого завдання. EDF зазвичай виконується в режимі витіснення, тобто завдання, що виконується, витісняється щоразу, коли з'являється інше завдання з найближчим кінцевим строком.

EDF дуже ефективний порівняно з RMS. Він може збільшити використання ЦП майже до 100 % (69 % у RMS), гарантуючи при цьому граничні строки виконання всіх завдань. Якщо використання ЦП менше 100 %, це означає, що всі завдання виконали свої граничні строки. Здійсненний графік – це той, в якому всі завдання в системі виконуються в установлені терміни. Якщо EDF не може знайти можливий розклад для всіх завдань у системі реального часу, це означає, що жоден інший алгоритм планування завдань у системах реального часу також не може дати можливий розклад.

Алгоритм EDF не потребує, щоб завдання були періодичними, а також вимагали фіксованого часу навантаження ЦП. Якщо два завдання мають однаковий абсолютний граничний строк виконання, обирається одне з них випадковим чином або те завдання, в якого строк менший, або вибір робиться з міркувань зменшення кількості перемикань між процесами.

Приклад планування за алгоритмом EDF для вихідних даних завдань з табл. 3.15 наведено на рис. 3.20 (відносні граничні строки також співпадають з періодами p_i).

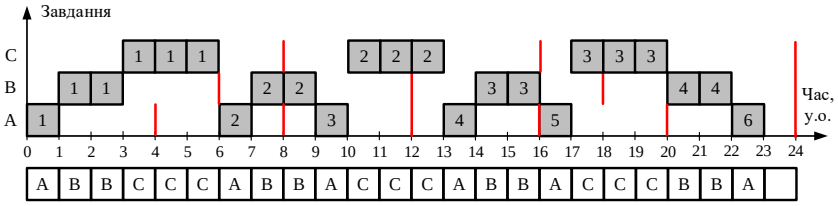


Рисунок 3.20 – Діаграма Ганта планування за алгоритмом EDF

Оскільки максимальне навантаження процесора $U = 0.958$ становить менше 100 %, даний набір завдань можна спланувати за алгоритмом EDF.

У момент часу 0 всі завдання починають свої періоди, але пріоритети визначаються відповідно до граничних строків, тому завдання A має вищий пріоритет, оскільки його строк самий малий (4).

У момент часу 1 знову порівнюються граничні строки, і завдання B має коротший строк, тому воно починає виконуватись. Далі починає роботу завдання C.

У момент часу 4 починається другий період виконання завдання A, у цей момент і A, і C мають однакові граничні строки, тому за умови мінімізації кількості перемикань контекстів процесів продовжується виконання завдання C.

У момент часу 6 розпочинає свій другий період завдання B, але крайній строк для A наступає раніше, ніж для B, тому в цей момент часу починає виконання завдання A, а в момент 7 – завдання B.

У момент часу 8 A і B мають однакові строки, тому завдання B продовжує виконання.

У момент часу 12 A і B розпочинають свої наступні періоди одночасно, тому, порівнюючи граничні строки, продовжується виконання завдання C.

Аналогічним чином далі система продовжує працювати, дотримуючись алгоритму EDF. Таким чином, на відміну від алгоритму RMS, порушень граничних строків немає. Послідовність повністю повторюється через 24 одиниці часу – найменше спільне кратне для періодів виконання всіх завдань.

Під час перенавантаження (короткочасного перевантаження) процесора при роботі за алгоритмом EDF виникає ефект «доміно» (англ. Domino Effect). Умова тимчасового перевантаження виникає, коли потреба в часі обчислення завдання, виконуваного в даний момент, перевищує доступну в цей момент потужність процесора. Через тимчасове перевантаження завдання прострочують свої граничні строки виконання. Це тимчасове перевантаження може виникнути через багато причин, наприклад зміни в середовищі, одночасне надходження асинхронних завдань, виключення системи тощо. В ОС реального часу під керуванням EDF стан, коли завдання в стані тимчасового перевантаження пропускає свій граничний строк і в результаті кожне з наступних завдань також починає пропускати свій

граничний строк, називається ефектом «доміно». Виникнення ефекту «доміно» ставить під загрозу поведінку всієї системи. Приклад такого стану наведено на рис. 3.21 з даними з табл. 3.16 (відносні граничні строки також співпадають з періодами p_i).

Таблиця 3.16 – Вихідні дані для прикладу роботи EDF (ефект «доміно»)

Завдання	Час виконання (c_i)	Період виконання (p_i)
A	2	5
B	2	6
C	2	7
D	2	8

Максимальне навантаження процесора $U = 1.27$. Починаючи з моменту часу 15 завдання не встигають виконувати свої дії в межах своїх періодів (на рис. 3.21 такі проміжки часу позначені жовтим кольором). Видно, що EDF має ефект «доміно», і в результаті критичні завдання можуть не встигати виконуватись у свої строки. Як рішення цієї проблеми пропонується використання іншого алгоритму планування – LLF, як більш оптимального.

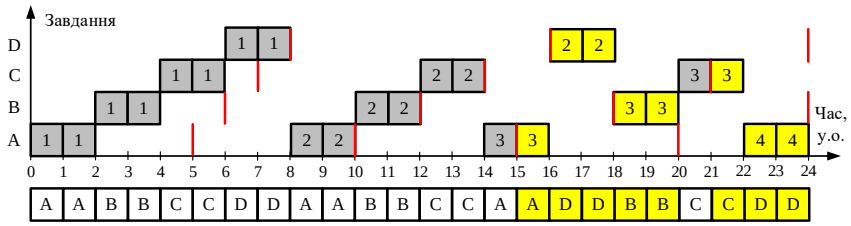


Рисунок 3.21 – Діаграма Ганта планування за алгоритмом EDF (ефект «доміно»)

Короткі висновки:

♦ переваги алгоритму – алгоритм EDF гарантує, що завдання з найбільш ранніми граничними строками виконуються першими, мінімізує ймовірність пропуску строків, максимізує використання ЦП – оптимізує використання системних ресурсів за рахунок мінімізації часу простою. EDF

забезпечує передбачуваність щодо часу та термінів виконання завдань – рішення щодо планування є детермінованими, їх можна проаналізувати та передбачити заздалегідь, що має вирішальне значення для систем реального часу. EDF може виконувати як періодичні, так і аперіодичні завдання;

♦ недоліки алгоритму – алгоритм EDF менш передбачуваний, ніж RMS. EDF забезпечує менший контроль над виконанням завдань та високі накладні витрати.

3.3.17. Алгоритм LLF

Алгоритм динамічного планування «з найменшим резервом часу – перший» LLF (англ. Least Laxity First) або «з найменшим слабким часом – перший» LSF (англ. Least Slack First) заснований на наступному принципі – у момент настання події планування найвищий пріоритет отримує завдання з найменшим резервом часу. Резерв часу або слабкий час (англ. Slack Time) визначається як різниця між часом, що залишився до граничного строку, і залишком часу, який потрібен завданню для завершення виконання, тобто проміжок часу, на який виконання завдання може бути відкладено без пропуску його граничного строку виконання.

Як і в алгоритмі EDF, формальною умовою даного алгоритму є те, що максимальне навантаження процесора має бути менше 100 %. LLF є оптимальним алгоритмом, тому що, якщо набір завдань пройде перевірку на використання, то він безперечно може бути запланованим за допомогою LLF. Ще одна перевага LLF полягає в тому, що в цьому алгоритмі заздалегідь можна визначити, яке саме завдання може пропустити свій граничний строк виконання.

Резерв часу L_i для кожного завдання i визначається як різниця між абсолютним граничним строком d_i, реальним часом, що минув з початку запуску t, та часом виконання завдання, що залишився, c_i':

$$L_i = (d_i - t) - c_i'.$$

Негативне значення резерву часу L_i сигналізує про порушення завданням i граничного строку виконання.

Наступний приклад пояснює роботу алгоритму LLF. В табл. 3.17 наведено вихідні дані для прикладу (відносний граничний строк для всіх завдань співпадає з періодом p_i). Значення навантаження процесора дорівнює 88.3 %.

Таблиця 3.17 – Вихідні дані для прикладу роботи LLF

Завдання	Час виконання (c_i)	Період виконання (p_i)
A	2	6
B	2	8
C	3	10

На рис. 3.22 наведено приклад роботи алгоритму LLF (замальовані сірим кольором блоки – завдання виконується на процесорі, номери на блоках вказують розраховані на початок інтервалу часу значення резервів L_i для кожного процеса, який очікує виконання). Червоними вертикальними лініями позначено граничні строки виконання завдань, що відповідають в даному випадку їх періодам p_i.

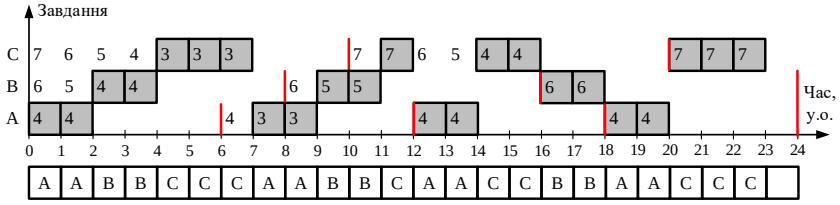


Рисунок 3.22 – Діаграма Ганта планування за алгоритмом LLF

На початку роботи всі 3 завдання готові до виконання. Виконується розрахунок резерву часу:

$$L_A = (6 - 0) - 2 = 4,$$
$$L_B = (8 - 0) - 2 = 6,$$
$$L_C = (10 - 0) - 3 = 7.$$

Найменший резерв у завдання A, саме воно на початку отримує процесорний час. У момент часу 1 резерви розраховуються знову:

$$L_A = (6 - 1) - 1 = 4,$$
$$L_B = (8 - 1) - 2 = 5,$$
$$L_C = (10 - 1) - 3 = 6.$$

Завдання A продовжує виконуватись. У момент часу 2 завдання A в своєму першому періоді завершило виконання, тому розрахунок резервів виконується тільки для готових до виконання завдань:

$$L_B = (8 - 2) - 2 = 4,$$
$$L_C = (10 - 2) - 3 = 5.$$

Процесорний час отримує завдання B. Далі розрахунки і вибір завдання виконуються аналогічним чином. Так, наприклад, для моменту часу 12, коли готові до виконання завдання A (почався новий період виконання) і C (вже в поточному періоді одну одиницю часу було використано), резерви будуть наступними:

$$L_A = (18 - 12) - 2 = 4,$$
$$L_C = (20 - 12) - 2 = 6.$$

Оскільки резерв у завдання A менше ніж резерв у раніше виконуваного завдання C, то завдання C витісняється і процесорний час віддається завданню A.

У разі, якщо найменші значення резервів співпадають у кількох завдань – обирається одне з них випадковим чином, або продовжує виконуватись раніше перерване планувальником завдання.

Ситуація в момент часу 12 демонструє стан, коли в системі з'являється завдання з меншим резервом. Очевидно, що інше завдання повинно бути витісненим. Іноді виникають ситуації, коли після кожного кроку планування мінімальний резерв буде не в поточного, а в іншого завдання, і тоді на

кожному кроці виконуються циклічні перемикання між двома (або більшою кількістю) завданнями – трешінг або буксування (англ. Thrashing). Такі перемикання контексту призводить до більшого споживання обчислювальних потужностей під час переходу від одного завдання до іншого, що супроводжується збереженням та завантаженням регістрів, відображенням пам'яті та оновленням багатьох таблиць і списків. Тобто призводить до значної втрати часу.

Для подолання ситуації буксування було розроблено вдосконалену версію LLF, яка є покращеним алгоритмом планування з найменшим резервом ELLF (англ. Enhanced Least Laxity First). В алгоритмі ELLF, коли більше ніж одне завдання має найменший резерв, вони групуються разом і всередині такої групи застосовується алгоритм EDF: коли LLF обирає таку групу для планування, всі завдання всередині групи планується за алгоритмом EDF, і після того, як всі завдання будуть сплановані, далі продовжує працювати LLF.

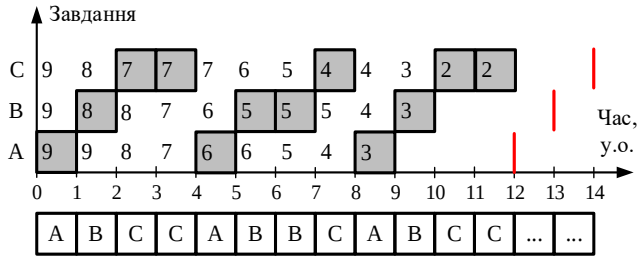
Припустимо, що є група завдань, параметри яких наведено в табл. 3.18.

Таблиця 3.18 – Вихідні дані для прикладу роботи LLF (буксування)

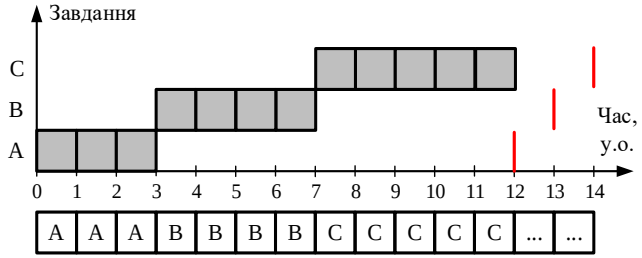
Завдання	Час виконання (c_i)	Період виконання (p_i)
A	3	12
B	4	13
C	5	14

Варіант планування завдань за алгоритмом LLF наведено на рис. 3.23, а.

При розрахунку резерву, на початку у всіх завдань буде однакове значення. Якщо виконувати планування, можна отримати і гірший варіант, ніж наведено – у прикладі вважається, що серед завдань з мінімальним резервом обирається завдання, яке в попередній момент часу вже виконувалось (для зменшення кількості перемикань контексту). Але, якщо використовувати випадковий вибір, кількість перемикань може бути навіть більшою. Якщо ж цю групу завдань планувати за алгоритмом EDF (рис. 3.23, б), кількість перемикань контексту буде значно менше.



а) планування групи завдань за алгоритмом LLF



б) планування групи завдань за алгоритмом EDF

Рисунок 3.23 – Ситуація буксування та її вирішення

Короткі висновки:

♦ переваги алгоритму – здатний обробляти як періодичні, так і аперіодичні завдання, забезпечує гарний баланс між дотриманням граничних строків і ефективним використанням системних ресурсів;

♦ недоліки алгоритму – алгоритм LLF вимагає сортування списку завдань на основі резерву часу, що може бути коштовним з точки зору обчислень, особливо в системах із великою кількістю завдань. Також алгоритм має великі накладні витрати на організацію частих перемикань контекстів (покращений алгоритм ELLF має менший обсяг накладних витрат).

3.3.18. Перший алгоритм планування Linux

В ОС Linux до ядра версії 2.4 планування процесів виконувалось на основі класичного планувальника UNIX. Версія 0.01 є першим ядром Linux.

Планувальник процесів у цій версії складається лише з 20 рядків коду і дуже простий.

У версії ядра 0.01 усі процеси представлені одним масивом. Цей масив є не тільки списком усіх процесів, але й чергою виконання. Розмір цього масиву становить 64 елементи. Це означає, що кількість процесів у цій версії не більше 64. У цьому масиві порожній запис виражено як *NULL*. Проміжок часу планувальника становить 150 мілісекунд. Апаратне забезпечення, яке називається інтервальним таймером, визначає, чи поточний час (*current*) вичерпує свій відрізок часу (квант). Інтервальний таймер перериває ЦП раз на 10 мілісекунд і викликається обробник, зареєстрований за планувальником *schedule*. Ця функція зменшує проміжок часу *current*, і якщо він стає нульовим, планувальник планує наступний процес, який можна виконати, з черги виконання.

Текст функції планувальника *schedule*:

```
void schedule(void)
{
   int i,next,c;
   struct task_struct ** p;
/* перевірити будильник, розбудити будь-які процеси, що
перериваються, які отримали сигнал */
   for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
      if (*p) {
         if ((*p)->alarm && (*p)->alarm < jiffies) {
               (*p)->signal |= (1<<(SIGALRM-1));
               (*p)->alarm = 0;
            }
         if ((*p)->signal && (*p)->state==TASK_INTERRUPTIBLE)
            (*p)->state=TASK_RUNNING;
      }
/* власне планування: */
   while (1) {
      c = -1;
      next = 0;
      i = NR_TASKS;
      p = &task[NR_TASKS];
      while (--i) {
         if (!*--p)
```

```
            continue;
         if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
            c = (*p)->counter, next = i;
      }
      if (c) break;
      for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
         if (*p)
            (*p)->counter = ((*p)->counter >> 1) +
                  (*p)->priority;
   }
   switch_to(next);
}
```

У змінній *i* зберігається індекс процеса в таблиці завдань (саме індекс з таблиці, а не ідентифікатор процеса), у змінній *p* – покажчик на структуру *task_struct*. Планувальник бере процес з найбільшим значенням *counter* і перемикається на нього. Якщо всі процеси, які можна виконувати, мають значення *counter* 0, він присвоює значенню *counter* кожного процеса *counter = (counter >> 1) + priority* (фактично реалізація формули з класичного планування UNIX) і перезапускає цикл. При оновленні лічильника також оновлюється значення лічильника процесів, які не можна виконувати. Нарешті, *switch_to(next)* – це макрос, що перемикає контекст ЦП на обраний процес (більш детально він описаний у розділі 2.3.2).

Планувальник процесів цієї версії підтримує значення *nice* (в планувальнику за це відповідає змінна *priority*), яке можна змінювати системним викликом *nice* (функція ядра *sys_nice*):

```
int sys_nice(long increment)
{
   if (current->priority-increment>0)
      current->priority -= increment;
   return 0;
}
```

Якщо значення *nice* менше нуля, *counter* збільшується і навпаки. Перевірка на початку системного виклику відсутня, тому в версії ядра 0.0.1 можна встановити будь-яке значення пріоритету для процеса.

До речі – «паніка ядра» (про яку згадували в розділі 1.5.4) в версії ядра 0.0.1 реалізована також дуже просто – виведення повідомлення та перехід у нескінченний цикл, єдиним можливим рішенням після чого є перезавантаження системи:

```
volatile void panic(const char * s)
{
   printk("Kernel panic: %s\n\r",s);
   for(;;);
}
```

Переваги та недоліки першого алгоритму планування Linux повністю відповідають перевагам і недолікам класичного планувальника UNIX (хоча розробники в коментарі перед реалізацією функції планувальника вказали: «This is GOOD CODE! There probably won't be any reason to change this, as it should work well in all circumstances» – «Це ГАРНИЙ КОД! Швидше за все, не буде жодних причин змінювати його, він повинен добре працювати за всіх умов»).

У версії ядра 2.4 став використовуватись алгоритм O(n), який ідеологічно мало чим відрізнявся від версії 0.01. Відповідно до назви, алгоритм мав лінійну часову складність O(n)[77].

[77] Часова складність алгоритму O() (англ. Order of) – термін у галузі аналізу складності алгоритмів і структур даних в інформатиці. Використовується для оцінки верхньої межі (найгіршого випадку), часової складності алгоритму. Якщо збільшується кількість вхідних даних, то може зростати кількість операцій і час, за який виконується алгоритм. Також може зростати обсяг пам'яті. O() показує швидкість зростання часу виконання алгоритму. O() має математичну нотацію, яка визначає, як алгоритм виконуватиметься у найгіршому випадку, залежно від розміру вхідних даних. Деякі з варіантів:

♦ O(1) Константна складність. Час виконання алгоритму залежить від розміру вхідних даних. Наприклад, доступ до елемента масиву за індексом;

♦ O(log n) Логарифмічна складність. Час виконання алгоритму зростає повільно зі збільшенням обсягу вхідних даних. Наприклад, бінарний пошук у відсортованому масиві;

♦ O(n) Лінійна складність. Час виконання алгоритму пропорційний до розміру вхідних даних. Наприклад, перегляд всіх елементів у масиві;

Планувальник версії ядра 2.4 розділив час на так звані «епохи», які означають тривалість життя (часовий зріз) кожного процеса в системі. Таким чином, до кінця «епохи» кожен процес виконується один раз, зазвичай використовуючи весь поточний часовий інтервал. У крайньому випадку, коли процес не використав свій фрагмент повністю, половина цього часу додається до нового часового інтервалу, що дозволить йому виконуватися довше протягом наступної «епохи». Алгоритм перебирає всі процеси в системі (так, як у версії 0.01), обчислює якість (англ. Goodness) і обирає «найкращий» процес для виконання.

Фрагмент функції планувальника *schedule* демонструє ітераційну частину:

```
asmlinkage void schedule(void)
{
   /*дескриптори процесів: prev: виконуваний зараз;
   next: наступний для запуска; p: поточний (для пошуку) */
   struct task_struct *prev, *next, *p;
   ...
   int this_cpu, c; /* weight */
   ...
   repeat_schedule:
      next = idle_task(this_cpu);
      c = -1000;
      ...
      list_for_each(tmp, &runqueue_head) {
         p = list_entry(tmp, struct task_struct, run_list);
```

♦ O(n log n) Лінійно-логарифмічна складність. Час виконання алгоритму зростає швидше, ніж лінійно, але повільніше, ніж квадратично. Наприклад, сортування злиттям (англ. Merge Sort);

♦ O(n^2) Квадратична складність. Час виконання алгоритму залежить від квадрата розміру вхідних даних. Наприклад, сортування бульбашкою (англ. Bubble Sort);

♦ O(n^3) Кубічна складність. Час виконання алгоритму залежить від обсягу вхідних даних у кубі. Наприклад, алгоритми, які мають три вкладені цикли;

♦ O(n!) Факторіальна складність. Найвища міра зростання часу виконання алгоритму. Час виконання алгоритму зростає факторіально від обсягу вхідних даних. Наприклад, перебір всіх можливих комбінацій елементів.

```
            if (can_schedule(p, this_cpu)) {
               int weight = goodness(p, this_cpu, prev->active_mm);
            if (weight > c)
               c = weight, next = p;
            }
         }
...
}
```

Функція *goodness()* визначає цінність процеса, засновану на кількості тактів процесора, що залишилися процесу, та деякій вазі на основі пріоритету завдання. Функція повертає цілі значення:

♦ -1000 – ніколи не обирати цей процес для запуску;

♦ 0 – поза часовим інтервалом, необхідно перерахувати лічильники (все ще може бути обраний для виконання);

♦ натуральне число – значення якості (чим більше, тим краще);

♦ +1000 – процес реального часу, обирається однозначно для виконання.

Незважаючи на те, що цей підхід був відносно простим, він був досить неефективним (витрачав надто багато часу на вибір найкращого процеса) з майже відсутнім потенціалом масштабованості та був слабким для систем реального часу. Також не вистачало функцій для використання нових апаратних архітектур, таких як багатоядерні процесори.

3.3.19. Алгоритм планування O(1) Linux

На зміну планувальнику O(n), починаючи з версії ядра 2.6.0, було запропоновано новий планувальник процесів, який міг планувати процеси протягом постійного проміжку часу.

У планувальнику O(1) для кожного процесора в системі визначається своя черга готових до виконання процесів. Кожна черга відстежує всі виконувані процеси, призначені для пов’язаного з нею ЦП, використовуючи 2 масиви: «активний» масив і «прострочений» масив. Планувальник обирає виконуваний процес з вищим динамічним пріоритетом з «активного» масиву для виконання на ЦП протягом попередньо визначеного кванта.

Черга готових до виконання процесів складається зі списків процесів:

```
struct runqueue {
 unsigned long nr_running; /* кількість готових процесів */
 ...
 struct prio_array *active; /* покажчик на «активний» масив */
 struct prio_array *expired; /* покажчик на «прострочений» масив */
 struct prio_array arrays[2]; /* поточні актуальні масиви */
}
```

У попередніх версіях планувальників процесів був явний спосіб перерахунку пріоритету кожного процеса в кінці епохи, коли всі процеси вичерпали свої кванти. Натомість O(1) перераховує пріоритет тільки одного процеса наприкінці його кванта. Процеси в межах одного пріоритету плануються циклічно. Після звільнення ЦП обчислюється наступний квант для процеса, і процес розміщується в «простроченому» масиві. Коли всі процеси в «активному» масиві вичерпають свої кванти (тобто «активний» масив стане порожнім), покажчики на «активний» і «прострочений» масиви міняються місцями і планування продовжується. Такий підхід дозволив позбутися циклів перерахунку і, відповідно, мінімізувати кількість процесів, які задіяні в плануванні, оскільки планувальник не враховує «прострочений масив».

Щоб забезпечити пріоритетність процесів у системі, кожен масив у черзі виконання є двовимірним, що представляє *MAX_PRIO* різних рівнів пріоритету. Кожна структура *prio_array* – це набір запущених процесів, який містить:

- *MAX_PRIO* двоспрямованих списків *list_head queue* заголовків (по одному списку для кожного значення пріоритету);
- бітову карту *bitmap* пріоритетів;
- кількість процесів *nr_active* у наборі.

```
struct prio_array {
 int nr_active; /* number of tasks */
 unsigned long bitmap[BITMAP_SIZE]; /* priority bitmap */
 struct list_head queue[MAX_PRIO]; /* priority queues */
};
```

MAX_PRIO – це кількість пріоритетів у системі за замовчуванням, у більшості випадків дорівнює 140. *BITMAP_SIZE* – це розмір масиву беззнакових даних, щоб забезпечити по одному біту для кожного рівня пріоритету (для 140 пріоритетів необхідно п’ять 32-розрядних слів). Встановлений біт у бітовій карті вказує, що список з відповідним пріоритетом містить процеси.

Алгоритм вибору наступного процеса для виконання в планувальнику O(1) виглядає наступним чином. У бітовій карті «активного» масиву для першого знайденого встановленого біту в відповідному списку виконуваних процесів (це буде список з найвищим пріоритетом) планувальник обирає перший процес.

Вибір наступного процеса для виконання та перехід до нього реалізується як і раніше за допомогою функції *schedule()*. Ця функція явно викликається кодом ядра. Функція *schedule()* виконується кожним процесором незалежно. Отже, кожен центральний процесор приймає власні рішення щодо того, який процес запускати далі.

Наступний фрагмент коду функції *schedule()* визначає процес з найвищим пріоритетом:

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array *array;
int idx;

prev = current;
array = rq->active;
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);
```

Оскільки число пріоритетів є статичним (140), час для завершення пошуку наступного процеса, який можна виконати, планувальником O(1) буде постійним і не буде залежати від кількості запущених процесів у системі.

Рівні пріоритету від 0 (найвищий пріоритет реального часу) до 99 (найнижчий пріоритет реального часу) використовують режим реального часу. Рівні пріоритету від 100 (найвищий статичний пріоритет) до 139 (найнижчий статичний пріоритет) представляють звичайні процеси в класі планування SCHED_NORMAL. На статичний пріоритет процеса впливає значення $nice$ в діапазоні від -20 (найвищий пріоритет) до 19 (найнижчий пріоритет). Кожному процесу призначається базовий квант Q на основі його статичного пріоритету (з врахованим $nice$) за допомогою формули:

$$Q\ (\text{мс}) = \begin{cases} \text{якщо пріоритет} < 120, \text{тоді } (140 - \text{пріоритет}) \times 20 \\ \text{якщо пріоритет} \geq 120, \text{тоді } (140 - \text{пріоритет}) \times 5 \end{cases}.$$

Таким чином, процес із вищим статичним пріоритетом отримає довший квант ЦП. Реалізуючи цю формулу, процеси з однаковим статичним пріоритетом отримують однаковий квант ЦП, що вважається справедливим. Окрім статичного пріоритету, для кожного процеса також розраховується динамічний пріоритет. Динамічний пріоритет застосовується для динамічної зміни пріоритету кожного процеса з урахуванням середнього часу очікування процеса. Середній час очікування відстежується, оскільки планувальник намагається ідентифікувати інтерактивні процеси з припущенням, що інтерактивні процеси часто очікують різні події, наприклад, введення від користувачів. Середній час очікування оцінюється шляхом відстеження середньої кількості наносекунд, які процес витратив на сплячий режим. Починаючи з 0 мс, за кожні 100 мс збільшення середнього часу очікування значення бонусу зростає з -5 до 5. На основі цього процесу надається або штраф від -5 до -1 або бонус від 1 до 5. Таким чином, динамічний пріоритет розраховується за формулою:

$$\begin{matrix}\text{динамічний} \\ \text{пріоритет}\end{matrix} = max\left(100, \min\left(\begin{matrix}\text{статичний} \\ \text{пріоритет}\end{matrix} - \text{бонус}, 139\right)\right)$$

Для забезпечення швидкої реакції на дії користувача, планувальник віддає перевагу інтерактивним процесам по відношенню до пакетних

процесів. Планувальник намагається зберегти інтерактивні процеси в активній черзі виконання, навіть якщо їх кванти вичерпано. Для ідентифікації інтерактивних процесів визначається евристика з використанням інтерактивної дельти. Процес вважається інтерактивним, якщо:

$$\text{бонус} \geq \text{інтерактивна дельта} = \left(\frac{\text{статичний пріоритет}}{4}\right) - 28$$

Інтерактивний процес буде розміщено в «простроченому» масиві, якщо найстаріший процес з вичерпаним квантом вже чекав тривалий час або в «простроченому» масиві є процес з вищим пріоритетом. Таким чином, прибирається ситуація «голодування» неінтерактивних процесів.

Короткі висновки:

♦ переваги алгоритму – O(1) краще масштабується, ніж O(n), працює швидше, не витрачає надто багато часу на перемикання процесів, у нього додані показники інтерактивності;

♦ недоліки алгоритму – планувальник O(1) більш складний, ніж O(n). Для позначення інтерактивного процеса або процеса введення-виведення використовуються складні евристики. Обчислення евристик схильне до помилок, що час від часу змушувало процеси поводитися не відповідно до їх передбачуваного рівня інтерактивності.

3.3.20. Алгоритми RSS і RSDS

У 2004 р. Кон Колівас[78] модифікував O(1), та в результаті отримав планувальник, який назвав RSS (англ. Rotating Staircase Scheduler). Ця розробка спрямована на спрощення існуючого коду та покращення обслуговування інтерактивних процесів. Було видалено приблизно 500 рядків старого коду (в основному щодо обчислення евристики ідентифікації

[78] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/Con_Kolivas

інтерактивних процесів), та додано менше 200. Евристику замінено простою схемою на основі рангів: планувальник реалізував чергу виконання як відсортований масив процесів (для кожного ЦП).

Спочатку кожен процес розміщується в масиві за рангом його пріоритету. Потім планувальник обирає перший процес із найвищого доступного пріоритету.

В алгоритмі O(1) процеси, які використали свої кванти, переміщувалися в «прострочений» масив. RSS має певну схожість на алгоритм MLFQ. В алгоритмі RSS процес, який вичерпав свій квант, «впаде на одну сходинку пріоритету» (англ. Fall One Priority Stair Down) – повернеться в чергу, але нижчого пріоритету (на один рівень). Таким чином, процес може продовжувати працювати, але з меншим пріоритетом. Після повторного вичерпання кванта в новій черзі процес переводиться в наступну чергу і т.д. Коли процес сягає черги найнижчого пріоритету та вичерпує свій квант, він повертається в чергу на один рівень пріоритету нижче попереднього максимуму та отримує вдвічі більший квант, ніж раніше. Таким чином, процес з пріоритетом *n* «падає зі сходів» і опиняється серед процесів з пріоритетом *n-1*, але з більшим квантом. Після того, як наступного разу процес опуститься вниз, він піднімається в чергу на два рівні пріоритету нижче максимального та отримує відрізок часу втричі більший за початковий. Якщо процес довгий час був у сплячому режимі, він повертається в чергу свого максимального попереднього пріоритету. Таким чином, інтерактивні процеси зазвичай залишаються на вершині та частіше отримують процесорний час, тоді як пакетні процеси витрачають багато часу з нижчими пріоритетами, але виконуються довше.

У 2007 р. Кон Колівас удосконалив свій алгоритм і випустив новий планувальник під назвою RSDS (англ. Rotating Staircase Deadline Scheduler) або RSDL (англ. Rotating Staircase DeadLine), пізніше відомий як SD (англ. Staircase Deadline). У цьому планувальнику повернуто «прострочений» масив і концепцію «епох».

RSDL підтримує пріоритети процесів, кожному пріоритету відповідає окремий рівень. На кожному рівні є список процесів, кожному процесу дається своя індивідуальна квота часу, який йому дозволено виконуватись з

таким пріоритетом. Планувальник розподіляє процесорний час процесам з найвищим пріоритетом за допомогою типового циклічного алгоритму. Коли процес використовує свою індивідуальну квоту на заданому рівні пріоритету, він переходить до наступного рівня (з нижчим пріоритетом) та отримує нову індивідуальну квоту часу. Таким чином, цей процес може продовжувати працювати, але лише після того, як процеси з вищим пріоритетом закінчать виконуватись у своїй черзі. У міру того, як процеси просуваються вниз по «сходах», їм все більше доводиться боротися з процесами нижчого пріоритету, які чекали на нижчих рівнях. Кінцевим результатом є те, що процеси з найнижчим пріоритетом зрештою отримують процесорний час.

Особливістю цього планувальника є те, що кожен рівень пріоритету має також власну квоту. Після того, як найвищий рівень пріоритету використає свою квоту, усі процеси, що виконуються на цьому рівні, переміщуються на наступний нижчий рівень, незалежно від того, використали вони свої індивідуальні квоти процесорного часу чи ні. Таке переведення процесів на нижні рівні має назву – «мала ротація» (англ. Minor Rotation). Квота ql рівня $prio\mathrm{N}$ дорівнює сумі індивідуальних квот q всіх $proc\mathrm{MAX}$ процесів, які на цьому рівні знаходяться на початку циклу роботи алгоритму:

$$ql(prio\mathrm{N}) = \\ = q(proc1, prio\mathrm{N}) + q(proc2, prio\mathrm{N}) + \ldots + q(proc\mathrm{MAX}, prio\mathrm{N}).$$

У результаті використання механізму квот процеси, які очікують на нижчих рівнях пріоритету, потребують очікування протягом обмеженого періоду часу (який залежить від індивідуальних квот часу процесів та квот часу рівнів з вищими пріоритетами), перш ніж усі інші процеси запустяться на їхньому рівні. Таким чином, максимальна затримка для будь-якого процеса, який очікує на виконання, обмежена і може бути обчислена, що гарантує відсутність «голодування» процесів.

«Епоха» tsl процеса – відрізок часу, який складається з індивідуальних квот відрізків часу на кожному рівні пріоритету, починаючи з його найкращого рівня $prio\mathrm{N}$ (статичний пріоритет)

$$tsl(proc\mathrm{N}) =$$
$$= \mathrm{q}(proc\mathrm{N}, prio\mathrm{N}) + \mathrm{q}(proc\mathrm{N}, prio\mathrm{N}+1) + \ldots + \mathrm{q}(proc\mathrm{N}, prio\mathrm{MAX}).$$

Коли процеси витрачають свій час tsl, вони переміщуються до «простроченого» масиву, в якому їм відновлюється їх початковий максимальний пріоритет. Процеси в «простроченому» масиві не запускаються; вони очікують, доки в поточному «активному» масиві більше не залишиться процесів. У цей момент «активний» і «прострочений» масиви міняються місцями, і обробка починається спочатку (нова «епоха»). Момент перемикання масивів має назву «основна ротація» (англ. Major Rotation).

Кількість рівнів пріоритету «епохи» для процеса визначається значенням $nice$ (від 0 до 19). Таким чином, максимальна тривалість епохи черги виконання визначається кількістю запущених процесів і їх значенням $nice$.

Параметр, який впливає на тривалість роботи, є RR_INTERVAL, який дорівнює 6 мс (значення для тактової частоти процесора 1000 Гц) та більше (для більших частот). Усім процесам спочатку надається індивідуальна квота на основі RR_INTERVAL. Вона дорівнює RR_INTERVAL між значеннями $nice$ від 0 до 19 і поступово збільшується для значень $nice$ в діапазоні від -1 до -20.

Якщо загальна кількість виконуваних процесів дорівнює *nr_proc*, а кількість процесів на рівні N дорівнює *nr_Nproc*, то кожний з N процесів виконуватиметься з MAX-N рівнями пріоритету, де MAX є найвищим пріоритетом. Максимальна затримка, яку матиме процес на рівні N, буде дорівнювати:

$$\left(\mathrm{nr}_{\mathrm{Nproc}} - 1\right) \mathrm{x}\ \mathrm{SUM}\left(\mathrm{k}\ \mathrm{x}\ \mathrm{RR}_{\mathrm{INTERVAL}} \mathrm{x}\ (\mathrm{MAX} - \mathrm{N})\right) +$$
$$+ \mathrm{SUM}(\text{максимальна затримка для процесів з nice } (\mathrm{N}-1),$$
$$\text{максимальна затримка для процесів з nice } (\mathrm{N}-2), \ldots,$$
$$\text{максимальна затримка для процесів з nice } (\mathrm{MAX})).$$

де друга складова обчислюється рекурсивно, k дорівнює 1 для рівнів $nice$ від 0 до 19 і більше 1 для рівнів від -20 до -1. Така максимальна затримка буде виникати під час виконання процесів із сильним навантаженням на процесор.

Планувальник RSDS намагається знайти інтерактивні процеси, відстежуючи, як часто кожен процес перебуває в сплячому режимі – для таких процесів виконується підвищення пріоритету. Інтерактивні процеси не використовують весь свій час на вищих рівнях пріоритету. Але коли вони працюють, вони отримують перевагу над конкурентами. Якщо процес перебуває в сплячому режимі протягом основного циклу алгоритму, його індивідуальна квота повертається до значення квоти поточного рівня. Таким чином, інтерактивний процес зможе працювати з більш високим пріоритетом, навіть якщо інші високопріоритетні процеси, які виконувалися протягом цього часу, були переведені на рівні нижчих пріоритетів. Найнижчі затримки отримують процеси, які найменше навантажують процесор.

Робота Коліваса не була сприйнята належним чином у спільноті через те, що вона була надто орієнтована на настільні комп'ютери, на які в той час ОС Linux не орієнтувалася. Жоден із планувальників Коліваса не був прийнятий до основної «гілки» Linux, але вони доступні як патч для версії ядра 2.6.20 (політика планування SCHED_NORMAL (SCHED_OTHER)).

3.3.21. Алгоритм повністю справедливого планування CFS

Інго Молнар[79], автор планувальника O(1) і, на той час, головний розробник механізмів планування ядра в ОС Linux, на основі деяких ідей з алгоритму RSDS розробив алгоритм CFS (англ. Completely Fair Scheduler), який відстежує витрачений час виконання (англ. Spent Executing Time) процесів в ефективному «червоно-чорному дереві» rbtree (англ. Red-Black Tree)[80]. Компоненти, що були присутні в планувальнику O(1), такі як масив

[79] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/Ingo_Molnár
[80] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/Red–black_tree

черг виконання, ідентифікація інтерактивних процесів і концепції квантів на основі пріоритетів, було видалено для підвищення ефективності. Планувальник CFS застосовується починаючи з версії ядра Linux 2.6.23.

Взагалі, в ОС Linux використовується не один алгоритм, а кілька. Варіант обирається залежно від типу процеса, як наведено на рис. 3.24.

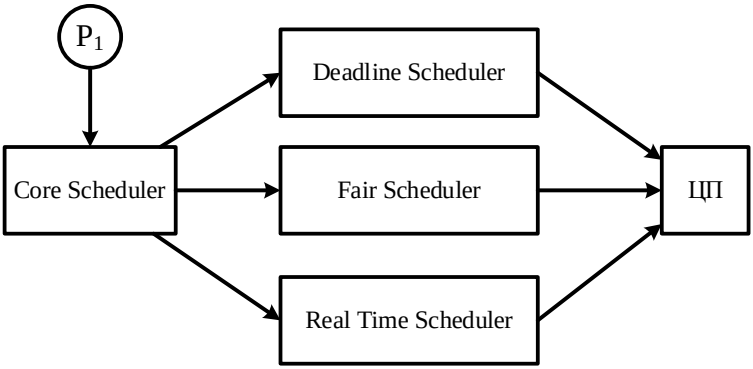


Рисунок 3.24 – Політики планування процесорного часу в ОС Linux

Для кожного процеса встановлено політику, яка визначає, як саме потрібно планувати його виконання. Всі політики реалізуються в рамках певного класу завдань. Наприклад, у класі *rt_sched_class* реалізовані 2 політики: SCHED_FIFO і SCHED_RR, а в класі *dl_sched_class* – SCHED_DEADLINE (політика SCHED_DEADLINE з'явилась у версії ядра 3.14).

Політика визначається пріоритетом *sched_priority*. CFS підтримує ті самі 140 рівнів пріоритету, що й у планувальнику O(1). Пріоритети *sched_priority* відповідають наступним класам:

- *sched_priority* = 0 – клас «Deadline» (*dl_sched_class*);
- 1 <= *sched_priority* <= 99 – клас «Real-Time» (*rt_sched_class*);
- *sched_priority* >= 100 – клас «Fair» (*fair_sched_class*).

Політики описані у файлі */include/uapi/linux/sched.h* (для версії ядра 3.7 та вище) або у файлі */include/linux/sched.h* (версії ядра до 3.7):

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL          0
#define SCHED_FIFO            1
#define SCHED_RR              2
#define SCHED_BATCH           3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE            5
#define SCHED_DEADLINE        6
```

Обробка процесів у політиках SCHED_FIFO і SCHED_RR виконується за алгоритмами FCFS та RR відповідно. Для політики SCHED_DEADLINE застосовується алгоритм EDF. За розподіл процесорного часу для процесів з політиками SCHED_NORMAL (SCHED_OTHER), SCHED_BATCH та SCHED_IDLE відповідає клас *fair_sched_class*, описаний у файлі */kernel/sched/fair.c*, та працюючий за алгоритмом CFS.

Основна ідея CFS полягає в підтримці балансу (справедливості) при розподілу процесорного часу. Якщо час для виконання процесів не збалансований (це означає, що одному чи кільком процесам не приділяється достатня кількість часу порівняно з іншими), тоді таким процесам слід дати час для виконання. Щоб визначити баланс, CFS зберігає кількість часу, наданого певному процесу, у так званому віртуальному часі виконання *vruntime* (англ. Virtual Runtime). Чим менший віртуальний час виконання процеса (чим менший час, протягом якого процес має доступ до процесора), тим вище його потреба в процесорі. CFS також реалізує концепцію справедливості для процесів у сплячому режимі.

Як було вказано вище, в алгоритмі CFS використовується відсортоване «червоно-чорне дерево» для кожного процесора. Властивості «червоно-чорного дерева»:

♦ самобалансування, що означає, що жоден шлях у дереві ніколи не буде більш ніж удвічі довшим за будь-який інший;

♦ операції над деревом відбуваються за O(log *n*) часу (де *n* — кількість вузлів у дереві).

«Червоно-чорне дерево» – це тип самозбалансованого бінарного дерева, структура даних, що використовується для реалізації асоціативних масивів. Приклад такого дерева наведено на рис. 3.25.

У «червоно-чорному дереві» створюється вузол для кожного працюючого процеса. «Червоно-чорне дерево» доволі складне, але воно дозволяє виконувати операції пошуку, додавання та видалення за однаковий час з часовою складністю алгоритму O(log n), де n – кількість елементів дерева (враховуючи особливості реалізації «червоно-чорного дерева» в Linux, часова складність операції пошуку складає лише O(1)).

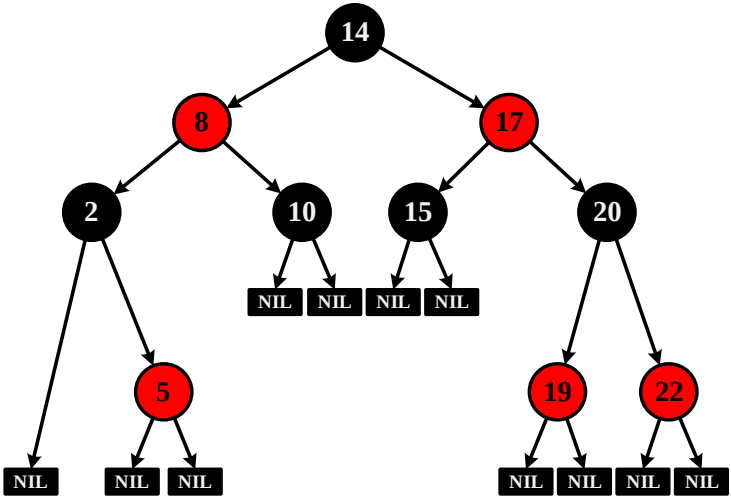


Рисунок 3.25 – «Червоно-чорне дерево» процесів в ОС Linux

Кінцеві вершини дерева не є інформативними та не містять даних. Для економії пам’яті іноді роль усіх кінцевих вершин грає один сигнальний вузол. Всі посилання із зовнішніх вузлів на кінцеві вершини вказують на цей сигнальний вузол.

Для процесів, що працюють у режимі розподілу часу, призначається вага, яка визначає частку часу ЦП, яку вони отримають. Частка $share$, виділена процесу – це відношення його ваги $se \rightarrow load.weight$ до суми ваг всіх процесів у черзі виконання $cfs\_rq \rightarrow load.weight$:

$$share = \frac{se \rightarrow load.weight}{cfs\_rq \rightarrow load.weight}.$$

Вага процеса $se \rightarrow load.weight$ визначається зі значення $nice$: кожне значення $nice$ має відповідну вагу. Рівні $nice$ зіставляються зі значеннями пріоритету за допомогою макросу NICE_TO_PRIO, а значення пріоритету зіставляються з вагами за допомогою таблиці пошуку *sched_prio_to_weight*, як наведено в табл. 3.19.

Таблиця 3.19 – Співвідношення $nice$, пріоритету та ваги

Рівень $nice$	Пріоритет	Вага $se \rightarrow load.weight$
-20 (мінімум)	100	$88761 \approx 1024 \times 1.25^{20}$
$n \in$ [-19, -1]	$120+n$	$\approx 1024 \times 1.25^{-n}$
0 (за замовчуванням)	120	1024 (NICE_0_LOAD)
$n \in$ [1, 18]	$120+n$	$\approx 1024 \times 1.25^{-n}$
19 (максимум)	139	$15 \approx 1024 \times 1.25^{-19}$

Кванти часу мають змінну довжину і визначаються динамічно. Квант $slice$, який має отримати процес за певний період часу $period$ (часовий проміжок, за який планувальник намагається виконати всі процеси), визначається як:

$$slice = \frac{se \rightarrow load.weight}{cfs\_rq \rightarrow load.weight} \times period.$$

На відміну від планувальника O(1), квант, який отримує кожний процес, не є константою і залежить від періоду $period$ (мінімальне значення якого 6 мс):

$$period = \begin{cases} 6 \text{ мс}, n \leq 8 \\ n \times sysctl\_sched\_min\_granularity \text{ мс}, n > 8 \end{cases}.$$

де n – кількість процесів; *sysctl_sched_min_granularity* – мінімальна тривалість кванта часу виконання, яка дорівнює 0.75 мс та визначена в файлі */kernel/sched_fair.c* як

```
unsigned int sysctl_sched_min_granularity = 750000ULL;
```

Якщо кількість процесів n у черзі збільшується, то період *period* також збільшується. Це робиться для запобігання надмірній кількості операцій при плануванні та різкому збільшенню перемикань контекстів процесів, коли кількість процесів значно перевищує кількість ЦП у системі.

Наприклад, є чотири процеси з однаковою вагою 1024. Розмір кванта для кожного процеса буде однаковим:

$$slice_1 = slice_2 = slice_3 = slice_4 = 0.25 \times period.$$

Якщо вага першого процеса зміниться з 1024 до 820 (зміна значення $nice$ з 0 на 1), тоді кванти будуть наступними:

$$slice_1 = \frac{820}{820+3\times1024} \times period \approx 0.2107 \times period,$$
$$slice_2 = slice_3 = slice_4 = \frac{1024}{820+3\times1024} \times period \approx 0.2631 \times period.$$

Віртуальний час виконання *vruntime* розраховується наступним чином:

$$vruntime += \frac{delta\_exec}{se\rightarrow load.weight} \times NICE\_0\_LOAD,$$

де $delta\_exec$ – кількість часу виконання процеса (дорівнює кванту, якщо не було витіснення процеса іншим процесом з вищим пріоритетом), яка розраховується у функції *update_curr* як різниця між поточним часом та часом початку його виконання в поточному кванті:

```
unsigned long delta_exec;
delta_exec = (unsigned long)(now - curr->exec_start);
```

NICE_0_LOAD – одиниця значення ваги, яка визначається в */kernel/sched/sched.h* таким чином:

```
# define SCHED_FIXEDPOINT_SHIFT      10
# define NICE_0_LOAD_SHIFT    (SCHED_FIXEDPOINT_SHIFT)
# define NICE_0_LOAD  (1L << NICE_0_LOAD_SHIFT)
```

Для 32-розрядної архітектури NICE_0_LOAD має значення 1024, і відповідає рівню $nice$ 0 (згідно табл. 3.19).

Усі запущені процеси сортуються в «червоно-чорному дереві» за значенням, яке повертає функція *entity_key* та вимірюється в наносекундах:

```
static inline s64 entity_key(struct cfs_rq *cfs_rq,
                             struct sched_entity *se)
{
        return (s64)(se->vruntime - cfs_rq->min_vruntime);
}
```

cfs->min_vruntime – мінімальне значення *vruntime* всіх процесів у системі, це значення також відповідає процесу, який виконувався мінімальний відрізок часу. Відповідно, у лівій частині дерева знаходяться процеси, які використали найменше часу, а в правій частині ті, що найбільше. Процес, що знаходиться в крайній лівій позиції «червоно-чорного дерева» (на рис. 3.25 це процес зі значенням 2), є наступним запланованим на виконання процесом. Коли процес припиняє роботу (через те, що він використав свій квант часу, або через те, що його було заблоковано чи перервано), планувальник повторно вставляє його в дерево на основі його оновленого часу виконання *se->vruntime* згідно розрахованому функцією *entity_key* значенню.

Нові процеси або процеси, що повертаються з довгого очікування, додаються в дерево зі значенням *se->vruntime = cfs->min_vruntime*, що дозволяє уникнути дефіциту уваги ЦП – такі процеси додаються в ліву частину дерева, і, відповідно, мають перевагу над іншими.

Для оптимізації обробки інтерактивних процесів, що знаходяться в сплячому режимі протягом певного періоду часу, виконується налаштовування значення *vruntime*. Коли процес виходить із режиму сну та вставляється в «червоно-чорне дерево», його значення *vruntime* оновлюється наступним чином:

$$vruntime -= \frac{\text{sysctl\_sched\_latency}}{cfs\_rq \rightarrow load.weight} \times NICE\_0\_LOAD,$$

де *sysctl_sched_latency* дорівнює періоду *period*.

Для багатопроцесорних систем SMP також було реалізовано балансувальник навантаження (англ. Load balancer). Для кожного процесора реалізовано власну чергу, час від часу ці черги можуть бути незбалансованими. Порожня черга, пов'язана з певним процесором, призводить до переведення його в режим очікування, що не дозволяє повною мірою використовувати переваги симетричних багатопроцесорних систем. Балансувальник навантаження викликається щоразу, коли системі потрібні процеси для планування. Якщо черги незбалансовані, балансувальник навантаження намагатиметься перенести неактивні процеси від найбільш завантажених процесорів до неактивного процесора.

Результати тестів показують, що CFS має перевагу в тому, що він більш справедливий у розподілі пропускної здатності процесора без істотного погіршення продуктивності інтерактивних процесів. Він також ефективніший, ніж O(1), оскільки не має складного алгоритму для визначення інтерактивних процесів.

3.3.22. Алгоритми BFS і MuQSS

У 2009 р. Кон Колівас випустив першу версію нового планувальника процесів BFS (англ. Brain F...k Scheduler). Як і раніше, його розробка орієнтована для підвищення інтерактивності та швидкості реагування для портативних пристроїв і настільних комп'ютерів з менш ніж 16 процесорами. Реалізація алгоритму BFS (і пізніше – MuQSS) також не входила до основної «гілки» Linux і розповсюджувалась як набір розширень

(патчів) – *«–ck patch set»*. Нова версія випускалась кожного разу після оновлення ядра до версії 5.14, після чого розробка була зупинена (в 2021 р.).

Метою BFS є створення планувальника процесів ЦП зі спрощеним дизайном і невеликим обсягом коду, замість складних попередніх планувальників. Під час першого випуску для ядра 2.6.31 розмір коду BFS був приблизно на 9000 рядків менше, ніж в основного планувальника Linux.

У BFS використовується єдина черга для всіх запущених процесів, незалежно від кількості доступних ЦП. Використання однієї черги в багатопроцесорній системі дозволяє уникнути необхідності дотримання узгодженості (оскільки в разі кількох черг кожна з них відповідає лише за власні процеси та підтримує справедливість лише локально). У планувальнику CFS визначення інтерактивності процеса виконується на основі аналізу тривалості часу сну/виконання. Колівас відмовився від такого визначення інтерактивності, оскільки практично неможливо виявити, чи процес перейшов у сплячий режим добровільно (наприклад, очікує на натискання клавіші), чи примусово (наприклад, очікує сигналу від іншого процеса (або нитки), очікує завершення операції введення/виведення та ін.). В алгоритмі BFS весь облік ведеться виключно на основі процесів, які використовують ЦП; час сну ніде не використовується для оцінки інтерактивності.

BFS використовує віртуальний граничний строк виконання (англ. Virtual Deadline) для досягнення справедливості. Кожен процес розміщується в черзі, впорядкованій за значеннями віртуальних граничних строків. Віртуальний граничний строк – це найдовший час, який доведеться чекати будь-яким процесам однакового пріоритету (з однаковим значенням $nice$), перш ніж їм буде надано квант часу на ЦП. Але процесам не гарантовано надання кванта до граничного строку. Після використання кванта процеси розміщується в черзі з новими граничними строками. Якщо процес призупинено до завершення кванта, він зберігає залишок часу та може його використати додатково до наступного виділеного йому кванта. Процеси з вищими пріоритетами отримують більш ранні граничні строки. Розмір кванта є регульованим цілим числом *rr_interval* (розміщено в */proc/sys/kernel/rr_interval*), значення якого встановлено в 6 мс за

замовчуванням для однопроцесорної системи. Для багатопроцесорних систем значення *rr_interval* збільшується (затримка зменшується через наявність більшої кількості ЦП), що дозволяє зменшити конкуренцію за кеш і збільшити пропускну здатність. Стандартне значення 6 мс засноване на тому факті, що людина може виявляти затримки приблизно з 7 мс.

rr_interval – максимальний час, протягом якого виконуватимуться процеси з політикою SCHED_OTHER (SCHED_NORMAL) одного рівня $nice$. Коли процес запитує процесорний час, для нього встановлюється квант, що дорівнює *rr_interval* , і віртуальний граничний строк *vdeadline* .

Віртуальний граничний строк *vdeadline* розраховується як зсув відносно поточного часу $jiffies$:

$$vdeadline = jiffies + (prio\_ratio \times rr\_interval),$$

де *prio_ratio* визначається як співвідношення на основі пріоритету процеса порівняно з базовим значенням $nice = -20$, і яке збільшується на 10 % для кожного наступного рівня $nice$; *jiffies*[81] – наносекундний лічильник таймера, який підраховує цикли процесора (вимірюється в тактах).

Віртуальний граничний строк *vdeadline* є віртуальним лише тому, що не надається гарантія, що процес дійсно буде заплановано до цього часу,

[81] $jiffies$ – глобальна змінна, яка містить кількість тактів, що відбулися з моменту завантаження системи. Під час завантаження ядро ініціалізує змінну нулем, і далі вона збільшується на одиницю під час кожного переривання таймера.

В ОС сімейства UNIX $jiffie$ – це час між двома послідовними тактами годинника, який історично дорівнює 10 мс. Однак, у Linux $jiffie$ може мати різні значення. Для отримання значення в секундах необхідно значення $jiffies$ поділити на частоту ЦП.

Колівасом у різних версіях планувальника було запропоновано використовувати такі варіанти лічильників: $gjiffies$ – лічильник для глобальної черги процесів; $miffies$ – лічильник з точністю в мікросекундах (використовується в ранніх версіях BFS); $niffies$ – лічильник з точністю в наносекундах, що містить значення також у наносекундах (використовується в алгоритмі MuQSS). Детальна інформація про лічильник $jiffies$ – https://litux.nl/mirror/kerneldevelopment/0672327201/ch10lev1sec3.html. Блог Кона Коліваса – https://ck-hack.blogspot.com/2010/10/of-jiffies-gjiffies-miffies-niffies-and.html

його значення використовується лише для вибору наступного процеса для виконання.

Існує три випадки, коли виконується перемикання контекстів процесів:

♦ якщо процес вичерпав свій квант часу, він витісняється в чергу, значення кванта заповнюється повторно, а граничний строк *vdeadline* розраховується як наведено вище;

♦ якщо процес перейшов у сплячий режим, квант та граничний строк не коригуються;

♦ якщо процес був витіснений до закінчення кванта в результаті надходження в чергу нового (або після сплячого режиму) більш пріоритетного процеса.

Для перших двох випадків наступний процес для виконання обирається з черги з мінімальним значенням граничного строку *vdeadline* (часова складність пошуку – O(n)). Додавання процеса в чергу виконується з часовою складністю O(1).

У системі з використанням BFS підтримується 103 пріоритетні черги, 100 з яких для процесів реального часу зі статичними пріоритетами, а решта 3 (у порядку від найкращого до найгіршого): SCHED_ISO (ізохронна політика), SCHED_NORMAL (описане вище планування за алгоритмом BFS) і SCHED_IDLEPRIO (політика планування процесів з пріоритетом простою). Коли процес ставиться в чергу, пошук наступного процеса для отримання часу ЦП виконується таким чином:

♦ спочатку перевіряється, які статичні пріоритетні процеси поставлено в чергу. Якщо такі є, перевіряється відповідна черга, виконується перший процес з цієї черги і пошук завершується;

♦ якщо є процеси з пріоритетом, що відповідають політиці SCHED_ISO, обирається перший процес з черги та виконується за алгоритмом RR (обслуговування таке саме, як для політики SCHED_RR);

♦ якщо пріоритет відповідає SCHED_NORMAL або SCHED_IDLEPRIO, тоді виконується пошук процеса за алгоритмом BFS.

Ізохронне планування (SCHED_ISO) призначене для забезпечення продуктивності майже в реальному часі непривілейованим користувачам без можливості безстроково отримувати ресурс ЦП, фактично, це звичайні

процеси, які обслуговуються за алгоритмом RR та мають пріоритет над іншими звичайними процесами. Проте, якщо такі процеси витратять весь час (задається в */proc/sys/kernel/iso_cpu* і є відсотком загального процесорного часу, за замовчуванням – 70 %), який їм було надано для виконання з пріоритетом політики SCHED_ISO, вони повертаються до планування з політикою SCHED_NORMAL.

Політика планування SCHED_IDLEPRIO призначається лише в тому випадку, коли ЦП неактивний. Ідея полягає в тому, щоб дозволити процесам із наднизьким пріоритетом виконуватися у фоновому режимі, що практично не впливає на інтерактивні процеси.

Основна реалізація BFS маніпулює процесами в рамках однієї глобальної черги для всіх ЦП, що дозволяє звести до мінімуму навантаження на систему від роботи самого планувальника, але призводить до проблем масштабування на багатоядерних системах. Маттіас Келер (Matthias Kohler) переробив архітектуру планувальника для забезпечення оптимальної роботи на багатоядерних системах. Суть внесених змін зводиться до додавання кількох черг виконання, кожна з яких прикріплюється до одного або кількох ядер ЦП і відповідає за планування виконання процесів тільки для цих ядер. Таким чином, вдається обійти проблеми з масштабуванням BFS на системах з великою кількістю процесорних ядер. Але виникає необхідність підтримки додаткових механізмів для балансування навантаження між чергами. Для зменшення негативного ефекту надано можливість гнучкого налаштування числа черг, що дозволяє вибрати оптимальну їх кількість для наявного числа процесорних ядер або особливостей робочого навантаження на систему.

У 2016 р. Кон Колівас представив перший публічний випуск нового планувальника MuQSS (англ. Multiple Queue Skiplist Scheduler). Як і BFS, він також націлений на підвищення інтерактивності процесів користувача.

MuQSS орієнтований на обробку процесів у кількох чергах та усуває обмеження BFS, пов’язані з масштабованістю на багатоядерних системах [22]. У MuQSS продовжено використання алгоритмів та спрощеної архітектури BFS, які перероблені для масштабування на устаткуванні з будь-яким числом процесорних ядер та систем із будь-яким числом активних процесів. Якщо в BFS використовувалася єдина черга готових до виконання

процесів, то в MuQSS застосована схема з роздільними чергами для кожного ядра ЦП, що дозволило отримати більш рівномірний розподіл навантаження по ядрах ЦП.

В алгоритмі використовується метод опитування черг, що не вимагає встановлення блокування. Також використовуються списки з пропусками (англ. Skip List)[82] замість раніше використовуваних зв'язаних списків. Для кожного ядра ЦП використовується свій 8-рівневий список з пропусками.

Список з пропусками працює як багатошаровий упорядкований пов'язаний список, де базовий (нульовий) рівень містить усі елементи, а кожен наступний рівень містить менший набір тих самих елементів. Кожного разу, коли елемент додається до рівня, він має певну ймовірність також бути доданим до наступного рівня (найчастіше 1/2). Пошук у списку з пропусками починається з самого верхнього рівня, який має найменшу кількість елементів, і переходить до нижчого рівня, якщо елемент не може бути знайдений, і так далі.

Щоб уникнути накладних витрат на створення та знищення структур списків пропуску MuQSS використовує статичні масиви з 8 рівнів замість динамічного створення та знищення структур. Приклад списку з пропусками наведено на рис. 3.26. На кожний елемент списку (*skiplist_node*) в БУП (*task_struct*) готових до виконання процесів містяться посилання (*node*). Структури, що описують список з пропусками в ОС Linux, виглядають таким чином:

```
typedef struct nodeStructure skiplist_node;
struct nodeStructure {
   int level;  /* Кількість рівнів з елементом */
   keyType key;
   valueType value;
   skiplist_node *next[8];
   skiplist_node *prev[8];
};
typedef struct listStructure {
```

[82] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/Skip_list

```
   int entries;
   int level;  /* максимальний рівень
         (на 1 більше ніж кількість рівнів) */
   skiplist_node *header; /* покажчик на початок */
} skiplist;
```

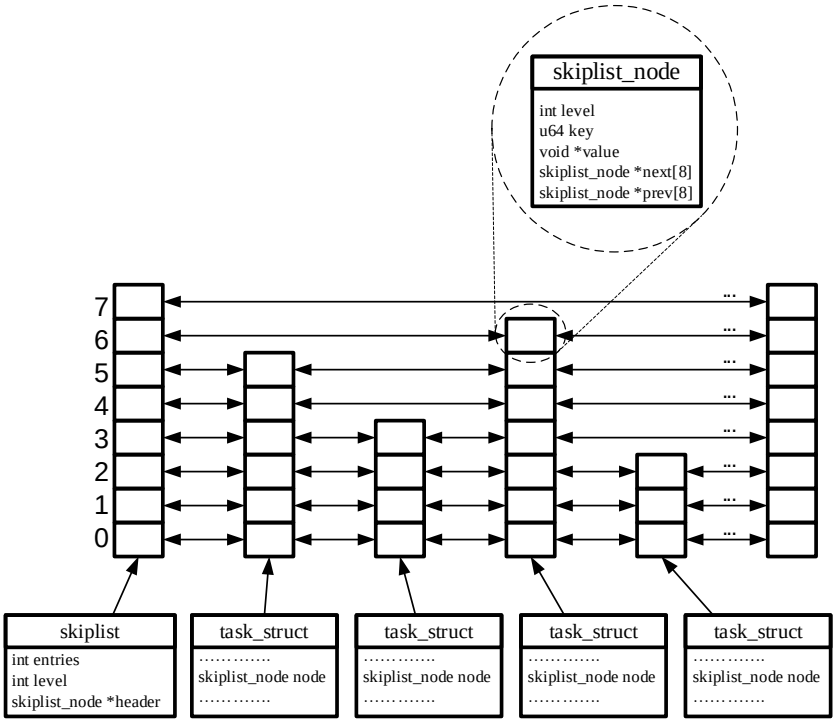


Рисунок 3.26 – Список з пропусками в алгоритмі MuQSS

Таким чином, кожна черга може мати лише 256 процесів. Однак, оскільки існує також спільна черга, то можлива кількість готових до виконання процесів збільшується в кількість разів, що дорівнює кількості ядер ЦП – це значно перевищує реалістичні обмеження в ОС щодо процесів, які можуть виконуватись на ЦП. Кожен вузол списку з пропусками є двоспрямованим, тому вставка та видалення має часову складність O(n) (вірніше O(k), де k – рівень, який приймає значення від 1 до 8). При пошуку

аналізується лише найперший запис у кожному списку з пропусками, тому складність пошуку для списку кожного ядра ЦП завжди буде O(1). Оскільки процеси в списках впорядковані, пошук наступного відповідного процеса в черзі готових процесів завжди полягає в виборі першого процеса в списку пропусків на 0-му рівні.

Вставка процесів у список з пропусками виконується впорядковано за статичним пріоритетом та віртуальним граничним строком *vdeadline*, який розраховується аналогічно як для алгоритму BFS, але з використанням *niffies*, отриманих за допомогою таймерів TSC (англ. Time Stamp Counter) з високою роздільною здатністю:

$$vdeadline = niffies + (prio\_ratio \times rr\_interval).$$

MuQSS не претендує на роль повнофункціональної заміни основного планувальника ядра Linux, основне його застосування – настільні системи з інтерактивними процесами. MuQSS не підтримує *cgroups* (властивість ядра ОС Linux, яка дозволяє об'єднати процеси в ієрархічних групах, і в цих групах відстежувати та обмежувати різні типи ресурсів), справедливий розподіл пріоритетів і точний облік граничного строку, але демонструє зниження затримок в інтерактивних застосунках, в умовах виконання в системі ресурсномістких процесів, таких як компіляція коду, обробка відео, розпакування архівів тощо.

3.3.23. Алгоритм EEVDF

Алгоритм «З найбільш раннім придатним віртуальним граничним строком – перший» EEVDF (англ. Earliest Eligible Virtual Deadline First) був вперше описаний у 1995 р. Але планувальник EEVDF [10] замінив CFS лише у версії 6.6 ядра ОС Linux (жовтень 2023 р.).

Алгоритм CFS не гарантує швидкого надання процесорного часу процесам. Як і CFS, EEVDF намагається справедливо розподілити доступний процесорний час між процесами. Якщо, наприклад, є п'ять процесів, які намагаються запуститися на одному ЦП, кожен з цих процесів має отримати 20 % доступного часу. Значення *nice* заданого процеса можна

використовувати для коригування розрахунку часу: процес із нижчим значенням *nice* (і, отже, вищим пріоритетом) має право на більшу частку процесорного часу за рахунок процесів із вищими значеннями *nice*. Відповідно, деякі процеси отримають більше часу, а деякі – менше. Для кожного процеса в EEVDF обчислюється різниця між часом, який процес мав отримати, і тим, скільки він насправді отримав; ця різниця називається відставанням або затримкою (англ. Lag). Процес із позитивним значенням затримки не отримав належної частки процесорного часу та має бути запланований раніше, ніж процес із негативним її значенням. Процес вважається придатним для планування, якщо його розрахована затримка більша або дорівнює нулю; будь-який процес із від'ємною затримкою не матиме права отримувати процесорний час. Для будь-якого процеса, який не відповідає критерію придатності, у майбутньому настане час, коли затримка стане позитивною, і він знову стане придатним; цей час називається «придатним часом» (англ. Eligible Time). Таким чином, розрахунок затримки є ключовою частиною планувальника за алгоритмом EEVDF.

Розглянемо поведінку алгоритму EEVDF на прикладі. Припустимо, що є три однотипні процеси (A, B і C), які почали працювати одночасно. На момент запуску всі вони матимуть нульову затримку (*Lag* A = *Lag* B = *Lag* C = 0 мс). Кожен з процесів має отримувати, при справедливому розподілу, третину процесорного часу.

Жоден з процесів не має від'ємної затримки, тобто всі вони придатні. Якщо планувальник першим для виконання обирає процес A з квантом 30 мс, і цей процес повністю використовує свій квант, то після звільнення процесора значення затримок *Lag* будуть наступними:

Lag A = -20 мс, *Lag* B = 10 мс, *Lag* C = 10 мс.

Протягом перших 30 мс кожний процес мав право на 10 мс процесорного часу. Насправді процес A отримав 30 мс, тому він отримав від'ємну затримку (10 мс - 30 мс) = -20 мс; інші два процеси, які взагалі не мали процесорного часу, отримали затримку 10 мс.

Процес A не відповідає критерію придатності, тому планувальник обирає процес B або C. Якщо процес B отримує квант 30 мс, після звільнення процесора затримки будуть наступними:

Lag A = -10 мс, *Lag* B = -10 мс, *Lag* C = 20 мс.

Процеси A та B використали по 30 мс з часу в 20 мс, на який вони мали право. Тепер лише процес C відповідає критерію придатності, тому планувальник наступний квант часу віддає процесу C.

У наведеному прикладі можна побачити, що сума затримок всіх процесів на кожному кроці завжди дорівнює 0.

Інший параметр, який враховується в планувальнику – віртуальний граничний строк виконання, який є часом, до якого процес повинен отримати процесорний час. Цей строк обчислюється шляхом додавання виділеного для процеса відрізку часу до придатного часу. Функція розрахунку (розташована у файлі */kernel/sched/fair.c*):

```
static void update_deadline(struct cfs_rq *cfs_rq,
                            struct sched_entity *se)
{
   if ((s64)(se->vruntime - se->deadline) < 0)
      return;
   /*
    * For EEVDF the virtual time slope is determined by w_i (iow.
    * nice) while the request time r_i is determined by
    * sysctl_sched_base_slice.
    */
   se->slice = sysctl_sched_base_slice;
   /*
    * EEVDF: vd_i = ve_i + r_i / w_i
    */
   se->deadline = se->vruntime + calc_delta_fair(se->slice, se);
   /*
    * The task has consumed its request, reschedule.
    */
   if (cfs_rq->nr_running > 1) {
      resched_curr(rq_of(cfs_rq));
```

```
        clear_buddies(cfs_rq, se);
    }
}
```

Наприклад, процес з квантом 10 мс і придатним часом 20 мс матиме віртуальний граничний строк виконання 30 мс. Сутність EEVDF полягає в тому, що для виконання обирається придатний процес з найменшим віртуальним граничним строком. Оскільки довший квант призведе до більш пізнього віртуального граничного строку, процеси з коротшими квантами (часто чутливі до затримки) зазвичай запускатимуться першими. Вибір, таким чином, обумовлюється поєднанням справедливості (значення затримки, яке використовується для обчислення прийнятного часу) та кількості часу, який наразі має кожен процес. В якості структури даних, так само як в алгоритмі CFS, використовується «червоно-чорне дерево» (для вибірки з «дерева» використовується функція *pick_eevdf*, яка реалізована у файлі */kernel/sched/fair.c*).

Якщо попередній планувальник з алгоритмом CFS використовував лише евристичні алгоритми та тонкі налаштування для визначення процесів, що потребують процесорного часу, то алгоритм EEVDF відстежує їх більш явно і не вимагає тонкого налаштування. Також він враховує процеси, які недоотримали або переотримали процесорний час у минулому циклі, що позначається в підвищенні продуктивності. Однак, розробники також акцентують увагу, що EEVDF у поодиноких випадках і при певних навантаженнях може призвести до зниження продуктивності, з часом це виправляється за допомогою патчів.

3.3.24. Альтернативні сучасні алгоритми планування процесорного часу в ОС Linux

На цей час існує певна множина неофіційних планувальників в ОС Linux. Використовуються планувальники, які є в основному похідними від алгоритмів CFS і BFS:

- MuQSS (описаний у розділі 3.3.22);

♦ PDS-MQ (англ. Priority and Deadline based Skiplist multiple queue Scheduler) – планувальник множинних черг на основі пріоритетів і граничних строків Skiplist, створений на основі планувальника BFS);

♦ BMQ (англ. BitMap Queue) – створений на основі PDS з застосуванням ідей планувальника з Zircon – ядра, на якому працює Fuchsia OS[83] від Google;

♦ Project C – модифікація BMQ на основі кодової бази Project C: злиття двох проєктів, з наступним оновленням PDS у вигляді Project C;

♦ TT (англ. Task Type) [41] – основна мета визначати типи завдань (REALTIME, INTERACTIVE, NO_TYPE, CPU_BOUND, BATCH) на основі їхньої поведінки і управляти плануванням на основі їх типів (наприклад, для типу процесів реального часу REALTIME використовується алгоритм HRRN, описаний у розділі 3.3.6);

♦ CacULE [8] – відгалуження від CFS, яке складається з набора патчів та змін. Основні його особливості: кожен процесор має власну чергу виконання; для звичайних процесів черга реалізована у вигляді списку (замість «червоно-чорного дерева»); коли процес прокидається, він витісняє поточний процес, якщо його показник інтерактивності (запозичений з планувальника ULE для ОС FreeBSD) вище;

♦ Baby Scheduler [4] від розробника CacULE– легкий та потужний для звичайного використання. Пропускна здатність у Baby Scheduler вище завдяки балансувальнику навантаження, який був спеціально для нього створений. Балансування навантаження виконується тільки на одному процесорі – CPU0, при цьому CPU0 сканує всі інші процесори і розміщує в черги до них по одному процесу в кожний такт. Балансування залежить тільки від кількості процесів. Планувальник Baby Scheduler займає 1036 рядків коду, з яких 254 – залежні функції, взяті з CFS без змін. Наразі існує три варіанти Baby Scheduler: Deadline Scheduling (dl) – основний, віртуальне

[83] Перша версія – 15.08.2016 р., остання версія F20 від 04.06.2024 р., ресурс у мережі Інтернет – https://fuchsia.dev/

планування часу виконання (vrt), Round Robin Scheduling (rr). Всі варіанти мають однаковий метод балансування навантаження процесів та відрізняються стратегією вибору процеса для виконання;

♦ BORE (англ. Burst-Oriented Response Enhancer) [6] – планувальник засновано на CacULE (та побудовано поверх CFS), основна ідея – пожертвувати деякою справедливістю заради меншої затримки планування інтерактивних процесів: на відміну від CFS коригується лише оновлення *vruntime*, тому загальні зміни досить малі. BORE працює краще, ніж CFS, з точки зору середньої затримки планування.

♦ ECHO (англ. Enhanced CPU Handling Orchestrator) [16] – політика планувальника є сумішшю алгоритмів SRTF (описано в розділі 3.3.5) і RR (описано в розділі 3.3.2), куди перенесено з алгоритму CFS обчислення віртуального часу виконання *vruntime*.

3.3.25. Алгоритм планування ULE для FreeBSD

Алгоритм ULE [33] був запропонований у 2003 р. для версії FreeBSD 5.1, але він був тимчасово заблокований і продовжував використовуватися планувальник, побудований на основі 4.3 BSD (див. розділ 3.3.14). Планувальником за замовчуванням ULE став з 2009 р. (версія FreeBSD 7.1).

Планувальник ULE був розроблений, щоб задовольнити зростаючі потреби FreeBSD на платформах SMP та SMT (англ. Simultaneous Multithreading) і при великих навантаженнях. Основні компоненти ULE: кілька черг, інтерактивний оцінювач, оцінювач використання ЦП та два алгоритми (алгоритм балансування навантаження ЦП та алгоритм розрахунку інтерактивності, оцінки використання ЦП, пріоритетів та кванта).

Повний код планувальника зберігається у вихідному файлі */sys/kern/sched_ule.c*.

В алгоритмі розрізняються інтерактивні та фонові нитки. Планувальник використовує евристику для визначення категорії нитки на основі її поведінки. Ця категорія може часто змінюватися протягом життєвого циклу нитки.

Оцінка інтерактивності нитки *Interactive_score* розраховується як відношення часу добровільного сну до часу виконання, нормалізоване в межах від 0 до 100. Ця оцінка не враховує час очікування в черзі виконання, поки нитка не отримає найвищий пріоритет у ній. Оцінка інтерактивності порівнюється з порогом інтерактивності (SCHED_INTERACT_MAX=100), який є граничною точкою для того, щоб вважати нитку інтерактивною. Поріг інтерактивності змінюється значенням *nice* процеса, що дає користувачеві певний контроль над основним механізмом зменшення затримки планування нитки. Нитка вважається інтерактивною, якщо відношення її часу добровільного сну *sleep* до часу виконання *run* нижче певного порогу. Поріг інтерактивності визначається в коді ULE і не налаштовується. Планувальник ULE використовує такі рівняння для обчислення оцінки інтерактивності нитки *Interactive_score*:

$$Scaling\_factor = \frac{Maximum\_Interactive\_Score}{2},$$

$$Interactive\_score = \begin{cases} \frac{Scaling\_factor}{sleep/run}, \text{при } sleep > run \\ \frac{Scaling\_factor}{run/sleep} + Scaling\_factor, \text{при } sleep \leq run \end{cases}$$

де *Scaling_factor* – це максимальна оцінка інтерактивності *Maximum_Interactive_Score*, зменшена в два рази.

Нитки, оцінка яких нижча за поріг інтерактивності, вважаються інтерактивними; всі інші – ні. Функція оновлення *sched_interact_update* викликається, коли нитка пробуджується (наприклад, викликом *wakeup*), щоб оновити час виконання та час сну. Значення часу сну та часу виконання можуть збільшуватися лише до певної межі. Коли сума часу роботи та часу сну перевищує ліміт SCHED_INTERACT_MAX, вони зменшуються, щоб повернути їх у діапазон від значення (4 / 5 * SCHED_INTERACT_MAX) до значення SCHED_INTERACT_MAX. Планувальник аналізує історію виконання нитки. Інтерактивна нитка, історія сну якої взагалі не запам’ятовувалась, не залишалася б інтерактивною, що призводило б до

поганої взаємодії з користувачем, а запам'ятовування довгої історії часу сну дозволило б їй отримувати невиправдано великий обсяг процесорного часу. Планувальник намагається зберігати приблизно 10 секунд історії. Обсяг історії і поріг інтерактивності – два значення, які найбільше впливають на прийняття рішень щодо визнання ниток інтерактивними.

При виділенні процесорного часу планувальник реалізує дисципліну RR, квант в якій для ниток одного пріоритету фіксований. Квант, помножений на кількість виконуваних ниток з даним пріоритетом, визначає максимальну затримку, яку матиме нитка, перш ніж зможе запуститися. Для обмеження величини затримки ULE динамічно регулює розмір квантів, які обчислюються залежно від навантаження системи. Обробник переривань викликає планувальник для оцінки розміру кванта під час кожного такту.

Планувальник ULE використовує набір із трьох масивів черг для кожного ЦП у системі. Наявність черг для кожного процесора дозволяє реалізувати закріплення процесорів у багатопроцесорній системі.

Перший масив – неактивні черги (англ. Idle Queue), у ньому зберігаються всі неактивні нитки. Масив впорядковується від найвищого до найнижчого пріоритету. Другий масив – черги реального часу. Як і неактивні черги, він впорядкований від найвищого до найнижчого пріоритету. Елементи масивів (для кожного пріоритету) організовані як подвійні зв'язані списки. Голова кожної черги зберігається в масиві. З масивом пов'язаний бітовий вектор *rq_status*, який використовується для ідентифікації непорожніх черг. Третій масив – черги розподілу часу. Замість того, щоб бути впорядкованим за пріоритетами, масив розподілу часу керується як календарна черга. Є покажчик *runq* на нитку, якій буде виділено квант часу, він посилається на якусь поточну позицію. Коли покажчик *runq* збільшується після останньої черги, він скидається та вказує на першу чергу. Жодна нитка не може бути додана до поточної позиції, це дозволяє розвантажити чергу за обмежений проміжок часу. Інший покажчик *insq* посилається на позицію для додавання нитки. Позиція для додавання в черги розподілу часу визначається відносною різницею між пріоритетом *priority* нитки та найкращим можливим пріоритетом розподілу часу *minimum_batch_priority* (числове значення 120). Коли нитка стає доступною

для виконання або поточна нитка звільняє процесор, черга *queue_index* у масиві розподілу часу обчислюється таким чином:

$$queue\_index = (insq\_index + \\ + priority - minimum\_batch\_priority) \ \% \ NQUEUE.$$

де *insq_index* – індекс поточної черги в масиві; *NQUEUE* – кількість черг в масиві.

Нитки з високим пріоритетом будуть розміщені близько до поточної позиції. Нитки з низьким пріоритетом будуть розміщені далеко від поточної позиції. Цей алгоритм гарантує, що будь-яка нитка з найнижчим пріоритетом, яка працює в режимі розподілу часу, потрапить до черги та виконається, незважаючи на те, що є нитки з вищим пріоритетом в інших чергах. Різниця в пріоритетах двох ниток визначає їх співвідношення часу виконання. Нитка з вищим пріоритетом може бути вставлена попереду нитки з нижчим пріоритетом кілька разів.

Наприклад, є дві нитки з пріоритетами 122 та 125. Поточний індекс черги, яка обробляється *insq_index* = 2. Перша нитка буде вставлена в чергу з індексом 4, друга – 7. Коли відбудеться обробка черги з індексом 4, перша нитка отримає процесорний час та для наступного періоду виконання буде розміщена в черзі з індексом 6. Коли вона отримає процесорний час в черзі з індексом 6, то після завершення кванта часу вона буде розміщена в черзі з індексом 8, і друга нитка з меншим пріоритетом (яка була розміщена в черзі з індексом 7) також зможе отримати процесорний час, після використання якого буде розміщена в черзі з індексом 12.

Загальний порядок обслуговування всіх трьох масивів черг у планувальнику ULE: нитки обираються для виконання в порядку пріоритету з черги реального часу, доки вона не стане порожньою, після чого будуть запущені нитки з поточної обраної черги розподілу часу. Нитки в неактивних чергах виконуються лише тоді, коли інші два масиви черг порожні. Нитки реального часу та нитки обробки переривань завжди вставляються в черги реального часу, щоб вони мали найменшу можливу затримку планування. Інтерактивні нитки також вставляються в чергу

реального часу, щоб інтерактивна відповідь системи була прийнятною. Неінтерактивні нитки розміщуються в чергах розподілу часу. Перемикання між чергами гарантує, що будь-яка нитка запускатиметься принаймні один раз за кожен обхід масиву черг розподілу часу незалежно від її пріоритету.

3.3.26. Планувальник Zircon Fair Scheduler

Планувальник Zircon Fair Scheduler [43] використовується в ОС Fuchsia. Ця ОС позиціонується як ОС реального часу на мікроядрі Zircon (Magenta) (похідному від мікроядра LK (Little Kernel)), вона в першу чергу призначена для вбудованих систем, але може також використовуватись у смартфонах та персональних комп'ютерах. У 2021 р. замінила для розумних колонок Google Nest Hub (попередня назва – Google Home) раніше використовувану в них Cast OS на базі Linux.

Планувальник Zircon Fair Scheduler побудований на основі алгоритму WFQ (варіація алгоритму FSS), головна ідея якого вказана в розділі 3.3.13 і є еволюцією планувальника LK. Основне застосування алгоритм WFQ знайшов для вирішення задач QoS у сучасному комутаційному обладнанні для управління чергами трафіка (наприклад, у маршрутизаторах Cisco).

Zircon Fair Scheduler приймає рішення щодо виконання ниток для кожного ЦП окремо: кожен ЦП запускає окремий екземпляр планувальника та керує власними чергами: по одній черзі для кожного рівня пріоритету в системі (32 рівні: 0 – найменший, 31 – найбільший). У кожній черзі є впорядкований список ниток, які очікують на виконання. Коли настає час для запуску нової нитки, планувальник з голови непорожньої черги з найвищим номером обирає нитку та запускає її. Якщо в чергах немає ниток для запуску – запускається нитка простою (англ. Idle Thread).

Нитка може конкурувати за процесорний час лише на одному ЦП одночасно. Цей планувальник використовує критерії впорядкування для порівняння та впорядкування ниток у черзі готових процесів.

Якщо нитка використовує весь виділений їй квант, вона буде повторно вставлена в кінець черги з відповідним пріоритетом. Якщо квант був використаний не повністю, нитка буде вставлена на початок черги, щоб

отримати процесорний час якомога швидше, але лише невикористану частку попереднього кванта.

Метою планувальника є забезпечення швидкого обслуговування інтерактивних ниток. Для визначення номеру черги використовуються ефективні пріоритети. Ефективним пріоритетом нитки є або успадкований пріоритет (якщо він є), або базовий пріоритет плюс його підвищення.

Успадкований пріоритет визначається таким чином: якщо нитка має доступ до спільного ресурсу і цим блокує іншу нитку з вищим пріоритетом, їй надається тимчасове підвищення до рівня пріоритету заблокованої нитки, щоб швидко завершити роботу з ресурсом та дозволити нитці з вищим пріоритетом відновити роботу.

Підвищення пріоритету – це значення від -MAX_PRIORITY_ADJ до MAX_PRIORITY_ADJ, яке використовується для компенсації базового пріоритету (0-31) та яке змінюється за правилами:

♦ коли нитку розблоковано після очікування на спільному ресурсі або після сплячого режиму, значення збільшується на одиницю;

♦ коли нитка добровільно звільнює процесор, значення зменшується на одиницю (але не повинно стати від'ємним);

♦ коли нитка витісняється після повного використання кванта, значення зменшується на одиницю (може стати від'ємним).

Нитки реального часу та нитки простою оброблюються трохи інакше. Нитка простою виконується, коли інші нитки не можуть бути запущені. Для кожного ЦП є нитка простою, що розташована фактично в черзі пріоритету -1. Вона використовується для відстеження часу простою та (на деяких апаратних платформах) для реалізації режиму очікування з низьким енергоспоживанням. Нитки реального часу (зі встановленим прапорцем THREAD_FLAG_REAL_TIME) працюють без витіснення за алгоритмом FCFS.

Для впорядкування ниток у черзі готових процесів використовуються дві характеристики: віртуальна часова шкала та нормалізована оцінка для кожної нитки. Віртуальна часова шкала відстежує, коли кожна нитка в черзі повністю витратить нормований квант. Нормований квант пропорційний нормованій оцінці нитки, яка, у свою чергу, обернено пропорційна вазі

нитки. Нитки впорядковуються у черзі за зростанням часу завершення на віртуальній шкалі часу.

Обернено пропорційна залежність від ваги призводить до того, що нитки з більшою вагою вставляються ближче до початку черги, ніж нитки з меншою вагою з подібним часом надходження в систему. Також, чим довше нитка чекає в черзі, тим менша ймовірність того, що нова нитка, незважаючи на велику вагу, буде вставлена перед нею. Механізм впорядкування ниток у чергах реалізовано за допомогою збалансованого бінарного дерева (пошук рішень щодо планування має часову складність O(log n)).

Для кожної нитки $P[i]$ визначаються такі характеристики:

♦ $w[i]$ – вага, дійсне число, що представляє відносну вагу нитки;

♦ $s[i]$ – час початку виконання нитки на віртуальній часовій шкалі;

♦ $f[i]$ – час завершення нитки на віртуальній часовій шкалі;

♦ $t[i]$ – розмір кванта для поточного періоду.

Для кожного ЦП $C[j]$ визначаються такі характеристики:

♦ $n[j]$ – кількість ниток, що конкурують за процесорний час на ЦП;

♦ $p[j]$ – період планування, протягом якого всі конкуруючі нитки виконуються хоча б один раз на ЦП;

♦ $W[j]$ – загальна вага: сума ваг всіх конкуруючих ниток на ЦП. Коли нитка конкурує за ЦП, її вага додається до загальної ваги ЦП. Так само, коли нитка блокується або переноситься на інший ЦП, її вага віднімається від загальної ваги ЦП.

Глобальні характеристики, які визначаються в системі:

♦ M – мінімальна гранулярність: найменший квант, виділений для будь-якої нитки;

♦ L – цільова затримка: період планування для ЦП, в який кожній нитці надається принаймні один мінімальний квант розміром M.

Період планування $p[j]$ для ЦП $C[j]$ контролює розмір квантів. Якщо конкуруючих ниток небагато, період планування за замовчуванням дорівнює цільовій затримці L. Це призводить до більших квантів і, відповідно, до меншої кількості перемикань контекстів процесів, що покращує пропускну здатність і енергоспоживання. Якщо кількість ниток

$n[j]$ велика, період планування розтягується так, що кожна нитка отримує принаймні мінімальний квант M:

$$N = floor\left(\frac{L}{M}\right),$$
$$p[j] = \begin{cases} M \times n[j], \text{при } n[j] > N \\ L, \text{при } n[j] \leq N \end{cases},$$

де N – розрахункове значення максимальної кількості конкуруючих ниток перед розтягуванням періоду (береться ціла частина результату розрахунку).

Коли нитка $P[i]$ потрапляє в чергу готових процесів (при надходженні нового процеса в систему або через завершення кванта), для неї обчислюється віртуальний час початку $s[i]$ та завершення $f[i]$. Час завершення $f[i]$ використовується для вибору позиції нитки $P[i]$ у черзі відносно інших ниток. Час початку $s[i]$ розраховується, але не використовується. Розробниками планується модифікація алгоритму Zircon Fair Scheduler, шляхом додавання ідей з алгоритму WF^{2}Q (Worst-Case Fair Weighted Fair Queuing)[84], тоді буде враховуватись як час початку, так і час закінчення.

Планувальник Zircon Fair Scheduler використовує період планування як ідеалізований уніфікований часовий проміжок для віртуальної шкали часу. Використання єдиного значення для всіх ниток дозволяє уникнути модифікації віртуальної шкали часу на користь ниток, які стартували раніше.

Якщо T – поточний системний час на ЦП $C[j]$, то, коли нитка $P[i]$ надходить у чергу готових процесів, віртуальні час початку $s[i]$ та час завершення $f[i]$ розраховуються так:

$$s[i] = T,$$

[84] J. C. R. Bennett and Hui Zhang, "WF^{2}Q: worst-case fair weighted fair queueing," Proceedings of IEEE INFOCOM '96. Conference on Computer Communications, San Francisco, CA, USA, 1996, pp. 120-128 vol.1, doi: 10.1109/INFCOM.1996.497885.

$$f[i] = s[i] + \frac{p[j]}{w[i]}.$$

Коли нитку $P[i]$ обрано для виконання, її квант $t[i]$ обчислюється на основі відносної оцінки R та періоду планування $p[j]$ на ЦП $C[j]$ (береться ціле значення, округлене в більшу сторону):

$$g = floor\left(\frac{p[j]}{M}\right),$$
$$R = \frac{w[i]}{W[j]},$$
$$t[i] = ceil(g \times R) \times M,$$

де g – кількість найменших квантів M у поточному періоді планування $p[j]$ на ЦП $C[j]$ (береться ціла частина результату розрахунку).

3.3.27. Планувальник Windows

В ОС Windows використовується пріоритетний алгоритм планування з витісненням [1, 34, 37, 42]. Запущеній нитці призначається квант часу, розмір якого може змінюватись. Розмір кванта можна налаштувати засобами ОС (рис. 3.27), обравши короткі (Programs) або довгі кванти (Background Services).

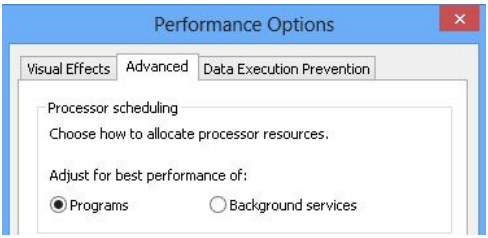


Рисунок 3.27 – Налаштування розміру кванта в ОС Windows

При плануванні не враховується, до якого процеса належить нитка. Це означає, що процес із багатьма виконуваними нитками отримуватиме

набагато більше процесорного часу, ніж процес з таким же пріоритетом, але з однією ниткою.

У версіях ОС Windows для робочих станцій нитки за замовчуванням виконуються протягом двох тактових інтервалів (короткі кванти). На серверних версіях – 12 тактових інтервалів (довгі кванти). Більший розмір кванта дозволяє мінімізувати кількість перемикань контекстів процесів. Довжина тактового інтервалу залежить від апаратної платформи. Тактовий інтервал зберігається в змінній ядра *KeMaximumIncrement* (у сотнях наносекунд) та для більшості однопроцесорних систем на базі архітектури x86 становить близько 10 мілісекунд, а для більшості багатопроцесорних систем на базі архітектур x86 і x64 – близько 15 мілісекунд. Безпосередньо сам тактовий інтервал не використовується: під час запуску системи на основі його значення виконується розрахунок кількості тактів в одному кванті (значення зберігається в змінній ядра *KiCyclesPerClockQuantum*). Величина кванта в цій змінній зберігається як число тактів таймера, помножене на 3. Відповідно, у Windows для робочих станцій нитки мають квант розміром 6 одиниць, а серверні системи мають квант розміром 36 одиниць за замовчуванням. Причина, чому квант зберігається частками в *KiCyclesPerClockQuantum*, а не цілими тактами, полягає в тому, щоб мати можливість зменшувати розмір квантів (у версіях ОС до Windows Vista). У сучасних версіях Windows ниткам процесорний час виділяється не в квантах, а в тактових циклах ЦП і, оскільки цей процес вже не залежить від інтервального таймера, такий перерахунок не потрібний.

У Windows завжди виконується хоча б одна нитка з найвищим пріоритетом. Якщо в системі багато ядер, то вони поділяються на групи по 64 ядра. Кожному процесу надається доступ до певної групи ядер, відповідно, нитки такого процеса можуть бачити лише свою групу ядер.

Для реалізації пріоритетного алгоритму планування в ОС Windows використовується 32 рівні пріоритету для ниток від 0 до 31 (31 – найвищий пріоритет):

- 16 - 31 – рівні пріоритетів реального часу;
- 1 - 15 – звичайні динамічні пріоритети;
- 0 – зарезервований для нитки обнуління сторінок.

Рівні пріоритетів призначаються на основі базових пріоритетів, які успадковуються від пріоритетів процесів, що задаються під час їх створення механізмами Windows API (для зміни класу пріоритету використовується функція *SetPriorityClass*). Існують такі класи пріоритетів:

♦ реального часу (Real-time, значення внутрішнього індексу PROCESS_PRIORITY_CLASS – 4, базовий пріоритет – 24);

♦ високий (High, значення PROCESS_PRIORITY_CLASS – 3, базовий пріоритет – 13);

♦ вище середнього (Above Normal, значення PROCESS_PRIORITY_CLASS – 6, базовий пріоритет – 10);

♦ звичайний (Normal, значення PROCESS_PRIORITY_CLASS – 2, базовий пріоритет – 8);

♦ нижче середнього (Below Normal, значення PROCESS_PRIORITY_CLASS – 5, базовий пріоритет – 6);

♦ низький (Idle, значення PROCESS_PRIORITY_CLASS – 1, базовий пріоритет – 4).

Далі призначається відносний пріоритет, який збільшує або зменшує пріоритет нитки (у дужках вказано максимальне відхилення від базового пріоритету):

♦ Time-Critical – критичний за часом (+15);

♦ Highest – найвищий (+2);

♦ Above-Normal – вище середнього (+1);

♦ Normal – звичайний (0);

♦ Below-Normal – нижче середнього (-1);

♦ Lowest – найнижчий (-2);

♦ Idle – рівень простою (-15).

Після отримання базового пріоритету та його коригування відносним пріоритетом розраховується динамічний пріоритет нитки. Відносні пріоритети ниток «*Time-Critical*» і «*Idle*» зберігають відповідні значення незалежно від класу пріоритету процеса (якщо це не Real-Time):

Якщо «Time-Critical»: ((HIGH_PRIORITY+1) / 2

Якщо «Idle»: -((HIGH_PRIORITY+1) / 2

де HIGH_PRIORITY дорівнює 31.

ЦП завжди обробляє якусь нитку. Коли немає готових до виконання ниток запускається спеціальна нитка простою «*Idle*». На кожне ядро процесора існує своя власна нитка простою (всі нитки простою належать одному процесу простою «*System Idle Process*» – псевдопроцес, який використовується для обліку простою процесорного часу, ідентифікатор процеса *pid*=0). Процес «*System Idle Process*» виконується майже 100 % часу в режимі ядра. Вихідні структури нитки та процеса простою розміщуються в пам'яті статично при початковому завантаження ОС перед ініціалізацією диспетчера процесів та диспетчера об'єктів. Пріоритети ниток простою не мають значення, тому що такі нитки обираються для диспетчеризації, тільки якщо немає ніяких інших ниток, готових до виконання. Їх пріоритет ніколи не порівнюється з пріоритетами інших ниток і не використовується для розміщення в чергах готових до виконання ниток – нитки простою ніколи не стоять у жодних чергах готових ниток.

Самий нижчий пріоритет у системі (0) має нитка «*Zero Page Thread*» в процесі *System*. Це – спеціальна нитка ОС, яка обнуляє дані на звільнених сторінках оперативної пам'яті. Робиться це для того, щоб після звільнення пам'яті, дані, які в ній були записані (наприклад, особисті дані, паролі тощо), ненароком не отримав інший процес. Періодично (коли звільняються 8 та більше сторінок в пам'яті) ОС планує на запуск «*Zero Page Thread*» з пріоритетом 0 (даний пріоритет не доступний ніяким іншим ниткам), тоді і тільки тоді, коли на процесорі виконується нитка простою «*Idle*» – вона витісняється і починає виконуватись «*Zero Page Thread*».

На рис. 3.28 наведено можливі пріоритети для ниток в системі.

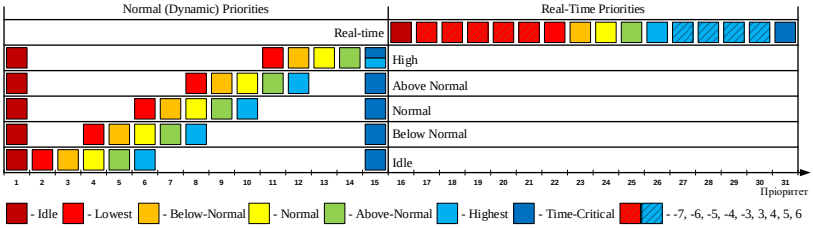


Рисунок 3.28 – Доступні пріоритети для ниток в ОС Windows

Нитки реального часу можуть мати будь-який пріоритет у діапазоні 16-31, на відміну від інших типів ниток. За замовчуванням при створенні процеса, всім його ниткам призначається пріоритет з класу Normal, з відносним пріоритетом Normal (значення – 8). Саме динамічний пріоритет враховується при плануванні процесорного часу. Тому, наприклад, нитка з базовим пріоритетом Normal та відносним пріоритетом Highest буде при плануванні ідентичною нитці з базовим пріоритетом Above Normal та відносним пріоритетом Normal. В обох випадках динамічний пріоритет буде дорівнювати 10. Нитки з однаковим динамічним пріоритетом плануються за алгоритмом RR.

Пріоритети ниток під час планування можуть періодично змінюватись:

♦ якщо нитка занадто довго чекає виконання, її пріоритет підвищується на 1 пункт;

♦ якщо відбувається введення з інтерфейсу користувача, пріоритет нитки підвищується на 1 пункт;

♦ після завершення операції введення/виведення пріоритет нитки підвищується. При очікуванні:

• диска, cd-rom, паралельного порту, відео – підвищення на 1 пункт;

• мережі, поштової скриньки, іменованого каналу, послідовного порту – підвищення на 2 пункти;

• клавіатури або миші – підвищення на 6 пунктів;

• звукової карти– підвищення на 8 пунктів;

♦ коли нитка очікує ресурс, який зайнятий іншою ниткою, система для нитки, яка зайняла потрібний ресурс, підвищує пріоритет до 15;

♦ для ниток - власників вікон (віконних функцій) після пробудження пріоритет підвищується на 2 пункти.

Для роботи мультимедійних процесів надається за замовчуванням 80 % процесорного часу та фіксація категорії планування (англ. Scheduling Category). Так, для ниток застосунку професійної обробки аудіо Pro Audio фіксується категорія High з пріоритетами 23-26, для ниток мультимедійного застосунку Windows Media Player – категорія Medium з пріоритетами 16-22. Для процесів реального часу пріоритет ніколи не підвищується.

3.4. Особливості планування в багатопроцесорних системах

У багатопроцесорному плануванні процесам доступні кілька ЦП, тому стає можливим розподіл навантаження між ними, що може значно підвищити загальну продуктивність і ефективність системи [13, 34, 35]. Багатопроцесорне планування працює шляхом розподілу процесів між кількома процесорами в комп'ютерній системі, що дозволяє обробляти процеси одночасно та зменшує загальний час, необхідний для їх виконання. Однак, багатопроцесорне планування є більш складним порівняно з однопроцесорним.

Один із підходів багатопроцесорного планування полягає в тому, що всі рішення щодо планування та обробки введення/виведення обробляються одним процесором, який називається головним процесором (англ. Master Processor), а інші процесори (англ. Slave Processors) виконують лише код процесів. Такий підхід називається асиметричною багатопроцесорністю AMP або ASMP (англ. Asymmetric Multiprocessing). Другий підхід використовує симетричну багатопроцесорну (SMP) обробку, коли кожен процесор самостійно планує свою роботу. Всі процеси можуть бути в загальній черзі готових до виконання процесів або кожен процесор може мати власну приватну чергу.

Використання спільної черги процесів або багатопроцесорне планування з однією чергою SQMS (англ. Single Queue Multiprocessor Scheduling) доволі нескладно реалізувати. Однак, SQMS має очевидні недоліки. Перша проблема – відсутність масштабованості. Щоб забезпечити коректну роботу планувальника на кількох процесорах необхідно забезпечити монопольний доступ до черги під час операцій з нею різними ЦП. Блокування, які використовуються для організації монопольного доступу, можуть значно знизити продуктивність, особливо зі збільшенням кількості процесорів у системах: система витрачає все більше часу на накладні витрати на блокування та менше часу на виконання корисної роботи.

Другою основною проблемою SQMS є проблема використання кеш-пам'яті. Коли процес виконується на процесорі, відбувається певний вплив

на кеш-пам'ять. Дані, до яких процес отримав останній доступ, заповнюють кеш для процесора, і в результаті, послідовний доступ процеса до пам'яті часто виконується в кеш-пам'яті. Це дозволяє процесу виконуватися швидше, оскільки він робитиме менше посилань на основну пам'ять. Якщо процес мігрує на інший процесор, вміст кеш-пам'яті має бути визнано недійсним для попереднього процесора, а кеш для іншого процесора має бути повторно заповнено – жодна частина процеса не буде присутня в кеші цього процесора, і процес запускатиметься повільно, поки буде заповнюватись кеш.

Через високу вартість операцій анулювання та заповнення кеш-пам'яті більшість систем SMP намагаються уникнути міграції процесів з одного процесора на інший і намагаються підтримувати роботу процеса на одному процесорі. Спорідненість процесора (англ. Processor Affinity) – це аспект планування в багатопроцесорній системі, де планувальник відстежує, на якому процесорі процес виконувався раніше, і намагається перепланувати цей процес у майбутньому на той самий процесор. Але реалізація такої схеми може бути складною.

Існує два типи спорідненості процесора:

♦ жорстка спорідненість (англ. Hard Affinity) гарантує, що процес завжди планується на той самий процесор або підмножину процесорів, на яких він може працювати;

♦ м'яка спорідненість (англ. Soft Affinity) – планувальник спробує запланувати процес на той самий процесор, але в деяких випадках може перемістити процес на інший процесор. Причина переміщення полягає в тому, що, навіть незважаючи на початкове зниження продуктивності для запуску процеса на іншому ЦП, це, ймовірно, краще, ніж простоювання ЦП без жодного процеса.

Деякі системи, такі як ОС Linux, реалізують м'яку спорідненість, але також надають деякі системні виклики, які підтримують жорстку спорідненість (наприклад, системний виклик *sys_sched_setaffinity* прив'язує процес до тих процесорів, котрим у бітовій масці виставлено відповідний біт: молодший біт відповідає першому логічному номеру процесора в

системі, а найстарший біт відповідає останньому логічному номеру процесора в системі).

Через проблеми, спричинені планувальниками з однією чергою, у деяких системах обирають кілька черг, наприклад, по одній на кожний ЦП – багаточергове багатопроцесорне планування MQMS (англ. Multi-Queue Multiprocessor Scheduling). У MQMS кожна черга дотримується певної дисципліни планування, наприклад RR, хоча, звичайно, можна використовувати будь-який інший алгоритм. Коли процес надходить у систему, він розміщується в певній черзі планування. Тоді він планується по суті незалежно від інших черг, уникаючи проблем обміну інформацією та синхронізації, які зустрічаються в підході з єдиною чергою.

MQMS має явну перевагу над SQMS у тому, що він має бути більш масштабованим. Зі збільшенням кількості ЦП зростає і кількість черг, і, отже, немає необхідності реалізовувати монопольний доступ до них за допомогою блокування. Крім того, MQMS забезпечує спорідненість процесора. Але в плануванні MQMS виникає проблема, яка є фундаментальною в підході на основі кількох черг: дисбаланс навантаження на процесори.

Для уникнення дисбалансу планувальник намагається збалансувати навантаження на процесорах, щоб переконатися, що вони мають достатньо процесів у черзі виконання. Балансування навантаження (англ. Load Balancing) – це механізм, завдяки якому робоче навантаження рівномірно розподіляється між усіма процесорами в системі SMP. Слід зауважити, що балансування навантаження необхідне лише в таких системах, де кожен процесор має власну приватну чергу готових до виконання процесів. У SMP важливо підтримувати збалансоване робоче навантаження між усіма процесорами, щоб повністю використовувати переваги наявності кількох процесорів, інакше один чи більше процесорів простоюватимуть, а інші процесори матимуть високе робоче навантаження з великими чергами готових до виконання процесів. Існує два загальні підходи до балансування навантаження:

♦ примусова міграція (англ. Push Migration) – ОС періодично перевіряє навантаження (кількість елементів у черзі готових до виконання процесів)

на кожному процесорі. Якщо є дисбаланс, деякі процеси будуть перенесені з перевантажених процесорів на неактивні або менш зайняті для рівномірного розподілу навантаження;

♦ міграція за запитом (англ. Pull Migration) – коли планувальник виявляє, що в черзі готових процесів для процесора більше немає процесів, то він переміщає процес (або кілька процесів) з черги готових процесів іншого процесора у свою чергу.

Примусова міграція і міграція за запитом не обов'язково є взаємовиключними та насправді часто впроваджуються паралельно в системах балансування навантаження. Наприклад, планувальник ОС Linux CFS (описаний у розділі 3.3.21) і планувальник ULE (описаний у розділі 3.3.25) для систем FreeBSD, реалізують обидві методики.

У багатоядерних процесорах кілька процесорних ядер розташовані на одному фізичному чіпі. Кожне ядро має набір регістрів для підтримки свого архітектурного стану і для ОС виглядає як окремий фізичний процесор.

Системи SMP, які використовують багатоядерні процесори, є швидшими та споживають менше енергії, ніж системи, у яких кожен процесор має власний фізичний чіп. Однак, багатоядерні процесори можуть ускладнити планування. Коли процесор звертається до пам'яті, він витрачає значний час на очікування, поки дані стануть доступними – виникає затримка пам'яті (англ. Memory Stall). Затримка може виникати з різних причин, одна з яких – промах кешу (англ. Cache Miss), тобто доступ до даних, яких немає в кеш-пам'яті. У таких випадках процесор може витрачати до 50 % свого часу на очікування, поки дані стануть доступними. Для вирішення цієї проблеми в останніх розробках апаратного забезпечення реалізовані багатониткові процесорні ядра, в яких дві або більше апаратних нитки (англ. Hardware Thread) призначено кожному ядру. Тому, якщо одна нитка зупиняється під час затримки пам'яті, ядро може переключитися на іншу нитку. З точки зору ОС кожна апаратна нитка зберігає свій архітектурний стан, такий як покажчик на поточну команду і набір регістрів. Таким чином, вона виглядає як логічний ЦП, який доступний для виконання програмної нитки. Ця техніка відома як багатонитковість мікросхем CMT (англ. Chip Multithreading), яка дозволяє одночасно виконувати декілька

незалежних ниток в одному процесорному ядрі. Приклади використання технології CMT:

♦ процесор UltraSPARC T1[85] компанії Sun Microsystems: у 2005 р. компанія Sun Microsystems (зараз – частина корпорації Oracle) представила процесор UltraSPARC T1 із технологією Chip Multithreading. Цей процесор може обробляти до 32 ниток одночасно на восьми процесорних ядрах. Останній процесор Oracle Sparc M8[86] (2017 р.) підтримує вісім апаратних ниток на ядро, по 32 ядра на процесор, таким чином забезпечуючи операційну систему 256 логічними процесорами;

♦ технологія Hyper-Threading[87] від Intel (HTT): Intel представила технологію Hyper-Threading у 2002 р. Ця технологія спочатку використовувалася в процесорах Intel Xeon і Pentium 4 високого класу. Сьогодні HTT реалізована у більшості процесорів Core i-серії, вона дозволяє кожному ядру виконувати дві нитки одночасно;

♦ процесор POWER5[88] від IBM – представлений у 2004 р., одночасна багатонитковість SMT в архітектурі POWER відома під назвою «віртуальна багатонитковість» (англ. Virtual Multithreading). Завдяки двом ниткам на кожне ядро процесор POWER5 дозволив ефективніше використовувати ресурси, гарантуючи, що процесор може продовжувати працювати над одним процесом, чекаючи на інший. Чип останнього процесора POWER10[89] (2021 р.) складається з 16 ядер (але використовується з них лише 15), кожне ядро може одночасно обробляти до восьми ниток завдяки підтримці SMT (виглядає як 120 логічних процесорів для ОС), а модуль з двома мікросхемами забезпечує 240 апаратних ниток на сокет.

Існує два способи забезпечення багатонитковості процесора:

---

[85] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/UltraSPARC_T1

[86] Ресурс у мережі Інтернет – https://www.oracle.com/us/products/servers-storage/sparc-m8-processor-ds-3864282.pdf

[87] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/Hyper-threading

[88] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/POWER5

[89] Ресурси у мережі Інтернет – https://en.wikipedia.org/wiki/Power10, https://www.redbooks.ibm.com/redpieces/pdfs/redp5675.pdf

♦ груба багатонитковість (англ. Coarse-Grained Multithreading) – нитка виконується на процесорі до виникнення події тривалої затримки (наприклад зупинка пам'яті), після чого процесор повинен переключитися на іншу нитку. Вартість перемикання між нитками суттєва, оскільки конвеєр команд повинен бути завершений, перш ніж інша нитка зможе почати виконання на ядрі процесора. Як тільки нова нитка починає виконуватися, вона починає заповнювати конвеєр своїми командами;

♦ точна багатонитковість (англ. Fine-Grained Multithreading) – перемикання між нитками на межі циклу команд. Архітектурний дизайн таких систем містить логіку для перемикання ниток, і, як наслідок, вартість перемикання між нитками невелика.

Важливо відзначити, що ресурси фізичного ядра (такі як кеш-пам'ять і конвеєри команд) спільні для його апаратних ниток, тому процесорне ядро може виконувати лише одну апаратну нитку за раз. Отже, багатонитковий багатоядерний процесор фактично потребує двох різних рівнів планування, як зображено на рис. 3.29.

На першому рівні рішення щодо планування приймає ОС – вона обирає, яку програмну нитку запускати на кожному логічному ЦП (апаратній нитці). На цьому рівні планування ОС може реалізовувати будь-який алгоритм планування, описаний раніше.

На другому рівні кожне ядро вирішує, яка апаратна нитка виконуватиметься. Існує кілька стратегій, за якими приймається рішення. Перший варіант – простий циклічний алгоритм для планування апаратних ниток (застосовано в процесорі UltraSPARC T3[90]). Другий варіант використовується в двоядерному процесорі з двома апаратно керованими нитками на кожне ядро Intel Itanium[91]. Кожній апаратній нитці призначається значення динамічної негайності в діапазоні від 0 до 7, де 0 означає найнижчу негайність, а 7 – найвищу. Процесор Itanium визначає п'ять різних подій, які

[90] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/SPARC_T3
[91] Ресурс у мережі Інтернет – https://en.wikipedia.org/wiki/Itanium

можуть викликати перемикання ниток. Коли відбувається одна з цих подій, логіка перемикання ниток порівнює негайність двох ниток і обирає нитку з найвищим значенням негайності для виконання на ядрі процесора.

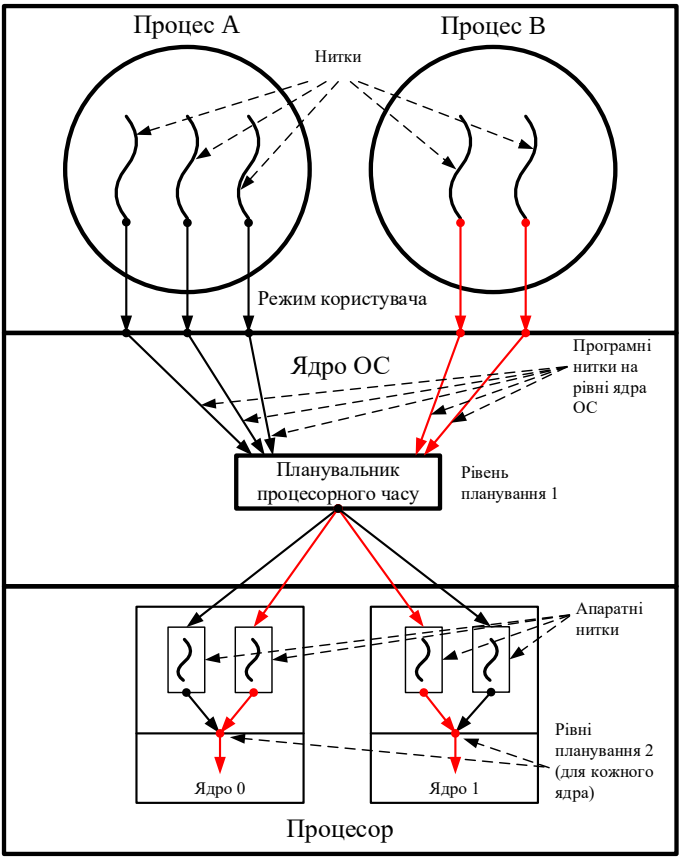


Рисунок 3.29 – Дворівнева архітектура планування процесорного часу

Планувальник ОС (перший рівень) може мати інформацію про спільне використання ресурсів процесора, тоді він може приймати більш ефективні рішення щодо планування. Припустимо, що ЦП має два процесорних ядра і кожне ядро має дві апаратних нитки (як зображено на рис. 3.29). Якщо в ОС

виконується дві програмні нитки ядра з одного процеса, вони можуть працювати як на одному ядрі, так і на окремих ядрах. Якщо заплановано, що обидві такі програмні нитки працюватимуть на одному ядрі, їм доведеться спільно використовувати процесорні ресурси, і тому, ймовірно, вони працюватимуть повільніше, ніж якби вони були заплановані на окремих ядрах. Якщо ОС має інформацію про рівень спільного використання ресурсів процесора, вона може планувати такі програмні нитки для апаратних ниток (логічних процесорів), розташованих на різних ядрах (на рис. 3.29 позначено лініями червоного кольору) і забезпечити реальну паралельність їх виконання.

Контрольні запитання

1. Чим відрізняються процеси, що обмежені можливостями процесора, та процеси, що обмежені можливостями пристроїв введення-виведення?
2. Які основні дії виконують планувальник та диспетчер процесорного часу?
3. Назвіть основні події в системі, коли активується планувальник процесорного часу, диспетчер процесорного часу.
4. У чому полягає основна відмінність алгоритмів планування без витіснення та з витісненням?
5. Назвіть базові критерії оцінки алгоритмів планування процесорного часу.
6. Назвіть вимоги до алгоритмів планування процесорного часу в пакетних системах, інтерактивних системах та системах реального часу.
7. Які переваги алгоритму FCFS над алгоритмом RR? RR над FCFS?
8. Які характеристики покращуються, а які погіршуються при переході з алгоритму SPN до алгоритму PSPN?
9. Які з базових алгоритмів планування процесорного часу можна віднести до пріоритетних? Чому?
10. Чи можна вважати алгоритм FCFS пріоритетним? Що може виступати в ролі пріоритету в цьому алгоритмі?

11. Назвіть базові алгоритми, які використовують більше однієї черги готових процесів?
12. В яких реалізаціях алгоритмів для сучасних ОС використовується базовий алгоритм MLFQ?
13. На що впливає розмір кванта в алгоритмах планування процесорного часу?
14. Які основні відмінності алгоритмів гарантованого та справедливого планування?
15. Назвіть алгоритми планування процесорного часу, які використовувались в ОС Linux, залежно від версій ядра ОС.
16. Які алгоритми планування процесорного часу використовуються в ОС реального часу?
17. Що мається на увазі під терміном «завдання» в ОС реального часу?
18. Чим відрізняються алгоритми Background Server, Deferrable Server та Sporadic Server?
19. Порівняйте між собою алгоритми RMS та EDF.
20. Що таке «буксування» процесів?
21. Які основні відмінності першого алгоритму планування в ОС Linux і алгоритму традиційного планування в ОС UNIX?
22. Чим відрізняється алгоритм CFS від алгоритму FSS?
23. Що було змінено при переході від алгоритму O(n) до алгоритму O(1) в ОС Linux?
24. Що спільного в усіх алгоритмах планування процесорного часу від Кона Коліваса?
25. Які моделі структур даних для черг готових процесів використовуються в алгоритмах планування процесорного часу?
26. Що таке $nice$, та в яких алгоритмах використовується?
27. В яких алгоритмах планування процесорного часу відбувається визначення інтерактивних процесів?
28. Для чого відбувається балансування навантаження в алгоритмах планування процесорного часу?
29. Чим відрізняється апаратна нитка від програмної нитки?
30. Назвіть ОС, в яких відбувається планування ниток, а не процесів.

СПИСОК ЛІТЕРАТУРИ

1. Allievi A., Russinovich M. E., Ionescu A., Solomon D. A. Windows Internals, 7th Edition. Part 2 (Developer Reference). Microsoft Press, 2021. – 912 p.

2. Anderson T., Dahlin M. Operating Systems Principles & Practice. Volume I : Kernels and Processes. Second Edition. Recursive Books, 2015. – 252 p.

3. Arpaci-Dusseau R. H., Arpaci-Dusseau A. C. Operating Systems : Three Easy Pieces. Arpaci-Dusseau Books, 2023. – 921 p.

4. Baby-CPU-Scheduler. URL : https://github.com/hamadmarri/Baby-CPU-Scheduler (дата звернення 23.08.2024).

5. Billimoria K. N. Linux Kernel Programming : A comprehensive and practical guide to kernel internals, writing modules, and kernel synchronization. Packt Publishing, 2024. – 826 p.

6. BORE (Burst-Oriented Response Enhancer) CPU Scheduler. URL : https://github.com/firelzrd/bore-scheduler (дата звернення 23.08.2024).

7. Bovet D. P., Cesati M. Understanding the Linux Kernel, Third Edition. O'Reilly Media, 2005. –944 p.

8. CacULE CPU scheduler. URL : https://github.com/hamadmarri/cacule-cpu-scheduler (дата звернення 23.08.2024).

9. Chakraborty P. Operating Systems. Evolutionary Concepts and Modern Design Principles. CRC Press, 2024. – 640 p.

10. Corbet J. An EEVDF CPU scheduler for Linux. URL : https://lwn.net/Articles/925371/ (дата звернення 23.08.2024).

11. Cottet F., Delacroix J., Mammeri Z. Scheduling in real-time systems. John Wiley & Sons Ltd, 2002. – 284 p.

12. Darnell L. S. Create Your Own Operating System. 1st Edition. CreateSpace Independent Publishing Platform, 2016. – 149 p.

13. Deitel H. M., Deitel P. J., Choffnes D. R. Operating Systems. Third Edition. Pearson Education, 2004. – 1209 p.

14. Dijkstra E. W. The structure of the 'THE'-multiprogramming system. Center for American History, University of Texas at Austin. 1965. URL :

http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD196.PDF (дата звернення 23.08.2024).

15. Doeppner T. W. Operating Systems In Depth: Design and Programming 1st Edition. Wiley, 2010. – 464 p.
16. ECHO CPU Scheduler. URL : https://github.com/hamadmarri/ECHO-CPU-Scheduler (дата звернення 23.08.2024).
17. Fox R. Linux with Operating System Concepts. CRC Press, 2015. – 680 p.
18. Garg R., Verma G. Operating Systems: An Introduction. Mercury Learning & Information, 2017. – 290 p.
19. Gregg B. Systems Performance: Enterprise and the Cloud, Second Edition. Pearson, 2020. – 928 p.
20. Haldar S., Subramanian D. K. Fairness in processor scheduling in time sharing systems. *ACM SIGOPS Operating Systems Review*, Volume 25, Issue 1, 1991. pp. 4–18. DOI : https://doi.org/10.1145/122140.122141
21. Holcombe J. Survey of Operating Systems, Seventh Edition. McGraw Hill, 2023. – 480 p.
22. Hussein N. The MuQSS CPU scheduler. URL : https://lwn.net/Articles/720227/ (дата звернення 23.08.2024).
23. Kernel Architecture Overview. URL : https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html (дата звернення 23.08.2024).
24. Liu Y., Yue Y., Guo L. UNIX Operating System. The Development Tutorial via UNIX Kernel Services. Springer, 2011. – 388 p.
25. Love R. Linux kernel development. 3rd ed. Addison-Wesley Professional, 2010. – 440 p.
26. McDougall R., Mauro J. Solaris™ Internals: Solaris 10 and OpenSolaris Kernel Architecture [2nd ed]. New-York : Prentice Hall, 2006. – 1020 p.
27. McHoes A., Flynn I. Understanding Operating Systems. 8th edition. Boston : Cengage Learning, 2018. – 591 p.

28. McKusick M. K., Neville-Neil G. V., Watson R. N. M. Design and Implementation of the FreeBSD Operating System, The, 2nd Edition. Addison-Wesley Professional, 2015. – 927 p.

29. Messier R. Operating System Forensics. 1st Edition. Syngress, 2015. – 386 p.

30. MIT Exokernel Operating System. URL : https://pdos.csail.mit.edu/archive/exo/ (дата звернення 01.10.2024).

31. Naghibzadeh M. Operating System Concepts and Techniques. iUniverse. Inc, 2005. – 296 p.

32. Padallan J. O. Introductory Guide to Operating Systems. Arcler Press, 2023. – 280 p.

33. Roberson J. ULE: a modern scheduler for FreeBSD. URL : https://www.researchgate.net/publication/41035925_ULE_a_modern_scheduler_for_FreeBSD (дата звернення 01.10.2024).

34. Silberschatz A., Galvin P. B., Gagne G. Operating system concepts, 10th edition. Wiley, 2018. – 1277 p.

35. Stallings W. Operating Systems: Internals and Design Principles. Ninth edition. Harlow: Pearson Education Limited, 2018. – 800 p.

36. Stuart B. Principles of Operating Systems: Design and Applications (Advanced Topics). 1st Edition. Cengage Learning, 2008. – 608 p.

37. Tanenbaum A. S., Bos H. Modern Operating Systems. 5th Edition, Global Edition. Pearson Education, 2023. – 1185 p.

38. Tanenbaum A. S., Woodhull A. S. Operating Systems: Design and Implementation. 3rd ed. Pearson Education, Inc., 2006. – 1071 p.

39. The Art of Linux Kernel Design: Illustrating the Operating System Design Principle and Implementation. 1st Edition / Y. Lixiang [et al.]. Auerbach Publications, 2017. – 534 p.

40. The KeyKOS nanokernel architecture. USENIX Workshop on Micro-Kernels and Other Kernel Architectures. A. C. Bomberger [et al.]. *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, WA, USA, 27-28 April 1992s. USENIX Association, 1992. pp. 95-112. URL : https://www.researchgate.net/publication/221397234_The_KeyKOS_nanokernel_architecture (дата звернення 01.10.2024).

41. TT CPU Scheduler. URL : https://github.com/hamadmarri/TT-CPU-Scheduler (дата звернення 23.08.2024).

42. Yosifovich P., Ionescu A., Russinovich M. E., Solomon D. A. Windows Internals, 7th Edition. Part 1. System architecture, processes, threads, memory management, and more. Microsoft Press, 2017. – 800 p.

43. Zircon Fair Scheduler. URL: https://fuchsia.dev/fuchsia-src/concepts/kernel/fair_scheduler (дата звернення 01.10.2024).

44. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. Черкаси : ЧНУ імені Богдана Хмельницького, 2018. – 524 с.

45. Зайцев В. Г., Дробязко І. П. Операційні системи: навчальний посібник для студентів спеціальності 123 «Комп'ютерна інженерія». Київ : КПІ ім. Ігоря Сікорського, 2019. – 240 с.

46. Панченко В. І., Коломійцев О. В., Межерицький С. Г. Системне програмне забезпечення. Програмування системних механізмів ОС: навчальний посібник для студентів спеціальності 123 «Комп'ютерна інженерія» денної та заочної форм навчання. Харків : НТУ «ХПІ», 2024. – 327 с.

Навчальне видання

ПАНЧЕНКО Володимир Іванович
ГЕЙКО Геннадій Вікторович
ГЛАВЧЕВ Максим Ігорович
СКОРОДЄЛОВ Володимир Васильович

ОПЕРАЦІЙНІ СИСТЕМИ. УПРАВЛІННЯ ПРОЦЕСАМИ

Навчальний посібник
для студентів спеціальності
123 «Комп'ютерна інженерія»
денної та заочної форм навчання

Відповідальний за випуск проф. Заковоротний О. Ю.

Роботу до видання рекомендував проф. Заполовський М. Й.

В авторській редакції

План 2025 р., поз. 8

Гарнітура Times New Roman. Ум. друк. арк. 12.

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, м. Харків, вул. Кирпичова, 2.

Електронне видання