1. Системні виклики

Базові тези про системні виклики (повторення)

- Найчастіше системні виклики написані мовами C, C++, Assembler.
- Системні виклики скрізь:
  - для роботи з файлами (відкриття, закриття, читання, запис, створення, вилучення, читання/запис атрибутів тощо);
  - для роботи з пристроями (приклади: вивести щось на екран, зчитати введену користувачем команду, ...);
  - для роботи з процесами (створення, завершення, виділення/вивільнення пам'яті, читання/запис атрибутів тощо);
  - для обміну повідомленнями всередині системи і т. д.
- Логіка роботи системного виклику - як у звичайної підпрограми (функції):
  - передавання параметрів і керування - у підпрограму;
  - виконання підпрограми;
  - повернення керування і результатів у точку виклику.
- На відміну від звичайної підпрограми, відбувається перемикання між режимами користувача і ядра. Це потребує більше часу.

1. Системні виклики

API (Application Programming Interface)

Надає прикладному програмісту функції, типи та структури даних, константи тощо для організації взаємодії прикладної програми з ОС.

API існують не лише для ОС, а й для багатьох інших систем.

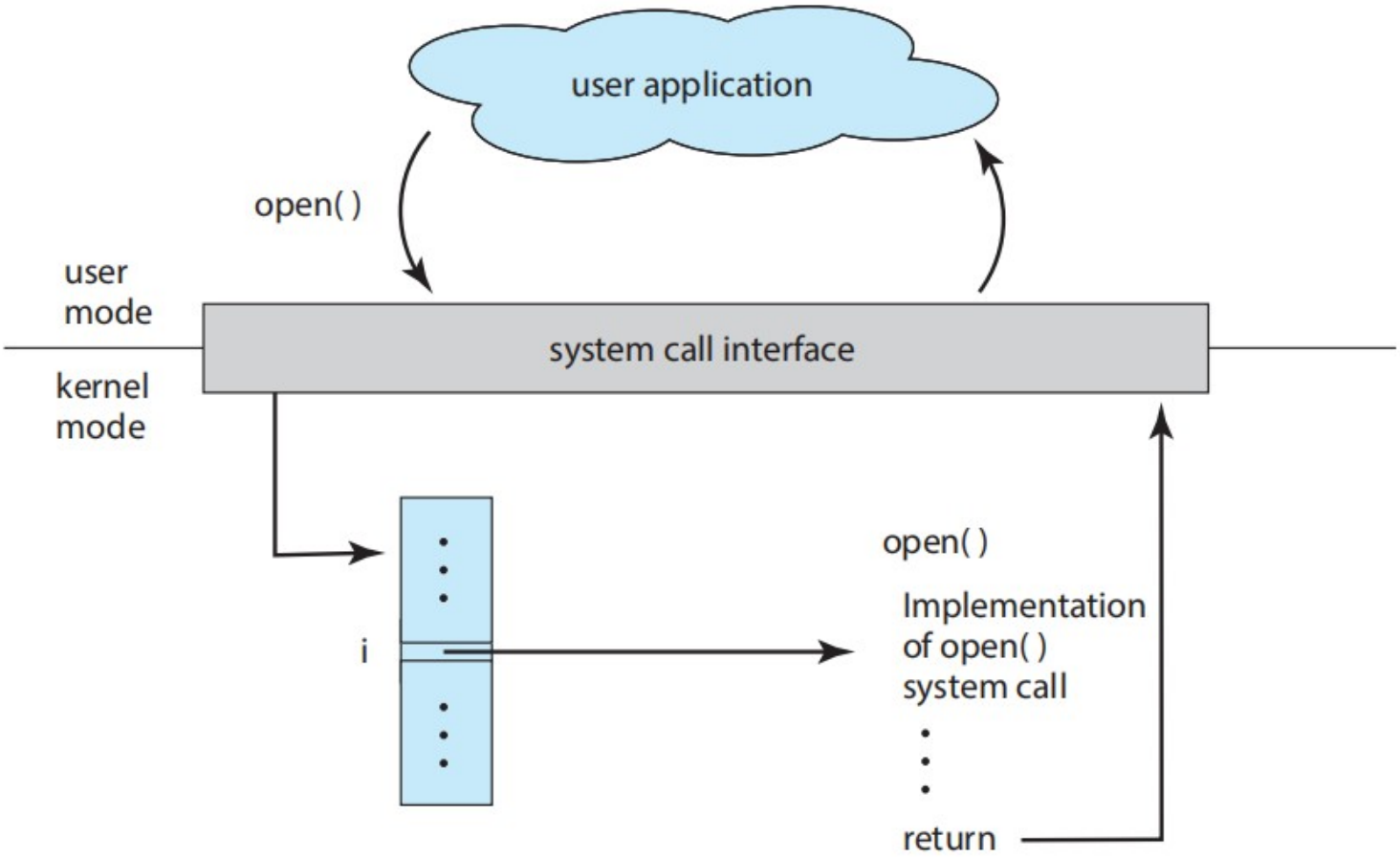
Приклади API для ОС:

- Windows API
- POSIX API (у Unix-подібних ОС)
- Java API

1. Системні виклики

API (Application Programming Interface) - продовження

У багатьох випадках виклику API відповідає системний виклик:



Джерело схеми: *A. Silberschatz, P. Galvin and G. Gagne,* Operating system concepts, 10th ed., Wiley, 2018. - p. 65, fig. 2.6

1. Системні виклики

POSIX (Portable Operating System Interface)

Що це: сімейство стандартів для забезпечення сумісності між різними ОС.

Хто розробляє: IEEE.

Що всередині: вимоги щодо API, командних оболонок, інтерфейсів базових утиліт.

Поточна версія: POSIX.1-2017

Які ОС підтримують:

(Якщо коротко) повністю - мало, значною мірою - багато, намагаються - всі.

(Якщо не коротко)

Сертифіковані POSIX-системи: macOS, VxWorks, z/OS, Integrity та деякі ін.

Загалом POSIX-сумісні: Linux, Android, MINIX, OpenSolaris, FreeBSD, Syllable, NetBSD, OpenBSD та ін.

Частково POSIX-сумісні (передусім - через додаткові механізми та середовища): Windows, eCos, Symbian, Plan 9, OpenVMS та ін.

1. Системні виклики

Відповідність деяких викликів Windows та викликів Unix

Що робить	Windows	Unix
Створює процес	`CreateProcess()`	`fork()`
Завершує процес	`ExitProcess()`	`exit()`
Чекає на завершення дочірнього процесу	`WaitForSingleObject()`	`wait()`
Створює файл	`CreateFile()`	`open()`
Зчитує з файлу	`ReadFile()`	`read()`
Записує у файл	`WriteFile()`	`write()`
Закриває файл	`CloseHandle()`	`close()`

Примітки

1. Деякі наведені функції роблять не лише те, що вказано у таблиці (`CloseHandle()` - закриває не лише файл, а й будь-який інший дескриптор).
2. У більшості наведених функцій є параметри. Іноді багато. Їх пропущено.
3. Для Windows наведено назви API-функцій. Назви відповідних системних викликів відрізняються.
4. Для Unix наведено назви системних викликів - вони часто співпадають або майже співпадають з назвами відповідних бібліотечних функцій.

2. Класифікація ОС за архітектурою

1. Монолітні ОС.

Найпоширеніший спосіб реалізації ОС. Усі базові функції ОС реалізовані у ядрі.

＋ Підвищення продуктивності (не потрібно перемикатися між режимами)

－ Зниження надійності (помилка у компоненті може призвести до краху системи)

2. Мікроядра.

У ядро поміщають мінімальний набір функцій. Решта компонентів працює у режимі користувача.

＋ Підвищення надійності (помилка у роботі драйвера не матиме фатальних наслідків, бо драйвер виконується у режимі користувача)

－ Зниження продуктивності

Приклади: Mach, MINIX 3, Symbian, QNX, Integrity, K42, L4, Pike OS.

2. Класифікація ОС за архітектурою

Взаємопроникнення:

- Сучасні монолітні ОС мають **модульну структуру** (модулі можуть за потреби підвантажуватися без перезавантаження системи) – Linux, FreeBSD, Solaris, Open VMS.
- Гібридні ОС (мікроядро з рисами монолітного ядра) – Windows NT, Syllable, DragonFly BSD, Mac OS X.
- Межі розмиваються. *Мікроядро Windows NT - це велиике мікроядро :)*

3. Багаторівневі ОС.

Історична архітектура.

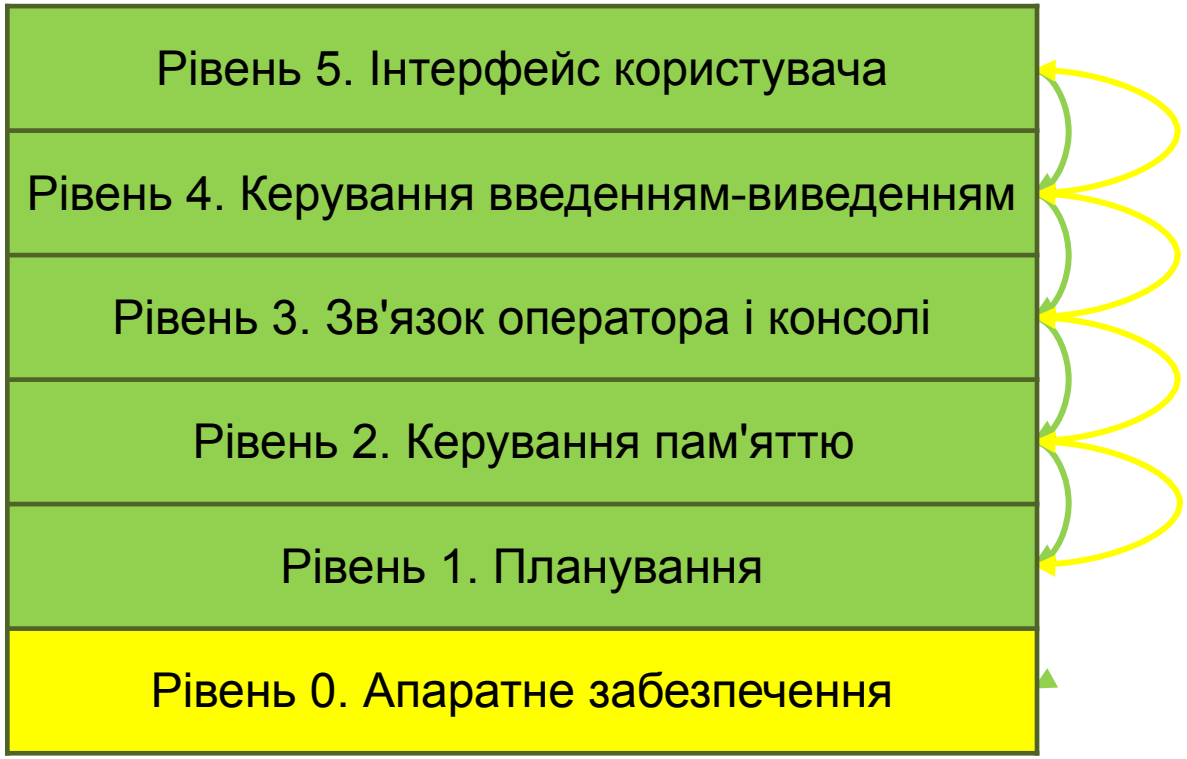
Замість двох режимів – багато рівнів.

Кожен наступний рівень має менше привілеїв доступу й спирається на функції попереднього рівня.

Доступ з верхнього рівня до функцій нижчого рівня – через системні виклики.

Приклади: THE, MULTICS.

Багаторівнева архітектура ОС
(на прикладі ОС *THE*)



\+ Зручність реалізації (створення окремих компонентів ОС): для використання операцій нижнього рівня програміст не мусить знати, як саме вони працюють, – йому досить знати, що вони роблять.

– Низька продуктивність (велика кількість переключень між рівнями).

5. Переривання

ЦП має **коректно реагувати на події**, які відбуваються з апаратним забезпеченням і програмними компонентами.

Які саме події?

Наприклад:

- читання або запис даних на диск завершено
- сталася помилка доступу до пам’яті
- користувач натиснув клавішу на клавіатурі
- під’єднано ще один пристрій тощо

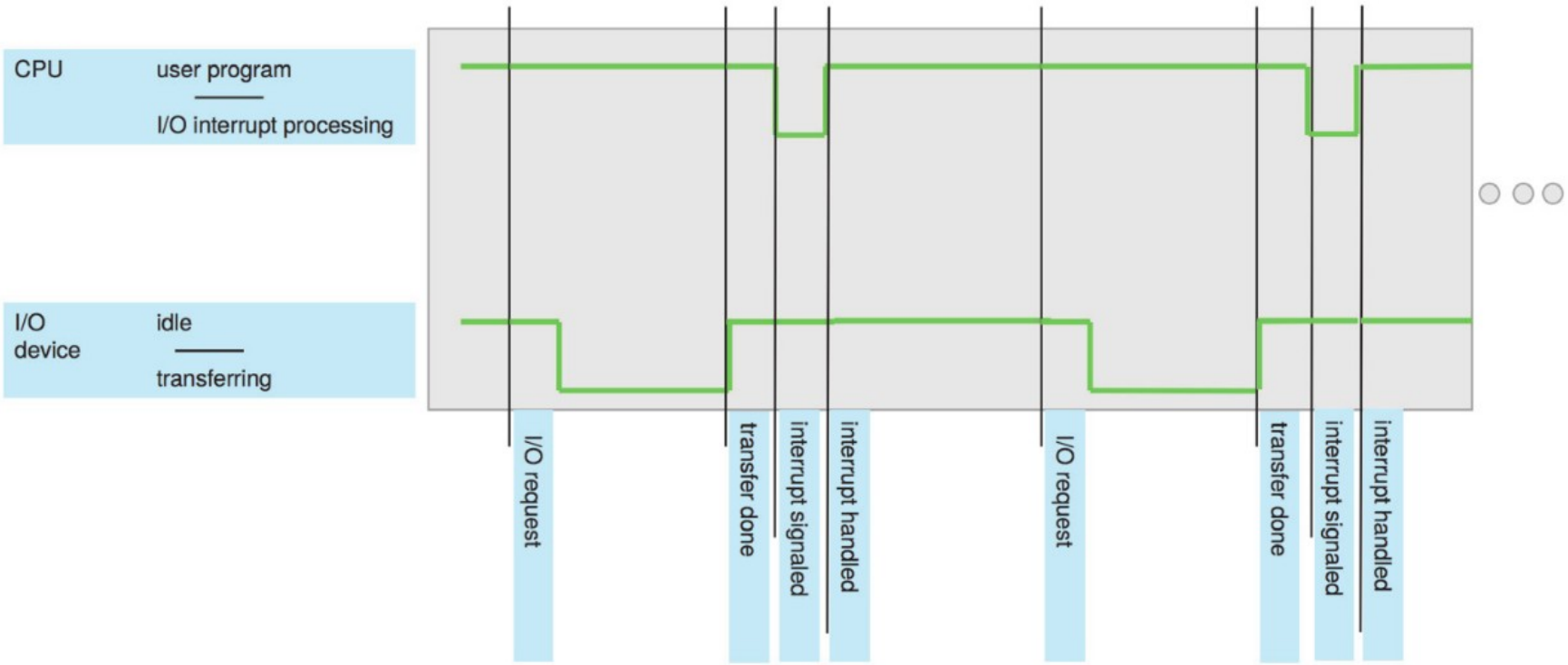
Щоб ЦП міг реагувати на ці та інші події, потрібно, щоб:

- ЦП якось дізнавався про ці події
- існували спеціальні процедури, котрі ЦП мав би виконати, якщо подія настала

Для таких потреб використовується механізм **переривань** (*interrupts*).

5. Переривання

Часова схема переривань для випадку одного процесу, який здійснює виведення даних



Джерело схеми: *A. Silberschatz, P. Galvin and G. Gagne,* Operating system concepts, 10th ed., Wiley, 2018. - p. 8, fig. 1.3

Для самостійного читання

1. **[*Silberschatz, Galvin, Gagne, 2018*]** Chapter 1 (patagr. 1.2.1, Chapter 3 (paragr. 3.1).
2. **[*Stollings, 2017*]** Chapters 3-4.
3. **[*Tanenbaum, Bos, 2014*]** Chapter 2 (paragr. 2.1) + chapter 1 (paragr. 1.4)
4. **[*Шеховцов, 2009*]** Розділи 2-3.