

---

# УПРАВЛІННЯ ПРОЦЕСАМИ ТА ПОТОКАМИ В ОПЕРАЦІЙНИХ СИСТЕМАХ

2019-2020 н. р.

Викладач: Марченко Станіслав Віталійович

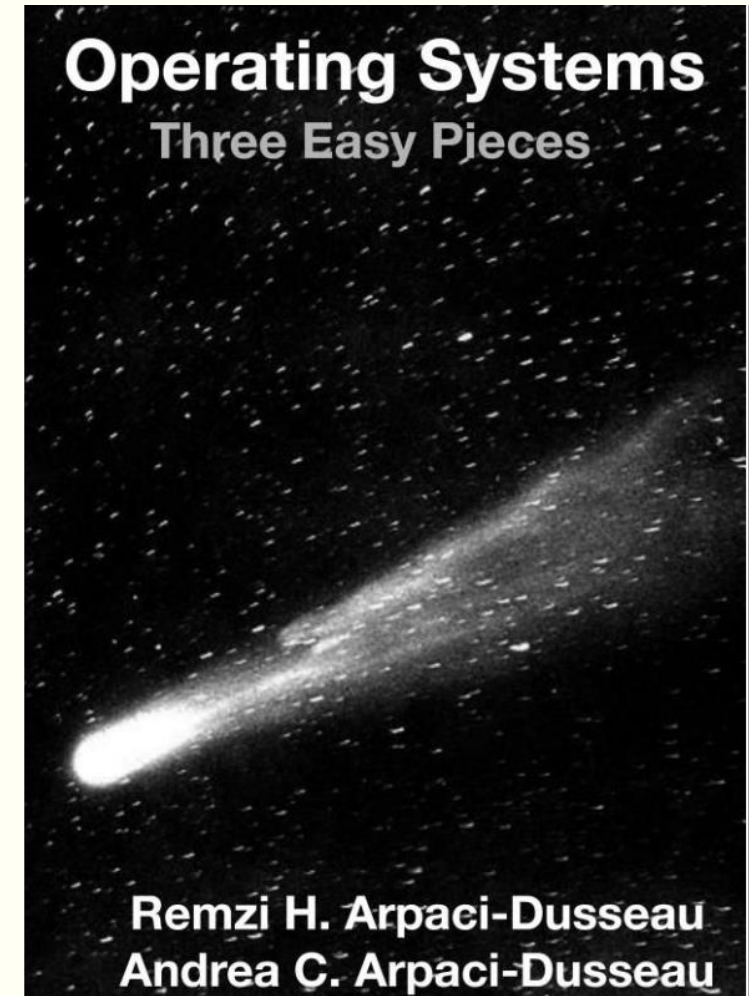
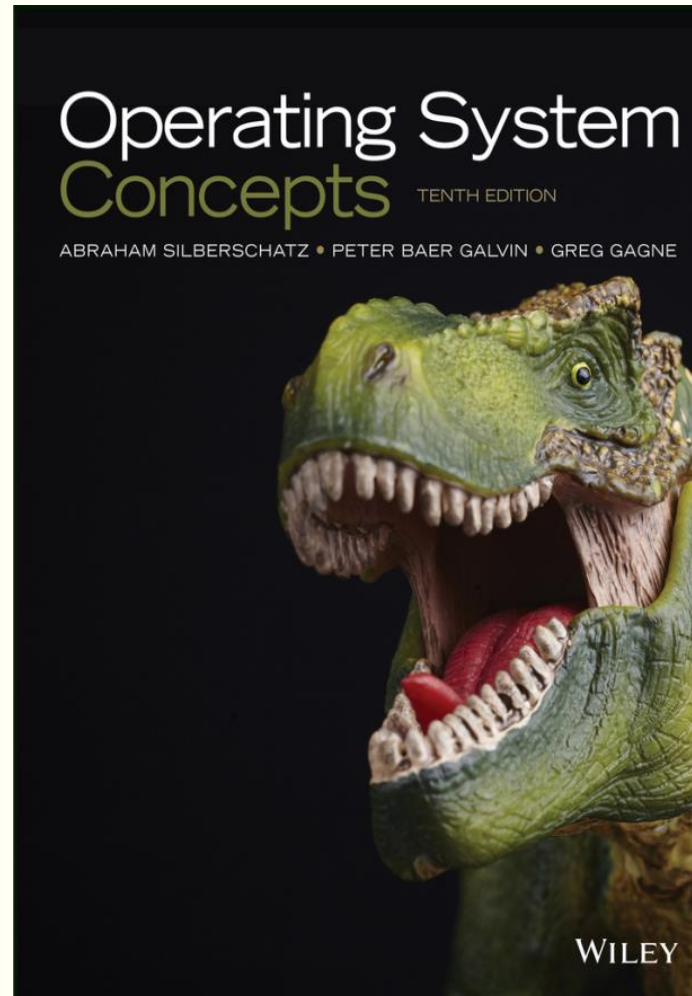
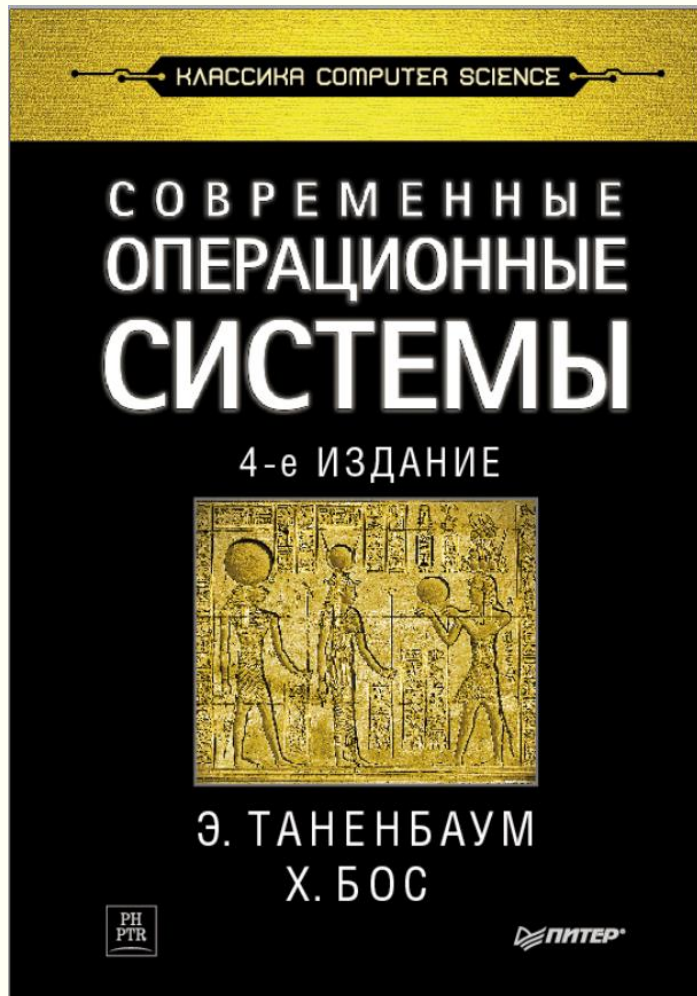
# Питання лекції

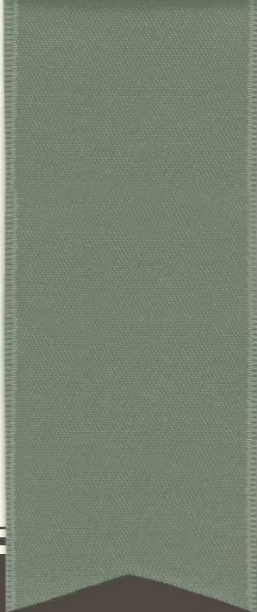
---

- Процеси в операційних системах.
- Потоки та моделі багатопоточного виконання коду.
- Принципи роботи планувальника центрального процесора.
- Синхронізація процесів на рівні операційної системи.

# Рекомендована література

---



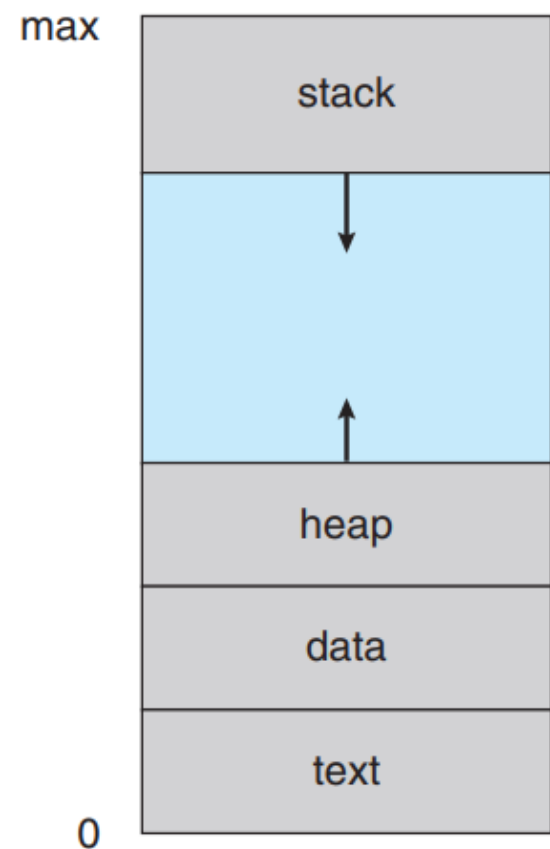


# ПРОЦЕСИ В ОПЕРАЦІЙНИХ СИСТЕМАХ

Питання 2.1

# Процес – програма під час виконання

---



Процес – одиниця роботи в сучасних комп'ютерних системах.

- Система складається з набору процесів, які виконують користувацький або системний код.
- Потенційно, вони працюватимуть конкурентно.

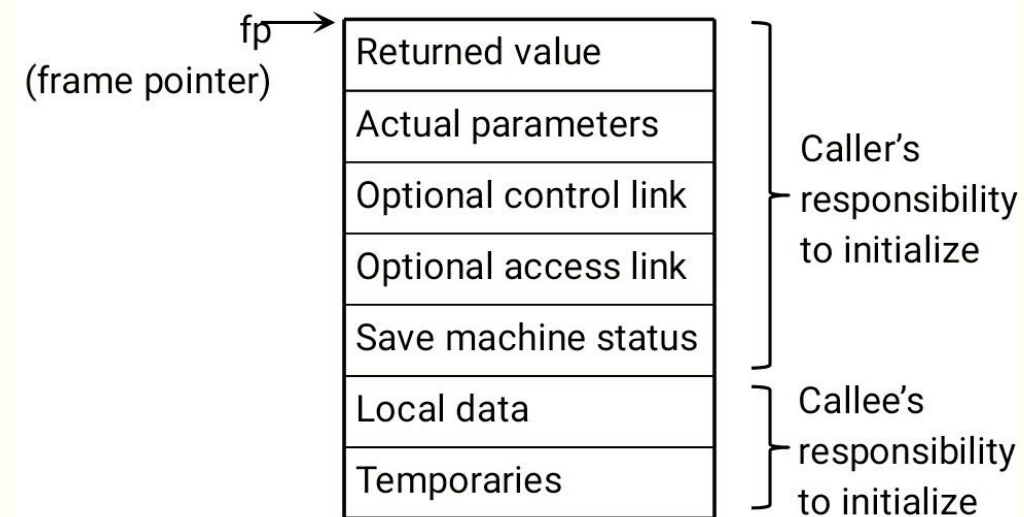
Ранні комп'ютери були пакетними системами, які виконували *завдання (jobs)*.

- Пізніше системи розподіленого часу запускали *користувацькі програми (задачі, tasks)*.
- Термін «процес» – аналогічна, проте більш сучасна концепція.

Модель пам'яті (memory layout) процесу передбачає кілька *сегментів (sections)*:

- Текстовий сегмент – частина адресного простору з виконуваним кодом (фіксований розмір)
- Сегмент даних – глобальні змінні (фіксований розмір)
- Куча (Heap section) – пам'ять, яка динамічно виділяється в процесі виконання програми
- Стек – тимчасове сховище даних при виклику функцій

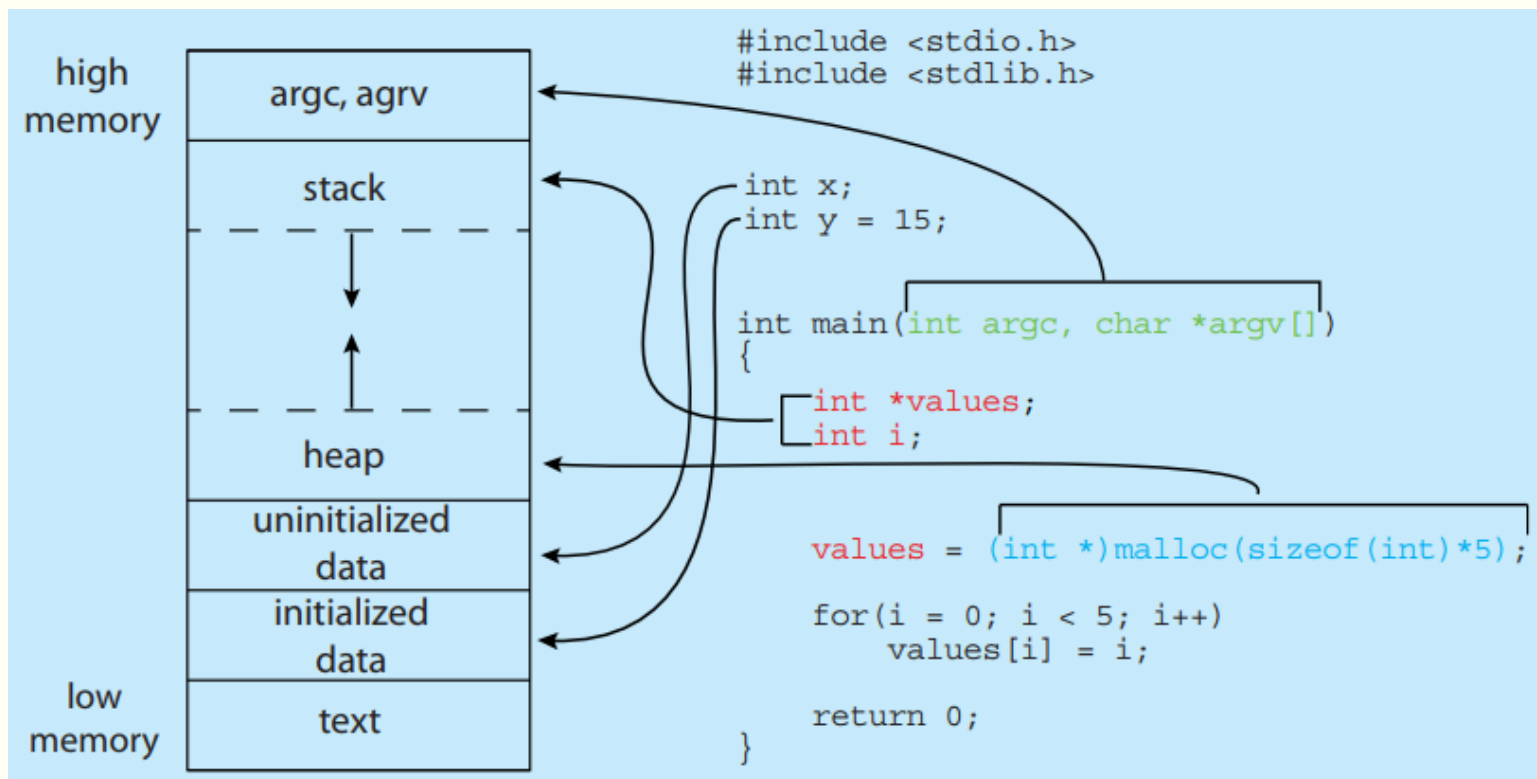
# Процес – програма під час виконання



- Кожного разу при виклику функції у стек додається **запис активації функції (активаційний запис, activation record)**.
  - Параметри функції, локальні змінні та return-адреса.
- Аналогічно, куча росте при динамічному виділенні пам'яті та зменшується при її вивільненні (поверненні системі).
  - Хоч стек і куча ростуть **назустріч** один одному, ОС повинна забезпечити неможливість їх перетину.

- Хоч 2 процеси можуть пов'язуватись однією програмою, вони все-одно вважаються 2 окремими послідовностями виконання.
  - Наприклад, кілька користувачів може працювати з різними копіями поштової програми або викликати багато копій веб-браузера.
  - Кожна копія є окремим процесом; хоч текстові секції ідентичні, існують відмінності в сегментах даних, кучі та стеку.

# Модель пам'яті програми мовою C



- GNU-команда `size` може використовуватись для визначення розміру (в байтах) деяких сегментів.
- Нехай назва файлу з даним первинним кодом – `memory`.
  - Виклик `size memory` згенерує наступний вивід:

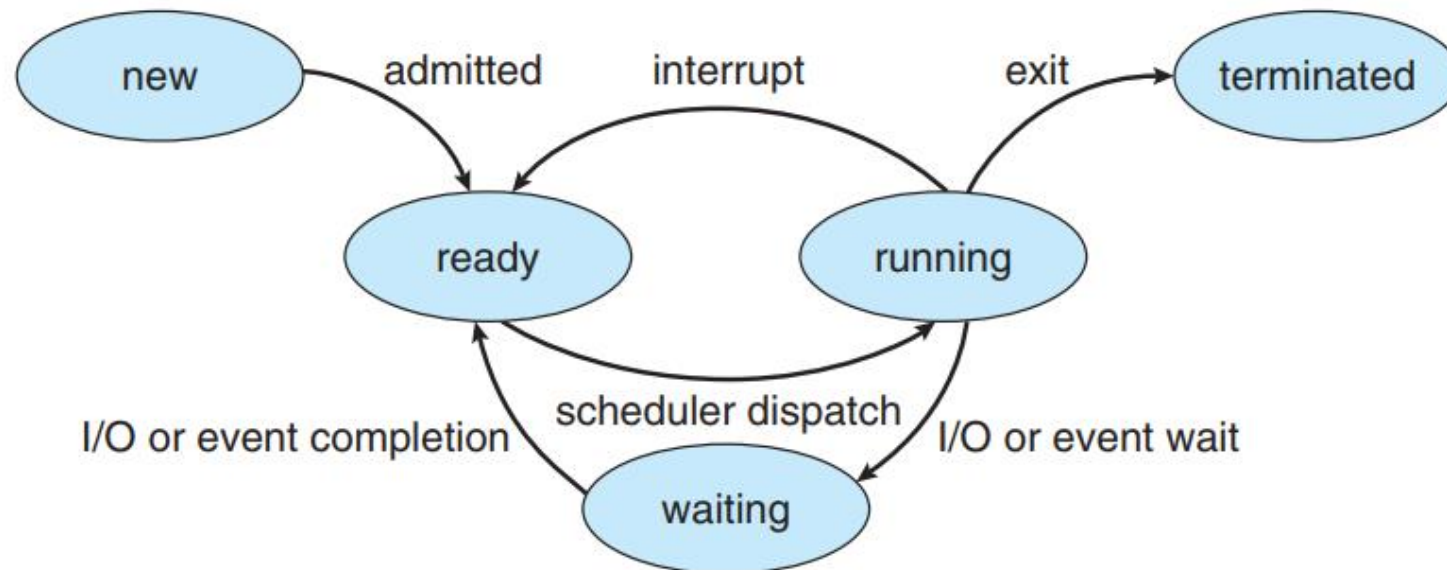
text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory



# Стан процесу

---

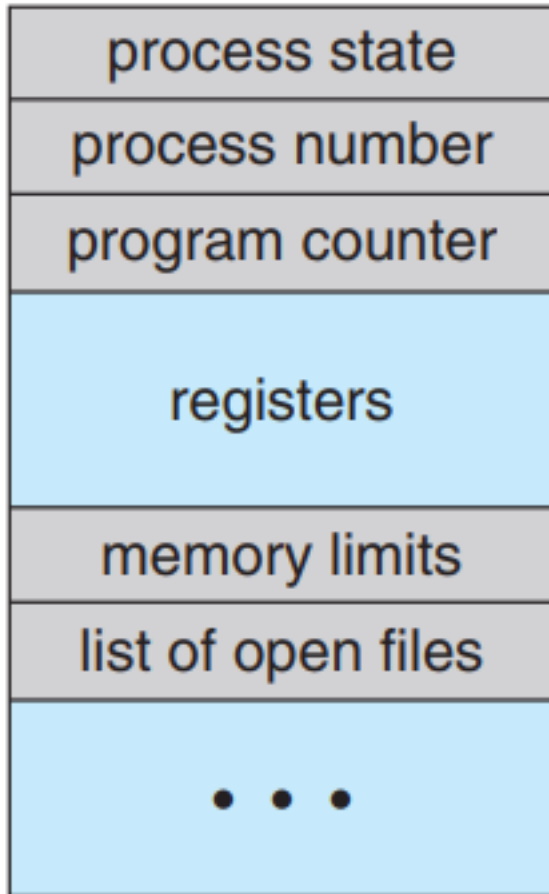
- Під час виконання процесу змінюється його стан:
  - **Новий (New)**. Процес створено.
  - **Запущений (Running)**. Інструкції виконуються.
  - **Очікування (Waiting)**. Процес очікує на настання деякої події (як закінчення вводу-виводу або отримання сигналу).
  - **Готовий (Ready)**. Процес очікує на своє призначення процесору.
  - **Завершений (Terminated)**. Процес завершив виконання.





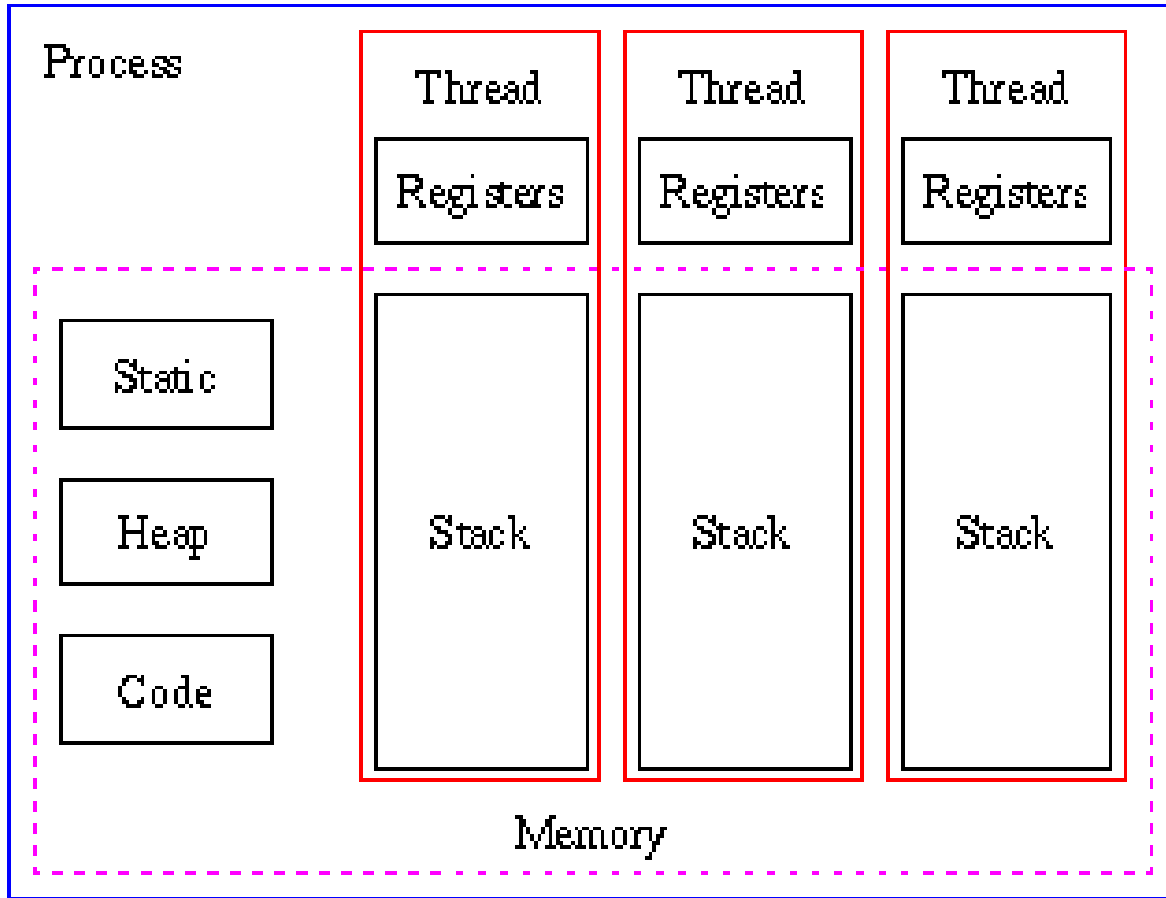
# Блок управління процесу (task control block, process control block, PCB)

---



- Містить багато частин інформації, пов'язаної з конкретним процесом:
  - **Стан процесу (Process state):** new, ready, running, waiting, halted та ін.
  - **Програмний лічильник (Program counter, PC).** Вказує на адресу наступної інструкції для виконання даним процесом.
  - **Регістри ЦП (CPU registers).** Відрізняються кількістю та типом залежно від архітектури.
    - Включають акумулятори, індексні регістри, вказівники стеку (stack pointers, SP) та регістри загального призначення + довільна condition-code інформація.
    - Разом з PC дана інформація про стан повинна зберігатись, коли відбувається переривання, щоб дозволити процесу коректно продовжуватись після перепланування його роботи.
  - **Інформація щодо планування роботи ЦП (CPU-scheduling information).** Включає пріоритет процесу, вказівники на планувальні черги (scheduling queues) та інші параметри планування.
  - **Інформація щодо керування пам'яттю (Memory-management information).** Може містити значення регістрів бази та границі, таблиць сторінок (page tables) або таблиць сегментів (segment tables) залежно від системи пам'яті, використаної в ОС.
  - **Статистична інформація.** Тривалість використання ЦП та загальний час роботи, часові ліміти, account numbers, номери робіт (job) або процесів.
  - **Інформація про статус вводу-виводу.** Включає список пристроїв вводу-виводу, виділених процесу, перелік відкритих файлів тощо.

# Потоки виконання (threads)



- Більшість сучасних ОС розширили поняття процесу до можливості мати кілька потоків виконання (threads of execution).
  - Особливо корисно на багатоядерних системах, коли багато потоків можуть працювати паралельно.
  - Наприклад, багатопоточний текстовий редактор може призначити один потік на керування вводом користувача, а інший – перевіряти правильність написання.
  - У системах з підтримкою потоків PCB розширяється для включення інформації по кожному потоку.
  - Інші зміни в системі також повинні підтримувати потоки.

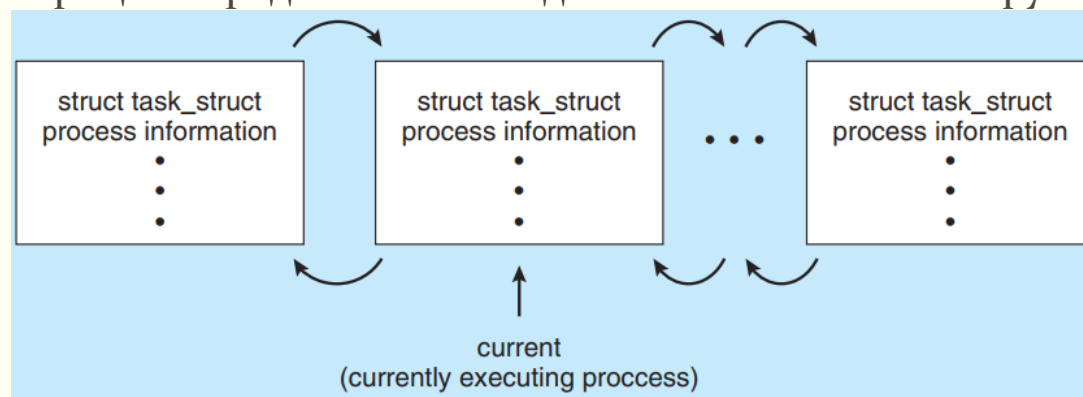
# Представлення процесу в Linux

- РСВ у Linux представлений С-структурою `task_struct` із файлу [include/linux/sched.h](#) первинного коду ядра.

- Структура містить усю необхідну інформацію щодо представлення процесу.
- Деякі з полів структури:

```
long state; /* стан процесу */
struct sched_entity se; /* інформація щодо планування */
struct task_struct *parent; /* батьківський процес */
struct list_head children; /* дочірні процеси */
struct files_struct *files; /* список відкритих файлів */
struct mm_struct *mm; /* адресний простір */
```

- У ядрі Linux усі активні процеси представляються двозв'язним списком структур `task_struct`.

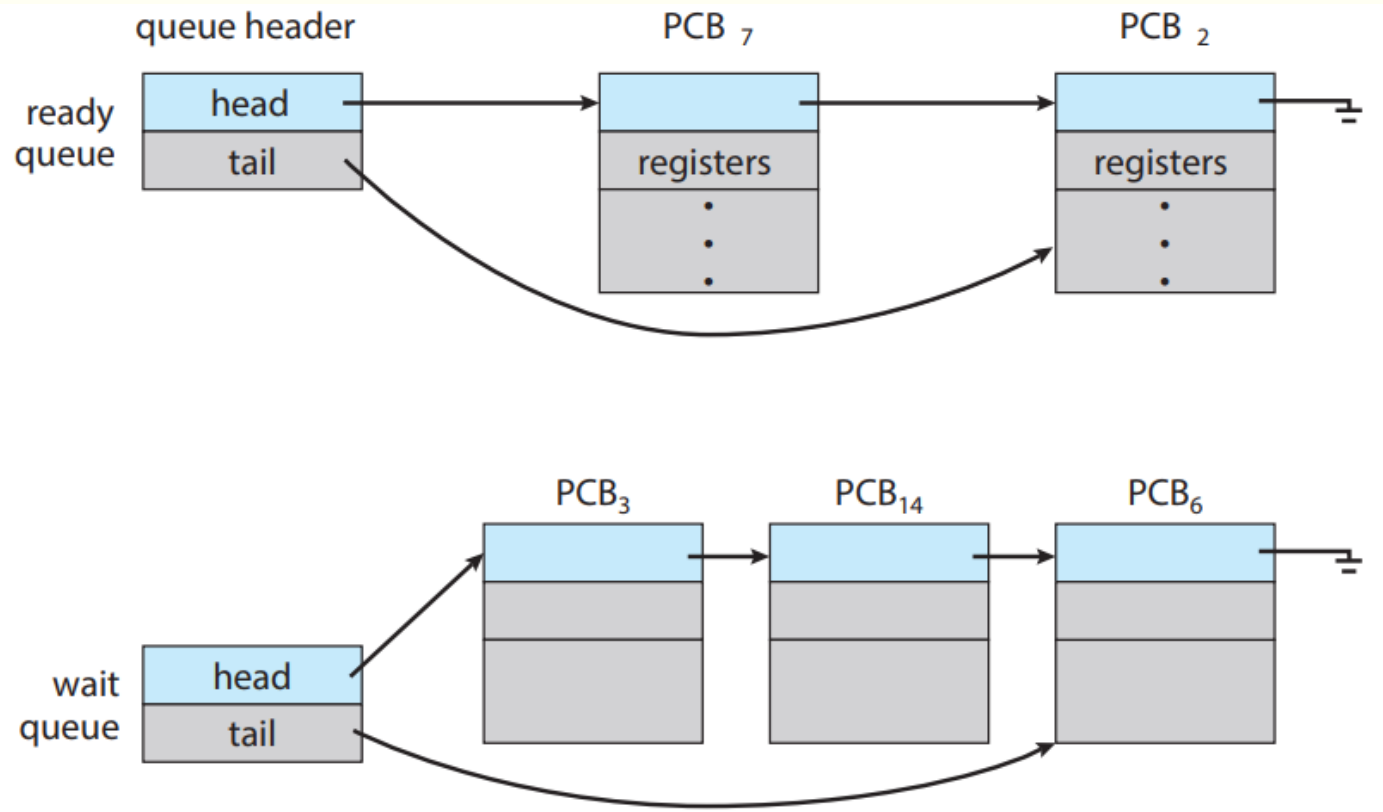


# Планування процесів (Process Scheduling)

---

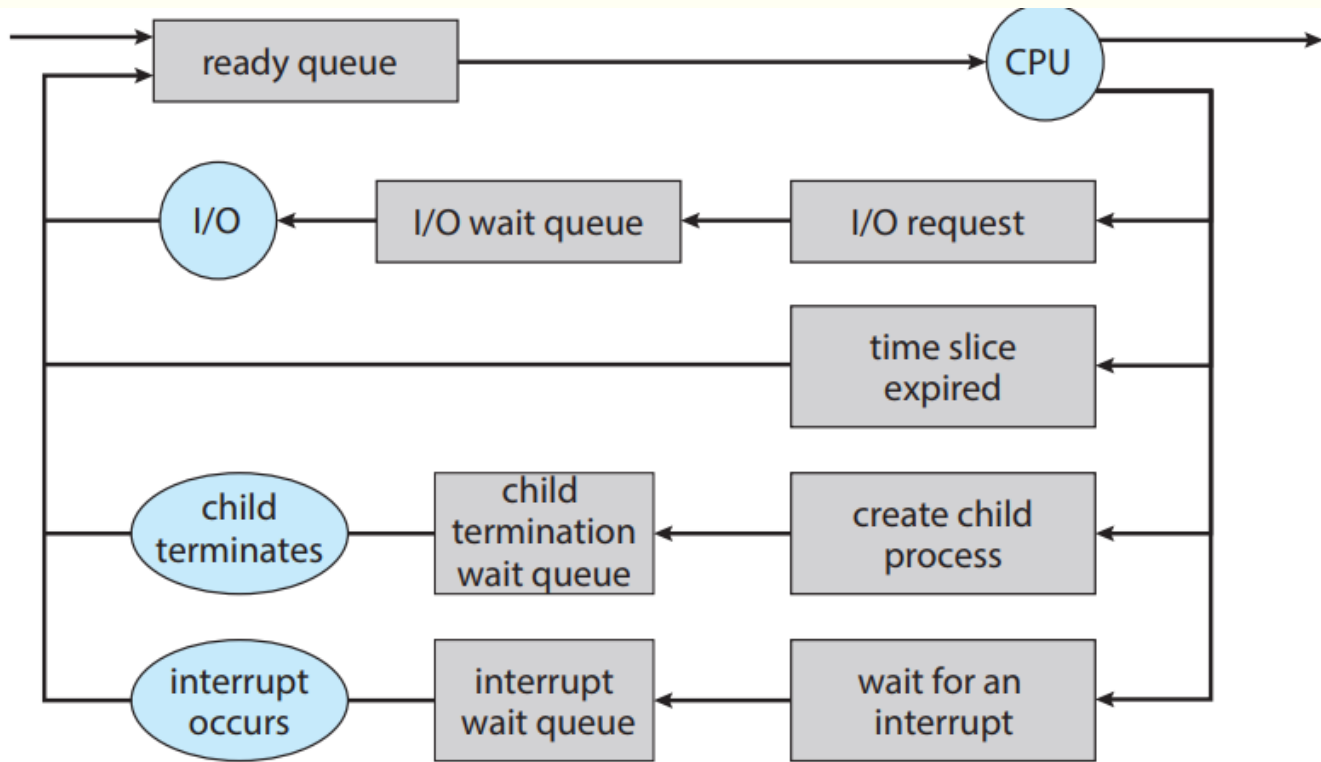
- Мета мультипрограмування – максимізувати використання ЦП процесами.
  - Розподіл часу передбачає перемикання ядра ЦП між процесами так часто, що користувачі можуть взаємодіяти з кожною запущеною програмою.
  - Планувальник процесів (process scheduler) обирає доступні процеси для виконання програми на ядрі.
  - Кожне ядро ЦП може працювати з одним процесом у кожний момент часу.
- Для одноядерної системи за раз може працювати лише 1 процес.
  - Якщо процесів більше, ніж ядер, надлишкові процеси чекають звільнення ядра та переплановують свою роботу.
  - Кількість процесів у пам'яті в поточний момент називають *рівнем мультипрограмування (degree of multiprogramming)*.
- Балансування завантаженості та розподіл часу вимагають врахування загальної поведінки процесу.
  - Загалом більшість процесів описуються **обмеженнями можливостями** вводу-виводу (I/O bound) або ЦП (CPU bound).
  - Обмежений можливостями вводу-виводу процес втрачає більшість часу на виконання операцій вводу-виводу, а не на обчислення.
  - Обмежений можливостями ЦП процес – навпаки.

# Планувальні черги (Scheduling Queues)



- При входженні процесів у систему їх поміщають у ready-чергу, де вони очікують на своє виконання ядром ЦП.
  - Загалом черга зберігається у вигляді зв'язного списку; голова ready-черги містить вказівник на перший PCB у списку, а кожний PCB включає вказівник на наступний PCB в ready-черзі.
- Система також містить інші черги.
  - Коли процесу надається ядро ЦП, він виконується деякий час, а потім завершується, переривається або очікує на настання певної події, наприклад, завершення запиту на ввід-вивід.
- Нехай процес надсилає запит на ввід-вивід до диску.
  - Диск працює набагато повільніше процесора, тому процес очікує доступність вводу-виводу.
  - Такий процес поміщається в чергу очікування (wait queue)

# Queueing-діаграма для представлення планування роботи процесу



- Представляє 2 типи черг: ready-чергу та набір черг очікування.
  - Круги представляють ресурси, які обслуговують черги, а стрілки вказують на хід процесів у системі.
- Новий процес спочатку ставиться в ready-чергу.
  - Він очікує, поки його оберуть для виконання або перенаправлять (dispatch).
  - Під час виконання процесу ядром може статись одна з кількох подій:
    - Процес видасть запит на ввід-вивід і буде поміщений в чергу очікування вводу-виводу (I/O wait queue).
    - Процес може створити новий дочірній процес, а потім переміститись у чергу очікування, поки не завершить/перерве роботу дочірній процес.
    - Процес може бути насильно переміщеним з ядра в результаті переривання або завершення періоду часу на виконання та перейде назад в ready-чергу.

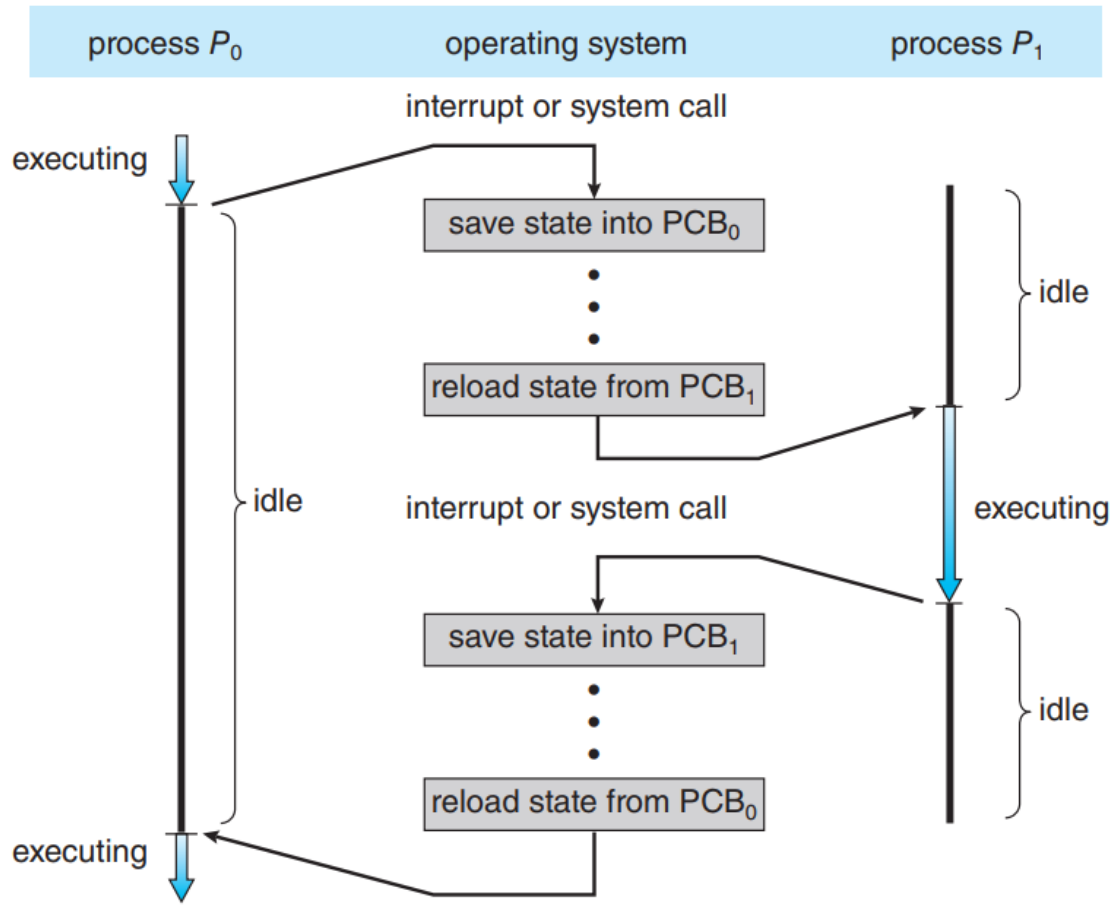
# Планування роботи ЦП (CPU Scheduling)

---

- Процес мігрує між ready-чергою та різними чергами очікування протягом свого існування.
  - Планувальник ЦП (CPU scheduler) обирає один процес з ready-черги та виділяє йому ядро ЦП.
  - Вибір нового процесу повинен відбуватись часто.
  - Обмежений можливостями вводу-виводу процес може виконуватись лише кілька мілісекунд перед очікуванням запиту на ввід-вивід.
  - Хоч CPU-bound-процес вимагатиме більше процесорного часу ядра, планувальник навряд чи дозволить розширити час використання ядра – скоріше, насильно передасть ядро іншому процесу.
  - Тому планувальник ЦП виконується принаймні раз кожні 100мс, зазвичай набагато частіше.
- Деякі ОС мають проміжну форму планування – **свопінг** (процес swapped out на диск).
  - Основна ідея: зменшити рівень мультипрограмування, вивантажуючи процес з пам'яті (і з конкуренції за ЦП).
  - Пізніше такий процес можна заново ввести в пам'ять, а виконання – продовжити з місця зупинки.
  - Свопінг зазвичай потрібен, коли пам'ять перевитрачається та повинна очиститись.

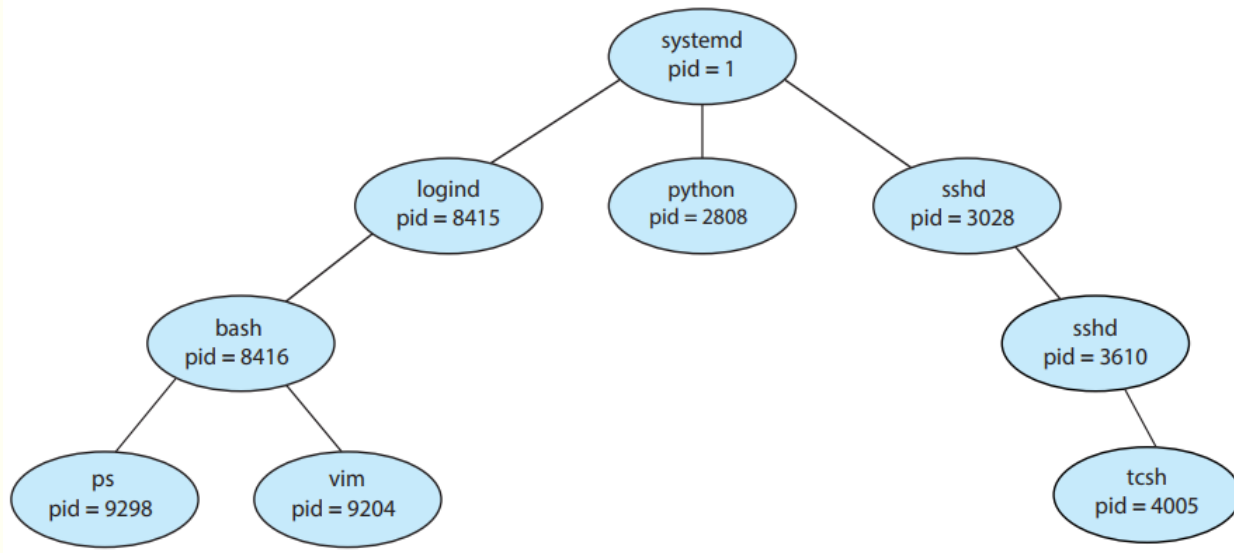


# Перемикання контексту (Context Switch)



- Коли трапляється переривання, системі потрібно зберегти поточний контекст працюючого на ядрі ЦП процесу, щоб відновити роботу після обробки переривання.
  - Контекст представляється в РСВ процесу: значення регістрів ЦП, стан процесу та інформація щодо управління пам'яттю.
- Перемикання ядра ЦП на інший процес вимагає збереження стану поточного процесу та відновлення стану іншого процесу.
  - Дана задача називається **перемиканням контексту**.
  - Контекст старого процесу зберігається в його РСВ, а потім завантажується збережений контекст нового процесу, запланованого на виконання.
  - Перемикання контексту – повністю накладні витрати.
  - Швидкість перемикання залежить від конкретної машини: швидкості пам'яті, кількості регістрів, необхідних для копіювання, існування спеціальних інструкцій тощо.
  - Типовий час перемикання – кілька мікросекунд.

# Операції над процесами. Створення процесу



Тут термін «процес» використовується rather loosely, оскільки для Linux переважно використовується «task».

Більшість ОС визначають процеси за їх унікальним ідентифікатором (pid), зазвичай представленим цілим числом.

- Його можна використовувати як індекс для доступу до різних атрибутів процесу всередині ядра.

Процес systemd (завжди pid=1) є кореневим батьківським процесом для всіх користувацьких процесів і створюється першим при завантаженні системи.

- Далі процес systemd створює процеси, які забезпечують додаткові служби (веб-сервер, print server, ssh-сервер тощо).
- Процес logind відповідає за управління клієнтами, які напряду входять у систему.
- Тут клієнт залогінився та використовує bash shell, якому присвоєно pid=8416.
- Процес sshd відповідає за управління клієнтами, підключеними до системи через ssh (secure shell).

# Операції над процесами. Створення процесу

---

- На ОС UNIX та Linux можна отримати перелік процесів за допомогою команди `ps`:
  - `ps -el`
  - Крім того, Linux-системи надають команду `pstree`, яка виведе дерево всіх процесів у системі.
- Загалом, коли процес створює дочірній процес, потрібно виділити деякі ресурси для виконання задачі: процесорний час, пам'ять, файли, пристрої вводу-виводу.
  - Дочірній процес може мати змогу отримувати ресурси напряму від ОС або обмежуватись підмножиною ресурсів батьківського процесу.
  - Батьківському процесу може бути потрібно розподіляти свої ресурси між дочірніми процесами.
  - Обмеження дочірнього процесу батьківськими ресурсами усуває можливість перевантаження системи процесом шляхом створення надто багато дочірніх процесів.
- Крім постачання різних фізичних та логічних ресурсів, батьківський процес *may pass along initialization data (input) to the child process*.
  - Нехай існує процес, чия задача – відобразити вміст файлу (наприклад, `hw1.c`) на екрані терміналу.
  - Коли процес створено, він отримає від батьківського процесу на вхід назву файлу (`hw1.c`).
  - Альтернатива: деякі ОС передають ресурси дочірнім процесам. Тоді новий процес може отримати з відкриті файли, `hw1.c` та термінальним пристроєм, та просто передавати дані між ними.

# Операції над процесами. Створення процесу

---

- Коли процес створює новий процес, існують 2 можливості для виконання:
  - 1. Батьківський процес конкурентно виконується з дочірніми.
  - 2. Батьківський процес очікує, поки деякі або всі дочірні процеси завершать роботу.
- Також існує 2 можливості стосовно адресного простору:
  - 1. Дочірній процес – дублікат батьківського процесу.
  - 2. Дочірній процес має нову завантажену в нього програму.
- Спочатку розглянемо ОС UNIX: кожен процес має унікальний ідентифікатор.
  - Новий процес створюється за допомогою системного виклику `fork()`.
  - Новий процес складається з копії адресного простору первинного процесу, що спрощує взаємодію.
  - Обидва процеси (parent + child) продовжують виконання з наступної після `fork()` інструкції з однією відмінністю: return-код для `fork()` буде 0 для дочірнього процесу та ідентифікатор дочірнього процесу для батьківського.

# Операції над процесами. Створення процесу

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

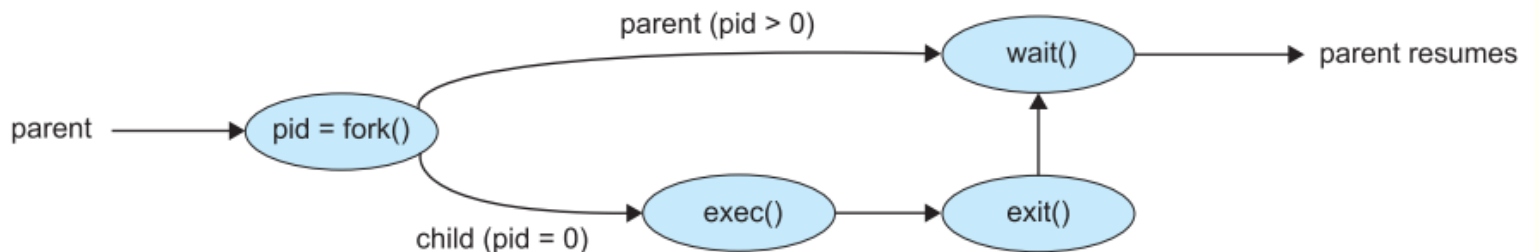
    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Після системного виклику `fork()` один з 2 процесів зазвичай використовує системний виклик `exec()`, щоб замінити простір пам'яті процесу новою програмою.

- Системний виклик `exec()` завантажує бінарний файл у пам'ять (знищуючи образ пам'яті програми, яка містить виклик `exec()`) та запускає його виконання.
- Батьківський процес може потім
  - створити більше дочірніх процесів;
  - або, не маючи роботи, поки працюють дочірні процеси, здійснити системний виклик `wait()`, щоб перемістити себе з ready-черги до завершення «дочок».
- Оскільки виклик `exec()` перекриває адресний простір процесу новою програмою, `exec()` не повертає управління, поки не трапиться помилка.



```
#include <stdio.h>
#include <windows.h>
```

```
int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

## Альтернативний приклад: створення процесу в ОС Windows

---

- Процеси створюються в Windows API за допомогою функції `CreateProcess()`, подібної до `fork()` у плані створення дочірнього процесу батьком.
  - Проте `CreateProcess()` вимагає завантаження заданої програми в адресний простір дочірнього процесу при його створенні.
  - Якщо `fork()` не приймає параметрів, `CreateProcess()` очікує не менше 10.
- Два переданих параметри у функцію `CreateProcess()` – структури `STARTUPINFO` та `PROCESS_INFORMATION`.
  - `STARTUPINFO` визначає властивості нового процесу, на зразок розмірів вікна, його зовнішнього вигляду, обробку файлів зі стандартних вводу та виводу.
  - `PROCESS_INFORMATION` містить обробник (handle) та ідентифікатори новоствореного процесу та потоку.
  - Викликаємо функцію `ZeroMemory()` для виділення пам'яті кожній із структур до роботи з `CreateProcess()`.
- Перші 2 параметри `CreateProcess()` – назва додатку (тут – `NULL`) та аргументи командного рядка (налаштовують завантаження додатку).
  - Тут завантажується додаток `mspaint.exe`.



# Операції над процесами. Переривання процесу

---

- Процес переривається при завершенні виконання останньої інструкції та запиту до ОС на його видалення за допомогою системного виклику `exit()`.
  - У цей момент процес може повертати свій статус (зазвичай ціле число) очікуючому батьківському процесу за допомогою системного виклику `wait()`.
  - Усі ресурси процесу, включаючи фізичну та віртуальну пам'ять, відкриті файли, буфери вводу-виводу, деалокуються та запитуються ОС на повернення (`reclaim`).
- Переривання може траплятись і в інших випадках.
  - Процес може спричинити переривання роботи іншого процесу за допомогою відповідного системного виклику (наприклад, `TerminateProcess()` in Windows).
  - Зазвичай такий системний виклик може здійснюватись лише батьківським процесом щодо запланованого на переривання процесу. Інакше, користувач або некоректно працюючий додаток можуть довільно «вбити» процеси іншого користувача.
  - Батьківський процес потребує ідентифікаторів дочірніх процесів, які бажає перервати, тому при створенні процесу такий ідентифікатор передається батьку.
- Батько може перервати виконання одного з дочірніх процесів з різних причин:
  - Дочірній процес перевищив використання деяких ресурсів, які йому були виділені. Батьку потрібно мати механізм перевірки стану «дочок».
  - Задача (`task`), призначена дочірньому процесу, більше не потрібна.
  - Батьківський процес завершується, а ОС не дозволяє дочірньому процесу продовжуватись без батька.



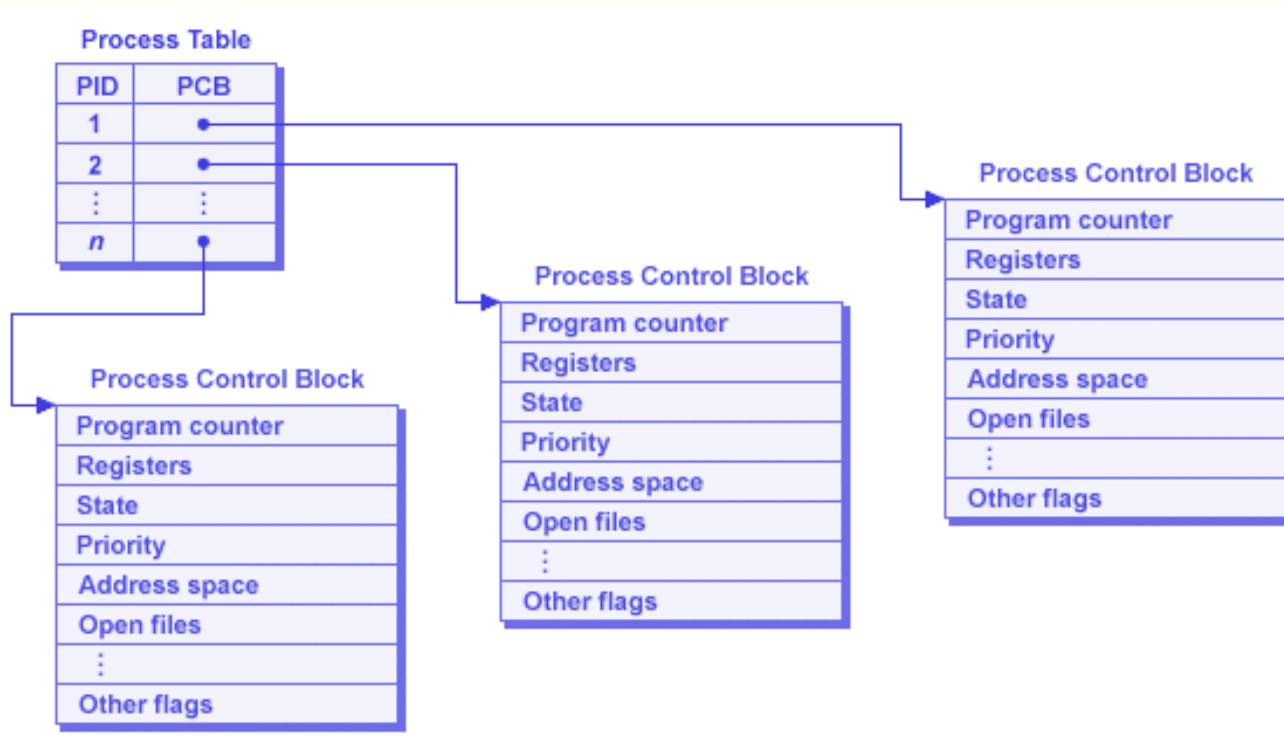
# Операції над процесами. Переривання процесу

---

- Деякі системи не дозволяють дочірньому процесу існувати, якщо «батько» завершив роботу.
  - Це явище називають каскадним перериванням (cascading termination), зазвичай воно ініціюється ОС.
- У Linux / UNIX-системах процес можна перервати системним викликом `exit()`, в який передається `exit status`: **`exit(1)`** ;
  - In fact, under normal termination, `exit()` will be called either directly (as shown above) or indirectly, as the C run-time library (which is added to UNIX executable files) will include a call to `exit()` by default.
- Батьківський процес може очікувати переривання дочірнього процесу за допомогою системного виклику `wait()`.
  - У виклик передається параметр, який дозволяє батьківському процесу отримати `exit status` «дочки».
  - Також виклик повертає ідентифікатор перерваного дочірнього процесу, щоб «батько» міг визначити, який з дочірніх процесів завершився:

```
pid_t pid;  
int status;  
pid = wait(&status);
```

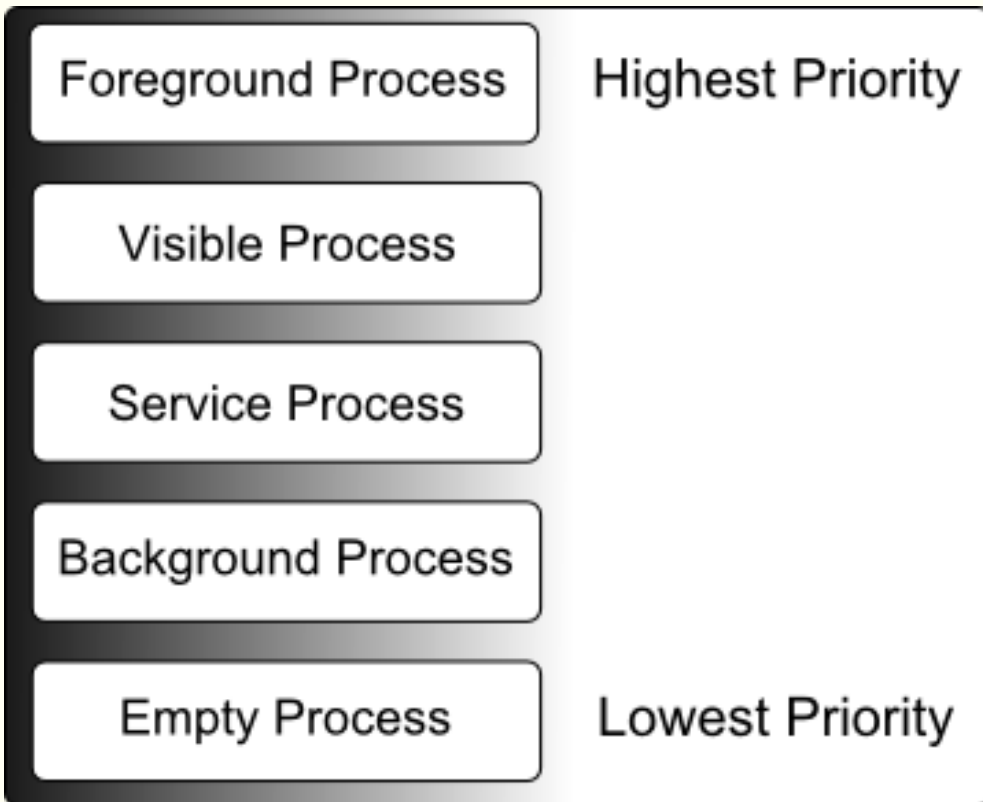
# Операції над процесами. Переривання процесу



- При перериванні процесу його ресурси деалокуються операційною системою.
  - Проте його входження в таблицю процесів (process table) повинно залишатись, поки батьківський процес викликає `wait()`, оскільки таблиця містить `exit status` процесів.
  - Завершений процес, батько якого ще не викликав `wait()`, називають **зомбі-процесом**.
  - Усі процеси переходять у цей стан при завершенні, проте загалом вони існують як зомбі зовсім трохи.
  - Як тільки батьківський процес викликає `wait()`, ID зомбі-процесу та його входження в таблицю процесів видаляються.
- Уявіть, що буде, якщо «батько» не викликав `wait()`, а перервав роботу.
  - Дочірній процес залишиться **сиротою (orphan)**.
  - Традиційні UNIX-системи вирішують проблему, присвоюючи процеси-сироти процесу `init`.
  - Процес `init` періодично викликає `wait()`, дозволяючи зібрати всі `exit`-статуси процесів-сирот та видалити їх ідентифікатори та входження в таблицю процесів.

# Ієрархія процесів в ОС Android

---



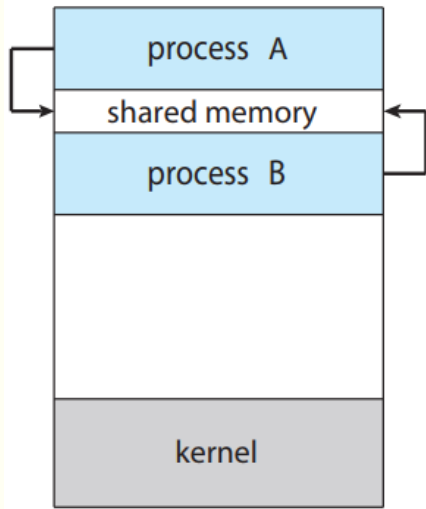
- Від найбільш до найменш пріоритетних процесів:
  - **Передньоплановий (Foreground) процес**—поточний процес видимий на екрані, представляючи додаток користувачу для взаємодії.
  - **Видимий (Visible) процес**—процес, який не видно на передньому плані, проте він виконує передньоплановий процес (активність)
  - **Служба (Service process)**—процес, подібний до фонового, проте виконує активність, видиму користувачу (як стримінг музики)
  - **Фоновий (Background) процес**—процес, який виконує активність, невидиму для користувача.
  - **Порожній процес**—процес, який не містить жодного активного компоненту, пов'язаного з додатком.

# Міжпроцесна взаємодія (Interprocess Communication, IPC)

---

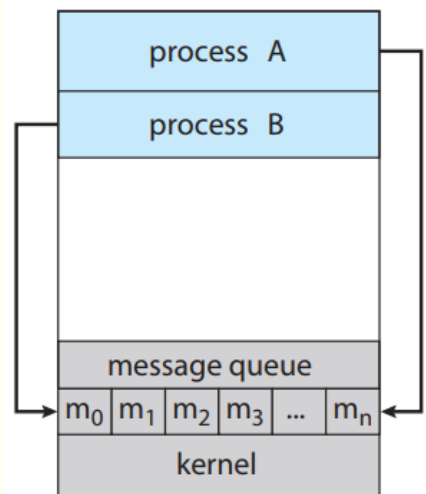
- Процес незалежний, якщо не має спільних даних з іншими процесами, що виконуються в системі. Інакше процес буде кооперованим (cooperating).
- Існує кілька причин для забезпечення середовища, яке дозволяє кооперацію процесів:
  - **Розкриття (*sharing*) інформації.** Необхідно забезпечити середовище для конкурентного доступу до інформації зі спільним доступом.
  - **Прискорення обчислень.** Розбиття крупної задачі на підзадачі для їх подальшого паралельного виконання. Прискорення можна отримати лише на багатопроцесорних системах.
  - **Модульність.** Розподіл системних функцій на окремі процеси або потоки.
- Кооперовані процеси вимагають механізм міжпроцесної взаємодії (IPC), який дозволить їм обмінюватись даними.
  - Фундаментальні моделі міжпроцесної взаємодії: shared memory та message passing.

# Міжпроцесна взаємодія (Interprocess Communication, IPC)



- У моделі зі спільною пам'яттю область пам'яті буде спільною для скооперованих процесів.

- Процеси зможуть обмінюватись інформацією, записуючи та зчитуючи дані в спільну область.
- Модель може бути швидшою за передачу повідомлень, оскільки системи з передачею повідомлень зазвичай реалізовані за допомогою системних викликів, тому вимагають більш time-consuming task of kernel intervention.
- У системах зі спільною пам'яттю системні виклики потрібні лише для визначення областей пам'яті зі спільним доступом.
- Як тільки визначається спільна пам'ять, усі звернення до неї обробляються як routine memory accesses, тобто від ядра не вимагається жодної підтримки.



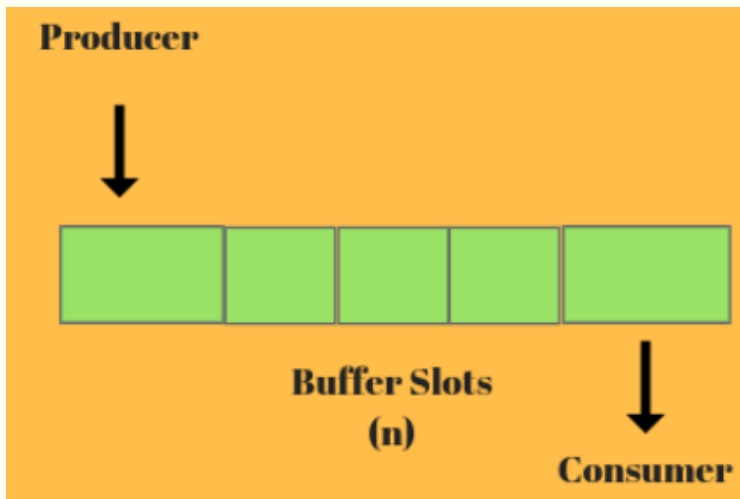
- У моделі з передачею повідомлень взаємодія відбувається шляхом обміну повідомленнями між скооперованими процесами.

- Передача повідомлень корисна для обміну невеликими об'ємами даних, оскільки немає конфліктів, які потрібно вирішувати.
- Передачу повідомлень також простіше реалізується в розподіленій системі, ніж модель зі спільною пам'яттю.

# IPC у системах зі спільною пам'яттю (Shared-Memory Systems)

---

- Зазвичай область спільної пам'яті розміщується в адресному просторі процесу, який створює shared-memory segment.
  - Інші процеси, які бажають взаємодії через цей shared-memory segment, повинні прикріплюватись до того ж адресного простору.
  - ОС зазвичай не дозволяє одному процесу отримувати доступ до адресного простору іншого, тому дані процеси повинні зняти це обмеження.
  - Форма даних та місце розташування спільної пам'яті визначається процесами, а не ОС, тому процеси відповідають за відсутність конфліктів при одночасному запису в одну комірку спільної пам'яті.



- Спільна пам'ять використовується в одному з розв'язків задачі «виробник-споживач» (producer–consumer problem).
  - Для їх конкурентної роботи необхідно мати доступний буфер елементів, який заповнюватиме виробник та спустошуватиме споживач.
  - Буфер буде розташований в області пам'яті, спільній для процесу-producer та процесу-consumer.
  - Виробник може створювати один елемент, а споживач – споживати інший, їх роботу треба синхронізувати.

# Можуть використовуватись 2 типи буферів

- Необмежений (unbounded) буфер не накладає межу на свій розмір.
  - Споживач може бути змушений очікувати на нові елементи, проте виробник може завжди продукувати нові елементи.
- Обмежений (bounded) буфер передбачає буфер фіксованого розміру.
  - У цьому випадку споживач повинен чекати, якщо буфер порожній, а виробник повинен чекати, якщо буфер повний.

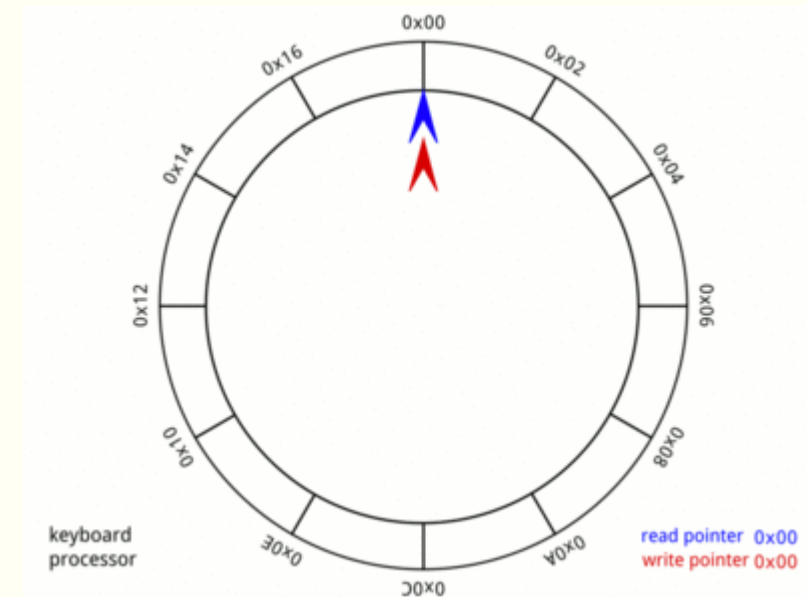
```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Обмежений буфер

- Спільний буфер реалізується як циклічний масив з 2 логічними вказівниками: in та out.
- Змінна in вказує на наступну вільну позицію в буфері; out – на першу full position у буфері.
- Буфер порожній при  $in == out$ , а заповнений – при  $((in + 1) \% BUFFER\_SIZE) == out$ .
- Не показано проблему конкурентного доступу до спільного буферу.





# Можуть використовуватись 2 типи буферів

---

- Процес-producer має локальну змінну next\_produced, у яку буде збережено згенероване значення.
  - Процес-consumer має локальну змінну next\_consumed, у яку зберігатиметься значення для споживання.

```
item next_produced;

while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Процес-producer при спільній пам'яті

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_consumed */
}
```

Процес-consumer при спільній пам'яті

# IPC у системах з передачею повідомлень (Message-Passing Systems)

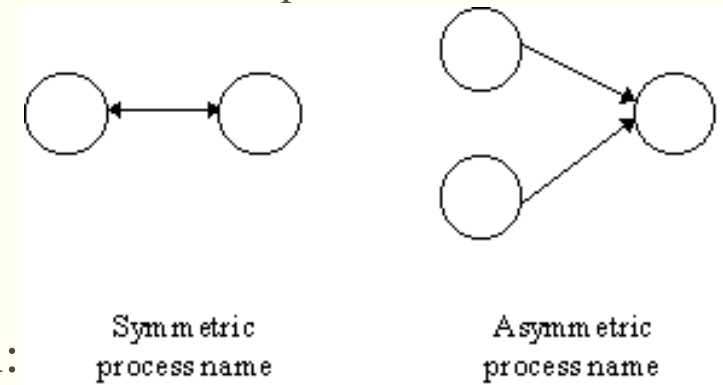
---

- Передача повідомлень забезпечує механізм комунікації процесів та синхронізації їх дій без спільного адресного простору.
  - Особливо корисно в розподілених середовищах, в яких взаємодіючі процеси можуть розміщатись на різних комп'ютерах мережі.
- Інфраструктура передачі повідомлень постачає принаймні 2 операції: `send(message)` та `receive(message)`
  - Надіслані процесом повідомлення можуть бути як фіксованого, так і змінного розміру.
  - Якщо надсилати повідомлення фіксованого розміру, системна реалізація спрощується, проте ускладнюється програмування. Для повідомлень змінної довжини – навпаки.
- Якщо процеси P і Q бажають взаємодіяти, вони повинні надсилати та отримувати повідомлення від один одного: між ними повинен існувати **комунікаційний зв'язок** (*communication link*).

# Naming

---

- Процеси, які бажають взаємодіяти, повинні мати спосіб звертатись один до одного.
  - При прямій взаємодії кожен процес, який хоче комунікувати, повинен явно назвати recipient або sender взаємодії.
- У даній схемі примітиви `send()` і `receive()` визначені так:
  - • `send(P, message)`—надіслати повідомлення процесу P.
  - • `receive(Q, message)`—отримати повідомлення від процесу Q.
- Комунікаційний зв'язок у даній схемі має наступні властивості:
  - Зв'язок (link) встановлюється автоматично між кожною парою процесів, які бажають взаємодіяти.
  - Процесам потрібно знати тільки ідентифікатори один одного.
  - Зв'язок стосується рівно 2 процесів, а між кожною парою процесів існує рівно 1 зв'язок.
  - Така схема показує симетрію адресації.



# Асиметричність адресації. Непряма адресація

---

- Тут тільки відправник називає отримувача.
  - `send(P, message)`—надсилає повідомлення процесу P.
  - `receive(id, message)`—отримує повідомлення від будь-якого процесу. У змінну `id` встановлюється назва процесу, з яким відбулась взаємодія.
- Недоліком обох схем є обмежена модульність результуючих process definitions.
  - Зміна ідентифікатора процесу може зробити необхідним огляд усіх інших process definitions.
  - Усі посилання на старий ідентифікатор повинні бути знайденими, щоб виконати заміну. Такий підхід небажаний, багато hard-coding-роботи.
- З непрямою (indirect) взаємодією повідомлення надсилаються та отримуються від mailboxes, або портів.
  - Кожен мейлбокс має унікальний ідентифікатор.
  - Наприклад, черги POSIX-повідомлень використовують ціле число для ідентифікатора mailbox.



# Непряма адресація

---

- Процес може взаємодіяти з іншим процесом через кілька різних мейлбоксів, проте 2 процеси можуть взаємодіяти тільки якщо мають спільний мейлбокс.
  - `send(A, message)`—надсилає повідомлення в мейлбокс A.
  - `receive(A, message)`—отримує повідомлення з мейлбоксу A.
- У цій схемі комунікаційна взаємодія має наступні властивості:
  - Зв'язок встановлюється між парою процесів тільки якщо обидва процеси мають спільний мейлбокс.
  - Зв'язок може асоціюватись більш, ніж з двома процесами.
  - Між кожною парою взаємодіючих процесів може існувати кілька різних зв'язків, кожному з яких відповідає 1 мейлбокс.

# Нехай процеси $P_1$ , $P_2$ та $P_3$ усі мають спільний мейлбокс $A$

---

- Процес  $P_1$  надсилає повідомлення до  $A$ , поки  $P_2$  та  $P_3$  виконують `receive()` від  $A$ .
  - Який процес отримає повідомлення, надіслане від  $P_1$ ?
- Відповідь залежить від того, який з наступних методів обрати:
  - Дозволити зв'язок максимум між 2 процесами.
  - Дозволити максимум одному процесу в даний момент часу виконувати операцію `receive()`.
  - Дозволити системі довільно обирати, який процес отримає повідомлення (або  $P_2$ , або  $P_3$ ). Система може визначити алгоритм вибору процесу-приймача (наприклад, циклічна диспетчеризація (`round robin`), *where processes take turns receiving messages*). Система може ідентифікувати отримувача для відправника.
- Власником мейлбоксу може бути або процес, або ОС.
  - Якщо власником є процес (мейлбокс – частина його адресного простору), потрібно розрізняти власника (*owner* – отримує повідомлення тільки через свій мейлбокс) та користувача (лише відправляє повідомлення в будь-який з мейлбоксів).
  - Оскільки кожен мейлбокс має унікального власника, завжди відомо, хто отримувач повідомлення.
  - Коли процес, який володіє мейлбоксом, завершується, мейлбокс зникає.
  - Кожен процес, який надіслав повідомлення в цей мейлбокс, повинен отримати сповіщення про зникнення.

- 
- 
- Якщо власником мейлбоксу буде ОС, він існуватиме сам по собі.
    - Незалежно та не прикріплений до конкретного процесу.
  - ОС повинна забезпечувати механізм, який дозволяє процесу робити наступне:
    - Створити нову поштову скриньку.
    - Надсилати та приймати повідомлення через поштову скриньку.
    - Видалити поштову скриньку.
  - Процес, який створює новий мейлбокс, буде його власником за умовчанням.
    - Спочатку тільки процес-власник може отримувати повідомлення через мейлбокс.
    - Проте володіння та привілеї з отримування можна передавати іншим процесам за допомогою відповідних системних викликів.
    - У результаті це може призвести до появи кількох приймачів для кожного мейлбоксу.



# Синхронізація

---

- Взаємодія між процесами відбувається через виклики примітивів `send()` та `receive()`.
- Передача повідомлень може бути блокуючою (синхронною) або неблокуючою (асинхронною).
  - **Блокуюче надсилання (*Blocking send*)**. Процес відправки блокується, поки повідомлення отримується за допомогою приймаючого процесу або мейлбоксу.
  - **Неблокуюче надсилання (*Nonblocking send*)**. Процес відправки надсилає повідомлення а відновлює операцію.
  - **Блокуючий прийом (*Blocking receive*)**. Приймач блокується, поки не буде доступне повідомлення.
  - **Неблокуючий прийом (*Nonblocking receive*)**. Приймач отримує повідомлення або `null`.

```
message next_produced;

while (true) {
    /* produce an item in next_produced */

    send(next_produced);
}
```

Процес-producer при передачі повідомлень

15.03.2020

```
message next_consumed;

while (true) {
    receive(next_consumed);

    /* consume the item in next_consumed */
}
```

Процес-consumer при передачі повідомлень

@Марченко С.В., ЧДБК, 2020

37

- 
- Можливі різні комбінації `send()` та `receive()`.
    - Коли обидва `send()` і `receive()` блокуючі, маємо *рандеву* між `sender` та `receiver`.
    - Вирішення `producer-consumer problem` стає тривіальним, коли використовуються блокуючі інструкції `send()` та `receive()`.
    - Виробник викликає блокуючу функцію `send()` та очікує, поки повідомлення доставляється приймачу або в мейлбокс.
    - Аналогічно, коли споживач викликає `receive()`, він блокується, поки повідомлення доступне.
  - Як для прямої, так і непрямої взаємодії, повідомлення, якими обмінюються взаємодіючі процеси, розташовуються в тимчасовій черзі.
  - Такі черги можуть реалізуватись трьома способами:
    - **Нульова місткість (capacity).** Черга має максимальну довжину 0; зв'язок не має очікуючих повідомлень. Тоді відправник повинен блокуватись, поки приймач отримує повідомлення.
    - **Обмежена (Bounded) місткість.** Черга має скінченну довжину  $n$ ; тому максимум  $n$  повідомлень можуть залишатись в ній. Якщо черга не повна, коли надходить нове повідомлення, воно додається в чергу (копіюється або зберігає вказівник на повідомлення). Місткість зв'язку скінченна: якщо зв'язок заповнений, відправник повинен блокуватись, поки в черзі не з'явиться вільне місце.
    - **Необмежена (Unbounded) місткість.** Довжина черги потенційно нескінченна; довільна кількість повідомлень може в ній очікувати. Відправник ніколи не блокується.

# Доповідь

---

- Examples of IPC Systems
- **Communication in Client–Server Systems**