Грайворонський Микола Владленович

Операційні системи

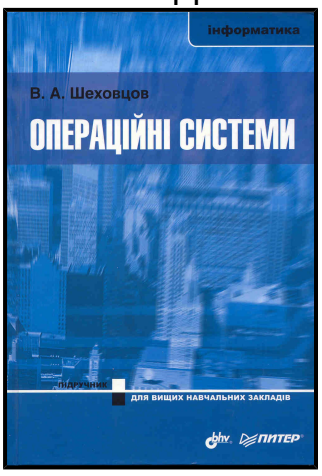
Конспект лекцій

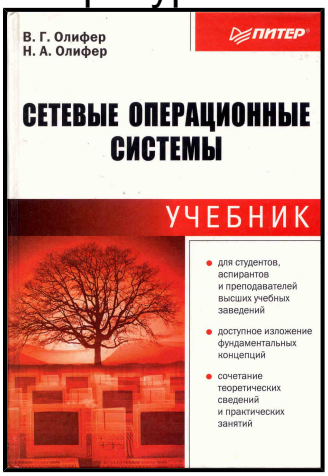
Лекція 1

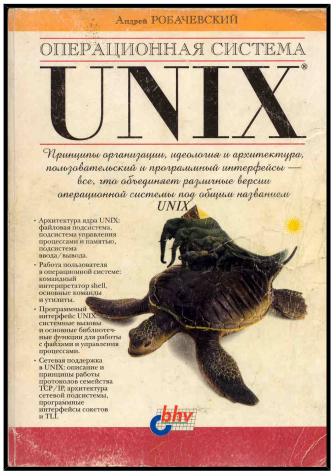
План лекційного курсу

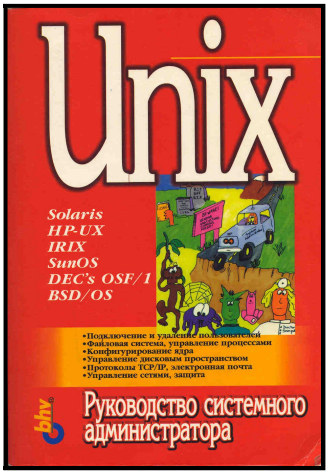
- Що таке ОС і якими вони бувають
  - Визначення, історія розвитку, класифікація
  - Архітектура
  - Вимоги до сучасних ОС
- Керування локальними ресурсами
  - Керування процесами
  - Керування пам’яттю
  - Керування пристроями введення/виведення
- Файлові системи
- Керування розподіленими ресурсами

Рекомендована література









План лекції

- Означення операційної системи
- Призначення ОС
- Історія розвитку
- Класифікація ОС
- Основні функції ОС

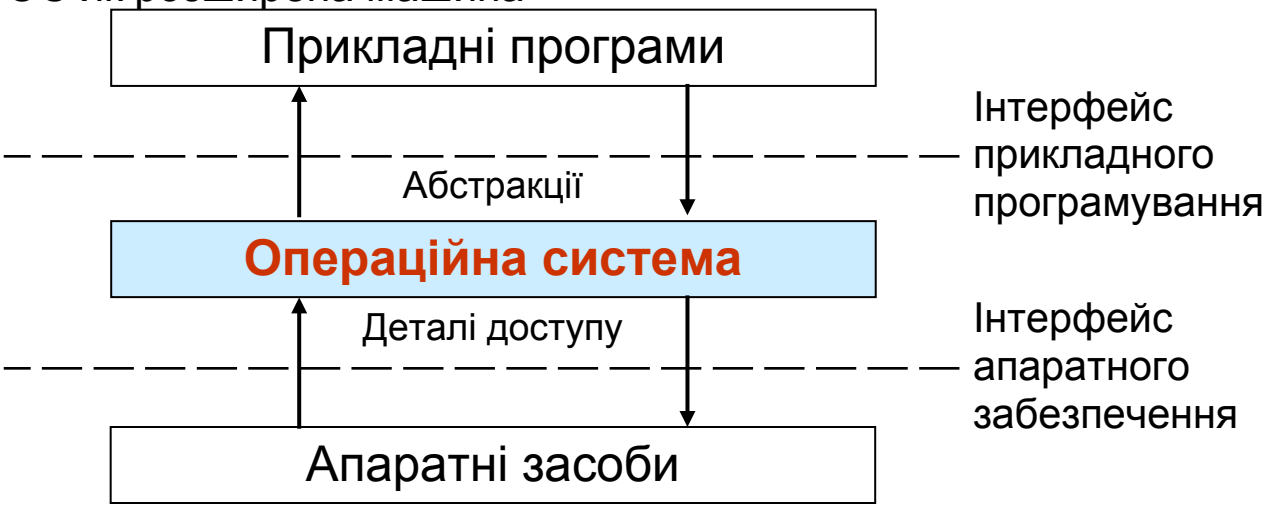
Важливі означення

- *Обчислювальна система* (або *комп'ютерна система*) – сукупність апаратного і програмного забезпечення комп'ютера
- Обчислювальні системи створюють для розв'язання практичних (прикладних) завдань користувачів – для цього створюють *прикладні програми* (*application programs*)
- Керування апаратним забезпеченням (у тому числі розподіл апаратних ресурсів) виокремили у спеціальний рівень програмного забезпечення, який і назвали *операційною системою* (*operating system*)

Означення операційної системи

- Операційна система – це комплекс взаємопов'язаних програм, що реалізує зв’язок (інтерфейс) між прикладними програмами (і користувачем) з одного боку і апаратними засобами комп'ютера з іншого боку
- ОС забезпечує:
  - Зручність і легкість взаємодії з апаратними засобами через *інтерфейс прикладного програмування* (*application programming interface, API*)
  - Раціональний розподіл апаратних ресурсів і керування ними

ОС як розширена машина



Приклад абстракції – файл

ОС надає *віртуальну машину*

ОС забезпечує *апаратну незалежність*

Керування ресурсами комп'ютера

- *Ресурси* – процесорний час, оперативна пам’ять, дисковий простір, пристрої введення-виведення (те, що може бути надано програмі у використання)
- ОС розподіляє ресурси, для чого розв'язує завдання:
  - Планування ресурсу (кому, коли і в якій кількості виділяти ресурс)
  - Виконання запитів на виділення ресурсів
  - Запобігання несанкціонованому доступу
- Розв'язання можливих конфліктів
  - Відстеження стану і облік використання ресурсів
- Два види розподілу ресурсів
  - *Просторовий розподіл* (пам'ять)
  - *Часовий розподіл* (процесор)

Історія розвитку ОС

- 1945-1955 (на електронних лампах)

  ОС не було

  Програмування виключно в машинних кодах
- 1955-1965 (на транзисторах)

  Системи пакетної обробки

  Алгоритмічні мови високого рівня, компілятори
- 1965-1980 (на інтегральних схемах)

  Багатозадачність
    - Багатозадачна пакетна обробка та системи розподілу часу,
    - Віртуальна пам'ять

  Програмно-сумісні сімейства ЕОМ (IBM/360, DEC PDP-11)
- 1980-… (на великих інтегральних схемах)

  “Дружній” інтерфейс, GUI, мережні ОС

  Персональні комп'ютери, стандартизовані обчислювальні мережі (Ethernet, Token Ring, TCP/IP)

Класифікація ОС за апаратною платформою

- ОС мейнфреймів
  - Продуктивність введення-виведення
  - Підтримка обробки значних обсягів даних
- Серверні ОС
  - Обслуговування великої кількості запитів до спільно використовуваних ресурсів
  - Підтримка мережної взаємодії
- Персональні ОС
  - Підтримка графічного інтерфейсу користувача
  - Підтримка мультимедіа-технологій
- Вбудовані ОС
  - Розміщення в малому обсязі пам'яті
  - Можливість прошивання в ПЗП

Класифікація ОС за областями застосування

- Системи пакетної обробки
  - ОС мейнфреймів
- Системи розділення часу
  - VMS
  - UNIX
  - Linux
  - Windows
- Системи реального часу
  - QNX

Лекція 2. Архітектура операційних систем

План лекції

- Поняття архітектури операційної системи
- Ядро і системне програмне забезпечення
- Привілейований режим і режим користувача
- Монолітна архітектура
- Багаторівнева архітектура
- Мікроядрова архітектура
- Архітектура ОС UNIX і Windows
- Об'єктна архітектура

Основні функції ОС

- Керування процесами і потоками
- Керування пам'яттю
- Керування введенням-виведенням
- Керування файлами (файлові системи)
- Мережна підтримка
- Безпека даних
- Інтерфейс користувача

Базові поняття

- *Архітектура операційної системи* визначає набір і структурну організацію компонентів, кожний з яких відповідає за певні функції, а також порядок взаємодії цих компонентів між собою та із зовнішнім середовищем.
- Фундаментальні можливості, які надають компоненти ОС, становлять *механізм* (*mechanism*). Рішення щодо використання цих можливостей визначають *політику* (*policy*). Механізм може бути відокремленим від політики, тоді компонент, що його реалізує, називають "*вільним від політики*" (*policy-free*).
- Базові компоненти ОС, які відповідають за найважливіші функції і виконуються у привілейованому режимі (і зазвичай перебувають у пам'яті постійно), називають *ядром операційної системи* (*operating system kernel*).

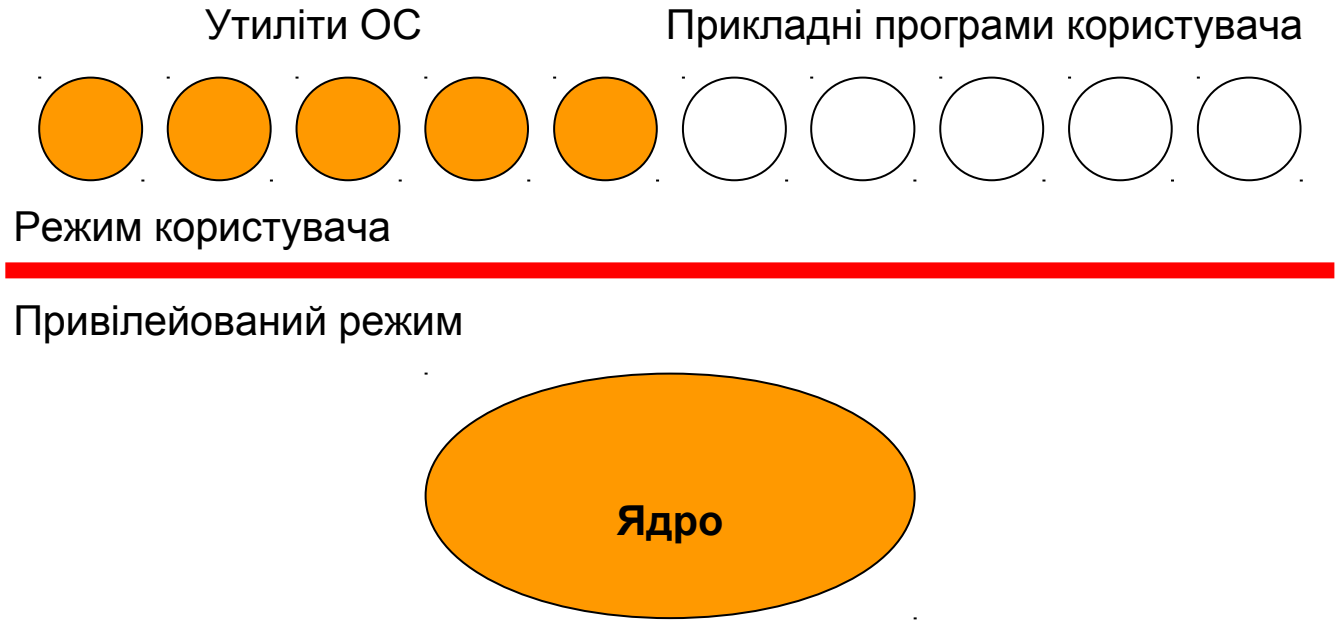
Ядро і системне програмне забезпечення

- Ядро
  - Виконується в привілейованому режимі
  - Постійно перебуває в оперативній пам'яті
  - Зазвичай виконує такі функції:
    - Обробка переривань
    - Керування пам'яттю
    - Керування введенням/виведенням
- Системне програмне забезпечення
  - Системні програми (утиліти)
    - Командний інтерпретатор
    - Програми резервного копіювання та відновлення даних
    - Засоби діагностики та адміністрування
  - Системні бібліотеки

Привілейований режим і режим користувача

- Привілейований режим (режим ядра)
  - Дозволяє втручатись в роботу будь-якої програми (наприклад, для перемикання контекстів або для розв'язання конфліктів)
- Режим користувача
  - Не дозволяє критичні команди (зупинка системи, перемикання контекстів, прямий доступ до пам'яті з заданими межами та до пристроїв введення-виведення)
  - Доступ до функцій ядра здійснюється через *системні виклики*
- Необхідна апаратна підтримка з боку процесора

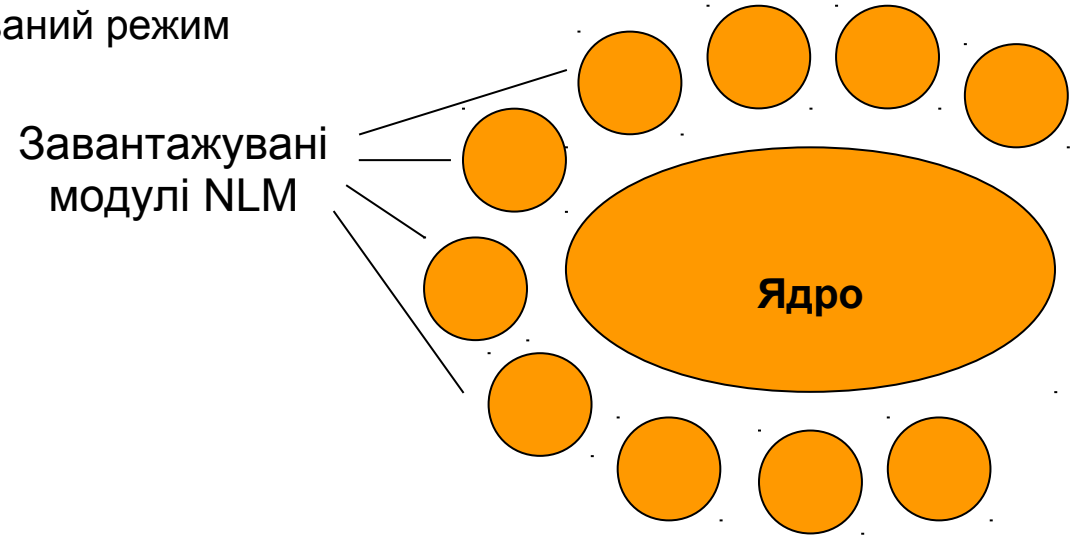
Типова архітектура ОС: ядро у привілейованому режимі



Архітектура ОС Novell NetWare: ядро і прикладні програми в одному режимі

Режим користувача

Привілейований режим

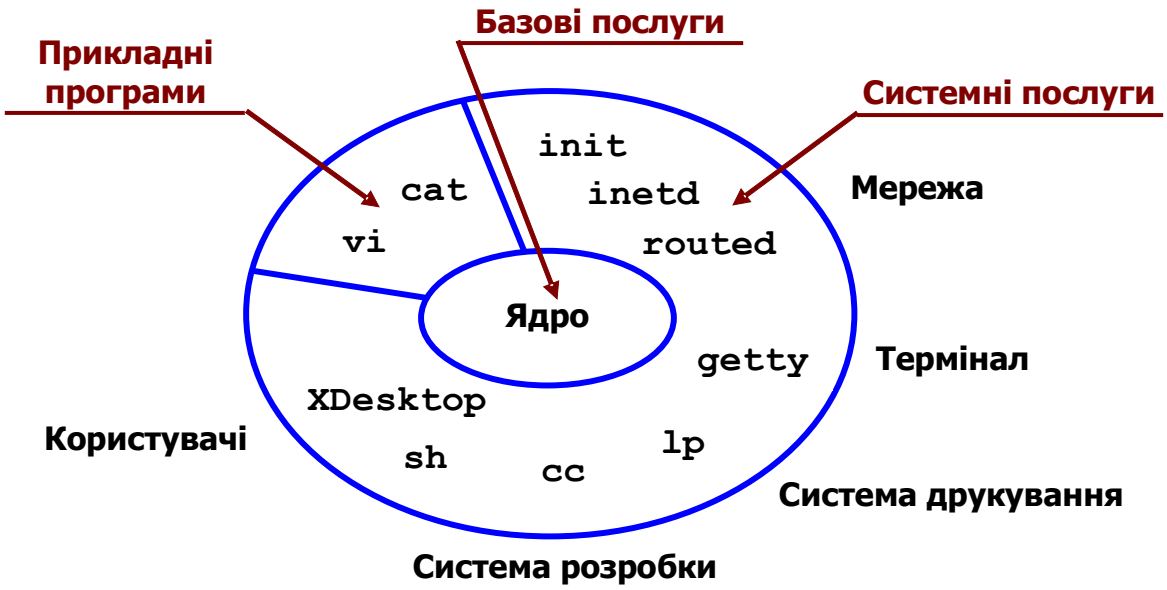


- Перевага – швидкодія
- Недолік – відсутній захист

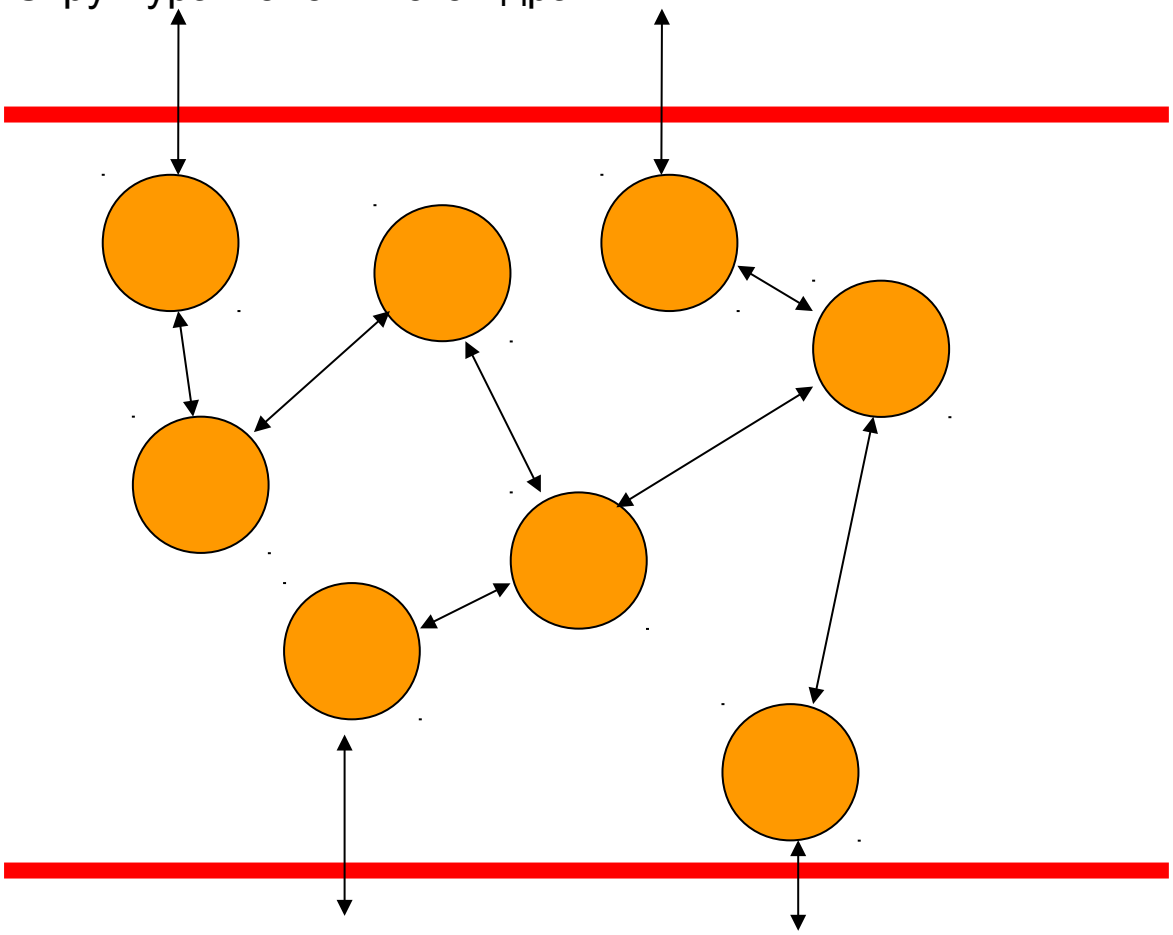
Різні архітектури ОС

- Монолітні системи
  - Усі компоненти знаходяться в ядрі
  - Немає чіткої ієрархії компонентів
- Багаторівневі системи
  - Компоненти утворюють ієрархію рівнів (шарів)
  - Кожний рівень спирається на функції попереднього рівня
- Мікроядрова архітектура
  - Реалізація більшості функцій винесена за межі ядра у прикладні сервери
  - Ядро підтримує взаємодію між компонентами

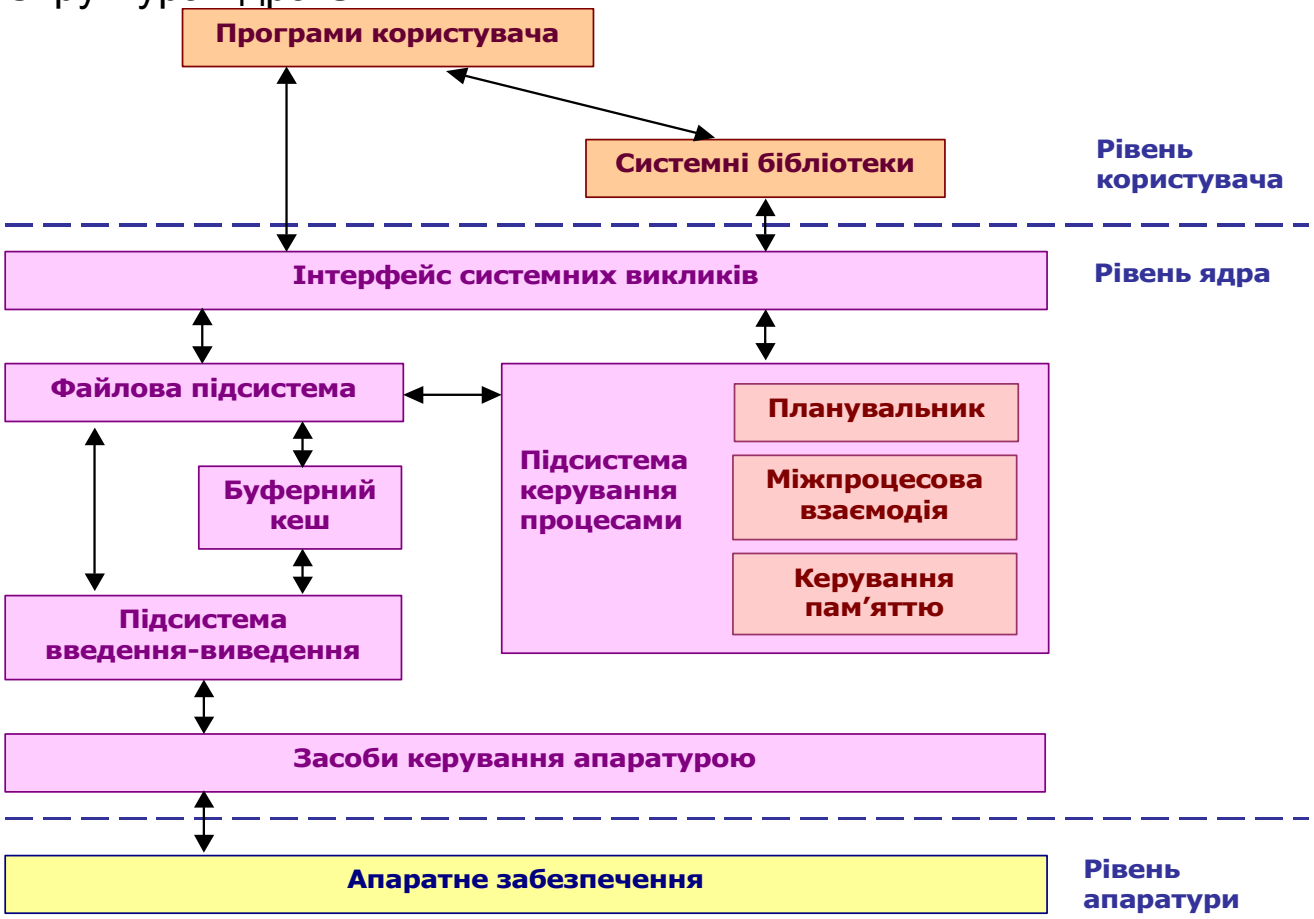
Архітектура системи UNIX (монолітне ядро)



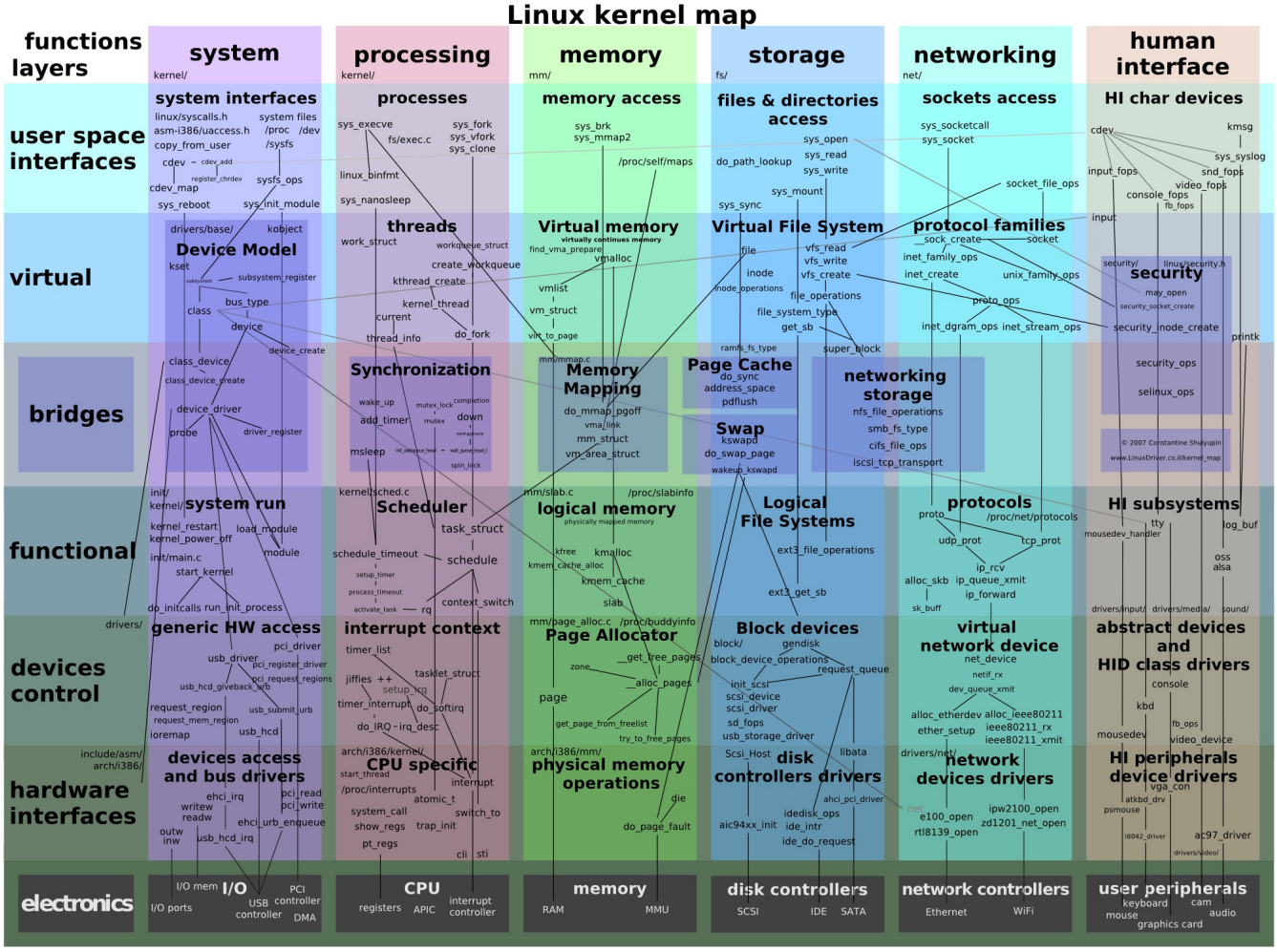
Структура монолітного ядра



Структура ядра UNIX

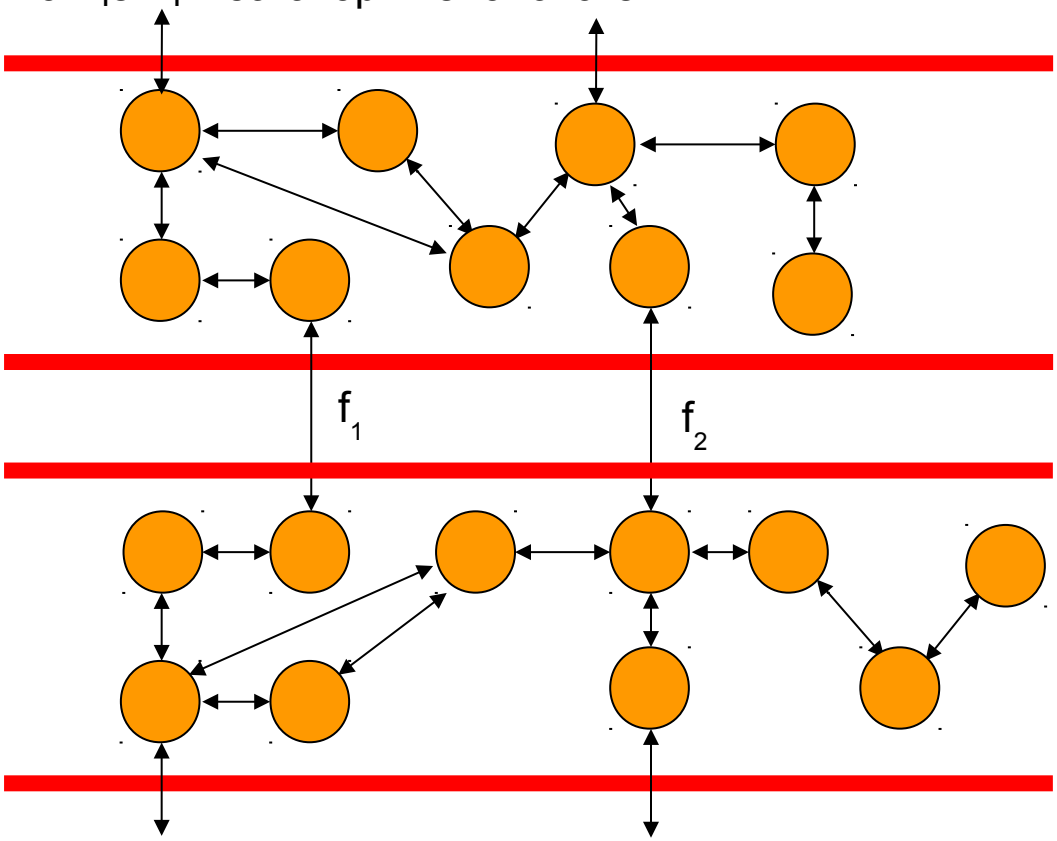


Linux kernel map

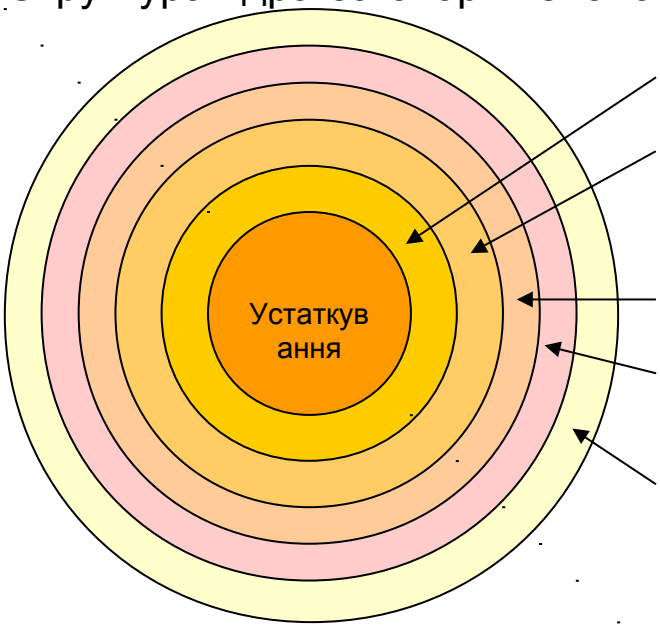


Ver 0.6, 1/1/2008

Концепція багаторівневої системи

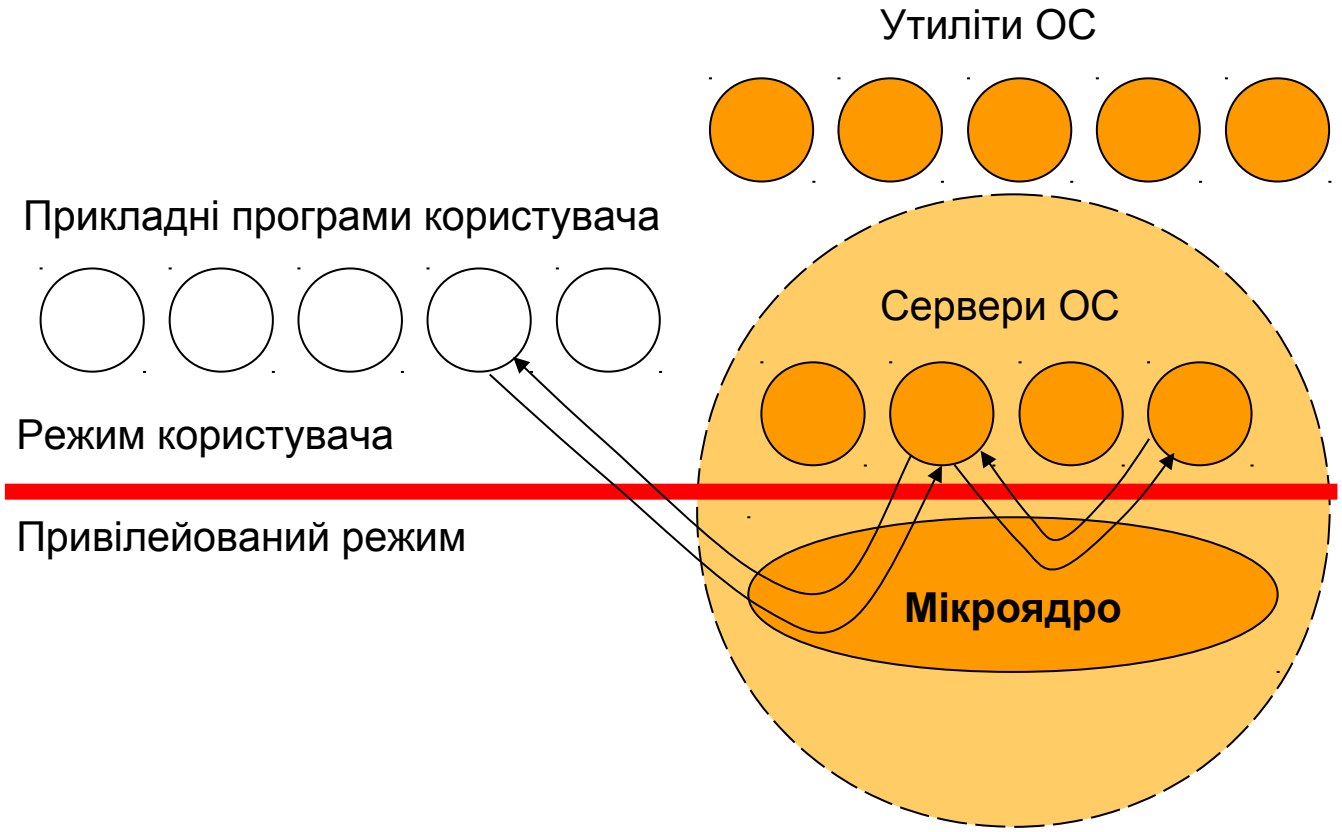


Структура ядра багаторівневої системи

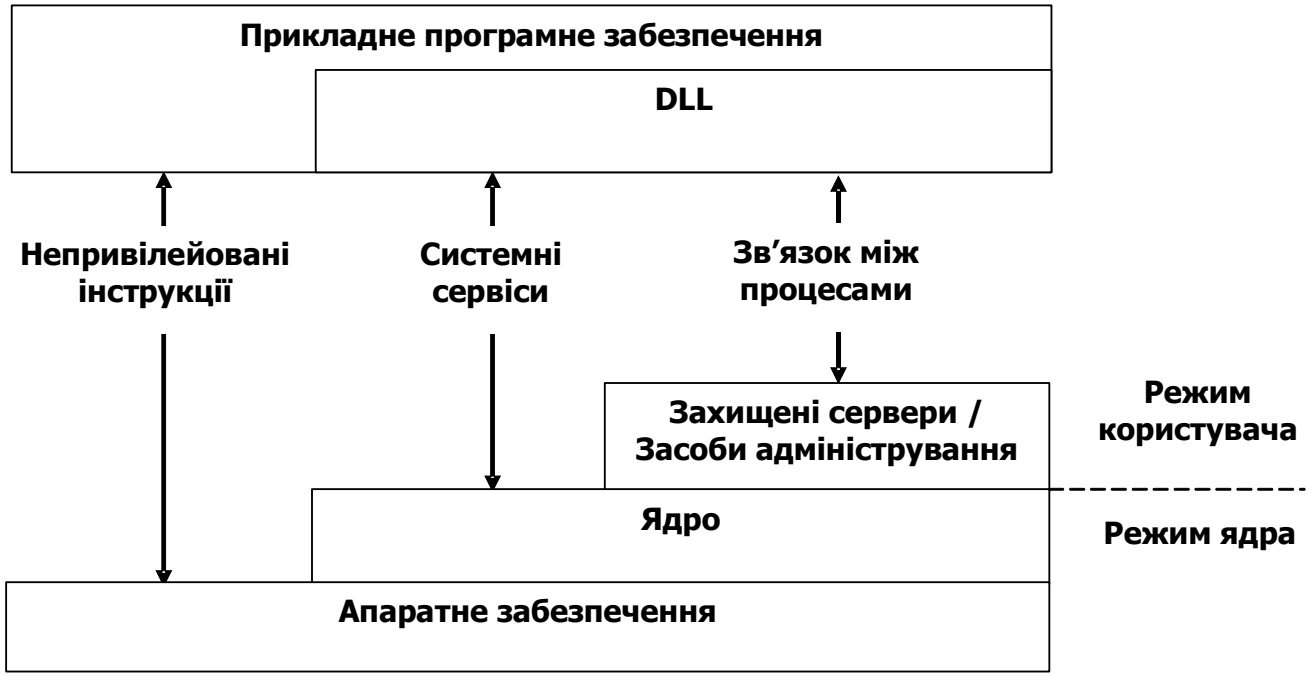


- Засоби апаратної підтримки ОС
- Засоби абстрагування від устаткування (hardware abstraction layer, HAL)
- Засоби, що реалізують базові механізми ядра
- Засоби керування ресурсами (менеджери ресурсів)
- Інтерфейс системних викликів

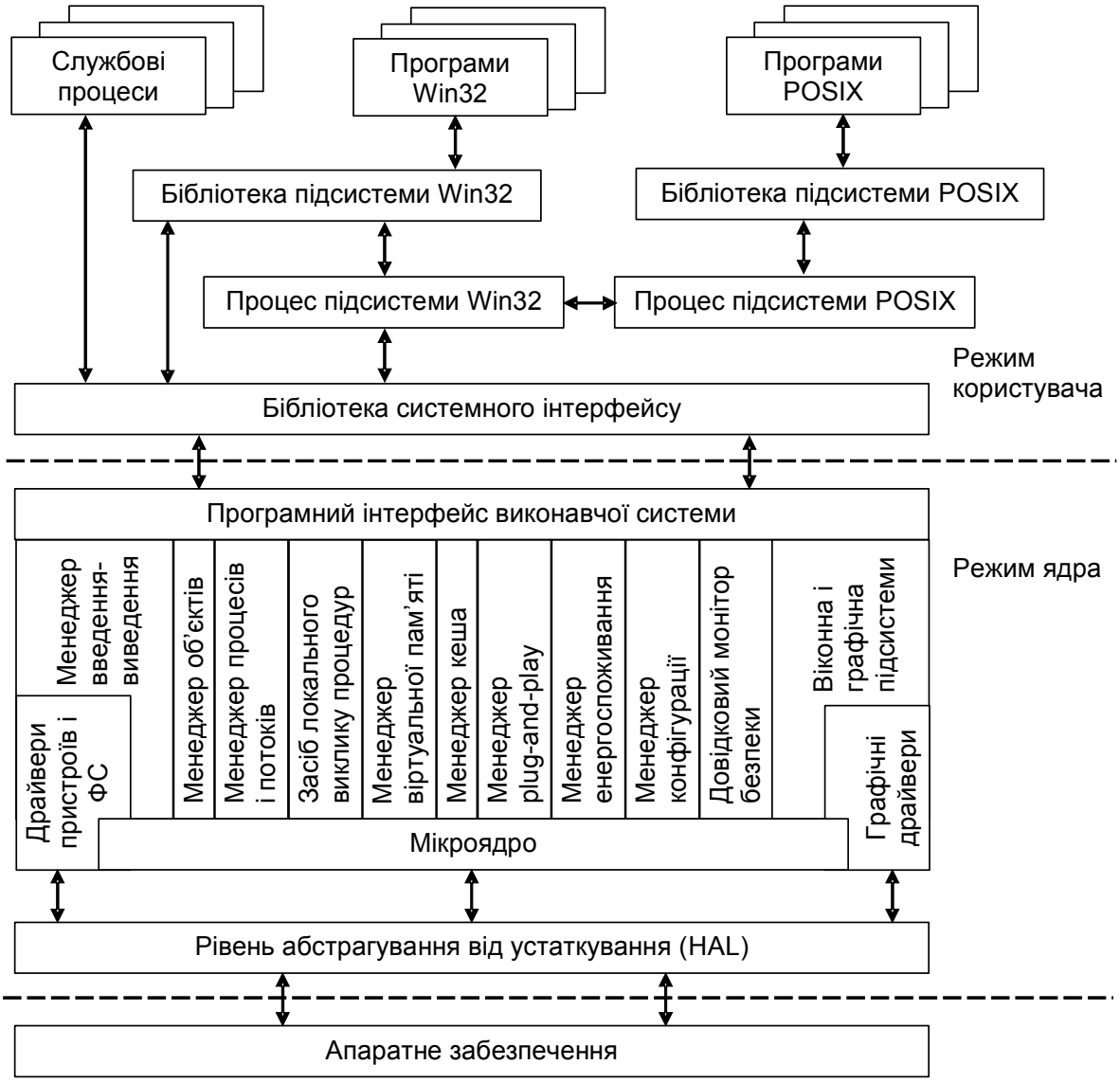
Мікроядрова архітектура



Вертикальна декомпозиція архітектури ОС Windows



Базові компоненти ОС Windows NT



Об'єктна архітектура (Windows)

- Імена об'єктів організовані в єдиний *простір імен*
- Об’єкти надають універсальний інтерфейс для доступу до системних ресурсів
  - Доступ до усіх об'єктів здійснюється однаково
  - Після створення об'єкта, або після отримання доступу до наявного, менеджер об'єктів повертає прикладній програмі *дескриптор об'єкта* (*object handle*)
- Забезпечено захист ресурсів
  - Кожну спробу доступу до об'єкта розглядає *підсистема захисту*
- Об'єкт має *заголовок* і *тіло*. Структура заголовка об'єкта:
  - Ім'я об'єкта, його місце у просторі імен
  - Дескриптор захисту
  - Витрата квоти (ціна відкриття дескриптора об'єкта)
  - Список процесів, що отримали доступ до дескрипторів об'єкта

Лекція 3. Вимоги до сучасних операційних систем

План лекції

- Функціональні і ринкові вимоги до ОС
- Апаратна незалежність і здатність ОС до перенесення
- Програмна сумісність, прикладні програмні середовища
- Розширюваність

Функціональні і ринкові вимоги до ОС

- *Функціональні* – вимоги до функцій, які підтримує ОС (вимоги користувача)
- *Ринкові* – вимоги до економічної ефективності розроблення і супроводження ОС (вимоги розробника)

Функціональні вимоги до ОС

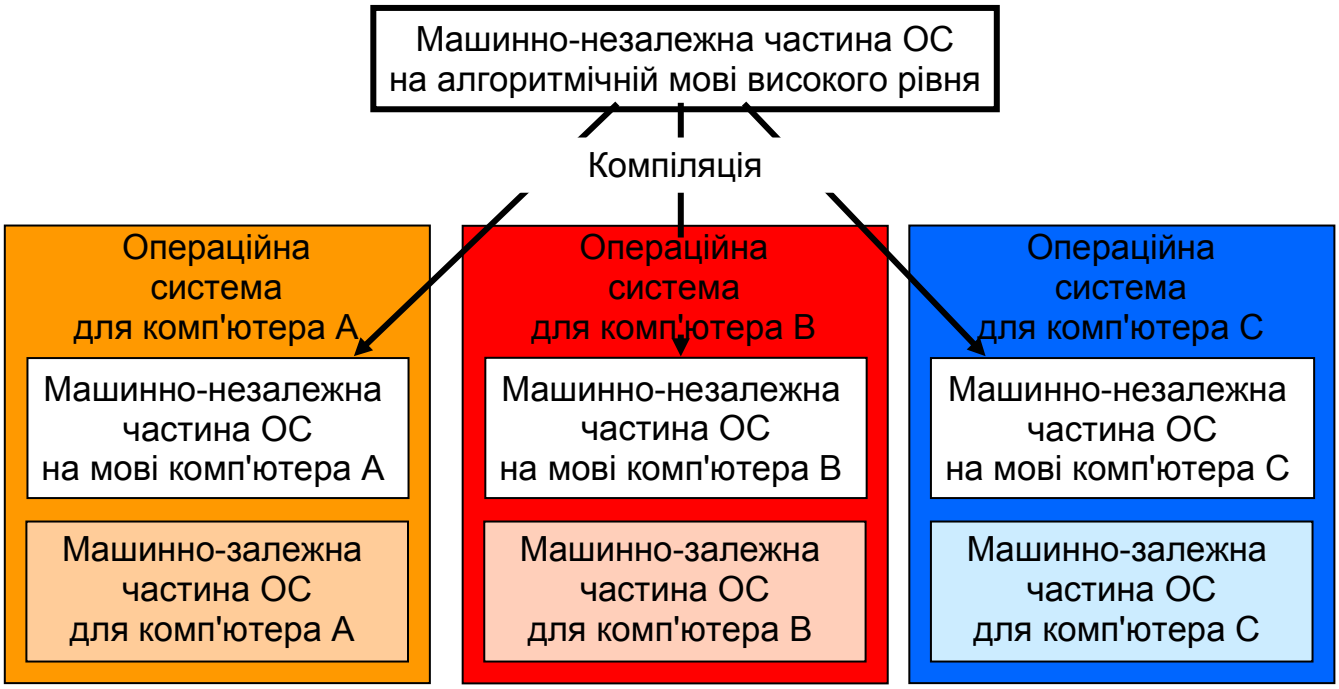
- Ефективне керування ресурсами
- Зручний інтерфейс користувача
- Зручний та ефективний інтерфейс прикладних програм
- Багатозадачність, багатопотоковість
- Віртуальна пам'ять
- Багатовіконний графічний інтерфейс
- Підтримка мережної взаємодії
- Надійність, відмовостійкість
- Безпека даних

Ринкові вимоги до ОС

- Здатність до перенесення (portability)
- Програмна сумісність (у тому числі – зворотна сумісність)
- Розширюваність

Апаратна незалежність і здатність ОС до перенесення

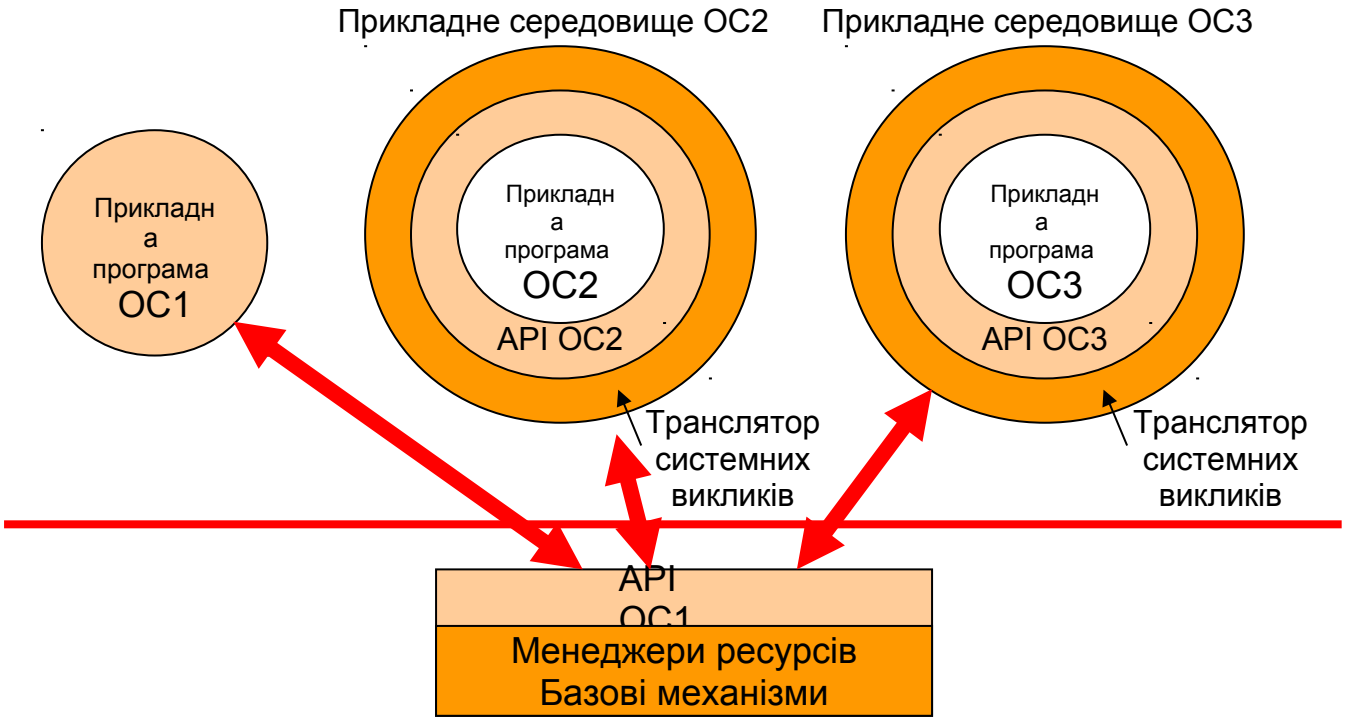
- Засоби апаратної підтримки ОС
  - Система переривань
  - Засоби підтримки привілейованого режиму
  - Засоби трансляції адрес
  - Засоби перемикання процесів
  - Системний таймер
  - Засоби захисту оперативної пам'яті
  - Захист пристроїв введення-виведення
- Здатність до перенесення (portability)
  - Більша частина коду має бути написана мовою високого рівня, для якої існують транслятори на різних апаратних платформах
  - Код, що залежить від апаратного забезпечення, має бути відокремленим від іншої частини системи
  - Обсяг машинно-залежного коду має бути мінімізованим



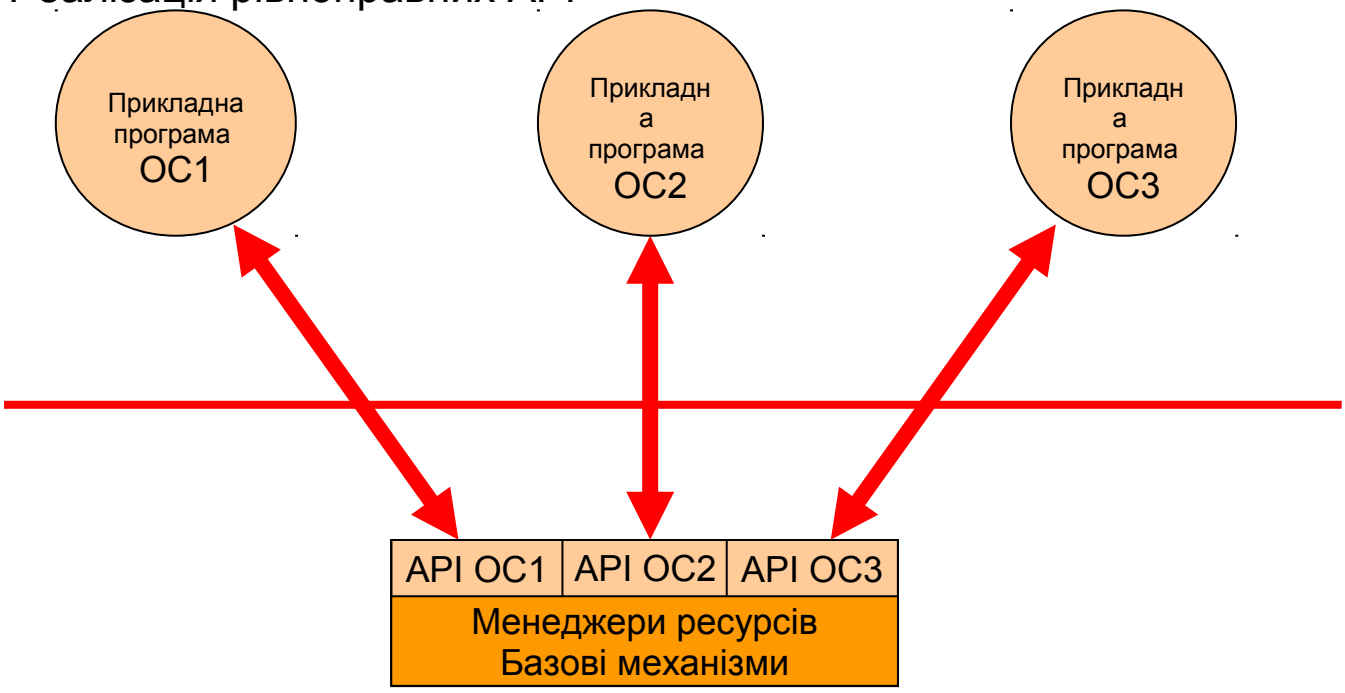
Програмна сумісність

- *Програмна сумісність* – можливість виконувати у середовищі ОС програми, розроблені для іншої ОС
- *Зворотна сумісність* – можливість виконувати у середовищі ОС програми, розроблені для попередньої версії ОС
- *Сумісність вихідних текстів* – можливість перенесення вихідних текстів
  - Необхідна наявність компілятора (стандартизація мов програмування, розробка стандартних компіляторів)
  - Необхідна сумісність API (стандартизація інтерфейсів)
- *Бінарна сумісність* – можливість перенесення виконуваного коду
  - Якщо архітектура процесора (набір команд, система адресації, діапазон адрес) сумісна, тоді необхідні лише
    - сумісність API
    - сумісність внутрішньої структури виконуваного файлу
  - Якщо архітектури процесорів несумісні, то необхідна *емуляція середовища виконання*
    - Для прискорення емуляції – трансляція бібліотек

Реалізація прикладних програмних середовищ



Реалізація рівноправних API



Розширюваність ОС

- ОС може жити довше за апаратуру!
- *Розширюваність* – можливість додавання нових функцій при збереженні основної частини коду
  - Підтримка нової апаратури (CD-ROM, flash)
  - Зв'язок з мережами нових типів
  - Нові технології інтерфейсу користувача (GUI)
  - Нова апаратна архітектура (багатопроцесорність)
- Шляхи досягнення розширюваності:
  - Модульна структура ОС
  - Використання об`єктів
  - Технологія клієнт-сервер із застосуванням мікроядрової архітектури
  - Завантажувані модулі драйверів

Лекція 4. Керування процесами і потоками

План лекції

- Мультипрограмування
- Означення процесу і потоку
- Моделі процесів і потоків
- Керування потоками, планування
- Опис процесів і потоків: керуючий блок, образ, дескриптор і контекст
- Стани потоків

Мультипрограмування

- *Мультипрограмування*, або *багатозадачність* (*multitasking*) – спосіб організації обчислювального процесу, в якому на одному процесорі почергово виконуються кілька програм
- Критерії ефективності обчислювальної системи:
  - *Перепускна спроможність* – кількість задач, яку здатна ефективно виконувати система в одиницю часу
  - Зручність роботи користувачів (насамперед, можливість одночасно інтерактивно працювати з низкою прикладних програм на одній машині)
  - *Реактивність системи* – здатність системи дотримувати наперед задані (короткі) інтервали між одержанням запиту і результатом його оброблення

Процеси і потоки

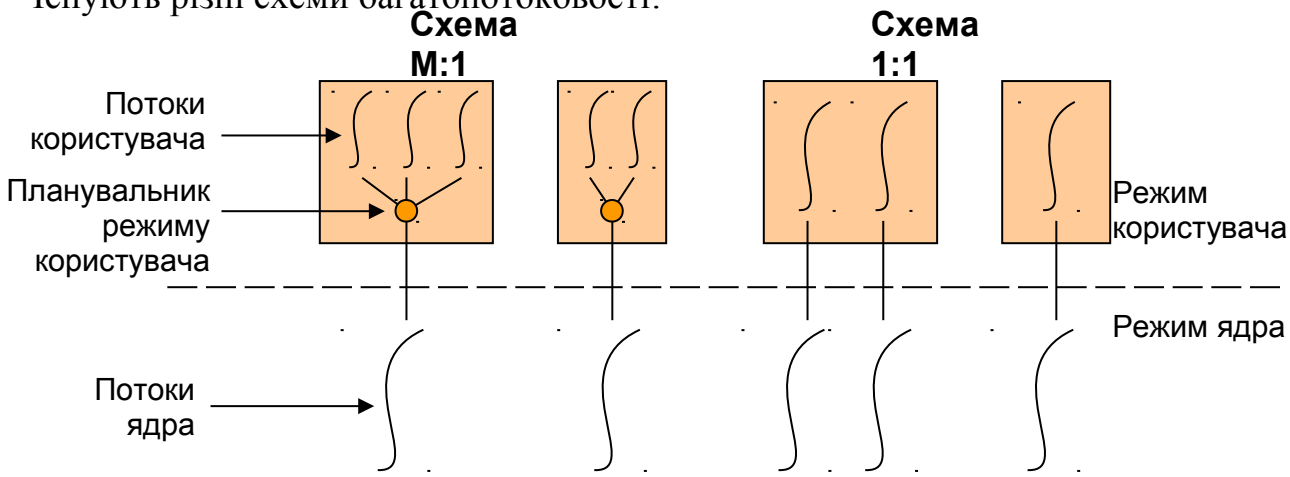
- *Процес* – це абстракція, що описує програму, яка виконується в даний момент
- Складові частини процесу:
  - Послідовність виконуваних команд процесора
  - Набір адрес пам'яті (адресний простір), у якому розташовані команди процесора і дані для них
- *Потік* (*нитка*, *thread*) – набір послідовно виконуваних команд процесора
- У багатопотокових системах процес – це сукупність одного або декількох потоків і захищеного адресного простору, у якому ці потоки виконуються

Моделі процесів і потоків

- Однозадачні системи – один адресний простір (один процес), у якому може виконуватись один потік
- Деякі сучасні вбудовані системи – один адресний простір (один процес), у якому можуть виконуватись кілька потоків
- Однопотокова *модель процесів* (традиційні системи UNIX) – багато процесів, але у кожному з них лише один потік
- Багатопотоковість, або *модель потоків* (більшість сучасних ОС) – багато процесів, у кожному з яких може бути багато потоків

Потоки ядра і потоки користувача

- Ядро ОС керує потоками ядра. Керування потоками користувача здійснюють спеціальні системні бібліотеки (підтримка *потоків POSIX*)
- Існують різні схеми багатопотоковості:



Переваги і недоліки багатопотоковості

- + (переваги)
  - Реалізація різних видів *паралелізму* (*concurrency*) – багатопроцесорних обчислень, введення-виведення, взаємодії з користувачем, розподілених застосувань
  - Масштабованість (особливо із зростанням кількості процесорів)
  - Необхідно менше ресурсів, ніж для підтримки процесів
  - Ефективний обмін даними через спільну пам'ять
- – (недоліки)
  - Складність розроблення й налагодження багатопотокових застосувань
  - Зниження надійності застосувань (можливі “гонки”, витоки пам'яті, втрата даних)
  - Можливість зниження продуктивності застосувань

Завдання підсистеми керування процесами (потоками)

- Створення та знищення процесів і потоків
- *Планування* виконання процесів або потоків, тобто розподіл процесорного часу між ними
  - Визначення моменту часу для зміни потоку, що виконується
  - Вибір наступного потоку для виконання
  - Переключення контекстів
- Забезпечення процесів і потоків необхідними ресурсами
- Підтримання взаємодії між процесами

Опис процесів і потоків

- *Керуючий блок потоку* (*Thread Control Block,* TCB)
  - Ідентифікаційні дані потоку
  - Стан процесора потоку (регістри процесора, лічильник інструкцій, покажчик на стек)
  - Інформація для планування потоків
- *Керуючий блок процесу* (*Process Control Block*, PCB)
  - Ідентифікаційні дані процесу
  - Інформація про потоки цього процесу (наприклад, покажчики на їхні керуючі блоки)
  - Інформація, на основі якої можна визначити права процесу на використання ресурсів
  - Інформація про розподіл адресного простору процесу
  - Інформація про ресурси введення-виведення та файли, які використовує процес
- *Таблиця процесів* (потоків) – зв'язний список або масив відповідних керуючих блоків

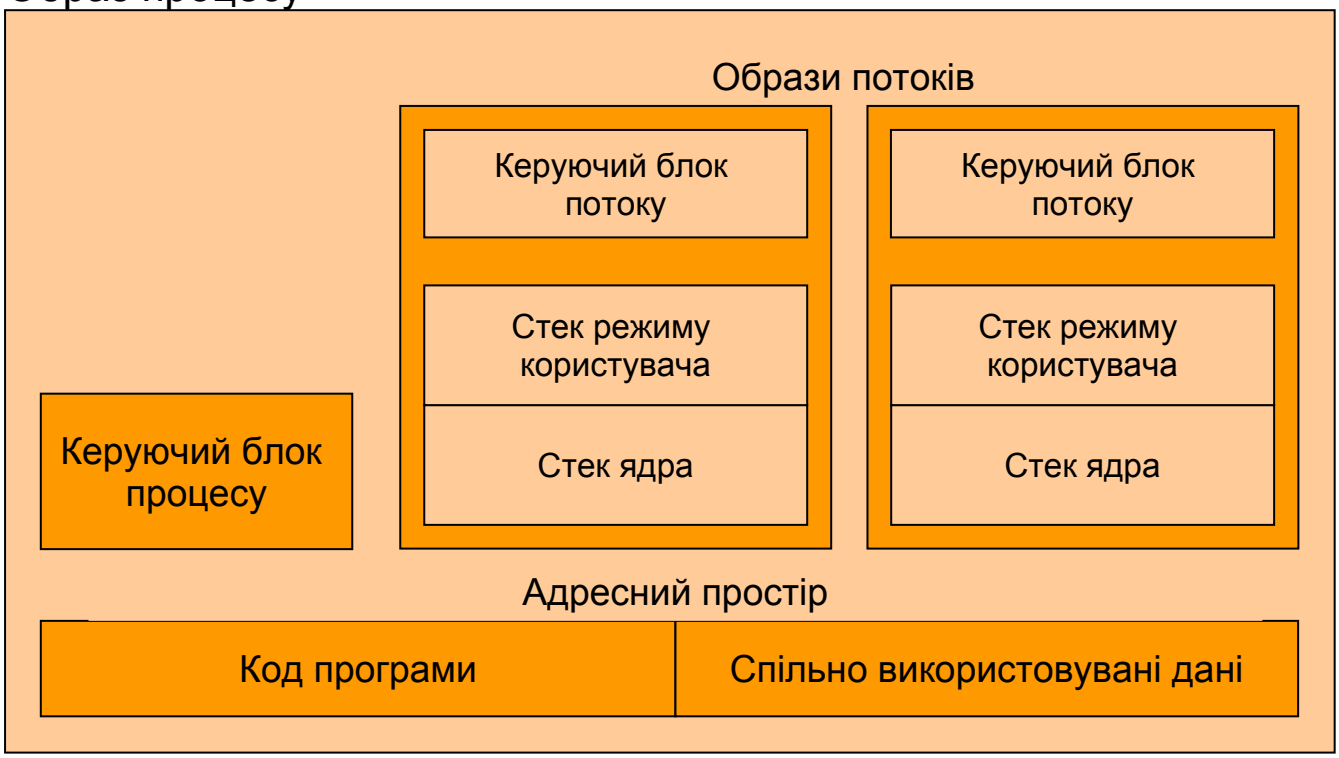
Приклади структур, що описують процеси

- OS/360 – *Task Control Block*, TCB (керуючий блок задачі)
- OS/2 – *Process Control Block*, PCB (керуючий блок процесу)
- UNIX – *proc*, дескриптор процесу
- Windows – *object-process* (об'єкт-процес)

Образи процесу і потоку

- Образ потоку
  - Керуючий блок потоку
  - Стек ядра
  - Стек користувача
- Образ процесу
  - Керуючий блок процесу
  - Програмний код користувача
  - Дані користувача
  - Інформація образів потоків процесу

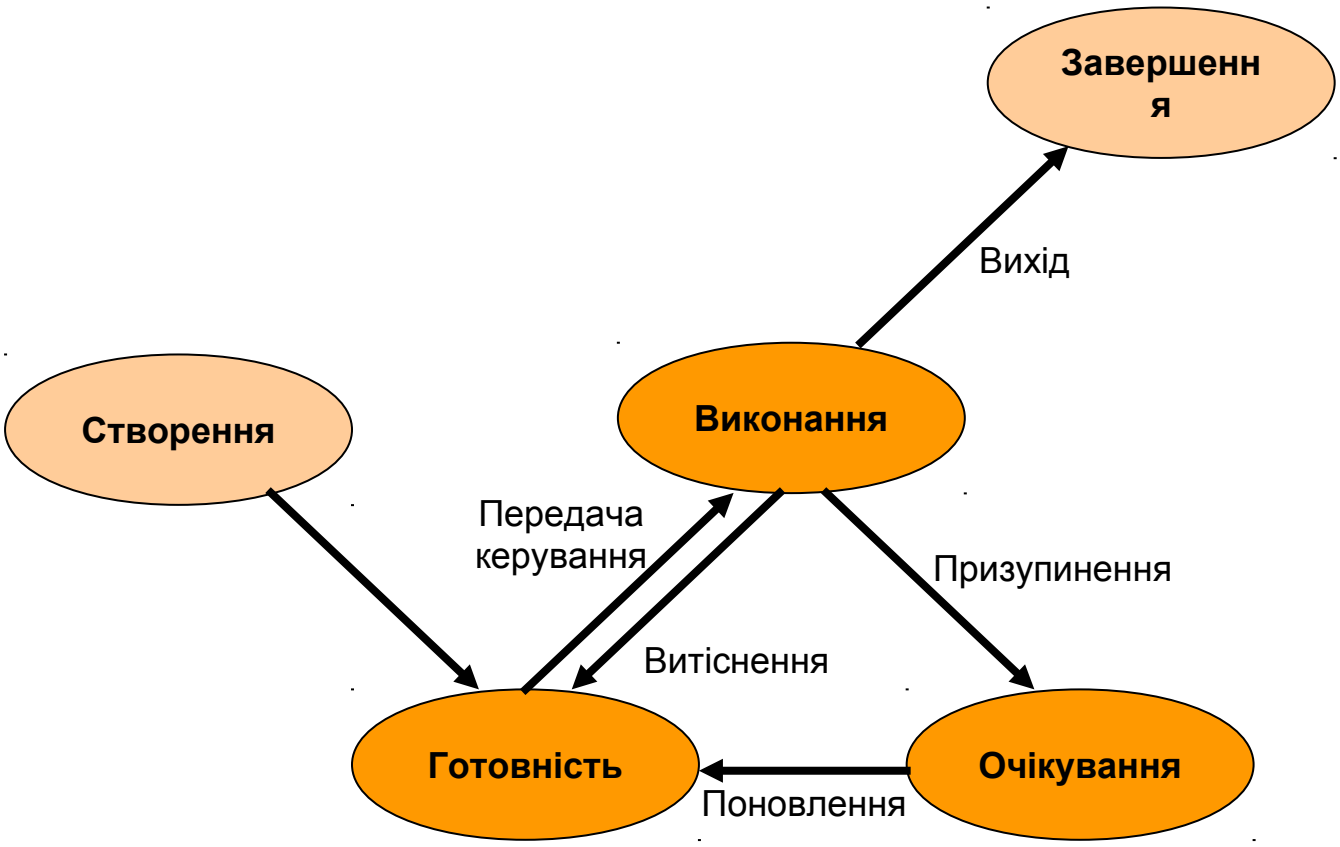
Образ процесу



Дескриптор і контекст процесу (UNIX)

- *Дескриптор* – структура, що містить інформацію, необхідну для планування процесів. Ця інформація необхідна протягом усього часу життєвого циклу процесу
  - Ідентифікатор процесу
  - Стан процесу
  - Пріоритет процесу
  - Розміщення процесу в оперативній пам'яті і на диску
  - Ідентифікатор користувача, що створив процес
- *Контекст* – структура, що містить інформацію, необхідну для відновлення виконання процесу (менш оперативна, але більш об'ємна частина інформації, ніж у дескрипторі)
  - Стан регістрів процесора
  - Коди помилок виконаних системних викликів
  - Інформація про відкриті файли і незавершені операції введення-виведення
- Таблиця процесів – список дескрипторів
- Контекст зв'язаний з образом процесу і переміщується разом з ним (наприклад, може бути витісненим з оперативної пам'яті на диск)

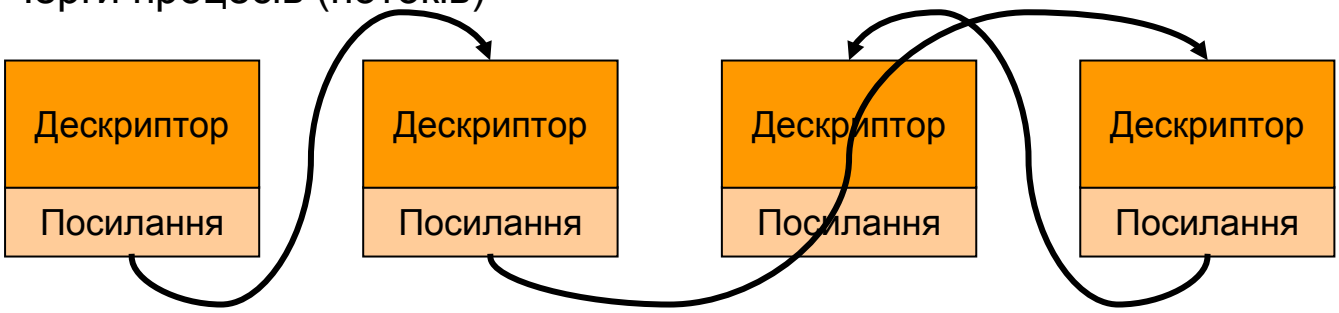
Стани потоків



Створення процесів

- Необхідні дії
  - Створити інформаційні структури, що описують процес (керуючий блок: дескриптор, контекст)
  - Виділити оперативну пам’ять
  - Завантажити кодовий сегмент процесу в оперативну пам’ять
  - Поставити дескриптор процесу у чергу готових процесів
- Створення у два етапи (POSIX)
  - fork() – створює точну копію поточного процесу
  - exec() – заміняє код поточного процесу на код іншого
- Створення в один етап (Windows) – CreateProcess()
  (це не системний виклик, а бібліотечна функція)
- Копіювання під час запису
  - Під час виклику fork() дані з пам'яті предка у пам’ять нащадка не копіюють, а натомість відображають адресний простір і помічають області пам'яті як захищені від запису
  - У разі спроби запису виділяють пам’ять і здійснюють копіювання

Черги процесів (потоків)



Лекція 5. Керування процесами і потоками (продовження) – планування

План лекції

- Завдання планування
- Витісняльна і невитісняльна багатозадачність
- Приорітетні і безпріоритетні дисципліни планування
- Квантування
- Алгоритми планування
- Керування процесами і потоками у сучасних ОС
  - UNIX
  - Linux
  - Solaris
  - Windows

Планування виконання процесів або потоків

- *Планування* – розподіл процесорного часу між процесами або потоками
- Завдання планування:
  - Визначення моменту часу для зміни потоку, що виконується
  - Вибір наступного потоку для виконання
  - Переключення контекстів
- Перші два завдання вирішуються здебільшого програмними засобами, третє – головним чином, апаратними засобами із застосуванням механізму переривань

Витісняльні і невитісняльні алгоритми планування

- *Витісняльні* (*preemptive*) – рішення про переключення з виконання одного потоку на виконання іншого (“витіснення” потоку з процесора) приймає операційна система
- *Невитісняльні* (*non-preemptive*) – активний потік виконується, поки він сам за власною ініціативою не віддасть керування операційній системі
- Не плутати з пріоритетними/безпріоритетними дисциплінами планування!

Квантування

- Зміна потоку відбувається, якщо:
  - Потік завершився
  - Виникла помилка (переривання)
  - Потік перейшов у стан очікування
  - Вичерпано квант процесорного часу
- *Квантування* – це один з підходів, що реалізує витісняльну багатозадачність
- Кванти можуть бути фіксованої величини, або змінюватись

Пріоритети

- Розрізняють *пріоритетні* і *безпріоритетні дисципліни планування*
- *Відносні пріоритети* впливають лише на вибір процесу з черги на виконання
  - Або завжди вибирають лише процес з найвищим пріоритетом
  - Або обслуговуються усі черги, але пропорційно пріоритетам (алгоритм *Weighed Round Robin*, *WRR*)
- *Абсолютні пріоритети* – це один з алгоритмів реалізації витісняльної багатозадачності. Зміна потоку:
  - Потік завершився
  - Виникла помилка (переривання)
  - Потік перейшов у стан очікування
  - У черзі з'явився потік з вищим пріоритетом

Алгоритми планування

- *Планування за принципом FIFO*
  - Одна черга потоків
  - Невитісняльне планування
  - Проблема – “ефект конвою”
- *Кругове планування* (*round-robin scheduling*)
  - Безпріоритетне планування (одна черга потоків)
  - Квантування у “чистому” вигляді
  - Рекомендована довжина кванта – 10-100 мс
- *Багаторівневі черги* (*multilevel queues*)
  - Кілька черг для груп потоків із різними пріоритетами
  - Якщо в черзі немає жодного потоку, переходять до черги з нижчим пріоритетом
  - У кожній черзі застосовують простий алгоритм (наприклад, кругове планування), не звертаючи уваги на потоки в інших чергах
  - Для різних черг можна застосовувати різні алгоритми
  - Проблема – “голодування” (*starvation*)

Планування на підставі характеристик подальшого виконання

- *STCF* (*Shortest Time to Completion First*, перший – із найкоротшим часом виконання)
  - Алгоритм є теоретично оптимальним за критерієм середнього часу відгуку
  - Недолік – не завжди можна передбачити час виконання (підходить для довготермінового планування, не підходить для короткотермінового)
  - Аналог з витісняльним плануванням (за принципом абсолютних пріоритетів) – *SRTCF* (*Shortest Remaining Time to Completion First*, перший – із найкоротшим часом виконання, що залишився)
- *Багаторівневі черги зі зворотним зв'язком* (*Multilevel Feedback Queues*)
  - Потокам дозволено переходити з черги в чергу
  - Потоки у черзі об'єднуються не за пріоритетом, а за довжиною інтервалу використання процесора
  - Коли потік не вичерпав квант – він становиться у кінець тієї ж черги, коли вичерпав – переводиться у нижчу чергу

Лотерейне планування (lottery scheduling)

- Принцип:
  - Потік отримує певну кількість лотерейних квитків, кожен з яких дає право користуватись процесором упродовж часу T
  - Планувальник через проміжок часу T проводить “розіграш”; потік, що “виграв”, дістає керування
- Шляхом розподілу і динамічного перерозподілу квитків можна:
  - Емулювати кругове планування
  - Емулювати планування з пріоритетами
  - Емулювати CRTCF
  - Забезпечити заданий розподіл процесорного часу між потоками
  - Динамічно змінювати пріоритети

Керування процесами в UNIX/Linux

- Образ процесу містить:
  - Керуючий блок
  - Код програми
  - Стек процесу
  - Глобальні дані
- Ідентифікатор процесу PID – унікальний
- Підтримується зв’язок предок-нащадок
  - Ідентифікатор процесу-предка PPID вказують під час створення
  - Якщо предок завершує роботу, то PPID:=1 (процес init)
- Створення процесу: fork(), exec() (Linux – execve())

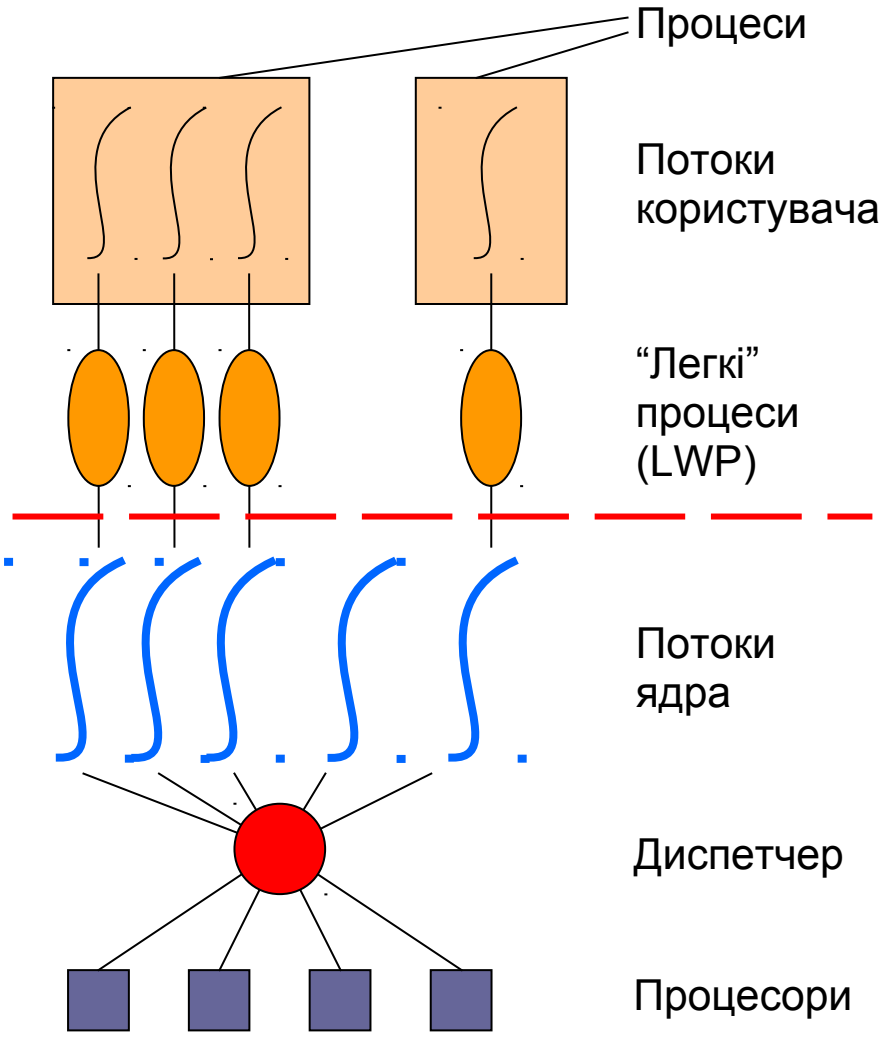
- Керуючий блок в Linux – структура task_struct, містить в собі і дескриптор, і контекст
  - В ядрі до версії 2.4 керуючі блоки зберігались у масиві максимального розміру 4 кБ (“таблиця процесів системи”)
  - В ядрі версії 2.4 і вище – дві динамічні структури без обмеження довжини
    - Хеш-таблиця (дає змогу швидко знаходити процес за його PID)
    - Кільцевий двозв’язний список, що забезпечує виконання дій у циклі для усіх процесів системи
- У UNIX SVR4 і BSD керуючий блок складається з двох структур
  - proc – дескриптор (утворюють таблицю процесів)
  - user (u) – контекст (пов'язаний з процесом)

Багатопотоковість у Linux

- Традиційна реалізація: бібліотека *LinuxThreads*
  - Потоки – це процеси, що користуються спільними структурами даних, але мають окремі стеки
  - Створення потоку: системний виклик clone()
  - Недоліки:
    - Створення потоку збільшує кількість процесів у системі
    - Кожний потік має власний PID (це суперечить POSIX)
    - Існує зв’язок предок-нащадок (чого не повинно бути)
    - Кожне багатопотокове застосування обов'язково створює додатковий потік-менеджер
- Нова реалізація: *NPTL* (*Native POSIX Threads Library*), що спирається на нові функціональні можливості ядра
  - Як процес у системі реєструють лише перший потік застосування
  - Усі потоки процесу повертають один і той самий PID
  - Зв’язок предок-нащадок між потоками не підтримується
  - Потік-менеджер не потрібен
  - Зняті обмеження на кількість потоків у системі
- У більшості UNIX-систем, на відміну від Linux, реалізована повна підтримка POSIX Threads

Багатопотоковість у Solaris

- Модель багатопотоковості – 1:1 (починаючи з Solaris 8)
- Кожен потік користувача відповідає одному окремому потоку ядра (Kernel Thread, kthread)
- Фактично планування здійснюється для потоків ядра – це найголовніша відмінність від традиційної системи UNIX
- Кожен потік ядра в процесі повинен мати власний “легкий” процес (Lightweight process, LWT) – віртуальне середовище виконання, що має власний стек



Планування в UNIX SVR4

- Поняття “потік” відсутнє, планування здійснюється для процесів
- Реалізована витісняльна багатозадачність, що заснована на пріоритетах і квантуванні
- Визначені три пріоритетні класи (кожний процес належить до одного з класів):
  - Реального часу
    - Фіксовані пріоритети, але користувач може їх змінювати
    - Характеристики: рівень глобального пріоритету і квант часу
    - За наявності готових до виконання процесів реального часу інші процеси взагалі не розглядаються
  - Системних процесів
    - Зарезервований для ядра системи
    - Рівень пріоритету призначується ядром і ніколи не змінюється
  - Розподілу часу
    - Цей клас призначається новому процесу за умовчанням
    - Пріоритет обчислюється з двох складових: користувацької і системної
    - Користувацьку частину можуть змінювати адміністратор (в обидва боки) і власник процесу (лише у бік зниження пріоритету)
    - Системну частину змінює планувальник: знижує пріоритет процесам, що не уходять в стан очікування, підвищує пріоритет процесам, що часто уходять в стан очікування

Планування в Solaris

- Порівняно з SVR4 введені додаткові пріоритетні класи
  - System (SYS) – лише для потоків ядра
  - Realtime (RT) – клас з найвищим пріоритетом (вищим за SYS)
  - Timeshare (TS) – традиційний клас розподілу часу
  - Interactive (IA) – “інтерактивний” клас, призначений для потоків, що пов'язані з віконною системою
    - Передбачене підвищення пріоритету вікна, що знаходиться у фокусі
    - Той самий діапазон пріоритетів, і та ж таблиця диспетчеризації, що й для TS
  - Fair Share Scheduler (FSS) – клас, що контролюється не динамічним пріоритетом, а виділеними й наявними ресурсами ЦП
    - Ресурси ЦП поділяються на частки (shares), які виділяє адміністратор
    - Той же діапазон пріоритетів, що й для TS/IA
  - Fixed Priority (FX) – клас, для якого ядро не змінює пріоритет
- Ядро завантажує планувальники як модулі, паралельно можуть діяти різні планувальники
- Ядро підтримує окремі черги (фактично, черги черг) для кожного процесора
  - Черги впорядковуються за пріоритетами потоків
  - Зайнятість черг подається у вигляді бітової маски

- Для потоків реального часу підтримується єдина системна черга (не поділяється на процесори, але може розподілятись на групи процесорів (processor sets), якщо такі призначені)
- Кожен потік має 2 пріоритети: глобальний, що виводиться з його пріоритетного класу, і успадкований пріоритет
- Здебільшого, потоки виконуються як потоки класів TS або IA
  - Потоки IA створюються, коли потік асоціюється з віконною системою
  - Потоки RT створюються явно
  - Потоки SYS – це потоки ядра
- Переривання мають пріоритет ще вищий, ніж потоки RT

Планування в Linux

- Як і в UNIX, планування здійснюється для процесів
- Визначені три групи процесів у системі:
  - Реального часу із плануванням за принципом FIFO
  - Реального часу із круговим плануванням
  - Звичайні
- Особливості планування процесів реального часу
  - Вони завжди мають пріоритет перед звичайними процесами
  - Процеси із плануванням за принципом FIFO або самі віддають процесор, або їх витісняють процеси реального часу з більшим пріоритетом
  - Процеси із круговим плануванням додатково витісняють по завершені кванту часу

Традиційний алгоритм планування в Linux (до 2.4)

- Процесорний час поділяється на *епохи*
- У кожній епосі процес має квант часу, який розраховують на початку епохи (*базовий квант*)
- Епоха закінчується, коли усі готові до виконання процеси вичерпали свої кванти
- Значення кванту може змінюватись системними викликами nice() і setpriority()
- Пріоритет буває фіксований (для процесів реального часу) і динамічний (для звичайних процесів)
- Динамічний пріоритет залежить від
  - Базового пріоритету (nice – задає величину, на якій ґрунтується базовий квант процесу)
  - Часу, що залишився до вичерпання кванту (counter – кількість переривань таймера, на початку епохи йому надають значення базового кванту і зменшують на одиницю в обробнику переривань таймера)

Планування в Linux з ядром 2.6

- Недоліки традиційного алгоритму
  - Значний час на розрахунки на початку кожної епохи
  - Значний час на обрання процесу (потрібний розрахунок динамічного пріоритету)
  - Одна черга процесів – погано пристосовується для багатопроцесорних систем
- Новий підхід
  - На кожний процесор – своя черга
  - Кожна черга готових процесів – це масив черг готових процесів, де елементи упорядковані за динамічним пріоритетом
  - Процеси, що вичерпали квант, переносять в інший масив – масив черг процесів, що вичерпали квант
  - По завершенні епохи масиви міняють місцями

Керування процесами у Windows

- Адресний простір процесу складається з набору адрес віртуальної пам'яті, які цей процес може використовувати
  - Адреси можуть бути пов'язані з оперативною пам'яттю, а можуть – з відображеними у пам’ять ресурсами
  - Адресний простір процесу недоступний іншим процесам
- Процес володіє системними ресурсами, такими як файли, мережні з'єднання, пристрої введення-виведення, об’єкти синхронізації
- Процес не має прямого доступу до процесора
- Процес містить стартову інформацію для потоків, які у ньому створюються
- Процес обов'язково має містити хоча б один потік

Структури даних процесу у Windows

- Доступні лише у привілейованому режимі:
  - Для виконавчої системи – об'єкт-процес виконавчої системи (*Executive Process Block*, *EPROCESS*)
    - KPROCESS
    - Ідентифікаційна інформація (PID, PPID, ім'я файлу)
    - Інформація про адресний простір процесу
    - Інформація про ресурси, доступні процесу
    - PEB
    - Інформація для підсистеми безпеки
  - Для ядра – об'єкт-процес ядра (Kernel Process Block, KPROCESS)
    - Покажчик на ланцюжок блоків потоків ядра
    - Інформація, необхідна ядру для планування
- Доступна у режимі користувача:
  - Блок оточення процесу (*Process Environment Block*, *PEB*)
    - Початкова адреса ділянки пам'яті, куди завантажився програмний файл
    - Покажчик на динамічну ділянку пам'яті, що доступна процесу

Керування потоками у Windows

- Елементи потоку:
  - Вміст набору регістрів, який визначає стан процесора
  - Два стеки (для режиму ядра і режиму користувача), що розміщені в адресному просторі процесу
  - Локальна пам’ять потоку (TLS)
  - Унікальний ідентифікатор потоку TID
- Розрізняють потоки ядра і потоки користувача

Структури даних потоку у Windows

- Доступні лише у привілейованому режимі:
  - Для виконавчої системи – об'єкт-потік виконавчої системи (*Executive Tread Block*, *ETHREAD*)
    - KTHREAD
    - Ідентифікаційна інформація (*PID*, покажчик на *EPROCESS*)
    - Стартова адреса потоку
    - Інформація для підсистеми безпеки
  - Для ядра – об'єкт-потік ядра (*Kernel Thread Block*, *KTHREAD*)
    - Покажчик на стек режиму ядра
    - Покажчик на *TEB*
    - Інформація, необхідна ядру для планування
    - Інформація, необхідна для синхронізації потоку
- Доступна у режимі користувача:
  - Блок оточення потоку (*Thread Environment Block*, *TEB*)
    - Ідентифікатор потоку *TID*
    - Покажчик на стек режиму користувача
    - Покажчик на *PEB*
    - Покажчик на локальну пам’ять потоку

Планування у Windows

- Планування здійснюється виключно для потоків (ядро не розрізняє, яким процесам належать потоки)
- Під час планування ядро працює з блоками KTHREAD
- Пріоритети (від 1 до 31, динамічні – від 1 до 15):
  - Real-time ~ 24
  - High ~ 13
  - Normal ~ 8
  - Idle ~ 4
- Відносні пріоритети потоків від -2 до +2 від базового
- Кванти
  - Короткі кванти змінної довжини (10 або 30 мс) – перевага інтерактивних застосувань
  - Довгі кванти фіксованої довжини (120 мс) – перевага фонових процесів
- Список готових потоків складається з 31 елементу (відповідно до рівнів пріоритетів), з кожним з яких пов'язана черга
- Динамічна зміна пріоритету і кванту часу: *підтримка* (*boosting*) і *ослаблення* (*decay*), запобігання голодуванню

Взаємодія між процесами

- UNIX
  - Сигнали (аналог переривань)
    - Синхронні – під час виконання процесу, наприклад, через ділення на нуль або через помилку звернення до пам'яті
    - Асинхронні – повідомлення від іншого процесу або в результаті апаратної події
  - Диспозиція сигналів
    - Викликати обробник
    - Проігнорувати сигнал
    - Застосувати диспозицію за умовчанням
  - Сигнали можна блокувати!
- Windows
  - Повідомлення
  - Опитування черги повідомлень

Лекція 6. Керування процесами і потоками – синхронізація

План лекції

- Проблема синхронізації
- Гонки (змагання)
- Критична секція
- Атомарні операції
- Блокування, змінна блокування
- Семафори
- Задача виробник-споживач
- Взаємні блокування
- М'ютекси, умовні змінні, монітори

Проблема синхронізації: приклад

- Нехай 2 потоки T_A і T_B (різних користувачів) отримують доступ до деякого спільного ресурсу (банківський рахунок) з метою внесення змін
  - На рахунку 100 у.о.
  - Користувач А намагається зняти 50 у.о.
  - Користувач В намагається покласти ще 100 у.о.
- Алгоритм роботи потоків: **total_amount = total_amount + new_amount**;
  - Зчитати значення **total_amount (var1 = total_amount)**
  - Обчислити нове значення **total_amount (var1 = var1 + new_amount)**
  - Записати нове значення **total_amount (total_amount = var1)**
- Послідовність подій:
  - T_A зчитав **total_amount var1A = 100**
  - T_A обчислив **total_amount var1A = 100 – 50**
  - T_A записав **total_amount total_amount = 50**
  - T_B зчитав **total_amount var1B = 50**
  - T_B обчислив **total_amount var1B = 50 + 100**
  - T_B записав **total_amount total_amount = 150**

Все вірно! Але результат може бути іншим

- Послідовність подій 2:
  - T_A зчитав **total_amount** **var1A = 100**
  - T_A обчислив **total_amount** **var1A = 100 – 50**
  - T_B зчитав **total_amount** **var1B = 100**
  - T_B обчислив **total_amount** **var1B = 100 + 100**
  - T_B записав **total_amount** **total_amount = 200**
  - T_A записав **total_amount** **total_amount = 50**

В результаті **total_amount == 50** (втрачена покладена сума)

- Послідовність подій 3:
  - T_A зчитав **total_amount** **var1A = 100**
  - T_B зчитав **total_amount** **var1B = 100**
  - T_B обчислив **total_amount** **var1B = 100 + 100**
  - T_A обчислив **total_amount** **var1A = 100 – 50**
  - T_A записав **total_amount** **total_amount = 50**
  - T_B записав **total_amount** **total_amount = 200**

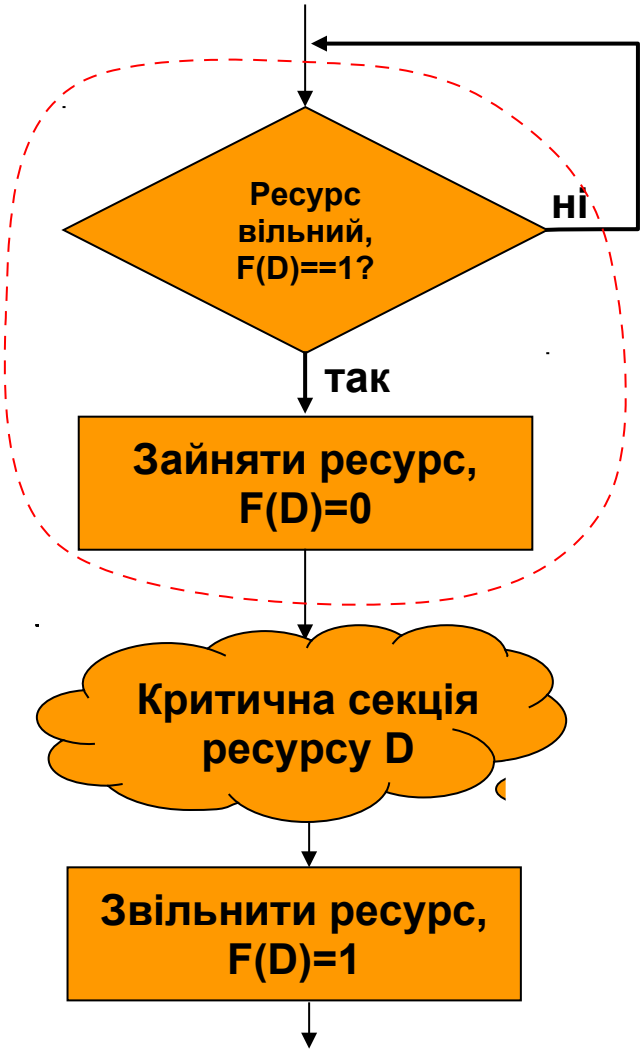
В результаті **total_amount == 200** (сума не знята)

Деякі означення

- Ситуація, коли 2 чи більше потоків обробляють спільні поділювані дані, і кінцевий результат залежить від співвідношення швидкостей потоків, називається *гонками* або *змаганням* (*race condition*)
- *Критична секція* (*critical section*) – частина програми, в якій здійснюється доступ до поділюваних даних або ресурсу
  - Щоби виключити гонки, у критичній секції, пов'язаній з певним ресурсом, повинно знаходитись не більше 1 потоку
- *Атомарна операція* – така послідовність дій, яка гарантовано виконується від початку до кінця без втручання інших потоків (тобто, є неподільною)
- Найпростіша реалізація – потік, що знаходиться у критичній секції, забороняє усі переривання
  - Це неприйнятно, оскільки внаслідок збою у критичній секції уся система може залишитись у непрацездатному стані

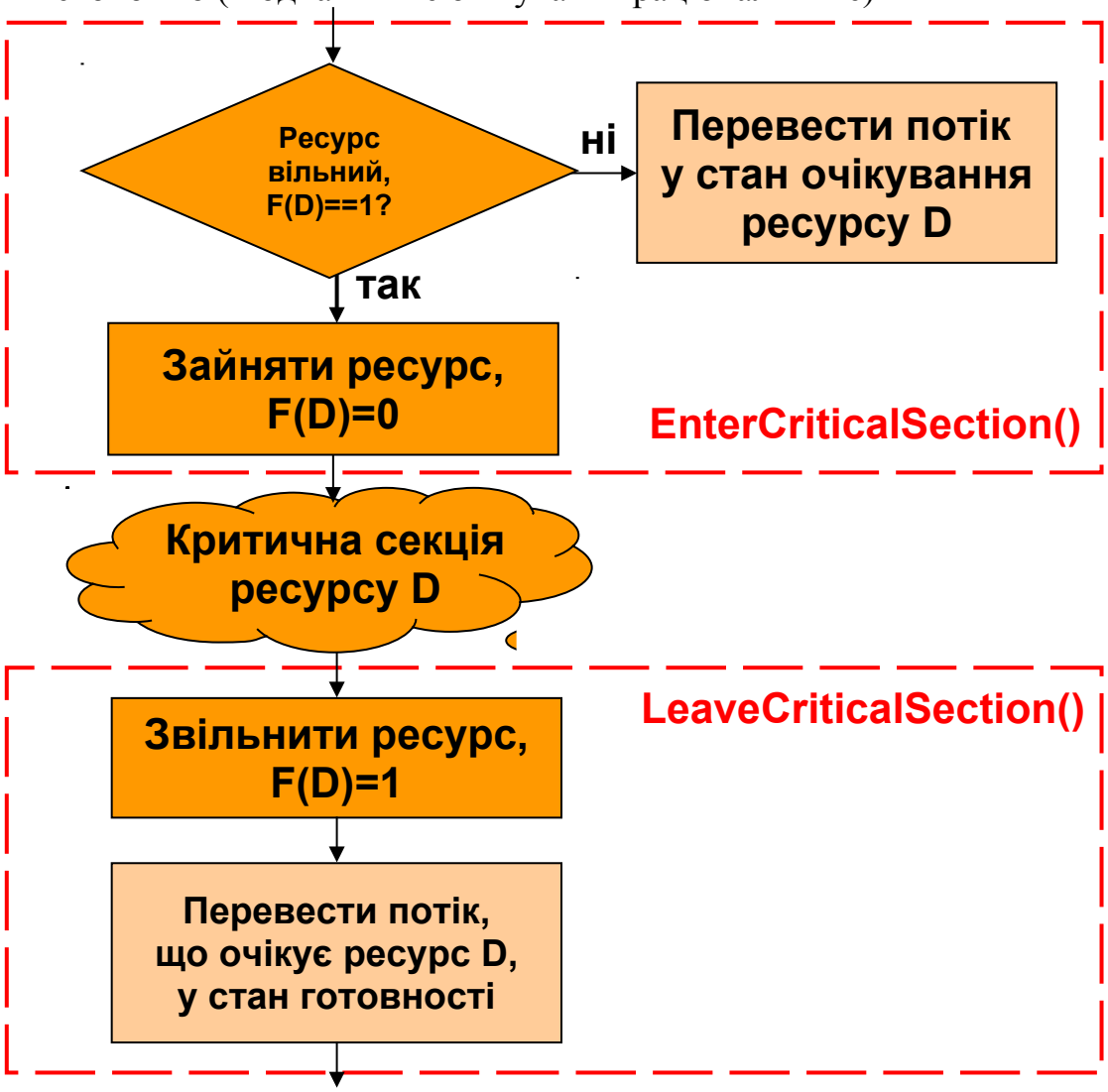
Блокування

- *Блокування* (*locks*) – це механізм, який не дозволяє більше як одному потокові виконувати код критичної секції
- Найпростіша (наївна) реалізація блокування – вводимо *змінну блокування* F(D) (1 – вільно, 0 – зайнято)
- Алгоритм *спін-блокування* (*spinlock*)
  - Здійснюємо опитування у циклі, доки не виявимо, що ресурс вільний – так зване *активне очікування* (*busy waiting*)
  - Встановлюємо відповідне значення змінної блокування і займаємо ресурс
- Проблема реалізації – операція перевірка-встановлення має бути атомарною (необхідна апаратна підтримка)
- Суттєвий недолік алгоритму – нераціональні витрати процесорного часу



Апарат подій для роботи з критичними секціями

- Функція WAIT(D) переводить потік у стан очікування події звільнення ресурсу D
- Функція POST(D) викликається після звільнення ресурсу D і переводить усі потоки, що його очікують, у стан готовності (планувальник обере один з них на виконання)
- У POSIX – виклики sleep() і wakeup()
- Проблема: накладні витрати ОС на здійснення викликів можуть перевищити економію (іноді активне очікування раціональніше)



Семафори

- *Семафор* (*semaphore*) – цілочисловий невід'ємний лічильник (позначають S)
- Для семафора визначені дві атомарні операції

```
V(S) {
   S++;
   if (waiting_threads()) POST(S);
}
P(S) {
   if (S > 0) S--;
   else WAIT(S);
}
```

- Окремий випадок: якщо семафор може приймати лише значення 0 і 1 (*двійковий семафор*), він фактично є змінною блокування
- Семафор – універсальний засіб, що забезпечує як взаємне виключення, так і очікування події

Задача виробник-споживач

- *Потік-виробник* створює об’єкти і поміщає їх у буфер. Операція розміщення об'єкта не є атомарною
- *Потік-споживач* отримує об’єкти і видаляє їх з буфера. Операція видалення об'єкта не є атомарною
- Буфер має фіксовану довжину N
- Вимоги:
  - Коли виробник або споживач працює з буфером, усі інші потоки повинні чекати завершення цієї операції
  - Коли виробник має намір помістити об'єкт у буфер, а буфер повний, він має чекати, поки в буфері звільниться місце
  - Коли споживач має намір отримати об'єкт з буфера, а буфер порожній, він має чекати, поки в буфері з'явиться об'єкт

Рішення задачі виробник-споживач

- Організуємо критичну секцію для роботи з буфером. Для цього використаємо двійковий семафор **lock**
- Для організації очікування виробника впровадимо семафор **empty**, значення якого дорівнює кількості вільних місць у буфері
  - Виробник перед спробою додати об'єкт у буфер зменшує цей семафор, а якщо той дорівнював 0 – переходить у стан очікування
  - Споживач після того, як забере об'єкт з буфера, збільшує цей семафор, при цьому, можливо, ініціюється пробудження виробника
- Для організації очікування споживача впровадимо семафор **full**, значення якого дорівнює кількості зайнятих місць у буфері
  - Споживач перед спробою забрати об'єкт з буфера зменшує цей семафор, а якщо той дорівнював 0 – переходить у стан очікування
  - Виробник після того, як додасть об'єкт до буфера, збільшує цей семафор, при цьому, можливо, ініціюється пробудження споживача

Псевдокод рішення задачі виробник-споживач

```
semaphore lock = 1, empty = n, full = 0;      // на початку буфер порожній

void producer() {
    while(1) {                          // працює постійно
        item = produce_new_item();      // створюємо об'єкт
        P(empty);                       // перевіряємо наявність місця
        P(lock);                        // входимо у критичну секцію
        add_to_buffer(item);            // додаємо об'єкт до буфера
        V(lock);                        // виходимо з критичної секції
        V(full);                        // повідомляємо про новий об'єкт
    }
}

void consumer() {
    while(1) {                          // працює постійно
        P(full);                        // перевіряємо наявність об'єкта
        P(lock);                        // входимо у критичну секцію
        item = get_from_buffer();       // забираємо об'єкт з буфера
        V(lock);                        // виходимо з критичної секції
        V(empty);                       // повідомляємо про вільне місце
        consume_new_item(item);         // споживаємо об'єкт
    }
}
```

Проблема взаємних блокувань

- Припустимо, у функції "виробник" ми поміняли місцями **P(empty)** і **P(lock)**:

```
void producer() {
    while(1) {                          // працює постійно
       item = produce_new_item();       // створюємо об'єкт
       P(lock);                         // входимо у критичну секцію
       P(empty);                        // перевіряємо наявність місця
       add_to_buffer(item);             // додаємо об'єкт до буфера
       V(lock);                         // виходимо з критичної секції
       V(full);                         // повідомляємо про новий об'єкт
    }
}
```

- Перевірка умови з можливим очікуванням здійснюється *всередині* критичної секції. Можлива така послідовність дій:
  - Виробник входить у критичну секцію, закриваючи семафор **lock**
  - Виробник перевіряє семафор **empty** і очікує на ньому (буфер повний)
  - Споживач намагається ввійти у критичну секцію і блокується на семафорі **lock**
- Така ситуація називається *взаємне блокування* або *тупик* (*deadlock*)

Типові причини виникнення взаємних блокувань

- Необережне застосування семафорів. Найчастіше – через застосування семафору всередині критичної секції
  - див. приклад вище
- Необхідність різним потокам одночасно захоплювати кілька ресурсів
  - "Філософи, що обідають"
  - Реальна ситуація (різна послідовність захоплення ресурсів):

Шляхи вирішення проблеми взаємних блокувань

- Запобігання взаємних блокувань
  - Запит ресурсів здійснюється у певній послідовності, спільній для усіх потоків
  - Якщо один з потрібних ресурсів зайнятий, то потік має звільнити усі ресурси, що йому необхідні одночасно, і повторити спробу через деякий час
  - Потік запитує усі ресурси у центрального диспетчера і очікує їх виділення.
- Розпізнання взаємних блокувань
  - Існують формальні методи, які вимагають ведення таблиць розподілу ресурсів і запитів до них
- Відновлення після взаємних блокувань
  - Аварійне завершення усіх або деяких заблокованих потоків
  - "Відкат" до контрольної точки

Спеціалізовані засоби синхронізації низького рівня – м’ютекс

- *М’ютекс* (*mutex* – від *mutual exclusion*) призначений для взаємного виключення
  - Має два стани: вільний і зайнятий
  - Визначені дві атомарні операції: зайняти і звільнити
  - На відміну від двійкового семафора, звільнити м’ютекс може лише той потік, що його зайняв (*власник м’ютекса*)
  - У деяких реалізаціях існує третя операція – спробувати зайняти м’ютекс
  - Повторна спроба власника м’ютекса зайняти той самий м’ютекс призводить до блокування
  - Існують рекурсивні м’ютекси, які діють за принципом семафора лише для свого власника

Спеціалізовані засоби синхронізації низького рівня – умовна змінна

- *Умовна змінна* призначена для очікування події
- Умовна змінна пов'язана з певним м’ютексом і даними, які він захищає
- Визначені три операції:
  - сигналізація (signal) – потік, що виконав дії з даними у критичній секції, перевіряє, чи не очікують на умовній змінній інші потоки, і якщо очікують – переводить один з них у стан готовності (потік буде поновлено після звільнення м’ютексу)
  - широкомовна сигналізація (broadcast) – те ж, що й сигналізація, але у стан готовності переводить усі потоки
  - очікування (wait) – викликається, коли потік у критичній секції не може продовжувати роботу через невиконання певної умови
    - м’ютекс звільняють і інші потоки можуть мати доступ до даних
    - після того, як інший потік здійснив виклик signal або broadcast, потік знову повертається до виконання
    - потік захоплює м’ютекс і продовжує роботу в критичній секції

    Операція очікування не атомарна

Особливості використання умовних змінних

- Перед викликом **wait()** необхідно перевіряти умову в циклі **while()**
  while(! condition_expr) // вираз для умови
  wait(condition, mutex);
- Умовні змінні використовуються лише всередині критичних секцій, на відміну від семафорів, використання яких всередині критичних секцій призводить до блокування
- Умовні змінні не зберігають стану, на відміну від семафорів, які зберігають стан
- Рекурсивні м'ютекси не можуть бути використані з умовними змінними, оскільки рекурсивний м'ютекс може не звільнитися разом із викликом **wait()** (гарантоване взаємне блокування)

Монітор

- *Монітор* – це набір функцій, які використовують один загальний м'ютекс і нуль або більше умовних змінних для керування паралельним доступом до спільно використовуваних даних
- Правила реалізації монітора:
  - Під час входу в кожну функцію монітора слід займати м'ютекс, під час виходу – звільняти
  - Під час роботи з умовною змінною необхідно завжди вказувати відповідний м'ютекс (для роботи з умовною змінною м'ютекс повинен завжди бути зайнятий)
  - Під час перевірки на виконання умови очікування необхідно застосовувати цикл, а не умовний оператор

Лекція 7. Керування оперативною пам'яттю

План лекції

- Завдання керування пам'яттю
- Типи адрес
- Пласка і сегментна моделі пам'яті
- Методи розподілу пам'яті
- Розподіл пам'яті без застосування дискового простору
- Оверлеї
- Свопінг
- Віртуальна пам'ять

Завдання керування пам'яттю

- Відстеження вільної та зайнятої пам'яті
- Виділення пам'яті процесам і звільнення пам'яті після завершення процесу
- Витіснення процесів з оперативної пам'яті на диск і повернення їх в оперативну пам'ять (віртуальна пам’ять)
- Перетворення адрес

Типи адрес

- *Символьні адреси* (ідентифікатори змінних, мітки переходів у програмах на алгоритмічних мовах)
  - Транслятор
- Віртуальні адреси (умовні адреси)
  - Переміщуючий завантажувач (статичне перетворення)
  - Динамічне перетворення апаратними засобами
- *Фізичні адреси* (номери комірок фізичної пам'яті)

- Сукупність віртуальних адрес процесу називається *віртуальним адресним простором* (у загальному випадку не дорівнює обсягу фізичної пам'яті)

Моделі пам'яті

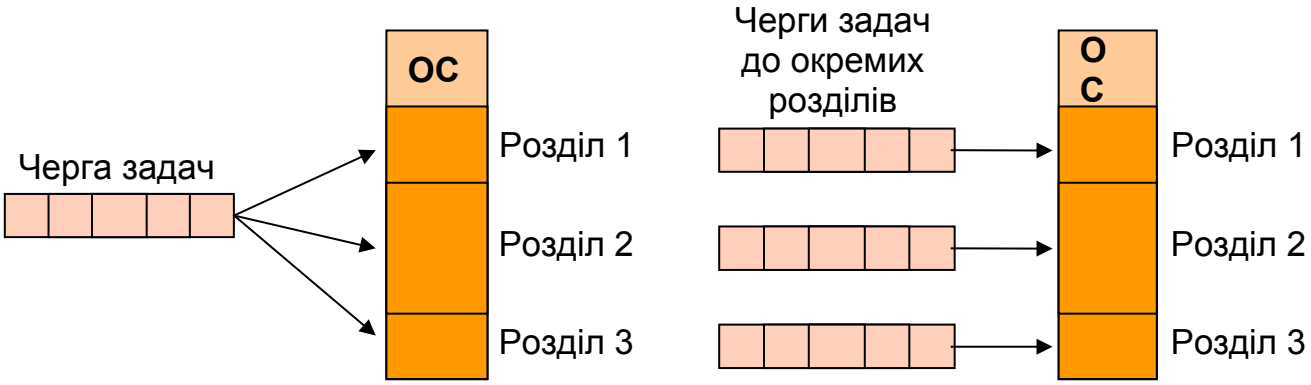
- Для забезпечення коректної адресації незалежно від розташування програми в оперативній пам’яті комп’ютера в якості віртуальних адрес використовуються *відносні адреси*, тобто зміщення від деякої *базової адреси*
- *Пласка* (*flat*) *модель пам’яті*
  - Кожному процесу виділяється єдина неперервна послідовність віртуальних адрес
  - Зміщення дозволяє однозначно вказати на положення даних або команди в адресному просторі процесу
- *Сегментна модель пам’яті*
  - Адресний простір процесу поділяється на окремі частини, які називаються *сегментами* (зустрічаються також інші назви: *секції*, *області*)
  - Віртуальна адреса задається парою чисел (*n*, *m*), де *n* визначає сегмент, а *m* – зміщення в даному сегменті
  - Сегментна модель є більш складною, але й більш гнучкою

Методи розподілу пам'яті

- Без застосування дискового простору
  - *Фіксовані розділи*
  - *Динамічні розділи* (розділи змінної величини)
  - *Переміщувані розділи*
- Із застосуванням дискового простору (віртуальна пам’ять)
  - *Сегментний розподіл*
  - *Сторінковий розподіл*
  - *Сегментно-сторінковий розподіл*

Фіксовані розділи

- Вибір розділу, що підходить за розміром
- Завантаження програми і налаштування адрес

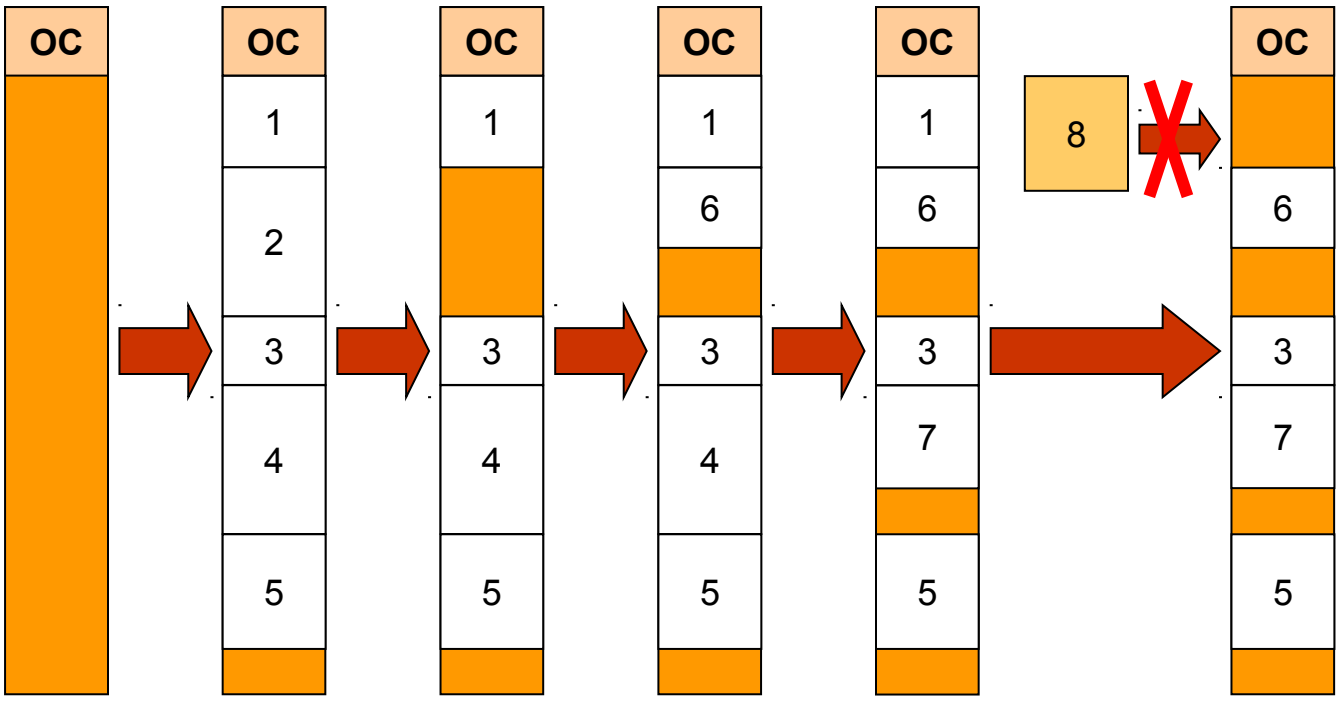


Динамічні розділи

- Завдання ОС:
  - Ведення таблиць вільних і зайнятих областей (стартові адреси і розміри ділянок пам'яті)
  - Під час надходження нової задачі – аналіз запиту, перегляд таблиці вільних областей і вибір розділу за одним з алгоритмів:
    - Перший знайдений розділ достатнього розміру
    - Найменший розділ достатнього розміру
    - Найбільший розділ (достатнього розміру)
  - Завантаження задачі у виділений розділ і коригування таблиць вільних і зайнятих областей
  - По завершенні задачі – коригування таблиць вільних і зайнятих областей

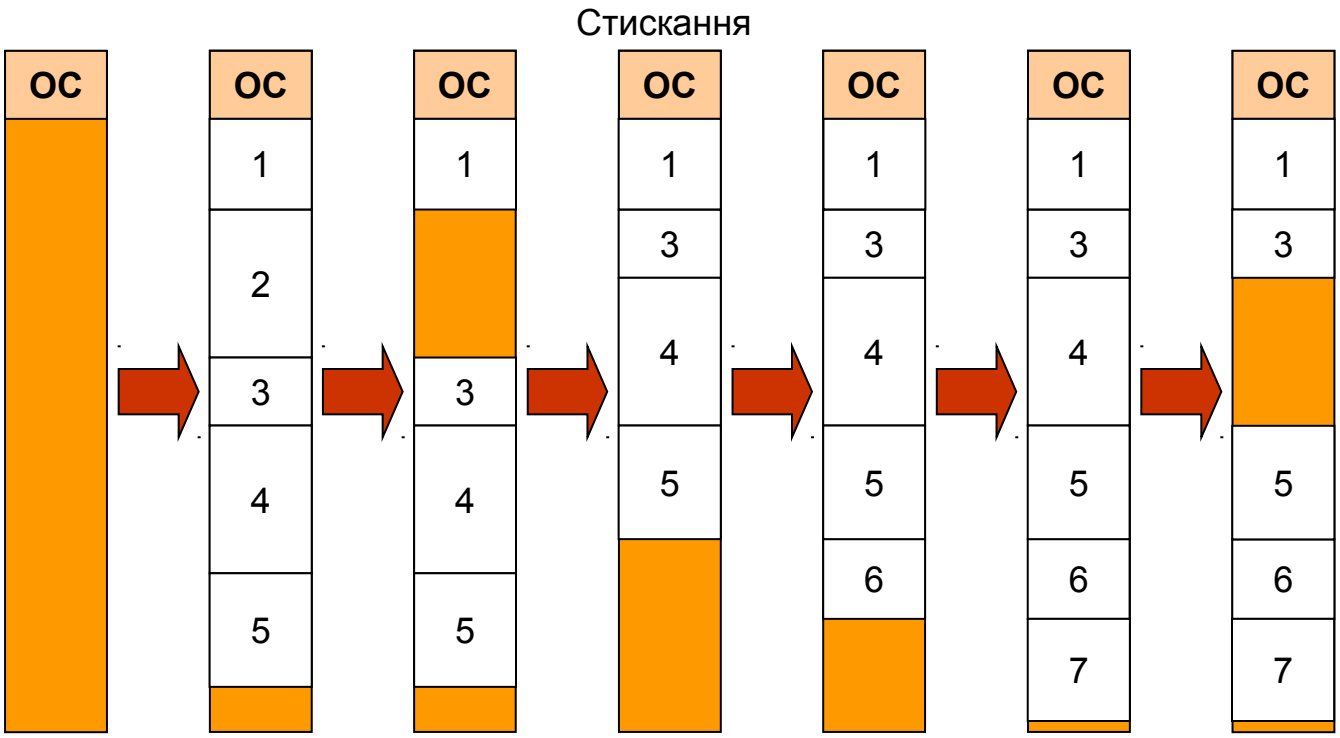
+ Перевага
  - У процесі виконання програмний код не переміщується – можна налаштовувати адреси одноразово

– Недолік
  - *Фрагментація!!!*



Переміщувані розділи

- Те ж саме, що й динамічні розділи, плюс:
- Система періодично усуває фрагментацію пам'яті шляхом переміщення усіх розділів у бік більших (або менших) адрес
  - Процедура називається стискання пам'яті
  - Може виконуватись:
    - Або завжди, коли завершується задача
    - Або лише тоді, коли для нового розділу не вистачає пам'яті
- – Недолік:
  - Необхідно динамічне перетворення адрес

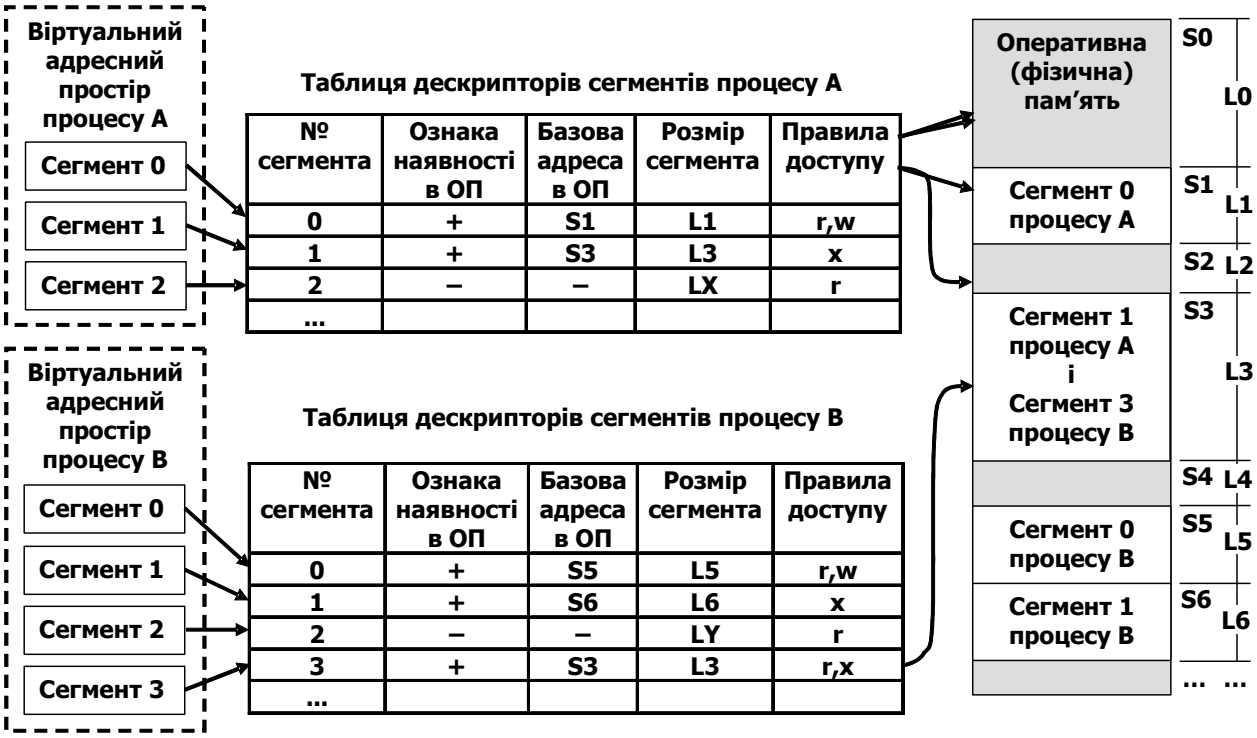


Розподіл пам'яті з використанням дискового простору

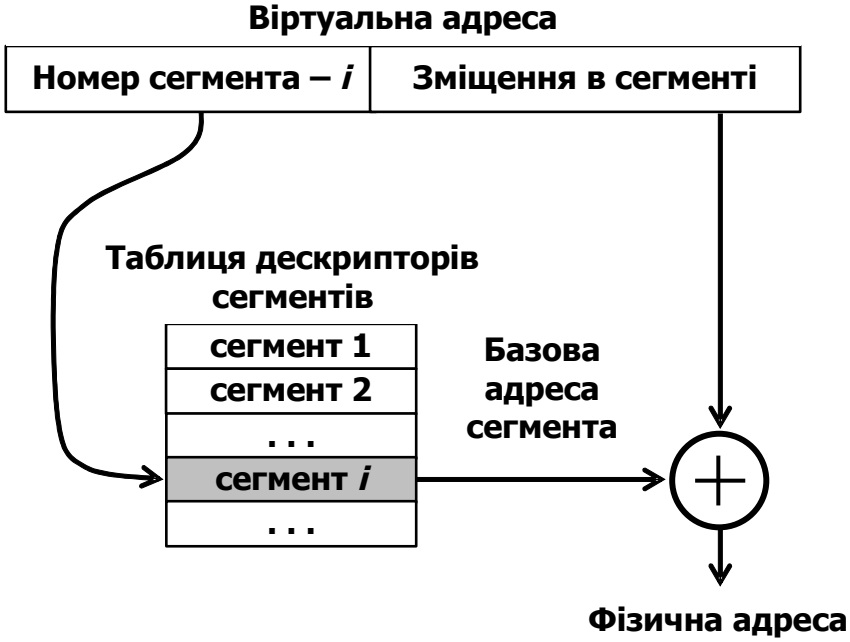
- *Оверлей (Overlay)*
  - У процесі виконання програми окремі програмні модулі завантажуються з диску
  - Реалізується засобами прикладних програм
- *Свопінг (Swapping)*
  - Процеси у стані очікування повністю вивантажуються на диск
- *Віртуальна пам'ять (Virtual Memory)*
  - Сукупність програмно-апаратних засобів, що дозволяє процесам використовувати більший обсяг пам'яті, ніж є наявної оперативної пам'яті
    - Розміщення коду і даних у пристроях пам'яті різного типу
    - Переміщення коду і даних між пристроями пам'яті різного типу
    - Перетворення віртуальних адрес у фізичні

Сегментний розподіл пам'яті

- *Сегмент* – це неперервна область віртуального адресного простору довільного розміру, виділена з урахуванням типу даних, які в ній знаходяться
- Віртуальний адресний простір процесу складається з окремих сегментів, розмір кожного з яких обмежується розрядністю адресації
  - При 16-розрядній адресації – до 64 кБ
  - При 32-розрядній адресації – до 4 ГБ
- Відомості про сегменти оформлюються у вигляді таблиці, кожний рядок якої містить інформацію про окремий сегмент (*дескриптор сегмента*):
  - Базова фізична адреса процесу в оперативній пам'яті
  - Розмір сегмента
  - Тип сегмента
  - Правила доступу до сегмента
  - Ознака наявності сегмента в оперативній пам'яті
  - Ознака модифікації сегмента
  - Інші відомості
- Сегментний розподіл передбачає, що деякі сегменти можуть бути повністю витіснені на диск

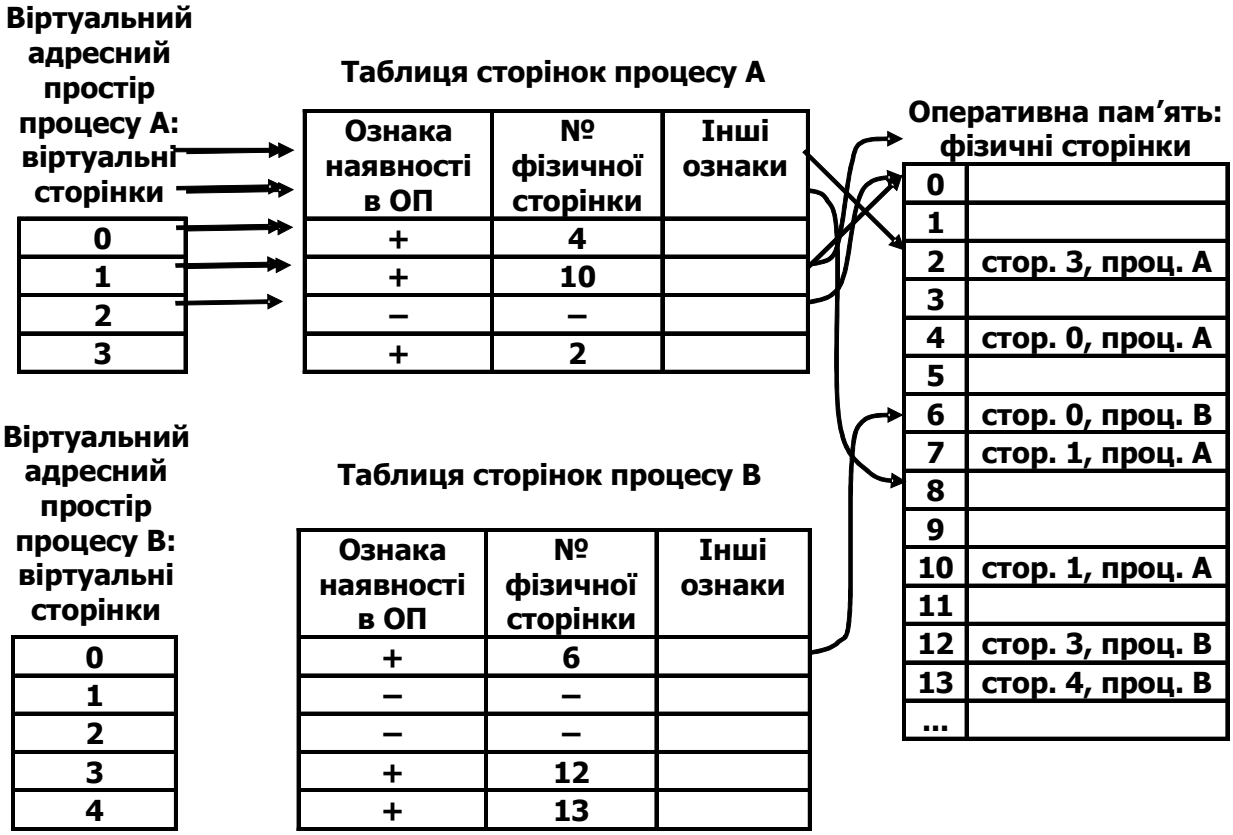


Трансляція віртуальної адреси при сегментній організації пам'яті

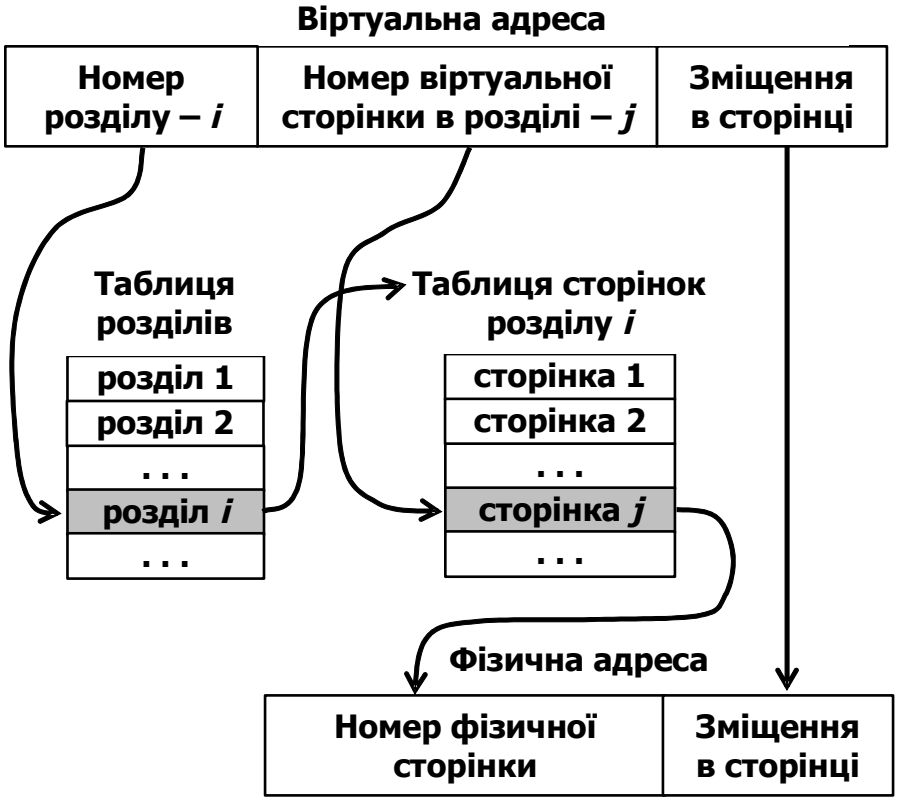


Сторінковий розподіл пам'яті

- *Сторінка* – це неперервна область віртуального адресного простору порівняно невеликого фіксованого розміру, виділена без урахування типу даних, які в ній знаходяться
- Для сторінок, як і для сегментів, застосовуються дескриптори, але структура їх значно простіша:
  - Номер фізичної сторінки в оперативній пам’яті, в яку завантажена ця віртуальна сторінка
  - Ознака наявності в оперативній пам’яті
  - Ознаку модифікації сторінки (якщо модифікації не було, в разі необхідності звільнити пам’ять сторінку можна просто “затерти”)
  - Ознаку звернення до сторінки (використовується для вибору сторінок-кандидатів для витіснення на диск)
- Типовий розмір сторінки – 4 кБ (величезна таблиця сторінок)
  - Віртуальний адресний простір поділяють на розділи однакового розміру, який підбирають таким чином, щоби таблиця сторінок одного розділу займала рівно одну сторінку
  - Для кожного розділу формують свою таблицю сторінок
  - Таблиці витискаються на диск разом із відповідними розділами
  - Дескриптори таблиць сторінок аналогічні дескрипторам звичайних сторінок, вони формують окрему таблицю, яку називають *таблицею розділів* або *каталогом сторінок*.



Трансляція віртуальної адреси при дворівневій сторінковій організації пам’яті



Завантаження-вивантаження сторінок

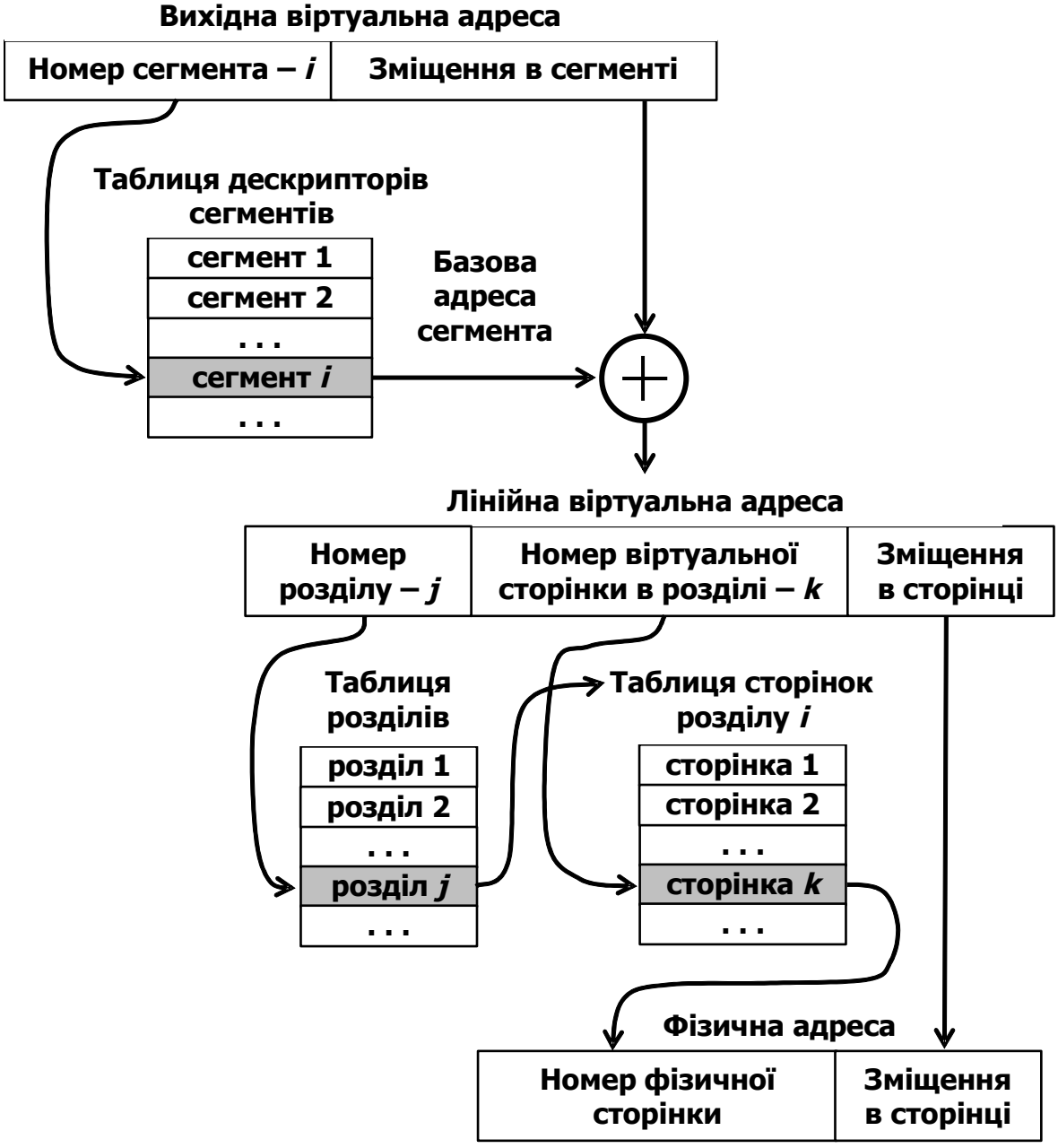
- Під час кожного звернення до пам'яті здійснюється зчитування інформації про віртуальну сторінку з таблиці сторінок
- Якщо сторінка є в пам'яті – здійснюється перетворення віртуальної адресу у фізичну
- Якщо сторінки немає – здійснюється така послідовність дій:
  - Сторінкове переривання
  - Процес переводять у стан очікування
  - Обробник сторінкового переривання знаходить сторінку на диску і намагається завантажити її у пам’ять
  - Якщо вільне місце є, сторінка завантажується у пам’ять
  - Якщо місця немає – здійснюється вибір сторінки, яку необхідно вивантажити з пам'яті
    - Перша знайдена сторінка
    - Сторінка, що довше за усіх не використовувалась
    - Сторінка, звернень до якої було менше за усіх
  - Якщо обрану сторінку модифікували – її записують на диск
  - Якщо обрану сторінку не модифікували – її просто видаляють з пам'яті

Сегментний і сторінковий розподіли пам’яті: переваги і недоліки

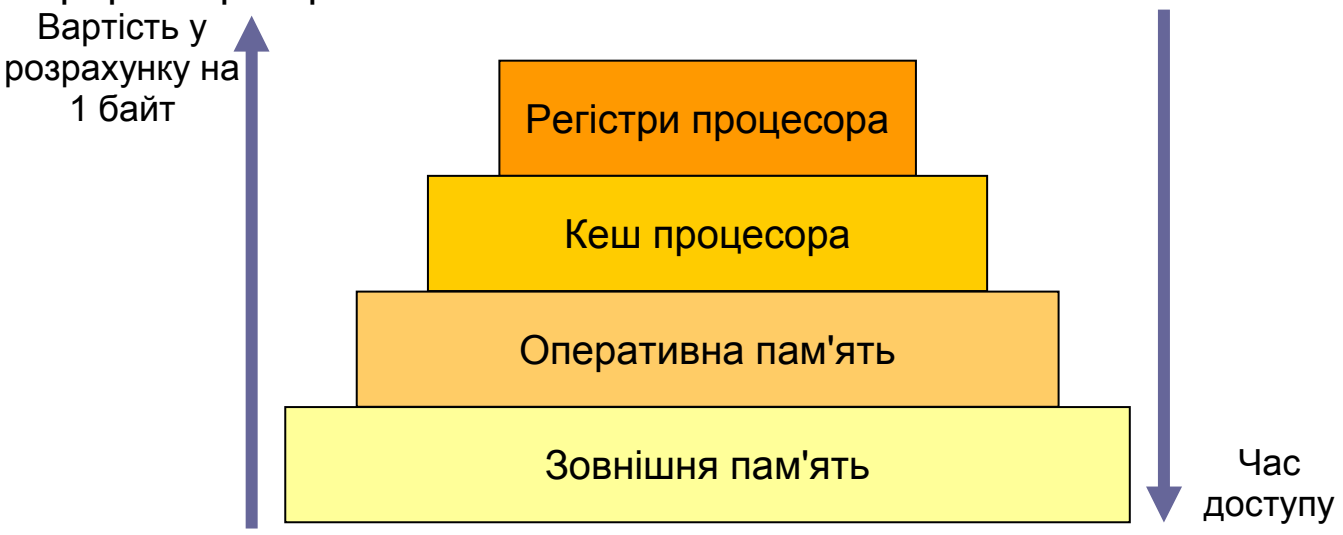
- Перевага сегментів – в їх типізації
  - Дозволяє здійснювати диференційоване керування доступом у відповідності до типу даних, що містяться у сегменті:
    - Заборона записування у сегмент, де містяться коди програми
    - Заборона виконання процесором фрагментів програмного коду, що містяться у сегменті даних
  - Сегментний розподіл є основою для реалізації захисту областей пам’яті
- Перевага сторінок – в однаковому і невеликому розмірі
  - Легше і швидше завантажити і вивантажити певну кількість сторінок однакового розміру, ніж один великий сегмент
  - Сторінковий розподіл переважно застосовується для реалізації механізму обміну інформацією між фізичною пам’яттю і диском.

Сегментно-сторінковий розподіл пам’яті

- Спочатку здійснюється сегментне перетворення адреси, а далі – сторінкове



Ієрархія пристроїв пам'яті



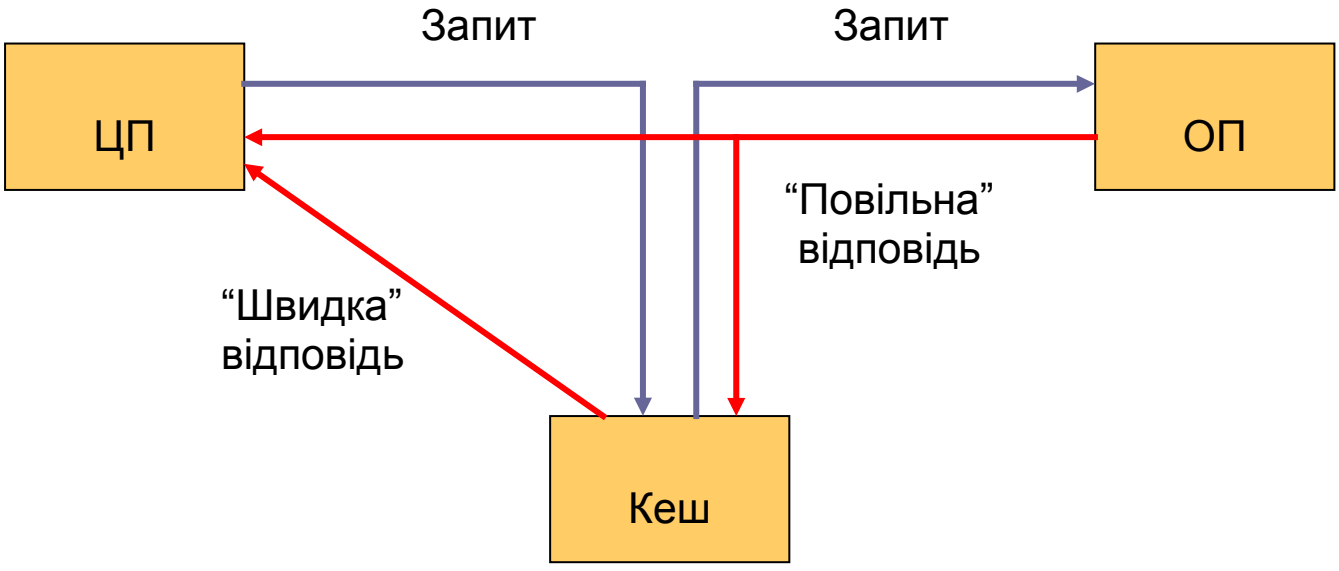
Кеш-пам'ять

- *Кеш-пам'ять* – це спосіб організації сумісного функціонування 2-х типів пристроїв пам'яті, що дозволяє знизити середній час доступу до даних за рахунок копіювання у "швидкий" пристрій частини даних з "повільного" пристрою
- Іноді кеш-пам'яттю називають не лише спосіб, але й сам швидкий пристрій
- Дані у кеш-пам'яті зберігаються прозоро (немає власної адресації, застосовуються адреси з повільного пристрою)
- Середній час доступу:

$$t = t_{\text{повільн}} (1 - p) + t_{\text{швидк}}\, p$$

p – ймовірність потрапляння в кеш (велика! ~0,9)

Схема функціонування



Локальність даних

- *Часова локальність*
  - Якщо відбулося звернення до пам'яті за певною адресою, то з великою ймовірністю найближчим часом відбудеться повторне звернення за тією ж адресою
  - Обґрунтовує ефективність копіювання даних в кеш під час зчитування
- *Просторова локальність*
  - Якщо відбулося звернення до пам'яті за певною адресою, то з великою ймовірністю наступне звернення відбудеться за сусідньою адресою
  - Обґрунтовує ефективність випереджаючого зчитування даних в кеш

Лекція 8. Керування оперативною пам'яттю у процесорах архітектури x86

План лекції

- Системні таблиці і регістри системних адрес
- Селектор і дескриптор сегмента
- Захист сегментів
- Завантаження селектора у сегментний регістр
- Звернення до пам’яті
- Сторінковий механізм керування пам'яттю

Керування пам'яттю у процесорах архітектури x86

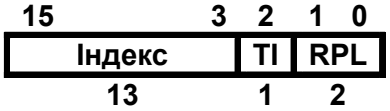
- У захищеному режимі підтримується сегментний і сегментно-сторінковий розподіл пам'яті
- Сторінкове перетворення включається або відключається окремим прапорцем у регістрі керування процесором **cr0**
- Максимальний розмір сегмента – 4 ГБ
- У 16-розрядні сегментні регістри процесора завантажуються *селектори*, які вказують на *дескриптори* сегментів
  - **cs** – сегмент коду
  - **ss** – сегмент стека
  - **ds**, **es**, **fs**, **gs** – сегменти даних
- Дескриптори містяться у спеціальних системних таблицях, на які вказують *регістри системних адрес*
- У дескрипторах містяться:
  - базова адреса сегмента
  - межа (розмір) сегмента
  - правила доступу до сегмента

Системні таблиці і регістри системних адрес

- *Регістри системних адрес* містять покажчики на системні таблиці, призначені для керування пам’яттю та диспетчеризації процесів
- Доступ до сегментів у пам'яті здійснюється через *дескриптори*
- Дескриптори містяться у двох таблицях, доступних процесу
  - *Глобальна таблиця дескрипторів* (*Global Descriptor Table*, GDT)
    - Містить дескриптори, що описують програмний код і дані, спільні для усіх процесів (наприклад, бібліотеки, драйвери пристроїв, тощо), а також численні системні об'єкти
    - На цю таблицю вказує регістр **gdtr**
  - *Локальна таблиця дескрипторів* (*Local Descriptor Table*, LDT)
    - Доступна лише тому процесу, який виконується в даний момент
    - Кожний процес має свою власну локальну таблицю
    - На цю таблицю вказує регістр **ldtr**
- Обробники переривань доступні через таблицю дескрипторів переривань (*Interrupt Descriptor Table*, IDT)
  - На неї вказує регістр **idtr**
- *Контекст процесу* знаходиться у спеціальному системному об'єкті, що називається *сегментом стану задачі* (*Task Status Segment*, TSS)
  - Цей об'єкт описується дескриптором, на який вказує регістр **ts**
- Сегменти, що вказують на глобальні системні об'єкти (**gdtr** та **idtr**) містять базові лінійні адреси цих об'єктів, а також задають розмір об'єктів
- Сегменти, що вказують на локальні для кожного процесу об'єкти (**ldtr** та **tr**) містять лише селектори, які адресують відповідні дескриптори у глобальній таблиці
  - Таким чином, таблиця GDT містить дескриптори, що описують сегменти стану задач і локальні таблиці дескрипторів усіх процесів, що виконуються в системі
  - Для переходу до виконання іншого процесу необхідно просто завантажити у регістри **ldtr** та **tr** відповідні дескриптори

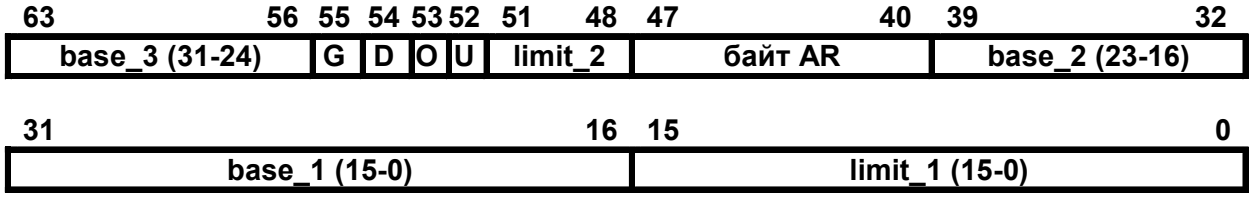
Регістр	**Таблиця (об'єкт)**		**Формат регістру**	
gdtr	GDT	Global Descriptor Table	48	Містить базову адресу і межу таблиці
ldtr	LDT	Local Descriptor Table	16	Селектор у таблиці GDT
idtr	IDT	Interrupt Descriptor Table	48	Містить базову адресу і межу таблиці
tr	TSS	Task Status Segment	16	Селектор у таблиці GDT

Селектор сегмента



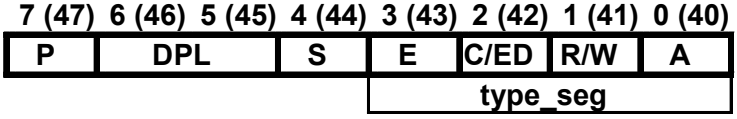
- *Селектор* – це 16-розрядна структура, яка завантажується в сегментні регістри
- Селектор адресує не сам сегмент, а його дескриптор
- 13 старших розрядів селектора є індексом в таблиці дескрипторів
  - Таким чином, кожна таблиця може містити 2^{13}=8192 дескрипторів
- Один розряд селектора (біт 2), який позначається як прапорець **TI**, вказує в якій з таблиць знаходиться дескриптор
  - **TI==0 →GDT**
  - **TI==1 →LDT**
- Останні 2 розряди селектора (біти 1,0) відведені для задавання рівня привілеїв (*Requested Privilege Level*, RPL), який використовується механізмом захисту

Дескриптор сегмента



- Дескриптор сегмента є 8-байтовою структурою
- Головні поля дескриптора:
  - 32-розрядна *базова адреса* (*base*)
  - 20-розрядна *межа сегмента* (*limit*) – визначає розмір сегмента в залежності від *прапорця гранулярності* **G**:
    - **G==0** →розмір у байтах (максимальний розмір сегмента 1 МБ)
    - **G==1** →розмір у 4-кБ сторінках (максимальний розмір сегмента – 4 ГБ)
  - *Байт захисту* (**AR**)
  - *Прапорець розрядності* (**D**):
    - **D==0** →16-розрядні операнди і режими 16-розрядної адресації
    - **D==1** →32-розрядні операнди і режими 32-розрядної адресації

Байт захисту



- Розряд 7 (**P** – *Present*) показує наявність сегмента у пам'яті
- Розряди 5 і 6 (**DPL** – *Descriptor Privilege Level*) визначають рівень привілеїв дескриптора
- Розряд 4 (**S** – *System / Segment*) визначає, чи є об’єкт, який описує цей дескриптор, сегментом у пам’яті чи спеціальним системним об’єктом
- Розряди 1 – 3 визначають тип сегмента і права доступу до нього
  - Розряд 3 (**E** – *Executable*)
  - Розряд 2
    - Для сегментів коду – біт підпорядкованості (**C** – *Conforming*)
    - Для сегментів даних – біт розширення вниз (**ED** – *Expand Down*)
  - Розряд 1
    - Для сегментів коду – дозвіл на зчитування (**R** – *Readable*)
    - Для сегментів даних – дозвіл на записування (**W** – *Writable*)
- Розряд 0 (**A** – *Accessed*) встановлюється при доступі до сегмента

Значення поля типу сегмента

Біт S	**Поле type_seg**	**Тип сегмента**
0	0100	Таблиця локальних дескрипторів (LDT)
0	0001 1000 1101 1101	Сегмент стану задачі (TSS)
1	000x	Сегмент даних, тільки для зчитування
1	001x	Сегмент даних, зчитування і записування
1	010x	Не визначено
1	011x	Сегмент стека, зчитування і записування
1	100x	Сегмент коду, тільки виконання
1	101x	Сегмент коду, зчитування і виконання
1	110x	Підпорядкований сегмент коду, тільки виконання
1	111x	Підпорядкований сегмент коду, дозволені зчитування і виконання

Завантаження селектора у сегментний регістр

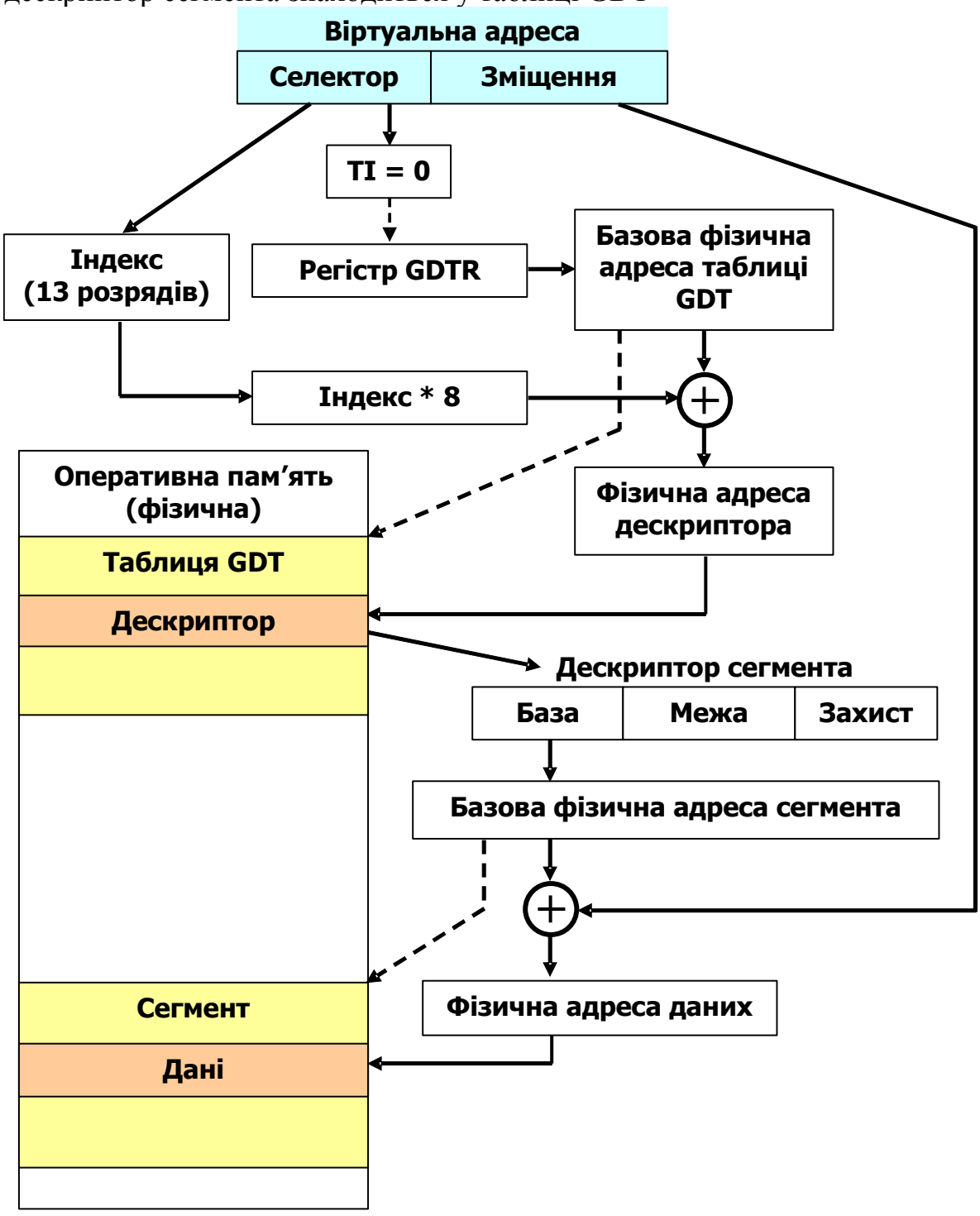
- Якщо у селекторі **TI == 0**, то дескриптор міститься в GDT
  - З регістру **gdtr** визначають базову адресу і розмір таблиці GDT
  - За індексом вибирають з GDT потрібний дескриптор
  - Перевіряють, чи не виходить дескриптор (з урахуванням його розміру – 8 байтів) за встановлену межу таблиці GDT. В разі виходу за межу – виняткова ситуація 11
  - Перевіряють сумісність типу дескриптора і достатність привілеїв для доступу до нього. Якщо щось не так – виняткова ситуація 13
- Якщо у селекторі **TI == 1**, то дескриптор міститься в LDT
  - З регістру **gdtr** визначають базову адресу і розмір таблиці GDT
  - З таблиці GDT вибирають дескриптор, що описує таблицю LDT, для чого в якості селектора використовують вміст регістру **ldtr**
  - Перевіряють, чи відповідає тип дескриптора таблиці дескрипторів і чи присутня таблиця у фізичній пам'яті
  - З дескриптора таблиці LDT визначають базову адресу таблиці та її межу
  - Далі здійснюють операцію вибору з таблиці необхідного дескриптора, яка аналогічна операції вибору дескриптора з таблиці GDT, з тими ж перевірками.

Дозволені комбінації бітів байту захисту дескриптора при завантаженні селектора у сегментний регістр

Регістр	**DPL**	**S**	**E**	**C/ED**	**R/W**	**Примітка**
cs	≥ RPL ≥ CPL	1	1	x	x	сегмент коду
ss	=RPL =CPL	1	0	1	1	сегмент стека
ds, es, fs, gs	≥RPL ≥CPL	1	1	x	1	сегмент коду
			0	x	x	сегмент даних або стека

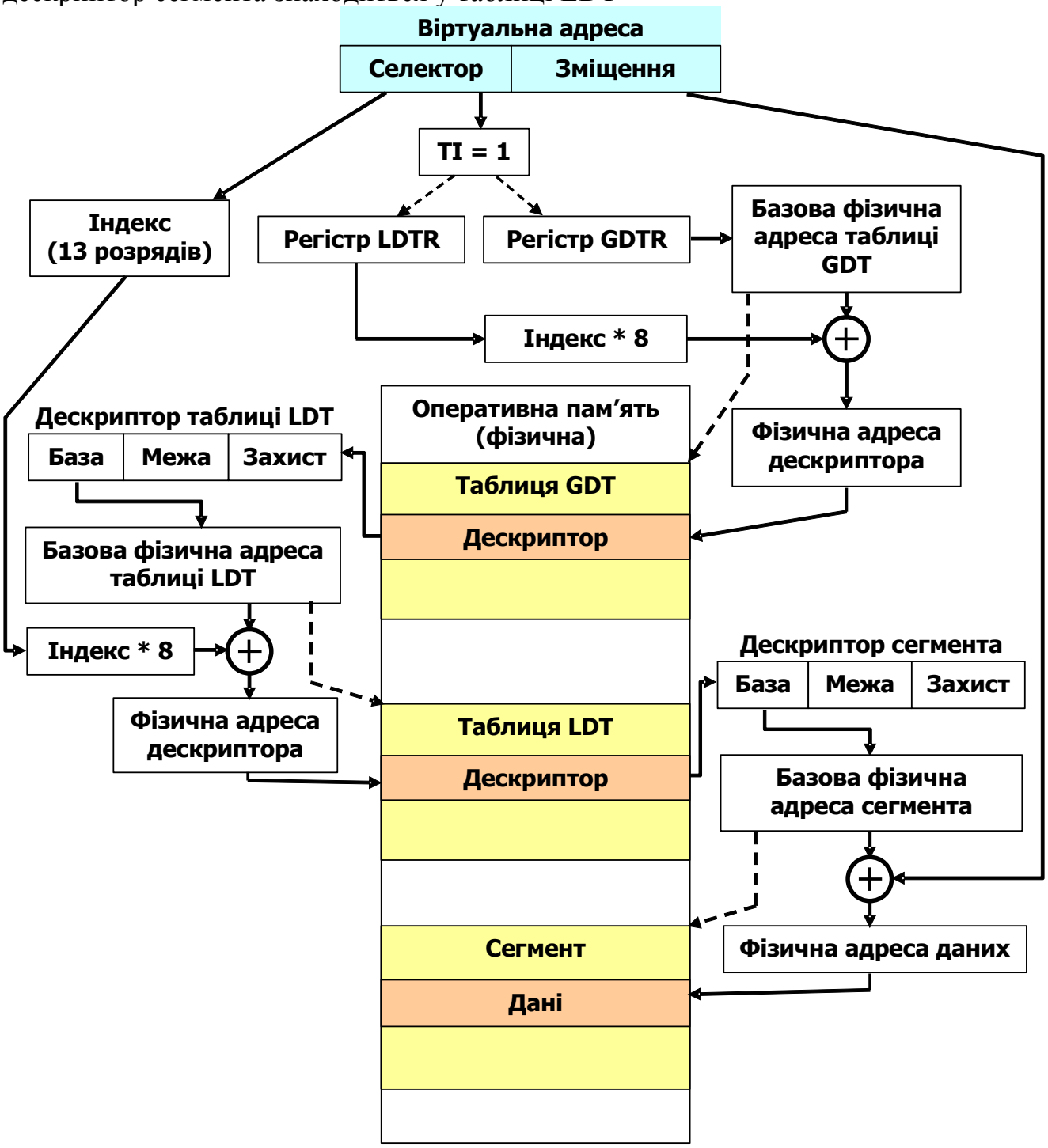
Перетворення адрес за сегментного розподілу пам'яті

дескриптор сегмента знаходиться у таблиці GDT



Перетворення адрес за сегментного розподілу пам'яті

дескриптор сегмента знаходиться у таблиці LDT



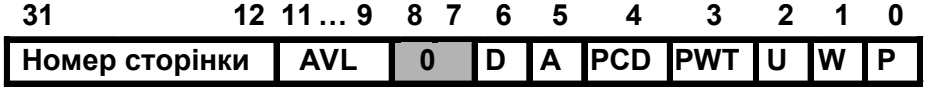
Звернення до пам’яті

- Перевіряють дозвіл на операцію
- Перевіряють коректність доступу (відсутність виходу за межу сегмента)
  - Сегмент стека зростає в бік молодших адрес, і обчислена адреса повинна бути не меншою за межу сегмента
  - Сегменти коду і даних зростають у бік старших адрес, тому до обчисленої адреси додається розмір даних, і отримана адреса повинна бути не більшою за межу сегмента
- Дозволені комбінації бітів байту захисту:

Операція	**S**	**E**	**C/ED**	**R/W**	**Примітка**
Зчитування з пам’яті	1	0	x	x	сегмент даних або стека
	1	1	x	1	сегмент коду
Записування у пам’ять	1	0	x	1	сегмент даних або стека

Сторінковий механізм керування пам'яттю

- Лінійна адреса є 32-розрядною, старші 20 розрядів інтерпретуються як номер сторінки, а молодші 12 – як зміщення в сторінці
- Номер сторінки інтерпретується як індекс дескриптора сторінки, а з дескриптора визначається номер сторінки у фізичній пам’яті



P (Present) Прапорець присутності сторінки у фізичній пам’яті.

W (Writable) Прапорець дозволу записування у сторінку

U (User mode) Прапорець користувач/супервізор

PWT Керують механізмом кешування сторінок (введені, починаючи з процесора i486)

PCD

A (Accessed) Ознака, що мав місце доступ до сторінки

D Ознака модифікації вмісту сторінки

0 Зарезервовані

AVL (Available) Зарезервовані для потреб операційної системи

Регістри, що забезпечують сторінковий механізм

cr0	містить прапорці, які суттєво впливають на роботу процесора і відображають глобальні (незалежні від конкретної задачі) ознаки його функціонування. Деякі важливі системні прапорці з цього регістру: **pe** (*Protect Enable*), біт 0 – вмикає захищений режим роботи процесора; **cd** (*Cache Disable*), біт 30 – вмикає використання внутрішньої кеш-пам'яті (кеш першого рівня); **pg** (*Paging*), біт 31 – вмикає сторінкову трансляцію адрес.
cr2	Містить лінійну віртуальну адресу команди, яка викликала виняткову ситуацію 14 – відсутність сторінки у пам'яті. Обробник цієї виняткової ситуації після завантаження необхідної сторінки у пам'ять має змогу відновити нормальну роботу програми, передавши керування на адресу з cr2
cr3	Містить фізичну базову адресу каталогу сторінок

Лекція 9. Керування оперативною пам'яттю в ОС Linux і Windows

План лекції

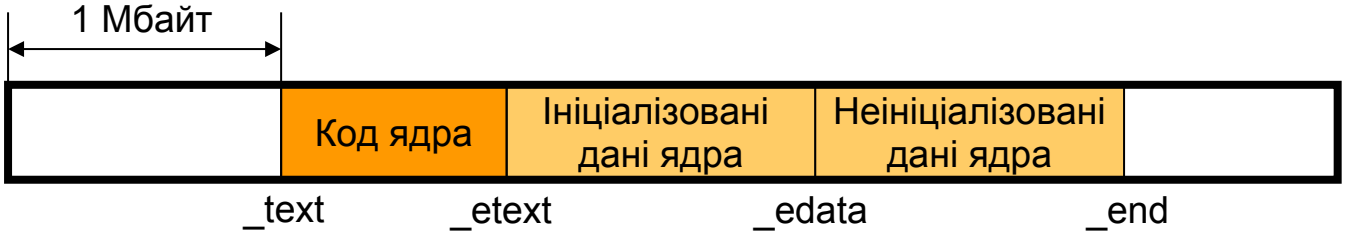
- Керування пам'яттю в ОС Linux
- Керування пам'яттю в ОС Windows

Керування пам’яттю в Linux

- Ядро системи практично не застосовує засобів підтримки сегментації, які надає процесор x86
- Підтримують мінімальну кількість сегментів
  - Сегменти коду і даних ядра
  - Сегменти коду і даних режиму користувача
- Код ядра і режиму користувача спільно використовує ці сегменти
  - Сегменти коду доступні і для виконання, і для зчитування
  - Сегменти даних доступні і для зчитування, і для записування
  - Сегменти даних ядра доступні лише з режиму ядра
- Для усіх сегментів визначають межу у 4 ГБ
  - Таким чином, керування пам’яттю фактично передають на рівень лінійних адрес (які є зміщенням у сегментах)

Розміщення ядра у фізичній пам’яті

- Ядро розміщають у фізичній пам’яті, починаючи з другого мегабайта
- Фрейми пам’яті, у яких розміщено ядро, заборонено вивантажувати на диск і передавати процесам користувача
- З ядра завжди можна визначити фізичні адреси початку та кінця його коду і даних



Особливості адресації процесів і ядра

- У лінійному адресному просторі процесу перші 3 ГБ відображають захищений адресний простір процесу
  - Використовують у режимі ядра та користувача
  - Елементи глобального каталогу, що визначають ці адреси, можуть задаватися самим процесом
- Останній 1 ГБ лінійного адресного простору процесу відображає адресний простір ядра
  - Використовують лише у режимі ядра
  - Елементи глобального каталогу, що визначають ці адреси, однакові для усіх процесів, і можуть задаватися лише ядром
  - Коли передають керування потоку ядра, глобальний каталог (значення регістру **cr3**) не змінюють, оскільки ядро використовує лише ту частину каталогу, яка є спільною для усіх процесів користувача

Керування адресним простором процесу в Linux

- Адресний простір процесу складається з усіх лінійних адрес, які йому дозволено використовувати
- Ядро може динамічно змінювати адресний простір процесу шляхом додавання або вилучення інтервалів лінійних адрес
- Інтервали лінійних адрес зображуються спеціальними структурами даних – *регіонами пам'яті* (*memory regions*)
  - Розмір регіону кратний 4 кБ
  - Регіони пам'яті процесу ніколи не перекриваються
  - Ядро намагається з'єднувати сусідні регіони у більший регіон

Опис адресного простору процесу в Linux

- Кожний регіон описують *дескриптором регіону* (**vm_area_struct**). Дескриптор регіону містить:
  - Початкову лінійну адресу регіону
  - Першу адресу після кінцевої адреси регіону
  - Прапорці прав доступу
    - зчитування, записування, виконання, заборона вивантаження на диск тощо
- Усю інформацію про адресний простір процесу описують *дескриптором пам'яті* (*memory descriptor*, **mm_struct**). Дескриптор пам'яті містить:
  - Кількість регіонів пам'яті
  - Покажчик на глобальний каталог сторінок
  - Адреси різних ділянок пам'яті
    - коду, даних, динамічної ділянки, стека
  - Покажчик на однозв'язний список усіх регіонів процесу
    - Цей список використовують для прискорення сканування всього адресного простору
  - Покажчик на бінарне дерево пошуку, що об'єднує усі регіони процесу
    - Це дерево застосовують для прискорення пошуку конкретної адреси пам'яті

Сторінкова організація пам'яті в Linux

- Три рівня
  - *Page Global Directory*, PGD
  - *Page Middle Directory*, PMD
  - *Page Table*
- Елементи таблиць сторінок PTE вказують на фрейми фізичної пам'яті
- Кожний процес має свій PGD і набір таблиць сторінок
- На архітектурі Intel x86 PMD пустий
  - PGD відповідає каталогу сторінок x86
  - Між таблицями сторінок Linux і таблицями сторінок x86 завжди дотримується однозначна відповідність
- Під час переключення контексту **cr3** зберігають у керуючому блоці процесу

Сторінкові переривання

- Виникають під час звернення до логічної адреси пам’яті, якій у конкретний момент не відповідає фізична адреса
- Якщо переривання відбулось у режимі ядра, поточний процес негайно завершують
- Якщо переривання відбулось у режимі користувача:
  - Перевіряють регіон пам’яті, якому належить адреса. Якщо регіон відсутній, процес завершують
  - Якщо переривання викликане спробою записування у регіон, відкритий лише для зчитування, процес завершують
  - Перевіряють таблицю сторінок процесу
  - Якщо сторінка відсутня, ядро створює новий фрейм і завантажує у нього сторінку
    - Так реалізують завантаження сторінок на вимогу
  - Якщо сторінка є, але позначена “тільки для зчитування”, переривання могло виникнути лише під час спроби записування. Тоді ядро створює новий фрейм і копіює у нього дані зі сторінки
    - Так реалізують технологію копіювання під час записування

Списки сторінок менеджера віртуальної пам’яті Linux

Керування пам’яттю в ОС Windows

- Сегментна модель – як і в Linux
  - Для усіх сегментів в програмі задають однакове значення бази й межі
    Тобто, також передають керування оперативною пам’яттю на рівень лінійних адрес
- Розподіл віртуального адресного простору
  - Перші 2 ГБ – доступні для процесу в режимі користувача
  - Інші 2 ГБ – доступні лише в режимі ядра і відображають системний адресний простір

Структура віртуального адресного простору процесу

- Перші 64 кБ – спеціальна ділянка, доступ до якої спричиняє помилки
- Ділянка, яку процес може використовувати під час виконання
- Блок оточення потоку TEB (4 кБ)
- Блок оточення процесу PEB (4 кБ)
- Ділянка пам’яті, в яку відображають системні дані (системний час, номер версії системи тощо)
  - для доступу до цих даних процесу не требі перемикатись у режим ядра
- Останні 64 кБ – ділянка, спроба доступу до якої завжди спричиняє помилки

Структура системного адресного простору (спрощена)

- Перші 512 МБ – для завантаження ядра системи
- 4 МБ – каталог сторінок і таблиці сторінок процесу
- 4МБ – гіперпростір (hyperspace) – використовують для відображення різних структур даних, специфічних для процесу, на системний адресний простір
- 512 МБ – системний кеш
- Вивантажуваний пул
- Невивантажуваний пул
- Приблизно 4 МБ – структури даних для створення аварійного дампу пам’яті

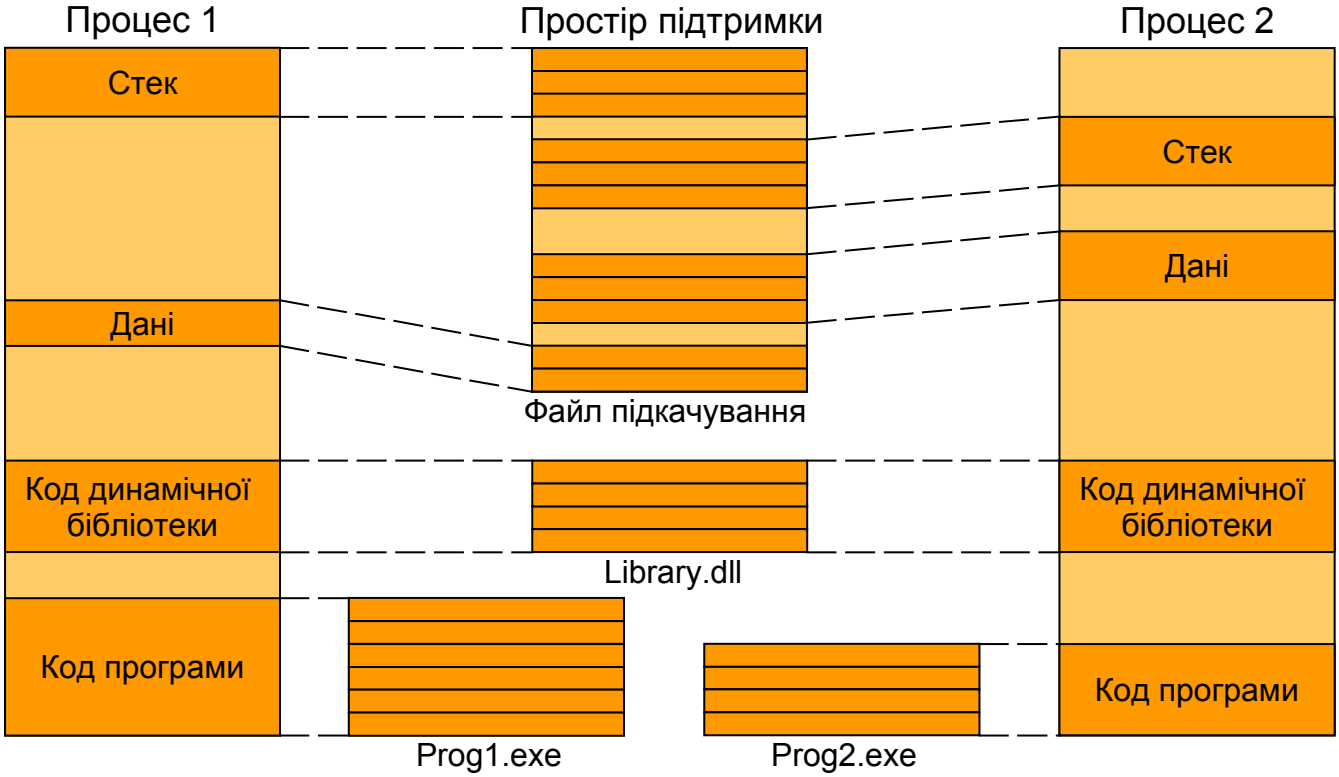
Сторінкова адресація в ОС Windows

- Здійснюється у повній відповідності до архітектури Intel x86
  - У кожного процесу є свій каталог сторінок, кожний елемент якого вказує на таблицю сторінок
  - Таблиці сторінок містять по 1024 елементи, кожний з яких вказує на фрейм фізичної пам'яті
  - Адресу каталогу сторінок зберігають у KPROCESS
- Лінійна адреса – 32 біти
  - 10 – індекс у каталозі сторінок,
  - 10 – індекс у таблиці сторінок,
  - 12 – зміщення
- Елемент таблиці сторінок (дескриптор сторінки) – також 32 біти
  - 20 – адресують конкретний фрейм, якщо сторінка є у фізичній пам'яті, або зміщення у файлі підкачування, якщо сторінка не перебуває у фізичній пам'яті
  - 12 – атрибути сторінки

Сторінки адресного простору

- Сторінки адресного простору можуть бути
  - *вільні (free)*
  - *зарезервовані (reserved)*
  - *підтверджені (committed)*
- Вільні сторінки використовувати не можна. Спочатку вони мають бути зарезервовані
  - Будь-який процес може зарезервувати сторінки
  - Після цього інші процеси не можуть резервувати ті самі сторінки
  - Для використання сторінок процесом вони мають бути підтверджені
- Підтверджені сторінки пов'язані з простором підтримки на диску. Вони можуть бути двох типів:
  - Сторінки, що пов'язані з файлами на диску
    - Простір підтримки – той самий файл
  - Сторінки, що не пов'язані з файлами на диску
    - Простір підтримки – файл підкачування (у файлі підкачування резервуються так звані *тіньові сторінки – shadow pages*)

Процеси і простір підтримки у Windows



Регіони пам’яті у Windows

- Регіон відображає неперервний блок логічного адресного простору процесу
- Регіон характеризується початковою адресою і довжиною
  - Початкова адреса регіону повинна бути кратною 64 кБ
  - Розмір регіону повинен бути кратним розміру сторінки – 4 кБ
- Регіони необхідно резервувати і підтверджувати
  - Після резервування регіони інші запити не можуть його повторно резервувати
  - У процесі підтвердження регіону для нього створюються тіньові сторінки – операція підтвердження вимагає доступу до диску і є значно повільнішою, ніж резервування
  - Типовою стратегією є резервування великого регіону, а далі поступове підтвердження його малих частин
  - Для резервування і підтвердження регіону використовують функцію **VirtualAlloc()** (з різними параметрами)

Причини виникнення сторінкових переривань

- Звернення до сторінки, що не була підтверджена
  - фатальна помилка
- Звернення до сторінки із недостатніми правами
  - фатальна помилка
- Звернення для записування до сторінки, що спільно використовується процесами
  - технологія копіювання під час записування
- Необхідність розширення стека процесу
  - оброблювач переривання має виділити новий фрейм і заповнити його нулями
- Звернення до сторінки, що була підтверджена, але не завантажена у фізичну пам’ять
  - застосовують випереджаюче зчитування

Лекція 10. Керування введенням-виведенням

План лекції

- Завдання керування введенням-виведенням
- Фізична організація пристроїв введення-виведення
- Організація програмного забезпечення введення-виведення
- Синхронне та асинхронне введення-виведення

Завдання керування введенням-виведенням

- Забезпечення доступу до зовнішніх пристроїв з прикладних програм
  - Передавати пристроям команди
  - Перехоплювати переривання
  - Обробляти помилки
- Забезпечення спільного використання зовнішніх пристроїв
  - Розв’язувати можливі конфлікти (синхронізація)
  - Забезпечувати захист пристроїв від несанкціонованого доступу
- Забезпечення інтерфейсу між пристроями і рештою системи
  - Інтерфейс повинен бути універсальним для різних пристроїв

Фізична організація пристроїв введення-виведення

- Два типи пристроїв – *блок-орієнтовані* (або блокові) та *байт-орієнтовані* (або символьні)
- Блок-орієнтовані пристрої
  - зберігають інформацію у блоках фіксованого розміру, які мають адресу
  - дозволяють здійснювати пошук

  Приклад: диск
- Байт-орієнтовані пристрої
  - не підтримують адреси
  - не дозволяють здійснювати пошук
  - генерують або споживають послідовність байтів

  Приклади: термінал, клавіатура, мережний адаптер
- Існують пристрої, які не належать ні до перших, ні до других

  Приклад: таймер
- До якого типу належить оперативна пам’ять?

Контролери пристроїв

- Пристрій може мати механічний компонент
- Пристрій обов'язково має електронний компонент
  - Електронний компонент пристрою, що взаємодіє з комп'ютером, називають *контролером* (також – *адаптером*)
  - Контролер завжди має кілька *регістрів*
- ОС взаємодіє саме з контролером
- Незалежно від типу пристрою, контролер, як правило:
  - Під час операції введення перетворює потік біт у блоки, що складаються з байтів (під час виведення – навпаки)
  - Здійснює контроль і виправлення помилок

Робота з регістрами

- Операції
  - Виведення даних – це записування даних у *регістр виведення* контролеру (**data out**)
  - Введення даних – це зчитування даних з *регістра введення* контролеру (**data in**)
  - Керування пристроєм – це записування даних у *регістр керування* контролеру (**control**)
  - Перевірка стану пристрою і результату попередньої операції – це зчитування даних з *регістра статусу* контролеру (**status**)
- Варіанти архітектури комп'ютерів:
  - Регістри контролерів є частиною адресного простору оперативної пам'яті (**memory-mapped I/O**)
  - Регістри контролерів утворюють власний адресний простір – порти введення-виведення (**I/O port**)

Драйвери пристроїв

- Драйвер пристрою – це програмний модуль, що керує взаємодією ОС із конкретним зовнішнім пристроєм
  - точніше – з його контролером
- Драйвер можна розглядати як транслятор, який:
  - отримує на свій вхід команди високого рівня, які зумовлені його (універсальним) інтерфейсом з ОС
  - на виході генерує інструкції низького рівня, специфічні для конкретного контролера
- Драйвери практично завжди виконують у режимі ядра
  - Драйвери можуть бути завантажені у пам'ять і вивантажені з неї під час виконання

Способи виконання операцій введення-виведення

- Опитування пристроїв
  - Драйвер у циклі зчитує біт **busy** регістру статусу, поки він не буде знятий
  - Драйвер встановлює біт **write** керуючого регістра і записує байт даних у регістр виведення
  - Драйвер встановлює біт **cready**
  - Контролер виявляє встановлений біт **cready** і встановлює біт **busy**
  - Контролер виявляє встановлений біт **write**, зчитує регістр виведення і передає отриманий байт пристрою
  - Контролер знімає біти **cready** і **busy** (операція завершена)
- Введення-виведення, кероване перериваннями
  - У регістри записують команди та(або) дані
  - Контролер працює
  - Контролер викликає переривання
  - Обробник переривання перевіряє результати операції
- Прямий доступ до пам’яті (DMA)
  - Процесор дає команду DMA-контролеру виконати зчитування блоку даних з пристрою, при цьому він передає контролеру адресу буфера у фізичній пам’яті
  - DMA-контролер починає пересилання (процесор може виконувати інші інструкції)
  - Після завершення пересилання усього блоку (наприклад, 4 кБ), DMA-контролер генерує переривання
  - Оброблювач переривання завершує обробку операції

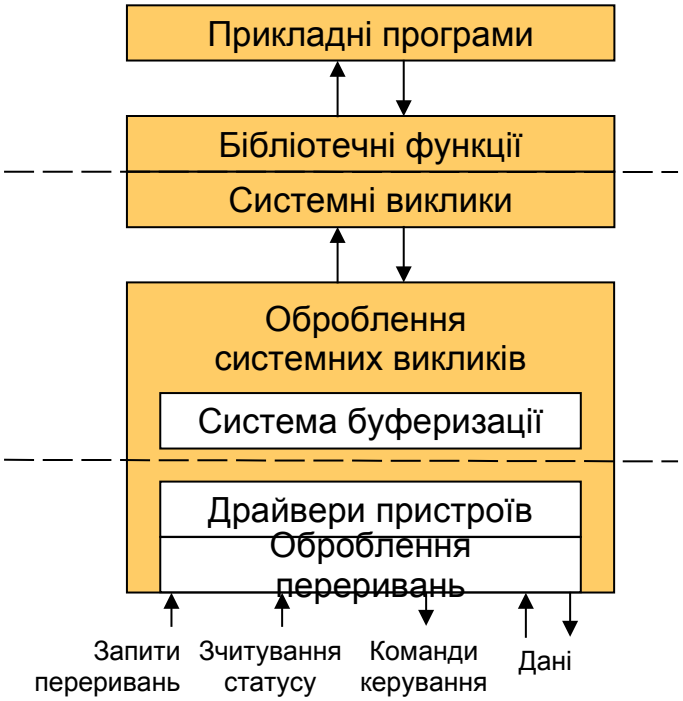
Особливості оброблення переривань

- Рівні переривань
  - Деякі переривання більш важливі, деякі – менш
  - Переривання поділяють на рівні відповідно до їхнього пріоритету (IRQL – *Interrupt Request Level*)
  - Можна *маскувати* переривання, які нижчі певного рівня (тобто, скасовувати їх оброблення)
- Встановлення оброблювачів переривань
  - Апаратні переривання обробляються *контролером переривань*, який надсилає процесору сигнал різними лініями (IRQ line)
    - Раніше контролери переривань були розраховані на 15 ліній
    - Сучасні контролери розраховані на більше ліній (Intel – 255)
  - Оброблювач переривання встановлюється драйвером пристрою, при цьому драйвер повинен визначити лінію IRQ, яку буде використовувати пристрій
    - Раніше користувачі мали вручну обрати номер лінії IRQ
    - Для деяких стандартних пристроїв номер лінії IRQ документований і незмінний
    - Драйвер може виконувати *зондування* (*probing*) пристрою. Цей підхід вважається застарілим
    - Пристрій сам повідомляє номер лінії. Цей підхід є найкращим

Особливості реалізації оброблювачів переривань

- В оброблювачах дозволено виконувати більшість операцій, за деякими винятками:
  - Не можна обмінюватись даними з адресним простором режиму користувача (оброблювач не виконується у контексті процесу)
  - Не можна виконувати жодних дій, здатних спричинити очікування
    - Явно викликати sleep()
    - Звертатись до синхронізаційних об'єктів з викликами, які можна заблокувати
    - Резервувати пам'ять за допомогою операцій, що призводять до сторінкового переривання
- Відкладене оброблення переривань
  - Для запобігання порушенню послідовності даних оброблювач переривань має на певний час блокувати (маскувати) переривання
  - Оброблювач повинен швидко завершувати свою роботу, щоби не тримати заблокованими наступні переривання
  - Для підвищення ефективності у сучасних ОС код оброблювача поділяють на дві половини:
    - *Верхня половина* (*top half*) – безпосередньо оброблювач переривання
    - *Нижня половина* (*bottom half*) – реалізує відкладене оброблення переривання. Не маскує переривання!

Ієрархія рівнів програмного забезпечення введення-виведення



- Користувальницький рівень ПЗ
- Незалежний від пристроїв рівень
- Драйвери пристроїв
- Оброблення переривань

Рівень, незалежний від пристроїв

- Забезпечення спільного інтерфейсу до драйверів пристроїв
- Іменування пристроїв
- Захист пристроїв
- Буферизація
- Розподіл і звільнення виділених пристроїв
- Повідомлення про помилки

Синхронні та асинхронні операції введення-виведення

- Введення-виведення на рівні апаратного забезпечення є керованим перериваннями, а отже – асинхронним
- На користувальницькому рівні організувати синхронне оброблення даних значно простіше, ніж асинхронне
  - Потік робить блокувальний (синхронний) виклик
  - Ядро переводить потік у стан очікування пристрою
  - Після завершення операції введення-виведення викликається переривання
  - Обробник переривання переводить потік у стан готовності
- Синхронне введення-виведення не підходить для:
  - Серверів, що обслуговують багатьох клієнтів
  - Програм, що працюють з журналом
  - Мультимедійних застосувань

Асинхронне введення-виведення на користувальницькому рівні

- Багатопотокова організація
  - У процесі створюють новий потік, який виконує звичайне синхронне введення-виведення
  - Проблеми:
    - Потрібно реалізовувати синхронізацію потоків
    - Може знизитись надійність прикладної програми
- Введення-виведення з повідомленнями (notification-driven I/O, I/O multiplexing)
  - Застосовується тоді, коли треба у циклі виконати блокувальний виклик для кількох файлових дескрипторів
    - Один із викликів може заблокувати потік тоді, коли на іншому дескрипторі є дані
  - Реалізують спеціальний виклик, який дозволяє потоку отримати повідомлення про стан дескрипторів
    - Після цього можна у циклі працювати з усіма дескрипторами, для яких у поточний момент введення-виведення можливе (блокування потоку не виникне)
- Асинхронне введення-виведення
  - Основна ідея – потік, який почав виконувати введення-виведення, не блокують
  - Послідовність:
    - Потік виконує системний виклик асинхронного в-в, який ставить операцію в-в у чергу і негайно повертає керування
    - Потік продовжує виконання паралельно з операцією в-в
    - Коли операцію завершено, потік отримує повідомлення
  - У разі асинхронного в-в не гарантується послідовність операцій
    - Якщо потік виконує поспіль операції 1 і 2, то цілком ймовірно, що операція 2 буде виконана до операції 1
- Порти завершення введення-виведення (I/O completion port)
  - Поєднує багатопотоковість з асинхронним в-в
  - Має переваги для серверів, що обслуговують велику кількість одночасних запитів
  - Принцип:
    - Створюють *пул потоків* (*thread pool*)
    - Кожний потік готовий до обслуговування запитів в-в
    - Кількість потоків обирають виходячи з кількості наявних процесорних ядер

Лекція 11. Керування введенням-виведенням в ОС Linux, UNIX, Windows

План лекції

- Керування введенням-виведенням в ОС UNIX і Linux
- Керування введенням-виведенням в ОС Windows

Введення-виведення в UNIX і Linux

- Введення-виведення здійснюється через файлову систему
- Кожному драйверу пристрою відповідає один або кілька спеціальних файлів пристроїв
- Файли пристроїв традиційно розміщені у каталозі /dev
- Кожний файл пристрою характеризується чотирма параметрами
  - Ім’я файлу
    - застосовується для доступу до пристрою з прикладних програм
  - Тип пристрою (символьний чи блоковий)
    - фактично вказує на таблицю – одна таблиця для символьних пристроїв, друга – для блокових
  - Major number – номер драйвера у таблиці
    - ціле число, як правило, 1 байт (може 2)
  - Minor number – номер пристрою
    - це число передають драйверу, драйвер може працювати з кількома пристроями, у тому числі різними

Робота з файлами пристроїв

- Файли пристроїв, які і звичайні файли, можна створювати й видаляти
  - На драйвер це ніяк не впливає
  - Команда створення файлу пристрою:

  mknod /dev/mydevice c 150 1
- Звернення до файлу:
  - Спочатку – звернення до файлу за іменем
  - Система звертається до індексного дескриптора
  - Перевіряє права доступу
  - З дескриптора визначає тип пристрою і номер драйвера
  - Звертається до драйвера і передає йому номер пристрою
  - Виконує задану файлову операцію

Операції роботи з пристроями

- Драйвер зобов'язаний підтримувати стандартні файлові операції
  - **open()**, **read()**, **write()**, **lseek()**
  - Прикладні програми виконують такі операції так, якби вони працювали із звичайними файлами
- Для деяких пристроїв крім стандартних операцій, можливо, визначені й деякі інші операції
  - Для реалізації нестандартних операцій передбачено універсальний виклик **ioctl()**
    ioctl(int d, int request, char *agrp);

    d – файловий дескриптор

    request – код операції

    *** agrp** – покажчик на довільну пам'ять
  - Наприклад, для приводу оптичного диску так можна визначити операцію EJECT

Структура драйвера

- Код ініціалізації (одна функція **init()**)
  - Код виконується під час завантаження ядра системи або під час завантаження модуля драйвера
  - Цей код реалізує реєстрацію драйвера у системі (вибір номера, реєстрацію оброблювачів переривань)
  - Цей код не може створювати спеціальні файли!
- Реалізацію файлових операцій і **ioctl()**
  - Для символьних пристроїв: **open(), close(), read(), write(), lseek(), select(), mmap()**
  - Для блокових пристроїв є особливість: реакцію на операції зчитування і записування викликають не прямо, а після проходження керування через буферний кеш; для цього їх реалізують в одній функції
    - UNIX – **strategy()**
    - Linux – **request()**
- Обробники переривань
  - Обробників переривань може і не бути: драйвер може не застосовувати переривання, а виконувати опитування пристроїв
  - Обробники переривань мають верхню і нижню половини

Введення-виведення з розподілом і об'єднанням

- За одну операцію здійснюється зчитування в, або записування з кількох не пов'язаних ділянок пам'яті
  - введення – *scatter* – **readv()**
  - виведення – *gather* – **writev()**

  ssize_t readv(int fdl, const struct iovec *iov, int count);

  fdl – дескриптор відкритого файлу;

  iov – масив структур, які задають набір ділянок пам'яті для введення і виведення;

  count – кількість структур у масиві **iov**
- Кожний елемент масиву містить два поля:
  - **iov_base** – задає базову адресу ділянки пам'яті
  - **iov_len** – задає довжину ділянки пам'яті
- Виклики повертають загальну кількість байтів (зчитаних або записаних)

Введення-виведення з повідомленням

- Введення-виведення з повідомленням про стан дескрипторів
  - Готують структуру даних **fdarr** з описом усіх дескрипторів, стан яких треба відстежувати
  - Передають **fdarr** у системний виклик повідомлення (у POSIX – виклики **select()** і **poll()**)
  - Після виходу з виклику **fdarr** містить інформацію про стан усіх дескрипторів
  - У циклі обходять усі елементи **fdarr** і для кожного з них визначають готовність дескриптора
- Введення-виведення з повідомленням про події (у FreeBSD – **kqueue**, Linux 2.6 – **epoll**)
  - Системний виклик (**epoll_create()**) створює структуру даних у ядрі (прослуховувальний об'єкт)
  - Для прослуховувального об'єкта формують набір дескрипторів, для кожного з них вказують події, які цікавлять (**epoll_ctl()**)
  - Виклик повідомлення (**epoll_wait()**) повертає інформацію лише про ті дескриптори, які змінили свій стан з моменту останнього виклику

Асинхронне введення-виведення

- Стандарт POSIX передбачає такі виклики для асинхронного введення-виведення:
  - **aio_read()** – зчитування
  - **aio_write()** – записування
  - **aio_suspend()** – очікування
  - **aio_cancel()** – переривання
  - **aio_return()** – отримання результату
  - **aio_error()** – отримання статусу операції
- Усі виклики, крім **aio_suspend()**, приймають параметром покажчик на структуру **aiocb** з полями:
  - **aio_fildes** – дескриптор файлу, для якого здійснюють введення-виведення
  - **aio_buf** – покажчик на буфер, у який зчитає дані **aio_read()** і з якого запише дані **aio_write()**
  - **aio_nbytes()** – розмір буфера
- Формат виклику **aio_suspend()**:
  int aio_suspend(struct aiocb *list[], int cnt, struct timespec *tout);

Послідовність виконання операції введення-виведення

- Процес користувача готує буфер у своєму адресному просторі
- Процес користувача виконує системний виклик **read()** для спеціального файлу пристрою, і передає у виклик адресу буфера
- Відбувається перехід у режим ядра
  - Переключення контексту не здійснюють
- На підставі інформації з індексного дескриптора спеціального файлу визначають необхідний драйвер і викликають функцію, яка зареєстрована як реалізація файлової операції **read()** для відповідного драйвера
- Функція:
  - Виконує необхідні підготовчі операції
    - Наприклад, розміщує буфер у пам'яті ядра
  - Відсилає контролеру пристрою запит на виконання операції зчитування
  - Переходить у режим очікування
    - Для цього як правило використовують функцію **sleep_on()**
    - При цьому процес, що викликав операцію, призупиняється
- Контролер здійснює зчитування
  - При цьому він, можливо, використовує буфер, наданий йому функцією драйвера
- Після завершення зчитування контролер викликає переривання
- Апаратне забезпечення активізує верхню половину оброблювача переривання
- Код верхньої половини ставить нижню половину у чергу на виконання
- Код нижньої половини:
  - Заповнює буфер, якщо він не був заповнений контролером
  - Виконує інші необхідні дії для завершення операції введення
  - Поновлює виконання процесу, що очікує
- Керування після поновлення виконання процесу повертається у реалізацію функції read() для драйвера – у код, що слідує за викликом sleep_on()
  - Цей код копіює дані з буфера ядра у буфер режиму користувача
- Керування повертають у процес користувача

Введення-виведення у Windows

- Базовий компонент – менеджер введення-виведення (I/O Manager)
- Операції – асинхронні
  - Синхронні операції реалізують як асинхронні операції + очікування
- Операції введення-виведення відображаються у вигляді структур даних – пакетів запитів введення-виведення (I/O Request Packet)
  - Менеджер введення-виведення створює пакет і передає покажчик на нього потрібному драйверу
  - Драйвер
    - отримує пакет,
    - виконує потрібну операцію,
    - повертає пакет як індикатор виконання операції або для передачі його іншому драйверу
  - Після завершення операції введення-виведення, менеджер вивільняє пам'ять, яку займав пакет

Асинхронне введення-виведення

- На відміну від стандарту POSIX, у Win32 API для асинхронного введення-виведення можна застосовувати стандартні функції файлового в/в – **ReadFile()** і **WriteFile()**
  - Для цього у функції передається покажчик на спеціальну структуру **OVERLAPPED**
  - Файл повинен бути відкритим з дозволом асинхронних операцій **FILE_FLAG_OVERLAPPED**
- Для очікування завершення введення-виведення використовують універсальну функцію очікування
  - Наприклад, **WaitForSingleObject()**
- Для отримання результату необхідно використати функцію **GetOverlappedResult()**
- Для переривання введення-виведення використовують функцію **Cancello()**

Порти завершення введення-виведення (*I/O completion port*)

- Спочатку створюють новий об'єкт порту, потім додають у нього файлові дескриптори
  - Застосовують виклик **CreateCompletionPort()**
    CreateCompletionPort (fdarr[key], ph, key, R_{max});

    Один з параметрів, який передають у виклик – це максимальна кількість потоків, що можуть виконуватись у системі R_{max}
    - Оптимально – дорівнює кількості процесорних ядер
- Після цього формують *пул робочих потоків* (*thread pool*)
  - Кількість має перевищувати R_{max}
  - Кожний з потоків повинен виконувати один і той самий код

```
For ( : : ) {
        // очікування на об'єкті порту, заданому дескриптором ph
        GetQueuedCompletionStatus (ph, nbytes, &key, &ov, INFINITE);
        // тут потік є активним
        ReadFile (fdarr[key], request, … );     // прочитати запит
        process_request (request);              // обслужити клієнта
}
```

Категорії драйверів

- Типи драйверів згідно *Windows Driver Model*, WDM (для ядра версії 5)
  - Драйвери шини
    - Керують логічною або фізичною шиною, відповідають за виявлення пристроїв
  - Функціональні драйвери
    - Керують пристроєм конкретного типу
  - Драйвери-фільтри
    - Доповнюють або змінюють поведінку інших драйверів
- Категорії драйверів ядра (крім WDM-драйверів)
  - Файлових систем
    - Перетворюють запити введення-виведення, що використовують файли, у запити до низькорівневих драйверів пристроїв
  - Відображення (*Display Drivers*)
    - Перетворюють незалежні від пристрою запити підсистеми GDI в команди графічного адаптера або команди записування у пам'ять
  - Успадковані
    - Розроблені для Windows NT 4
- Категорії драйверів режиму користувача
  - Наприклад, драйвери принтера
    - Перетворюють незалежні від пристрою запити підсистеми GDI в команди конкретного принтера і передають їх WDM драйверу

- Категорії драйверів в залежності від рівня підтримки конкретного пристрою
  - Клас-драйвери
    - Реалізують інтерфейс оброблення запитів введення-виведення, специфічних для конкретного класу пристроїв (диски, CD-ROM)
  - Порт-драйвери
    - Реалізують інтерфейс оброблення запитів введення-виведення, специфічних для певного класу портів (SCSI)
  - Мініпорт-драйвери
    - Керують реальним конкретним пристроєм

Структура драйвера пристрою

- Процедура ініціалізації *Driver Entry*
  - Її виконує менеджер введення-виведення під час завантаження драйвера у систему
  - Здійснює глобальну ініціалізацію структур даних драйвера
- Процедура додавання пристрою *Add-device routine*
  - Для реалізації технології Plug and Play
  - Менеджер Plug and Play викликає цю процедуру, якщо знаходить пристрій, за який відповідає драйвер
- Набір процедур диспетчеризації *Dispatch Routines*
  - Аналогічні функціям файлових операцій в UNIX і Linux
- Процедура оброблення переривання *Interrupt Service Routine*, ISR
  - Аналогічна верхній половині оброблювача переривання
  - Основне завдання – запланувати для виконання нижню половину оброблювача
- Процедура відкладеного оброблення переривання DPC *Routine*
  - Аналогічна нижній половині оброблювача переривання

Послідовність виконання операції введення-виведення

- Запит в/в перехоплює динамічна бібліотека
  - Наприклад, Win32 перехоплює виклик **WriteFile()**
- Динамічна бібліотека викликає внутрішню функцію **NTWriteFile()**, яка звертається до менеджера в/в
- Менеджер в/в створює пакет IRP, розміщає його у пам'яті і відправляє посилання на нього драйверу викликом функції **IoCallDriver()**
- Драйвер бере дані з IRP і передає їх контролеру пристрою, після чого дає команду розпочати операцію в/в
- Для синхронного в/в драйвер викликає функцію очікування
  - Поточний потік призупиняють
- Коли операція завершується, контролер викликає переривання
- Драйвер викликає функцію **IoCompleteRequest()** для повідомлення менеджеру в/в про завершення дій з пакетом, після чого виконують код завершення операції

Завершення запиту введення-виведення

- Звичайно завершення зводиться до копіювання даних в адресний простір процесу користувача
- У разі синхронного введення-виведення адресний простір належить до процесу, що робив виклик
  - Дані можуть бути записані у нього безпосередньо
- Якщо запит був асинхронним, активний потік ймовірно належить до іншого процесу
  - Необхідно дочекатися, поки адресний простір потрібного процесу не стане доступний
  - Для цього менеджер в/в планує до виконання спеціальну *APC-процедуру* (від *Asynchronous Procedure Call*)
  - APC-процедура виконується лише у контексті конкретного потоку
    - Отримує керування
    - Копіює потрібні дані в адресний простір процесу
    - Вивільняє пам'ять з-під пакета IRP
    - Переводить файловий дескриптор (або порт завершення введення-виведення) у сигналізований стан

Лекція 12. Файлові системи

План лекції

- Основні поняття про файли і файлові системи
- Імена файлів
- Типи файлів
- Логічна організація файлів
- Файлові операції
- Міжпроцесова взаємодія через файлову систему
- Загальна модель файлової системи
- Фізична організація файлів

Основні означення

- *Файл* – це набір даних, до якого можна звертатись за іменем
  - Файли є найпоширенішим засобом організації доступу до інформації, що зберігається в енергонезалежній пам'яті
- *Файлова система* – це підсистема ОС, яка призначена для того, щоби забезпечити користувачеві зручний інтерфейс для роботи з даними, що зберігаються в енергонезалежній пам'яті (на диску), і забезпечити спільне використання файлів кількома користувачами і процесами
  - Файлова система надає прикладним програмам абстракцію файлу
- До складу файлової системи входять:
  - сукупність усіх файлів;
  - набори структур даних, що застосовуються для керування файлами:
    - каталоги,
    - дескриптори,
    - таблиці розподілу дискового простору;
  - комплекс системних програмних засобів, що реалізують керування файлами, зокрема створення, видалення, зчитування, записування, іменування, пошук та інші операції

Імена файлів

- Для користувачів – символьні імена
  - Обмеження на алфавіт
    - Які символи припускаються (ASCII, Unicode)
    - Чутливість до регістра (myfile.txt *vs* MYFILE.TXT)
  - Наявність і формат розширення
    - У деяких системах розширення обов'язкове (система і прикладні програми розрізняють типи файлів за розширенням)
  - Обмеження довжини імені
    - DOS (FAT) – 8.3
    - UNIX System V – 14
    - Windows (NTFS) – 255
- Унікальне ім'я
  - У більшості систем – повний шлях до файлу + ім'я файлу
  - Система не обов'язково працює із символьними іменами
    - Наприклад, у UNIX за іменем файлу визначають його індексний дескриптор (i-node)

Типи файлів

- Звичайні файли
  - Текстові – рядки символів (деякі символи можуть мати спеціальне значення – кінець рядка, кінець файлу
  - Бінарні – послідовність байт (біт)
    - Окремий тип – виконувані файли
- Файли-каталоги (directory)
  - З одного боку це група файлів
  - З іншого боку, це спеціальний файл, який містить інформацію про файли, що до нього входять
- Спеціальні файли
  - Файли пристроїв
    - Блок-орієнтовані пристрої – файли із прямим доступом
    - Байт-орієнтовані (символьні) пристрої – файли із послідовним доступом
  - Дозволяють замінити спеціалізовані операції введення-виведення на стандартні операції зчитування і записування у файл

Каталоги

- Каталог містить список файлів і встановлює відповідність між файлами та їхніми характеристиками (атрибутами)
  - Ім’я
  - Тип (бінарний/символьний, каталог, зв’язок, спеціальний)
  - Розмір файлу
  - Атрибути безпеки (власник, read only, hidden, system, temporary, атрибути доступу)
  - Часові атрибути (час створення, час останньої модифікації)
- Каталоги можуть містити
  - усю інформацію (MS-DOS)
  - лише посилання на таблиці (UNIX)
- Каталоги можуть утворювати ієрархічну структуру
  - дерево (MS-DOS)
  - мережа (UNIX)
- Шлях до файлу – перелік каталогів через які можна отримати доступ до файлу
  - Може бути абсолютним і відносним

Розділи

- *Розділ* (*partition*) – частина фізичного дискового простору, що призначена для розміщення на ній структури одної файлової системи і з логічної точки зору розглядається як єдине ціле
- Розділ – це логічний пристрій, що з погляду ОС функціонує як окремий диск
  - Може відповідати усьому фізичному диску
  - Найчастіше відповідає частині фізичного диску
- Кожний розділ може мати свою файлову систему і, можливо, використовуватись різними ОС
- Існують два підходи до реалізації взаємозв’язку розділів і структури каталогів файлової системи
  - Розділи із встановленими на них файловими системами об’єднуються (монтуються) в єдине дерево каталогів
    - Характерно для UNIX-подібних систем
    - Приховує від користувача структуру розділів
  - Кожний розділ є окремим логічним томом, що має власне позначення (C:, D: тощо), власний кореневий каталог, власне дерево каталогів

Зв’язки (або посилання)

- Жорсткі зв’язки (*hard links*)
  - Встановлення жорсткого зв’язку – це фактично надання файлу додаткового імені (у більшості випадків, воно знаходиться в іншому каталозі)
  - Усі жорсткі зв’язки посилаються на один і той самий файл з його атрибутами (повинна підтримуватись таблиця дескрипторів)
  - Усі жорсткі зв’язки рівноправні
  - Спроба видалення файлу приводить до видалення лише окремого його імені і зменшення на одиницю кількості імен, сам файл видаляється лише після видалення останнього з імен
- Символічні зв’язки (*symbolic links*)
  - Символічний зв’язок – це спеціальний файл, який містить ім’я файлу, на який він вказує
  - Видалення зв’язку ніяк не впливає на файл
  - Видалення або переміщення файлу зробить зв’язок недійсним (відновлення файлу поновить зв’язок)

Логічна організація файлів

- Структура, з якою має справу програміст
- Елемент логічної структури файлу називають *логічним записом*
  - Логічний запис – це найменший елемент даних, яким може оперувати програміст під час обміну із зовнішнім пристроєм
  - Фізичний обмін з пристроєм, як правило, здійснюється більшими порціями (блоками)
- Логічні записи можуть бути
  - Фіксованої довжини
  - Змінної довжини
  - З використанням індексних таблиць
- Для ідентифікації логічні записи можуть мати спеціальне поле, яке називають *ключем*
- У сучасних файлових системах файли, як правило, мають найпростішу структуру у вигляді послідовності однобайтових записів
  - Довжина логічного запису фіксована і дорівнює 1 байту,
  - Ключ відсутній

Файлові операції

- Відкриття файлу
  - Необхідне для того, щоби розпочати роботу із файлом
  - Зазвичай передбачає завантаження у пам’ять дескриптора файлу, який визначає його атрибути і місце розташування на диску
- Закриття файлу
  - Структуру, створену під час відкриття файлу, видаляють з пам’яті
  - Усі зміни, що не були збережені, записують на диск
- Створення файлу
  - На диску створюють файл нульової довжини
  - При цьому його автоматично відкривають
- Видалення файлу
  - Структури, що описують файл (індексний дескриптор, запис у каталозі) вилучають або помічають як недійсні
  - Дисковий простір, який займав файл, помічають як вільний (але, як правило, не очищають)
  - Як правило, ця операція недопустима для відкритих файлів
- Читання з файлу
  - Пересилання певної кількості байтів із файлу, починаючи з поточної позиції, у заздалегідь виділений буфер пам’яті режиму користувача
- Записування у файл
  - Здійснюють з поточної позиції
  - Якщо у цій позиції вже є записані дані, їх перезаписують
  - Дані пересилають із заздалегідь виділеного буфера
  - Операція може змінити розмір файлу
- Переміщення покажчика
  - Покажчик поточної позиції можна встановити на будь-яке місце всередині файлу, або безпосередньо за його кінець
- Отримання і призначення атрибутів файлу
  - Операції дозволяють зчитати усі або деякі атрибути файлу і встановити їх нові значення

Операції над каталогами

- Створення нового каталогу
  - Новий створений каталог, як правило, порожній (у деяких реалізаціях у нього одразу додають елементи “.” та “..”)
- Видалення каталогу
  - На рівні системного виклику ця операція дозволена лише для порожнього каталогу
- Відкриття і закриття каталогу
  - Каталог, подібно до звичайного файлу, має бути відкритим перед використанням і закритим після використання
  - Деякі операції, пов’язані з доступом до елементів каталогу, допустимі лише для відкритих каталогів
- Зчитування елементів каталогу
  - Зчитує один елемент каталогу і переміщає поточну позицію на один елемент
- Перехід у початок каталогу
  - Переміщує поточну позицію на початок каталогу

Відображення файлів у пам’ять

- Для відображення файлу у пам’ять застосовуються спеціальні системні виклики
  - У POSIX – **mmap()**, **munmap()**
  - Перед виконанням відображення файл має бути відкритим
  - У визначену частину адресного простору процесу відображають заданий файл або його частину
  - Після виконання цього виклику доступ до такої пам’яті спричинятиме прямий доступ до вмісту цього файлу
  - У разі закриття файлу або завершення процесу модифіковану інформацію зберігають у файлі на диску
- Кілька процесів можуть відобразити той самий файл у свої адресні простори
  - Таким чином можна реалізувати обмін даними між процесами
- Переваги відображення файлів у пам’ять
  - Лише один системний виклик
  - Прямий доступ до будь-якої ділянки (не потрібно переміщати покажчик у файлі)
  - Не потрібно копіювати дані між системною пам’яттю і буфером режиму користувача
- Недоліки
  - Не можна змінити розмір файлу!
  - Файл може бути більшим за обсяг віртуального адресного простору!

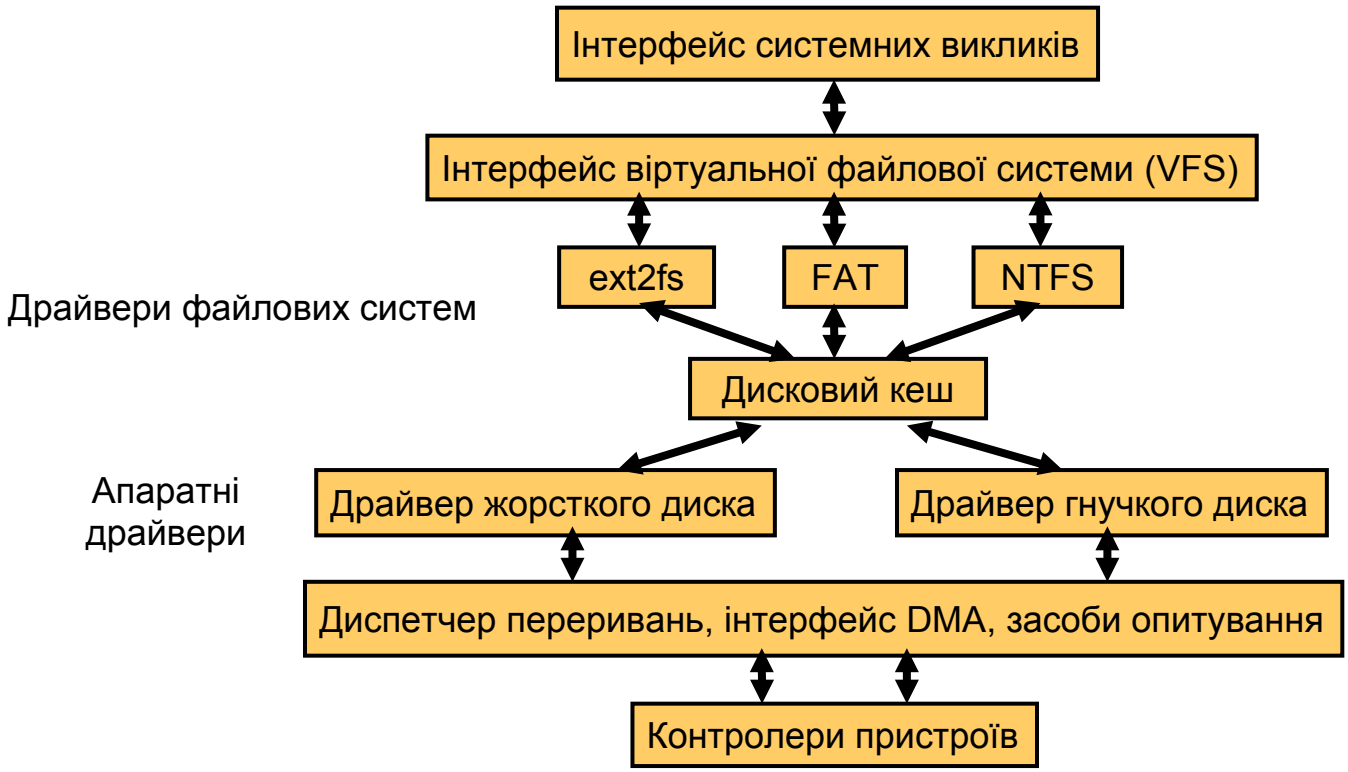
Міжпроцесова взаємодія через файлову систему

- Файлові блокування (*file locks*)
  - Засіб синхронізації процесів, які намагаються здійснити доступ до одного файлу
  - Консультативне, або кооперативне блокування (*advisory lock*)
    - Процес перед здійсненням операції перевіряє наявність блокування
    - У разі блокування – відмовляється від операції
    - За відсутності блокування – сам блокує
    - Якщо здійснює операцію без перевірки, то її дозволяють
  - Обов'язкове блокування (*mandatory lock*)
    - Здійснюється на рівні ядра
    - Небезпечно!
- Поіменовані канали
  - У POSIX – однобічні FIFO канали
  - У Win32 – двобічний обмін повідомленнями

Загальна модель файлової системи

- Символьний рівень
  - За символьним іменем файлу визначається його унікальне ім'я
- Базовий рівень
  - За унікальним іменем файлу визначаються його характеристики (атрибути)
  - Перевіряються права доступу
  - При відкриванні файлу його атрибути переміщаються в оперативну пам'ять
- Логічний рівень
  - Визначаються координати логічного запису у файлі
- Фізичний рівень
  - Визначається номер блока

Багаторівнева структура

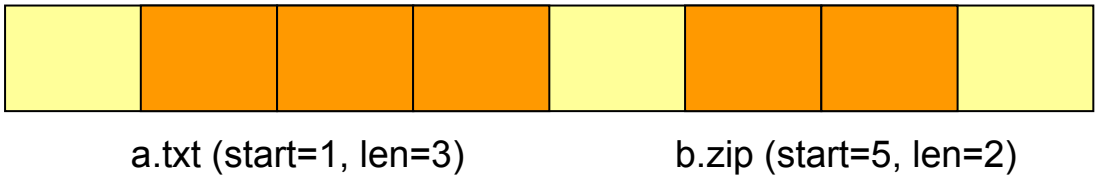


Фізична організація файлів

- Описує правила розміщення файлу на пристрої зовнішньої пам’яті
- Файл складається з фізичних одиниць – блоків
  - Блок – найменша одиниця, якою здійснюється обмін із зовнішнім пристроєм
    - У деяких файлових системах замість терміна “блок” вживають термін “кластер”
- Схеми фізичної організації
  - Неперервне розміщення файлів
  - Розміщення файлів зв’язними списками
    - Прості зв’язні списки
    - Зв’язні списки з таблицею розміщення файлів
  - Індексоване розміщення файлів

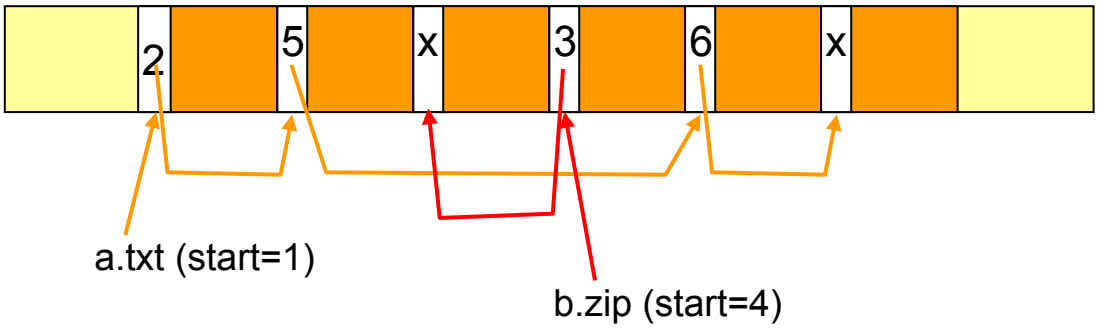
Неперервне розміщення

- Кожному файлу надають послідовність блоків диска, що утворюють єдину неперервну ділянку
- Переваги:
  - Простота та ефективність
  - Для знаходження будь-якого блоку достатньо задати лише перший блок
  - Дуже швидкий доступ
- Недоліки:
  - Під час створення файлу часто невідома його довжина, відповідно, невідомо, яку ділянку треба для нього зарезервувати
  - Велика зовнішня фрагментація
- Приклади реалізації:
  - HPFS (OS/2)
  - Файлова система компакт-дисків



Прості зв'язні списки

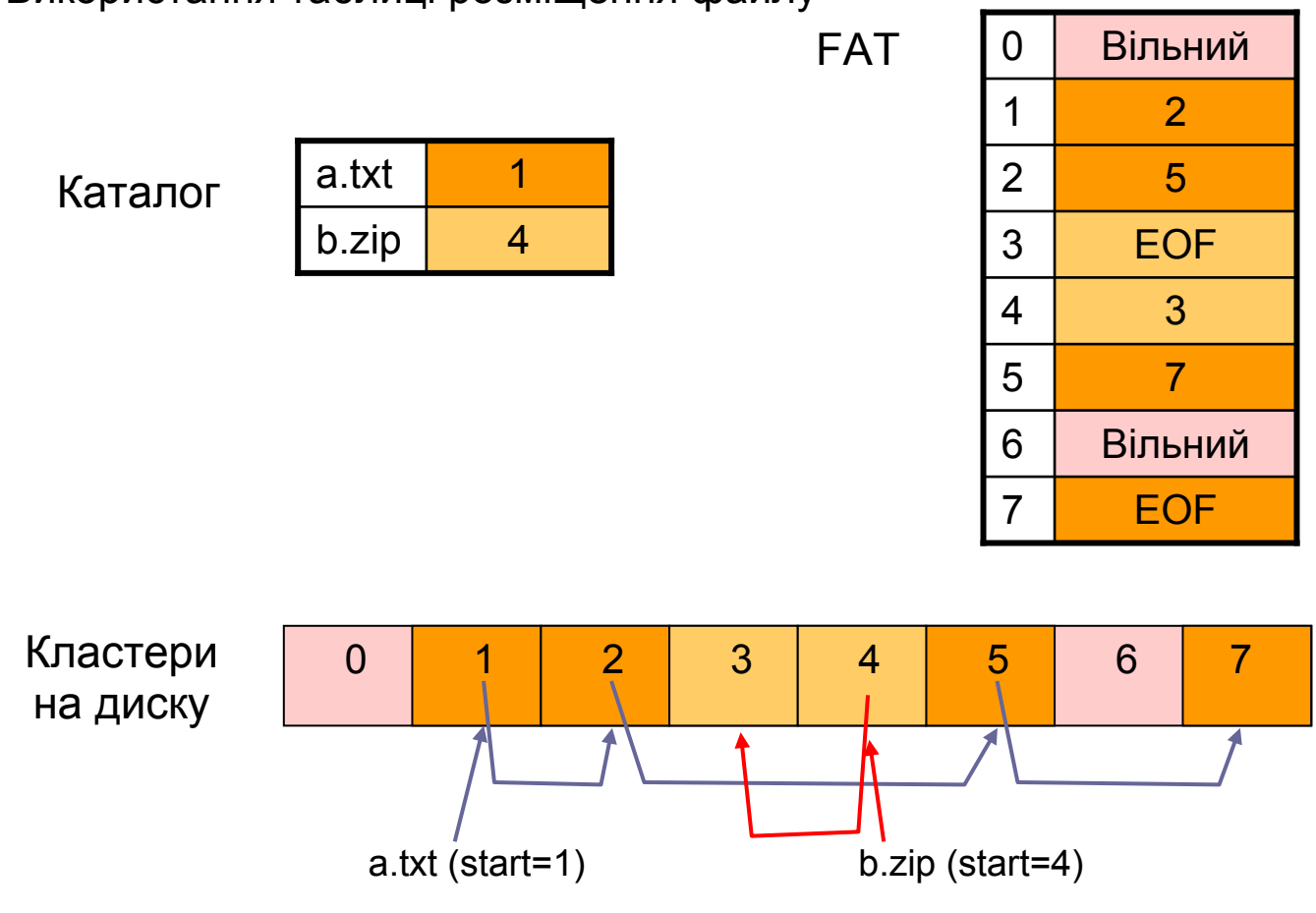
- На початку кожного блока міститься покажчик на наступний блок
- Переваги:
  - Достатньо задати початковий блок
  - Відсутня зовнішня фрагментація
  - Файл може змінювати довжину шляхом додавання блоків
- Недоліки:
  - Складність доступу до довільного блоку (необхідно прочитати усі попередні блоки)
  - Обсяг даних у блоці не дорівнює 2^n, оскільки частину блока виділяють для покажчика на наступний блок



Зв'язні списки з таблицею розміщення файлів

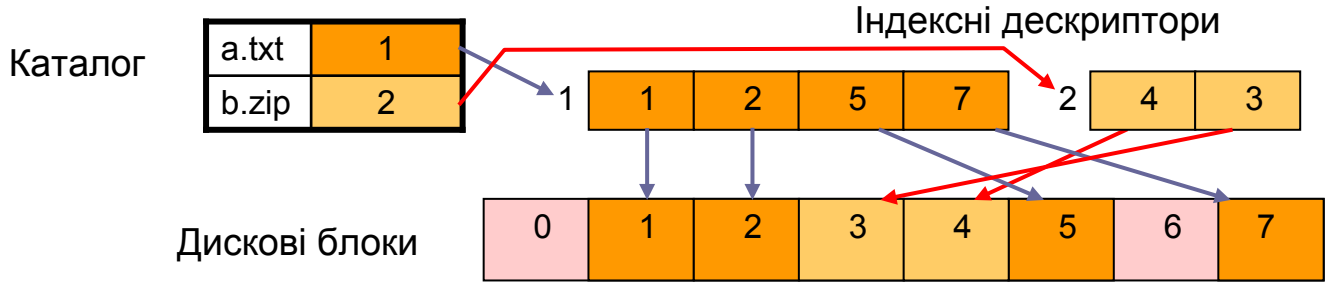
- Усі посилання на номери блоків зберігають в окремій ділянці файлової системи, формуючи *таблицю розміщення файлів* (*File Allocation Table*, FAT)
- Кожний елемент такої таблиці відповідає блоку (кластеру) на диску, і може містити:
  - Номер наступного кластера
  - Індикатор кінця файлу
  - Ознаку вільного кластера
- Переваги:
  - Для доступу до довільного кластеру зчитують не попередні кластери, а лише елементи FAT
  - Розміри FAT дозволяють кешувати її у оперативній пам'яті
- Обмеження:
  - Маленькі кластери –> багато елементів –> завеликий обсяг FAT
  - Великі кластери –> збільшення непродуктивних витрат дискового простору для малих файлів

Використання таблиці розміщення файлу



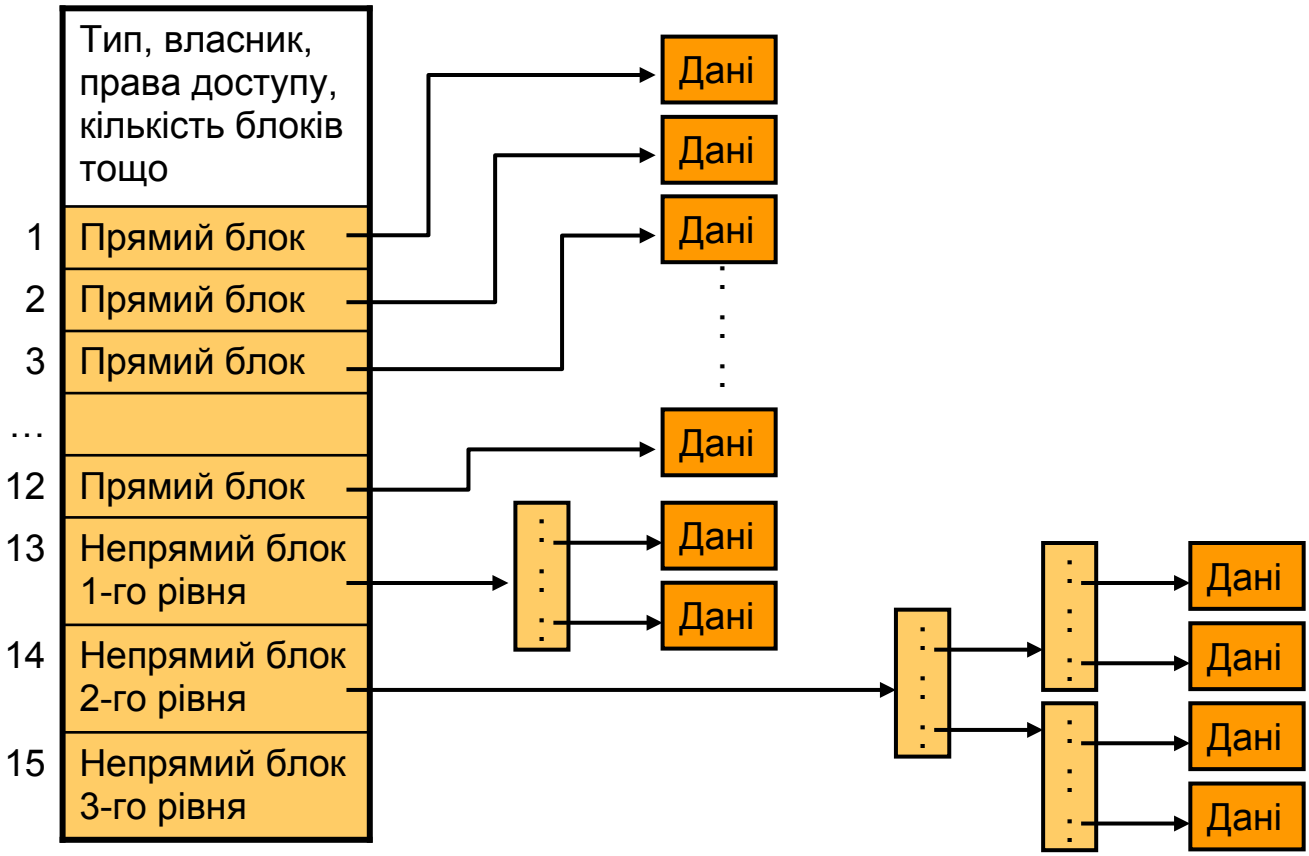
Індексоване розміщення

- У спеціальній структурі, пов'язаній з файлом (індексний дескриптор, i-node), безпосередньо описують розміщення усіх блоків файлу
  - У найпростішому варіанті індексний дескриптор – це масив, у якому перелічені адреси (номери) усіх блоків цього файлу
- Ефективно здійснюється як послідовний, так і випадковий доступ
  - Для підвищення ефективності індексний дескриптор повністю завантажують у пам'ять
- Основною проблемою є вибір розміру і структури індексного дескриптора



Приклад реалізації індексованого розміщення – файлові системи UNIX

- Перші 12 елементів індексного дескриптора безпосередньо вказують на дискові блоки з даними (прямі)
- 13-й елемент вказує на непрямий блок 1-го рівня, який містить масив адрес наступних блоків файлу
- 14-й елемент вказує на непрямий блок 2-го рівня, який містить масив адрес непрямих блоків 1-го рівня
- 15-й елемент вказує на непрямий блок 3-го рівня


Тип, власник, права доступу, кількість блоків тощо
1
Прямий блок
2
Прямий блок
3
Прямий блок
…
12
Прямий блок
13
Непрямий блок 1-го рівня
14
Непрямий блок 2-го рівня
15
Непрямий блок 3-го рівня
Дані
Дані
Дані
Дані
Дані
Дані
Дані
Дані
Дані

Лекція 13. Реалізація файлових систем

План лекції

- FAT
- ufs
- ext2fs
- ext3fs
- /proc
- VFS
- NTFS

Фізична організація FAT

- Логічний розділ, відформатований під файлову систему FAT, містить такі розділи:
  - Завантажувальний сектор (*boot sector*) містить програму початкового завантаження ОС
  - Основна копія FAT
  - Резервна копія FAT
  - Кореневий каталог (*root directory*) займає фіксовану ділянку у 32 сектора (16 кБ), що дозволяє зберігати 512 записів про файли і каталоги (кожний запис – 32 Б)
  - Область даних призначена для розміщення усіх файлів і усіх каталогів, крім кореневого каталогу

Особливості FAT

- Елемент (індексний покажчик) FAT може мати такі значення:
  - Кластер вільний
  - Кластер використовується (номер наступного кластера)
  - Останній кластер файлу
  - Дефектний кластер
  - Резервний
- Розрядність елементів
  - FAT12 – 12 біт – максимум 4096 кластерів
  - FAT16 – 16 біт – максимум 65536 кластерів
    - Максимальний розмір розділу 4 ГБ (кластер 64 кБ)
  - FAT32 – 32 біт – максимум $2^{32} = 4294967296$ кластерів
- Кожний запис у каталозі – 32 Б
  - Ім’я файлу – 11 Б у форматі 8.3
  - Довге ім’я файлу – до 255 двохбайтних символів Unicode
  - Довге ім’я поміщається порціями по 13 символів у записи, що безпосередньо йдуть за основним записом каталогу (кожний такий запис містить ще 6 Б службової інформації)

Фізична організація ufs (UNIX File System)

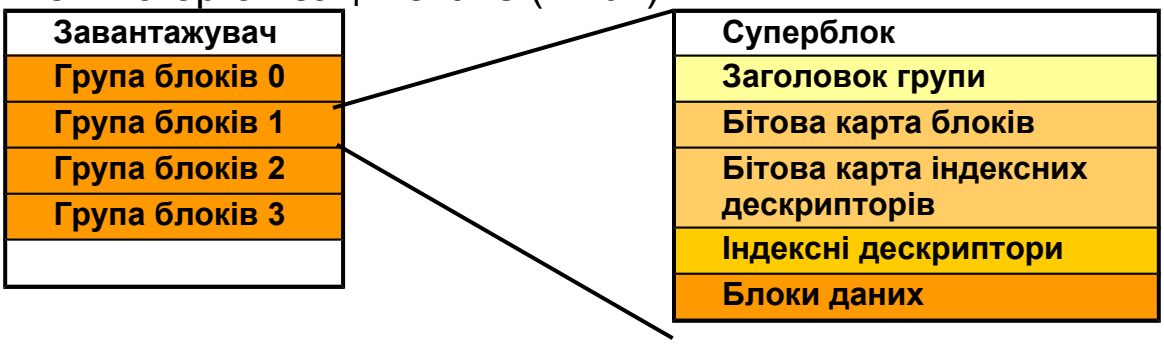
- Розділ містить завантажувальний блок, після якого кілька разів повторюється послідовність:
  - Суперблок – містить загальну інформацію про файлову систему

        - Розмір файлової системи
        - Розмір області індексних дескрипторів
        - Число індексних дескрипторів
        - Список вільних блоків
        - Список вільних індексних дескрипторів
    - Блок групи циліндрів – описує кількість індексних дескрипторів і блоків даних, що розташовані у цій групі циліндрів диска
    - Область індексних дескрипторів
        - Дескриптори розташовані за їхніми номерами
    - Область даних
        - Містить звичайні файли і каталоги (у тому числі кореневий каталог)
        - Спеціальні файли в області даних не відображені

Особливості ufs

- Індексний дескриптор містить
    - Ідентифікатор власника файлу
    - Тип файлу
    - Права доступу до файлу
    - Час останньої модифікації файлу, час останнього звернення до файлу, час останньої модифікації індексного дескриптора
    - Кількість посилань на цей дескриптор (імен файлу)
    - Адресна інформація (була розглянута нами раніше)
    - Розмір файлу в байтах
- Запис про файл у каталозі містить усього два поля: ім’я файлу і номер індексного дескриптора
    - Ім’я файлу в ufs може мати довжину до 255 символів (код ASCII – по одному байту на символ)
- Розмір індексного дескриптора – 64 кБ
- Розмір блоку – 4 або 8 кБ

Фізична організація ext2fs (Linux)



- Група блоків на відміну від групи циліндрів у ufs не прив'язана до геометрії диску
- При створенні файлу намагаються знайти для нього індексний дескриптор у тій групі блоків, де міститься його каталог
- За допомогою бітових карт ведуть облік вільних блоків і індексних дескрипторів
- Розмір бітової карти – 1 кБ, отже група може містити до 8192 блоків
- Розмір блоку – 1 кБ
- Розмір індексного дескриптора – 128 байт

Файлова система ext3fs (Linux)

- Відрізняється від ext2fs наявністю журналу
- Можуть бути задані три режими роботи з журналом
  - Режим журналу (*journal*)
    - Усі зміни даних зберігаються у журналі
    - Суттєво впливає на продуктивність
  - Упорядкований режим (*ordered*)
    - Зберігаються тільки зміни в метаданних
    - Блоки даних зберігаються на диску перед метаданними, тому ймовірність їх ушкодження низька
    - Це режим за умовчанням
  - Режим мінімального записування
    - Зберігаються лиш зміни в метаданних
- Журнал зберігають у схованому файлі **.journal** у кореневому каталозі

Файлова система /proc (UNIX, Linux)

- Це спеціальна файлова система, яка насправді взагалі не працює з диском
- Вміст кожного файлу і каталогу генерує програмне забезпечення у відповідь на запит файлової операції
- Кожному процесу відповідає каталог файлової системи **/proc**
  - Наприклад, процесу з **PID=25** відповідає каталог **/proc/25**
- Зчитування файлів цього каталогу надає певну інформацію про процес
  - **/proc/<pid>/cmdinfo** – вміст командного рядка
  - **/proc/<pid>/cpu** – відомості про поточне завантаження процесора
  - **/proc/<pid>/status** – різноманітна статистика
- У сучасних системах доступна також інша інформація
  - **/proc/modules** – завантажені модулі ядра
  - **/proc/mounts** – змонтовані файлові системи
  - **/proc/devices** – зовнішні пристрої
  - **/proc/cpuinfo** – процесори
  - **/proc/meminfo** – стан пам'яті
- **/proc/sys** і **/proc/sys/kernel** надають доступ до внутрішніх змінних ядра
  - Можна не лише зчитувати значення, а й записувати нові, змінюючи тим самим характеристики системи без перекомпіляції ядра і без перезавантаження системи

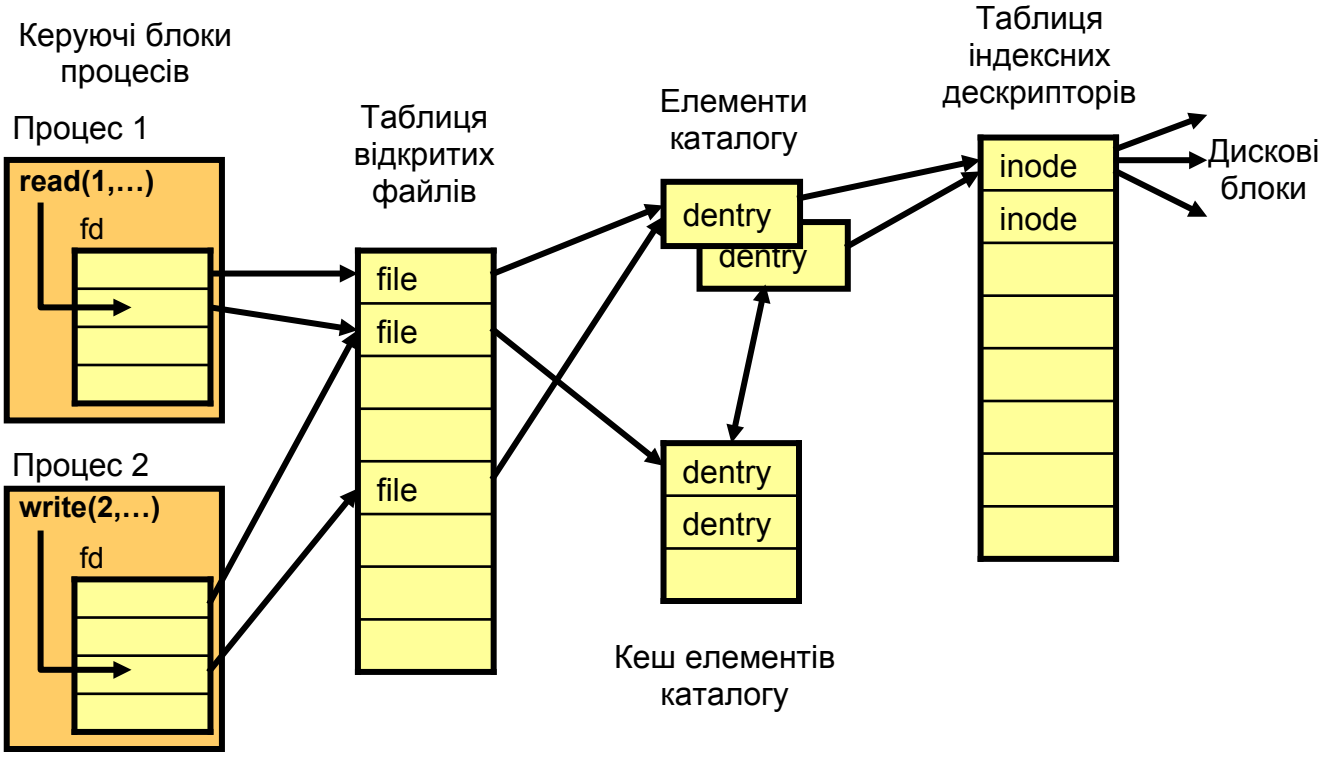
Віртуальна файлова система VFS

- Основною метою є забезпечення роботи ОС з максимально широким набором різних файлових систем
  - Дискові файлові системи
    - ext2fs, ext3fs, ReiserFS, ufs, xfs, FAT, NTFS, ISO9660
  - Мережні файлові системи
    - NFS, SMB
  - Спеціальні (віртуальні) файлові системи
    - /proc
- Рівень VFS забезпечує доступ через стандартні файлові системні виклики до будь-якого рівня програмного забезпечення, що реалізує інтерфейс файлової системи
  - Програмні модулі, що реалізують інтерфейс файлової системи, називаються *модулями підтримки файлових систем*

Об’єкти, що підтримує віртуальна файлова система VFS

- Об’єкт файлової системи (*filesystem object* або *superblock object*)
  - Відображає пов’язаний набір файлів, що міститься в ієрархії каталогів
  - Ядро підтримує по одному такому об’єкту для кожної змонтованої файлової системи
- Об’єкт індексного дескриптора (*inode object*)
  - Описує набір атрибутів і методів, за допомогою яких відображують файл на рівні файлової системи
  - Відображає файл як ціле
- Об’єкт відкритого файлу (*file object*)
  - Відображає відкритий файл на рівні процесу
  - Цей об’єкт створюють системним викликом відкриття файлу
  - В об’єкті зберігається режим доступу до файлу і покажчик поточної позиції
- Об’єкт елемента каталогу (*dentry object*) – у Linux
  - Відображає елемент каталогу і його зв’язок з файлом на диску
  - Знаходиться між об’єктами відкритих файлів і об’єктами індексних дескрипторів
  - Об’єкти елемента каталогу, як і об’єкти індексного дескриптора існують незалежно від процесів

Доступ до файлу з процесу у Linux



Файлова система NTFS

- Була розроблена як основна файлова система для ОС Windows NT на початку 90-х років
- Підтримує великі файли і великі диски (обсягом до 2^{64} байт)
- Стійка до відмов
- Забезпечує високу швидкість операцій
- Має гнучку структуру, що дозволяє додавання нових типів записів і атрибутів файлів
- Підтримує довгі символьні імена (до 255 символів Unicode)
- Забезпечує керування доступом до каталогів і файлів

Структура тому NTFS

- Основою структури є *головна таблиця файлів* (*Master File Table*, MFT)
  - MFT містить щонайменше 1 запис для кожного файлу тому (у тому числі для самої себе), розмір запису однаковий для усього тому і може бути 1, 2 або 4 кБ (типово 2 кБ)
  - Усі файли тому ідентифікуються номером файлу, який визначається його позицією в MFT
- Увесь том є послідовністю кластерів (а не лише область даних)
  - Порядковий номер кластера у томі називають *логічним номером кластера* (Logical Cluster Number, LCN)
  - Номери кластерів зберігаються у 64-розрядних покажчиках
- Базова одиниця розподілу дискового простору в NTFS – це не один кластер, а неперервна область кластерів, яку називають *відрізком*, або *екстентом*
  - Екстент задається логічним номером його першого кластера і кількістю кластерів в екстенті
- Завантажувальний сектор містить
  - Стандартний блок параметрів BIOS
  - Кількість кластерів у томі
  - Початковий логічний номер кластера основної копії MFT і дзеркальної копії MFT
- Копія завантажувального сектора знаходиться у середині тому
- Перший екстент MFT містить 16 стандартних записів, які створюють під час форматування, про системні файли NTFS
- Нульовий запис MFT містить опис самої MFT, зокрема, адреси усіх її екстентів
  - Безпосередньо після форматування MFT має один екстент, за яким записані системні файли
  - В момент створення першого не системного файлу створюють другий екстент MFT

Завантажувальний сектор	
0	
1	
2	**1-й екстент MFT**
…	
15	
Системний файл 1	
Системний файл 2	
…	
Системний файл n	
Копія MFT (перші 3 записи)	
Файли	
Копія завантажувального сектора	
2-й екстент MFT	
Файли	
3-й екстент MFT	

Системні файли NTFS

0	$Mft	Головна таблиця файлів
1	$MftMirr	Резервна копія перших 16 записів MFT
2	$LogFile	Файл журналу
3	$Volume	Ім’я тому, версія NTFS та інша інформація про том
4	$AttrDef	Таблиця визначення імен, номерів і описів атрибутів
5	$.	Кореневий каталог
6	$Bitmap	Бітова карта кластерів тому, що описує вільні й зайняті кластери
7	$Boot	Адреса завантажувального сектора розділу
8	$BadClus	Список зіпсованих кластерів тому
9	$Quota	Квоти дискового простору для кожного користувача
10	$Upcase	Таблиця перетворення регістру символів для Unicode
11 – 15		Зарезервовані

Структура файлу NTFS

- Файл складається з набору *атрибутів*
  - Кожний з атрибутів є незалежним *потоком* (*stream*) байтів, який можна створювати, вилучати, зчитувати та записувати
- Деякі атрибути є стандартними для усіх файлів
  - Ім'я (*FileName*)
  - Версія (*Version*)
  - Стандартна інформація про файл, що включає час створення, час відновлення тощо (*Standard Information*)
- Інші атрибути залежать від призначення файлу
  - Каталог має атрибут *Index Root* – структуру даних, що містить список файлів цього каталогу
- Атрибут *Data* містить усі дані файлу
  - В NTFS файл може містити більше одного атрибуту *Data*, які розрізняють за іменами (дозволена наявність кількох поіменованих потоків даних всередині файлу)
  - За умовчанням файл містить один потік даних, що не має імені
- Атрибути невеликого розміру, які зберігають безпосередньо в запису MFT, називають *резидентними*
  - Атрибут *Data* може бути резидентним!
- Атрибути великого розміру зберігають на диску окремо, їх називають *нерезидентними*

Розміщення на диску файлів NTFS

- Файли малого розміру (*small*)
  - Складаються щонайменше з таких атрибутів:
    - Стандартна інформація (SI – standard information)
    - Ім’я файлу (FN – file name)
    - Дані (Data)
    - Дескриптор захисту (SD – security descriptor)
  - Можуть бути повністю розміщені у запису MFT (усі атрибути резидентні)
- Великі файли (*large*)
  - Атрибут Data є нерезидентним, у його резидентній частині зберігають покажчики на усі екстенти
- Дуже великі файли (*huge*)
  - Якщо файл має багато атрибутів або сильно фрагментований, атрибут Data не вміщує усіх покажчиків на екстенти
  - У такому випадку файл займає кілька записів MFT
    - Основний запис називають базовим записом файлу, а решту – записами переповнення
- Надвеликі файли (*extremely huge*)
  - У базовому записі може не вистачати місця для записів переповнення
  - Атрибут Attribute List можна зробити нерезидентним
  - Можна застосовувати подвійну непряму адресацію (нерезидентний атрибут посилається на інші нерезидентні атрибути)

Лекція 14. Концепція розподіленого оброблення інформації

План лекції

- Обмін повідомленнями як єдиний спосіб керування розподіленими ресурсами
- Базові примітиви обміну повідомленнями

Особливості керування розподіленими ресурсами

- Найважливіша відмінність від централізованих систем – організація взаємодії між процесами
- Концептуально можливі лише два способи організації взаємодії між процесами:
  - За допомогою спільного використання одних і тих самих даних (спільна пам’ять)
  - За допомогою обміну повідомленнями
- У централізованих системах практично завжди застосовують спільну (поділювану) пам’ять
  - Семафор – окремий випадок спільної пам’яті (усі потоки звертаються до однієї змінної)
- У розподілених системах фізична спільна пам’ять відсутня
  - Тому єдина можливість – обмін повідомленнями
  - Повідомлення – це блок інформації, сформований процесом-відправником таким чином, щоби він був зрозумілий процесу-одержувачу

Формати повідомлень

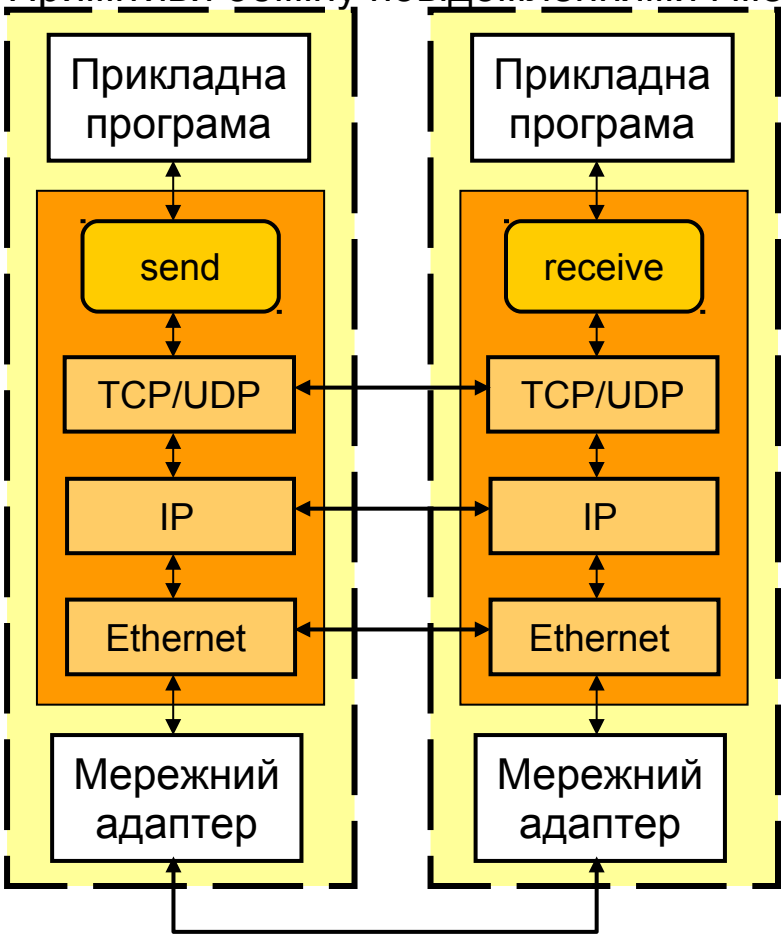
- Повідомлення як правило має:
  - Заголовок
  - Тіло
- Заголовок часто (але не завжди) має фіксовану довжину і містить дані чітко визначених типів і чітко визначеного обсягу
- Заголовок як правило містить такі елементи:
  - Адреса – набір символів, що однозначно ідентифікує процес-відправник і процес-одержувач
  - Номер, який є ідентифікатором повідомлення
  - Ідентифікатор типу даних, що містяться у повідомленні
  - Поле, яке визначає обсяг даних, що містяться у повідомленні
  - Інформація, яка дозволяє перевірити цілісність даних (контрольна сума)
- Повідомлення може не мати форматного заголовка. Тоді воно містить структуровану інформацію, яка у загальному випадку складається з таких полів:
  - Тип даних
  - Довжина даних

- Значення даних

Базові примітиви обміну повідомленнями

- Будь-яка мережна ОС має підсистему обміну повідомленнями, яку також називають транспортною підсистемою
  - Її завдання, як і інших підсистем ОС – приховати від прикладних програм деталі складних мережних протоколів і надати прикладним програмам абстракції у вигляді простих примітивів обміну повідомленнями
- У найпростішому випадку системні засоби обміну повідомленнями можуть бути зведені до двох базових примітивів:
  - **send** (відправити)
  - **receive** (отримати)
- На основі примітивів будують більш потужні засоби мережних комунікацій, наприклад, такі як:
  - Розподілена файлова система
  - Служба виклику віддалених процедур

Примітиви обміну повідомленнями і мережні протоколи



- Виконання примітивів **send** і **receive** спирається на виконання усіх протоколів стеку, які лежать нижче, відповідно до моделі взаємодії відкритих систем (OSI)

- Наприклад, для стеку TCP/IP
  - Примітив send звертається до засобів транспортного рівня (протокол TCP або UDP)
  - Той у свою чергу звертається до засобів мережного рівня (протокол IP)
  - Той звертається до засобів нижнього рівня (наприклад, драйвера адаптера Ethernet)
  - Драйвер керує апаратним засобом (мережним адаптером)
  - Примітив receive на іншому боці приймає повідомлення від засобів транспортного рівня після того, як воно було послідовно оброблено усіма нижчими рівнями у зворотному порядку

Варіанти реалізації базових примітивів

- Від реалізації базових примітивів залежить ефективність роботи мережі
- Основні питання, на які треба дати відповіді:
  - Як задають адресу одержувача?
  - Одержувач конкретного повідомлення завжди один, чи їх може бути кілька?
  - Чи необхідні гарантії доставки повідомлень?
  - Чи повинен відправник дочекатись відповіді на повідомлення перед тим, як продовжити роботу?
  - Як відправник, одержувач і комунікаційна підсистема повинні реагувати на відмови вузлів і каналів мережі?
  - Що робити, якщо приймач не готовий прийняти повідомлення? Чи відкинути його, чи зберегти у буфері?
  - Що робити, коли буфер переповнений?
  - Чи дозволено приймачу змінювати порядок оброблення повідомлень відповідно до їх важливості?
- Відповіді на ці запитання складають семантику конкретного протоколу передавання повідомлень

Синхронізація

- Спосіб синхронізації процесів у мережі цілком залежить від реалізації базових примітивів обміну повідомленнями, які бувають:
  - Блокуючі (синхронні)
    - Процес, що викликав примітив send, призупиняється до отримання з мережі підтвердження, що адресат отримав повідомлення
    - Процес, що викликав примітив receive, призупиняється до моменту отримання повідомлення
    - Якщо в процесі взаємодії процесів обидва примітиви є блокуючими, кажуть, процеси взаємодіють синхронно, в іншому випадку взаємодію вважають асинхронною
  - Неблокуючі (асинхронні)
    - Після виклику примітиву, керування повертається процесу миттєво
    - У процесі виклику ядру передається інформація про буфер, з якого треба взяти повідомлення для відправлення в мережу, або у який треба покласти повідомлення, що надійшло з мережі

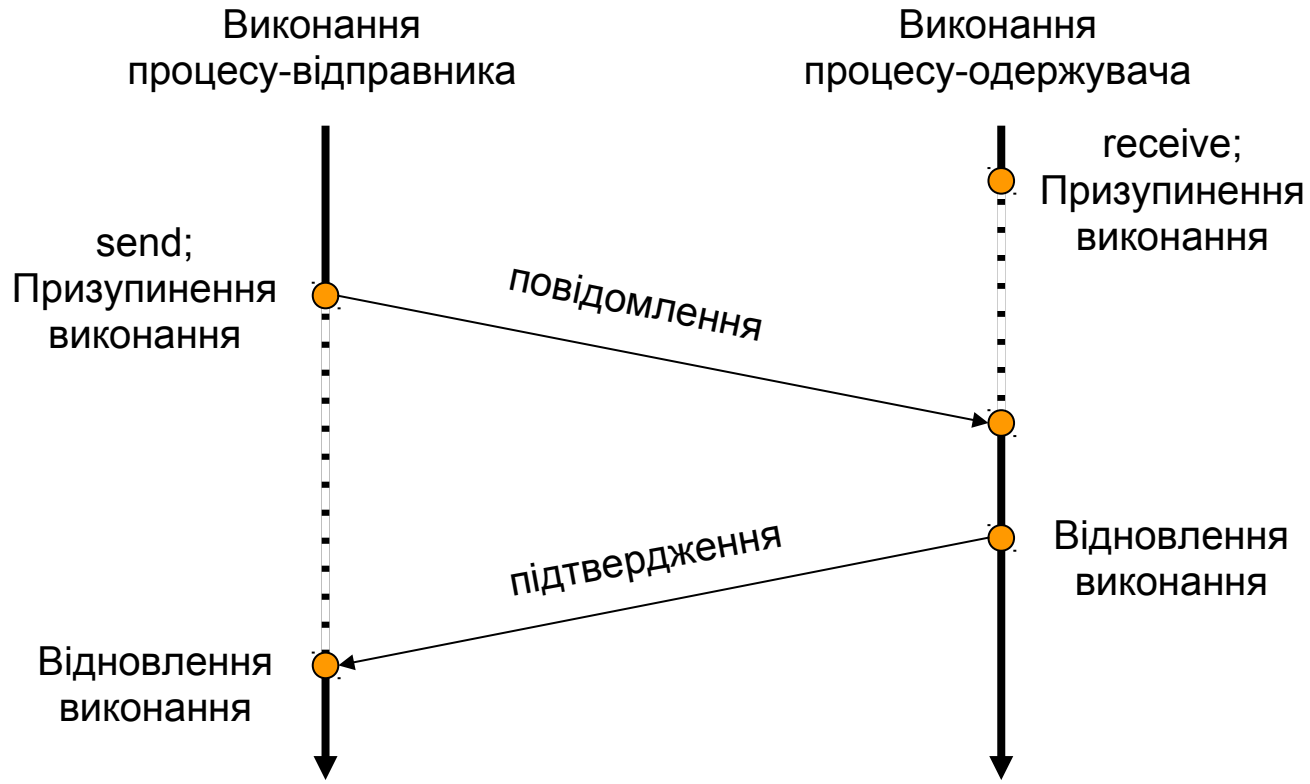
Тайм-аути у блокуючих примітивах

- Проблема блокуючих примітивів: процес може бути заблокованим назавжди
  - Це рівною мірою стосується і процесу, що здійснив виклик send, і процесу, що здійснив виклик receive
  - Блокування може статись, якщо
    - Повідомлення було втрачено в мережі внаслідок помилки
    - Інший процес-учасник обміну потерпів крах
- Для запобігання таким ситуаціям впроваджують механізм тайм-аутів
  - Задають інтервал часу, протягом якого триває очікування повідомлення (або підтвердження)
  - Якщо повідомлення (або підтвердження) не отримують протягом цього інтервалу, виклик завершується зі статусом “помилка”

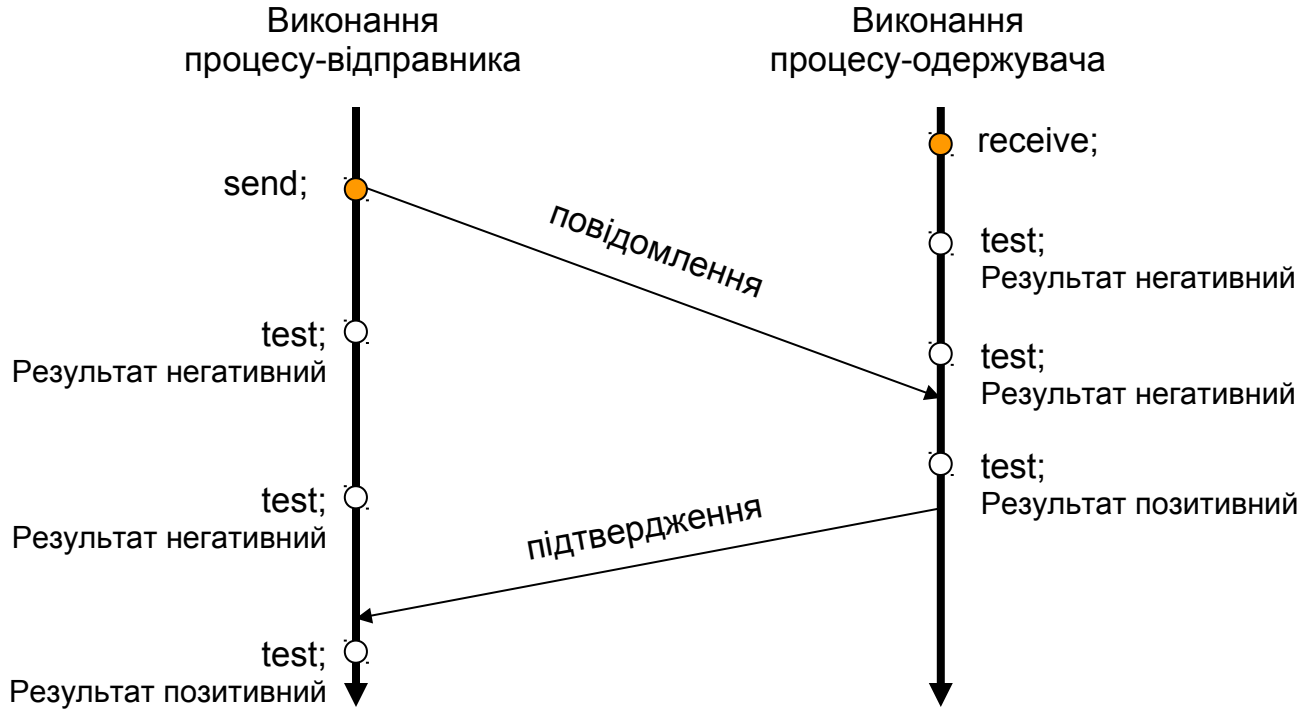
Особливості неблокуючих примітивів

- Великою перевагою неблокуючих примітивів є паралельне виконання процесу, що викликав примітив, і процедур, які цей примітив реалізують
  - Ці процедури не обов’язково працюють у контексті процесу, що їх викликав
- Проблемою реалізації неблокуючих примітивів є робота з буфером повідомлень
  - Наприклад, необхідно реалізувати повідомлення процесу-одержувача про те, що повідомлення надійшло і знаходиться у буфері
  - Застосовують один із двох способів:
    - Опитування (*polling*). Для цього вводять додатковий базовий примітив test (перевірити)
    - Переривання (*interrupt*). Застосовують програмне переривання процесу-отримувача повідомлення. Перевагою є підвищена ефективність, недоліком – ускладнене програмування

Синхронна взаємодія



Асинхронна взаємодія



Буферизація у примітивах обміну повідомленнями

- Примітиви бувають із застосуванням буферизації або без неї
  - (рос. – «буферизуемые» і «небуферизуемые»)
- Асинхронні примітиви завжди вимагають буферизацію
  - Для виключення втрат даних необхідним є буфер необмеженої довжини
  - Оскільки на практиці буфер завжди має обмежену довжину, можливі ситуації з переповненням буфера
    - Вихід – у керуванні потоком повідомлень (тобто, певне обмеження асинхронності)
- Синхронні примітиви можуть обходитись без буферизації взагалі
  - Повідомлення потрапляє у мережу безпосередньо з пам'яті процесу-відправника, а після одержання його з мережі – у пам'ять процесу-одержувача
  - На практиці для використання синхронних примітивів все ж передбачають буферизацію, обираючи буфер розміром в одне повідомлення

Примітив створення буфера

- Зазвичай ОС надає прикладним програмам спеціальний примітив для створення буферів **create_buffer**
- Процес повинен використовувати такий примітив перед відправленням і одержанням повідомлень
  - Тобто, перед **send** і **receive**
  - Часто буфер, що створений таким примітивом, називають порт (port) або поштова скринька (*mailbox*)
- Необхідно вирішити питання, що робити з одержаними повідомленнями, для яких буфер створено не було
  - Наприклад, якщо send на одному комп'ютері виконали раніше, ніж **create_buffer** на іншому
  - Один варіант – просто відмовитись від повідомлення
    - Перевага – простота
  - Інший варіант – помістити "неочікуване" повідомлення у спеціальний системний буфер на визначений час, очікуючи на виконання **create_buffer**
    - Недолік – проблема підтримання системного буфера

Надійні і ненадійні примітиви

- Повідомлення у мережі можуть втрачатись
- Три підходи до того, що з цим робити:
  - Система не бере на себе зобов'язань щодо доставки повідомлень (дейтаграмний режим)
    - Реалізація надійної доставки – турбота прикладного програміста
  - Ядро ОС одержувача повідомлень надсилає квитанцію-підтвердження на кожне повідомлення або групу повідомлень
    - У разі синхронних примітивів ОС розблоковує процес-відправник лише після одержання квитанції
    - Обробленням квитанцій займається підсистема обміну повідомленнями, прикладним процесам вони взагалі не видимі
  - В якості підтвердження використовується відповідь
    - Застосовується у тих системах, де передбачається відповідь на кожний запит (характерно для архітектури клієнт-сервер)
- Надійність передавання повідомлень може також передбачати гарантії упорядкованості повідомлень

Способи адресації

- Адреси у вигляді констант
  - Можна застосовувати у дуже простій мережі
- Апаратні адреси мережних адаптерів
  - Адресують адаптер, але не процес
  - Важко знайти адресата у складній структурованій мережі
- Адресація машина-процес
  - Можна адресувати процеси за їх унікальними ідентифікаторами
  - Частіше адресують не процеси, а служби – за добре відомими номерами
  - Приклад – IP адреса і № порту
- Символьні адреси замість числових
  - Значно підвищують прозорість адресації
  - Приклад – система доменних імен
  - Перетворення символьних адрес на числові здійснюється
    - Або із застосуванням широкомовних запитів
    - Або із застосуванням централізованих служб

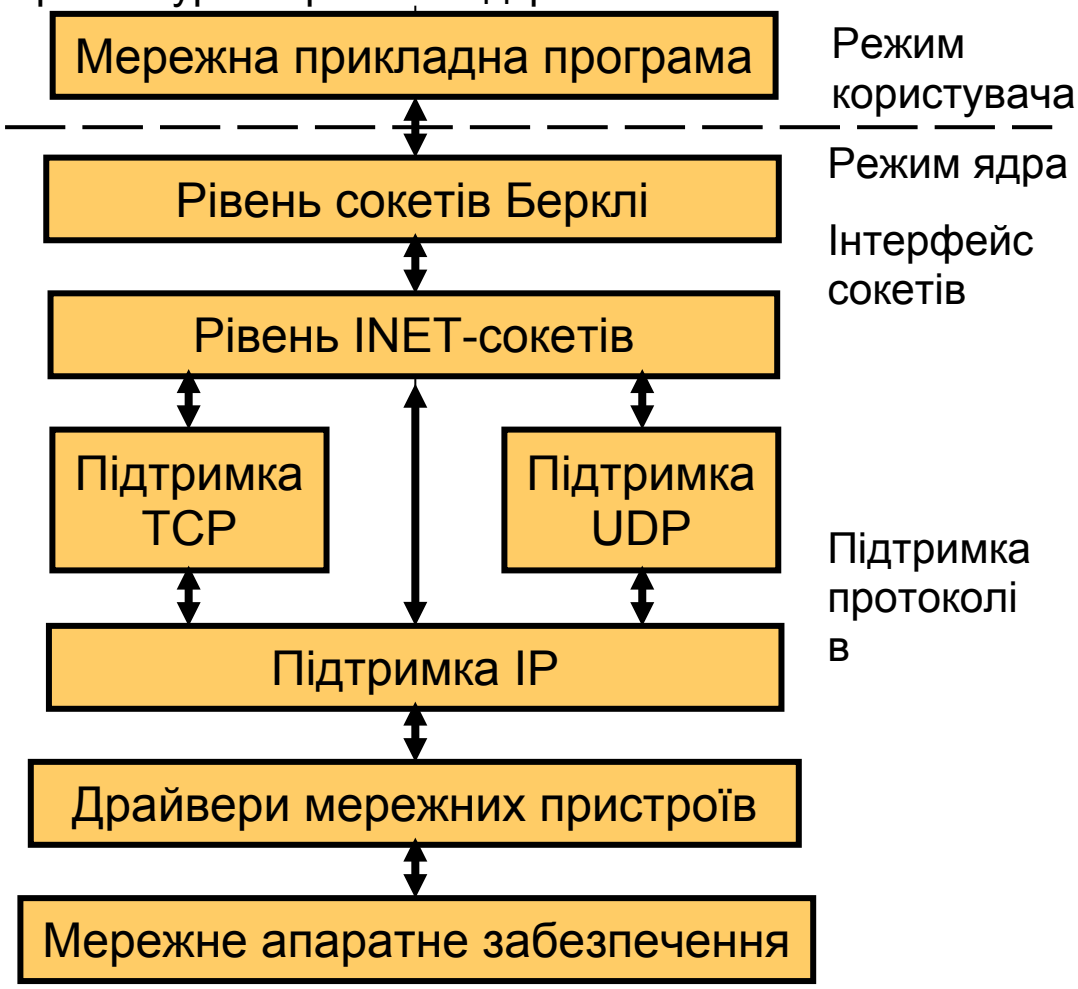
Механізм сокетів (sockets)

- Механізм сокетів є найпоширенішою системою обміну повідомленнями у мережі
  - Вперше був розроблений для версії 4.3 BSD UNIX (дотепер називають Berkeley Sockets – сокети Берклі)
  - Реалізацію для Windows називають Windows Sockets (WinSock)
- Програмний інтерфейс сокетів Берклі – це засіб зв'язку між прикладним рівнем мережної архітектури TCP/IP і транспортним рівнем
  - Фактично, це засіб зв'язку між кодом прикладної програми (що реалізує прикладний рівень) і реалізацією стека TCP/IP в ядрі ОС
  - Таким чином, інтерфейс сокетів – це набір системних викликів ОС

Основні концепції механізму сокетів

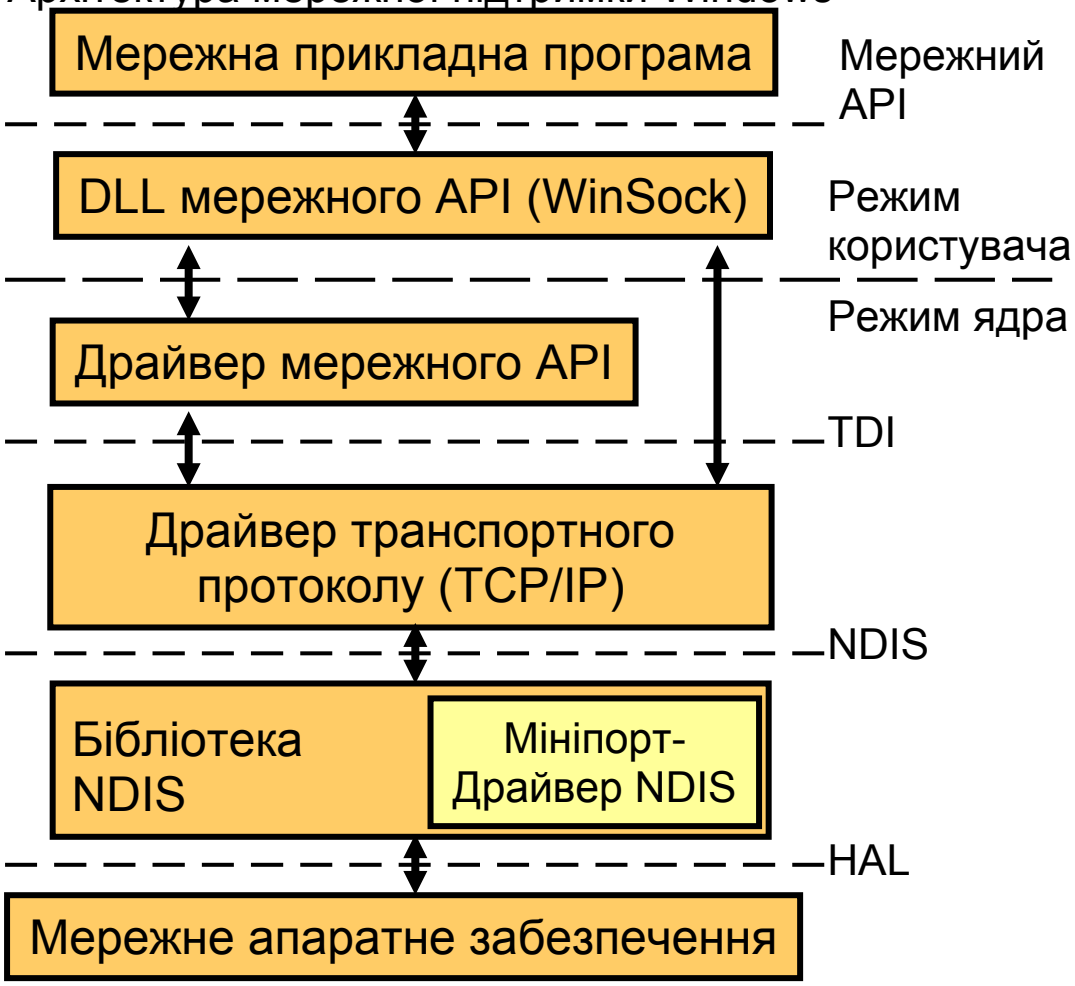
- Застосування абстрактної кінцевої точки з'єднання
  - Саме ця точка має назву *сокет* (*socket* – гніздо)
  - З'єднання між двома процесами здійснюється через пару сокетів
- Незалежність від мережних протоколів і технологій
  - Застосовується поняття *комунікаційний домен* (*communication domain*), який характеризується певним набором властивостей
  - Приклади – домен Інтернету (протоколи стеку TCP/IP), домен UNIX, або локальний домен (комп'ютер з його файловою системою)
- Сокет може мати як високорівневе символьне ім'я (адресу), так і низькорівневе
  - У домені Інтернету високорівневе ім'я – URL, низькорівневе – IP-адреса + порт
  - У домені UNIX ім'я сокету – це ім'я файлу
- Для кожного комунікаційного домену можуть існувати сокети різних типів
  - Наприклад, дейтаграмні (datagram) сокети і потокові (stream) сокети
  - Потокові з'єднання гарантують надійну упорядковану доставку

Архітектура мережної підтримки Linux



- Ядро розрізняє два рівня підтримки інтерфейсу сокетів
- На верхньому – інтерфейс сокетів Берклі, реалізований за допомогою універсальних *об'єктів сокетів Берклі*
  - Об'єкти реалізовані як файли, і містять:
    - тип сокета,
    - стан сокета,
    - універсальні операції сокетів,
    - покажчик на відповідний об'єкт INET-сокета
- На нижньому – реалізації інтерфейсу сокетів, що залежать від конкретної мережної архітектури
  - Наприклад, INET-сокети

Архітектура мережної підтримки Windows



- Драйвери транспортних протоколів надають драйверам мережних API універсальний інтерфейс транспортного драйвера (*Transport Driver Interface*, TDI)
- Мініпорт-драйвери NDIS відповідають за взаємодію драйверів транспортних протоколів і мережного апаратного забезпечення
  - Наприклад, драйвери конкретних мережних адаптерів
  - NDIS – *Network Driver Interface Specification* (специфікація інтерфейсу мережного драйвера)

Лекція 15. Виклик віддалених процедур Remote Procedure Call (RPC)

План лекції

Концепція віддаленого виклику процедур

- Добре відомий механізм передачі керування і даних всередині програми, що виконується на одній машині, поширюється на передачу керування і даних через мережу
- Найбільша ефективність RPC – коли існує інтерактивний зв'язок між віддаленими компонентами з
  - невеликим часом відповідей
  - відносно малим обсягом даних, що передають
- Характерні риси локального виклику процедур
  - Асиметричність – одна із сторін є ініціатором взаємодії
  - Синхронність – процедуру, яка викликає, блокують до повернення з процедури, яку викликали
- Ідея полягає в тому, щоби віддалений виклик процедур виглядав для прикладної програми точно так, як виклик локальної процедури

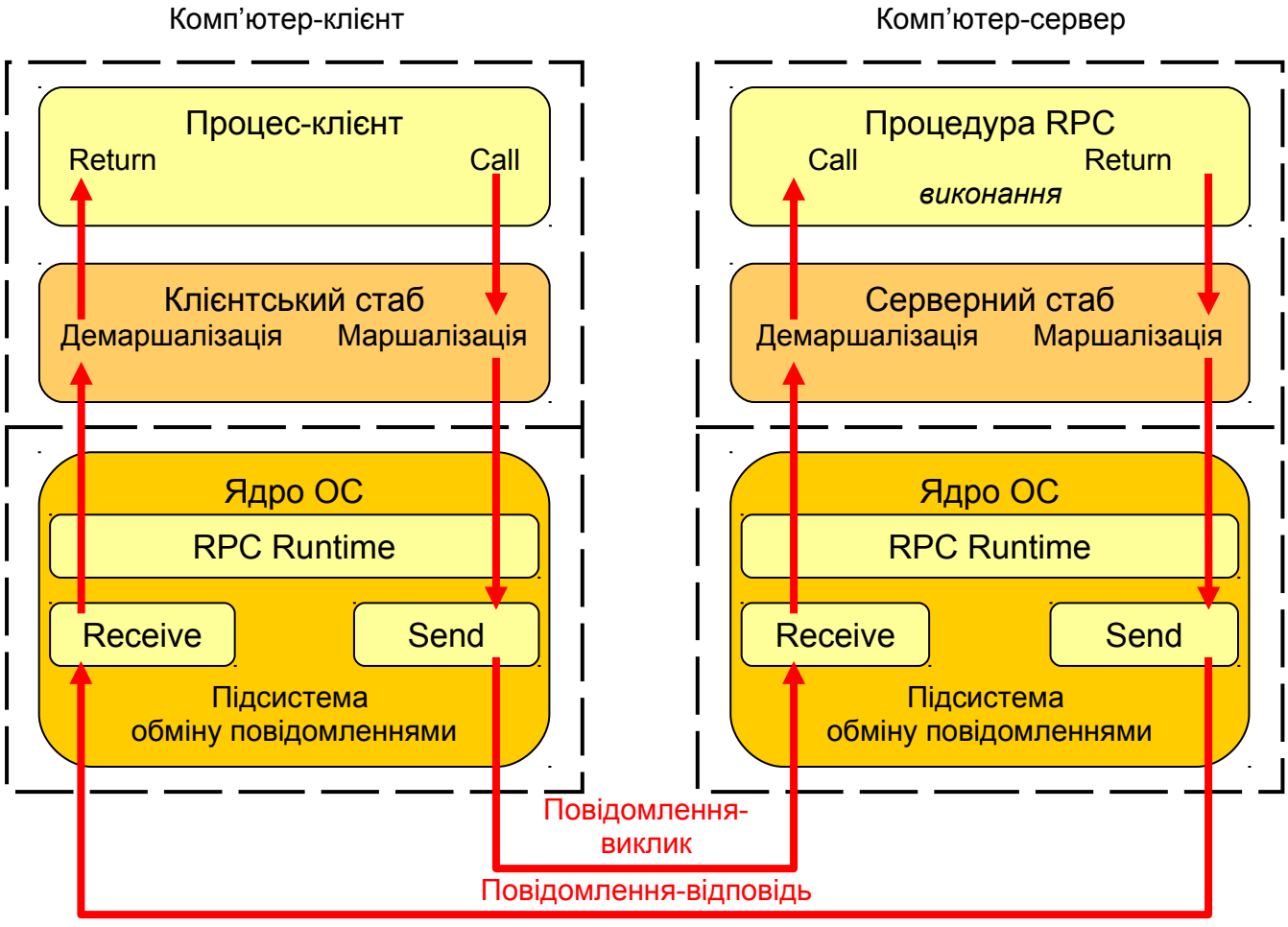
Можливі проблеми реалізації RPC

- Процедури виконуються на різних машинах, вони мають різні адресні простори, і не мають спільної пам'яті
  - Параметри не повинні містити покажчиків на пам'ять (в тому числі на стек)
  - Значення параметрів виклику слід передавати з одного комп'ютера на інший
- RPC обов'язково використовує систему обміну повідомленнями
  - Необхідно забезпечити прозорість RPC для прикладних програм
- Існує можливість аварійного завершення одного з процесів без повідомлення про це іншого, а також можливість втрати повідомлень у мережі
  - У разі краху програми, що викликає, віддалені процедури стають "сиротами"
  - У разі краху віддаленої процедури, програма, яка їх викликала, стає "обездоленою"
- Існують розбіжності у форматах подання чисел у різних архітектурах, у порядку параметрів викликів, у порядку байтів, у кодуваннях символів

Досягнення прозорості RPC

- В бібліотеку процедур на клієнтському комп'ютері замість коду процедури поміщають так званий *стаб* (*stub* – заглушка)
  - Клієнтський стаб викликають шляхом звичайної передачі параметрів через стек
- На комп'ютер-сервер поміщають оригінальний код процедури, а також серверний стаб
- Призначення клієнтського та серверного стабів – організувати передачу параметрів виклику процедури і повернення результату через мережу
  - Клієнтський стаб формує повідомлення, що містить ім'я процедури і параметри виклику (упаковка, або маршалізація повідомлення)
  - Серверний стаб отримує повідомлення, розпаковує (демаршалізує) параметри, і здійснює звичайний локальний виклик процедури
- Стаби використовують системні засоби обміну повідомленнями (send і receive)
  - Іноді у підсистемі обміну повідомленнями виділяють окремий програмний модуль для організації зв'язку стабів з примітивами обміну повідомленнями (RPC Runtime)

Виконання віддаленого виклику процедури

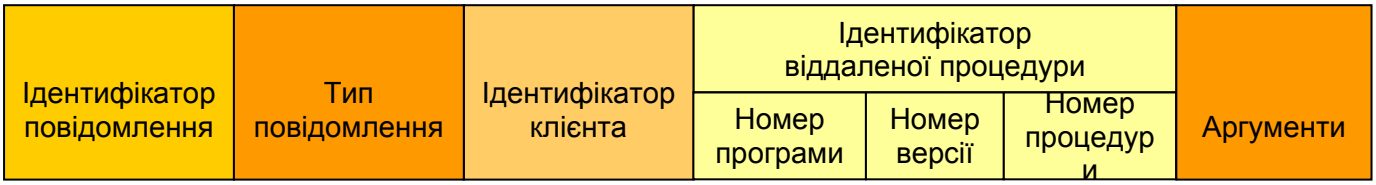


Генерація стабів

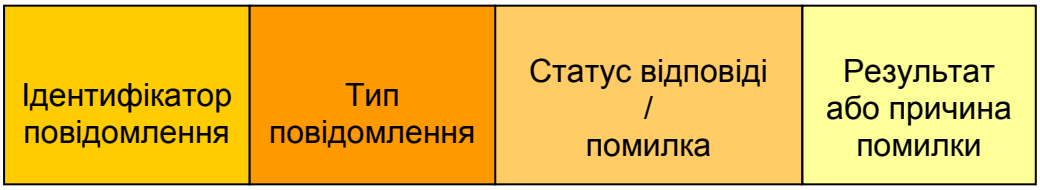
- Ручна генерація стабів
  - Програміст використовує ряд допоміжних функцій, наданих розробниками засобів RPC
  - Є свобода вибору способу передачі параметрів виклику і застосування тих чи інших примітивів передачі повідомлень
  - Вимагає значного обсягу ручної праці
- Автоматична генерація стабів
  - Застосовується мова визначення інтерфейсу (Interface Definition Language, IDL)
  - Опис інтерфейсу між клієнтом і сервером RPC містить список імен процедур, а також список типів аргументів і результатів цих процедур
  - Опис достатній для перевірки стабом типів аргументів і генерації повідомлення-виклику
  - IDL-компілятор створює вихідні модулі клієнтських і серверних стабів, а також генерує файли-заголовки з описом типів процедур і їхніх аргументів

Формат повідомлень RPC

Повідомлення-виклик



Повідомлення-відповідь



Зв’язування клієнта з сервером

- Процедуру, що встановлює відповідність між клієнтом і сервером RPC, називають зв’язуванням (*binding*)
- У різних реалізаціях RPC, можуть відрізнятись:
  - Способи завдання сервера, з яким хотів би бути зв’язаним клієнт
  - Способи визначення мережної адреси (місцезнаходження) потрібного сервера
  - Стадії, на якій відбувається зв’язування
- Статичне зв’язування:
  - Ім’я або адреса RPC-сервера задається явно
  - Відсутня гнучкість і прозорість
- Динамічне зв’язування:
  - Ім’я інтерфейсу RPC задається у вигляді **<тип інтерфейсу><екземпляр інтерфейсу>**
  - Тип визначає усі характеристики, крім місцезнаходження
  - Якщо клієнту важливий лише тип інтерфейсу, то процес знаходження сервера може застосовувати або широкомовні запити, або централізованого агента зв’язування