



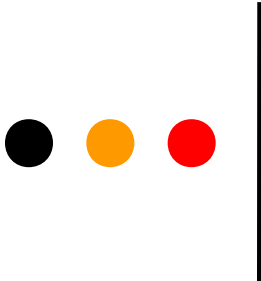
Операційні системи

Лекція 6

Взаємодія між потоками

Синхронізація

Взаємодія між процесами



План лекції

©Взаємодія між потоками

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

©Взаємодія між процесами

- Передавання повідомлень
- Розподілювана пам'ять
- Відображувана пам'ять



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: **$total_amount = total_amount + new_amount$** ;
 - 1. Зчитати значення **$total_amount$** (**$var1 = total_amount$**)
 - 2. Обчислити нове значення **$total_amount$** (**$var1 = var1 + new_amount$**)
 - 3. Записати нове значення **$total_amount$** (**$total_amount = var1$**)
- Послідовність подій:

1. T_A зчитав $total_amount$	$var1A = 100$
2. T_A обчислив $total_amount$	$var1A = 100 - 50$
3. T_A записав $total_amount$	$total_amount = 50$
<hr/>	
4. T_B зчитав $total_amount$	$var1B = 50$
5. T_B обчислив $total_amount$	$var1B = 50 + 100$
6. T_B записав $total_amount$	$total_amount = 150$

Все вірно

Проблема синхронізації: приклад (продовження)

Але результат може бути іншим!

- Послідовність подій 2:

1. T _A зчитав <code>total_amount</code>	<code>var1A = 100</code>
2. T _A обчислив <code>total_amount</code>	<code>var1A = 100 - 50</code>
<hr/>	
3. T _B зчитав <code>total_amount</code>	<code>var1B = 100</code>
4. T _B обчислив <code>total_amount</code>	<code>var1B = 100 + 100</code>
5. T _B записав <code>total_amount</code>	<code>total_amount = 200</code>
6. T _A записав <code>total_amount</code>	<code>total_amount = 50</code>

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

- Послідовність подій 3:

1. T _A зчитав <code>total_amount</code>	<code>var1A = 100</code>
2. T _B зчитав <code>total_amount</code>	<code>var1B = 100</code>
3. T _B обчислив <code>total_amount</code>	<code>var1B = 100 + 100</code>
4. T _A обчислив <code>total_amount</code>	<code>var1A = 100 - 50</code>
5. T _A записав <code>total_amount</code>	<code>total_amount = 50</code>
6. T _B записав <code>total_amount</code>	<code>total_amount = 200</code>

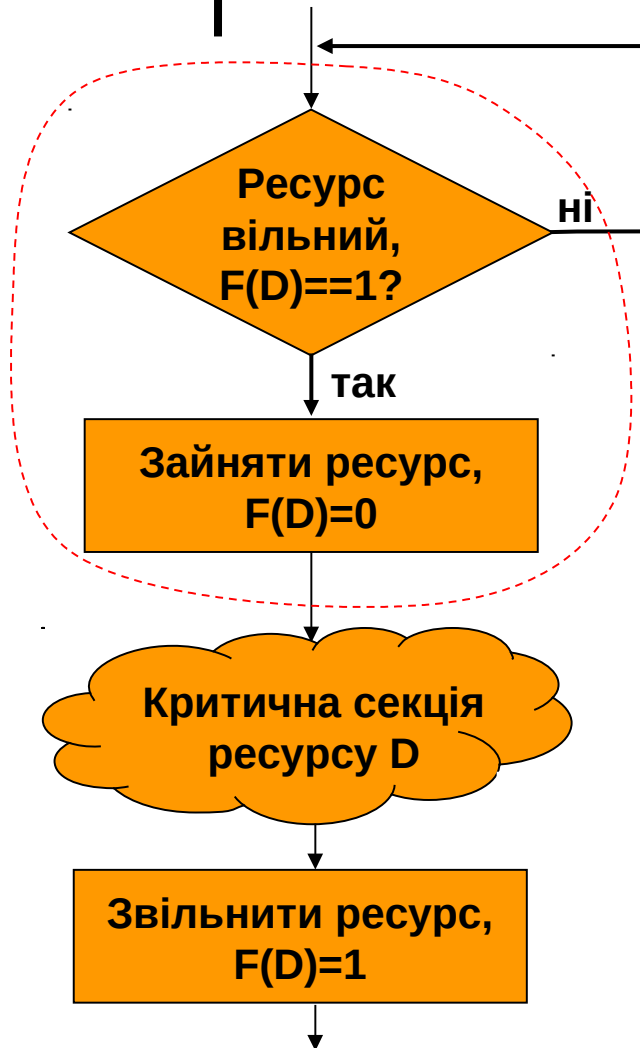
В результаті `total_amount == 200` (сума не знята)



Деякі визначення

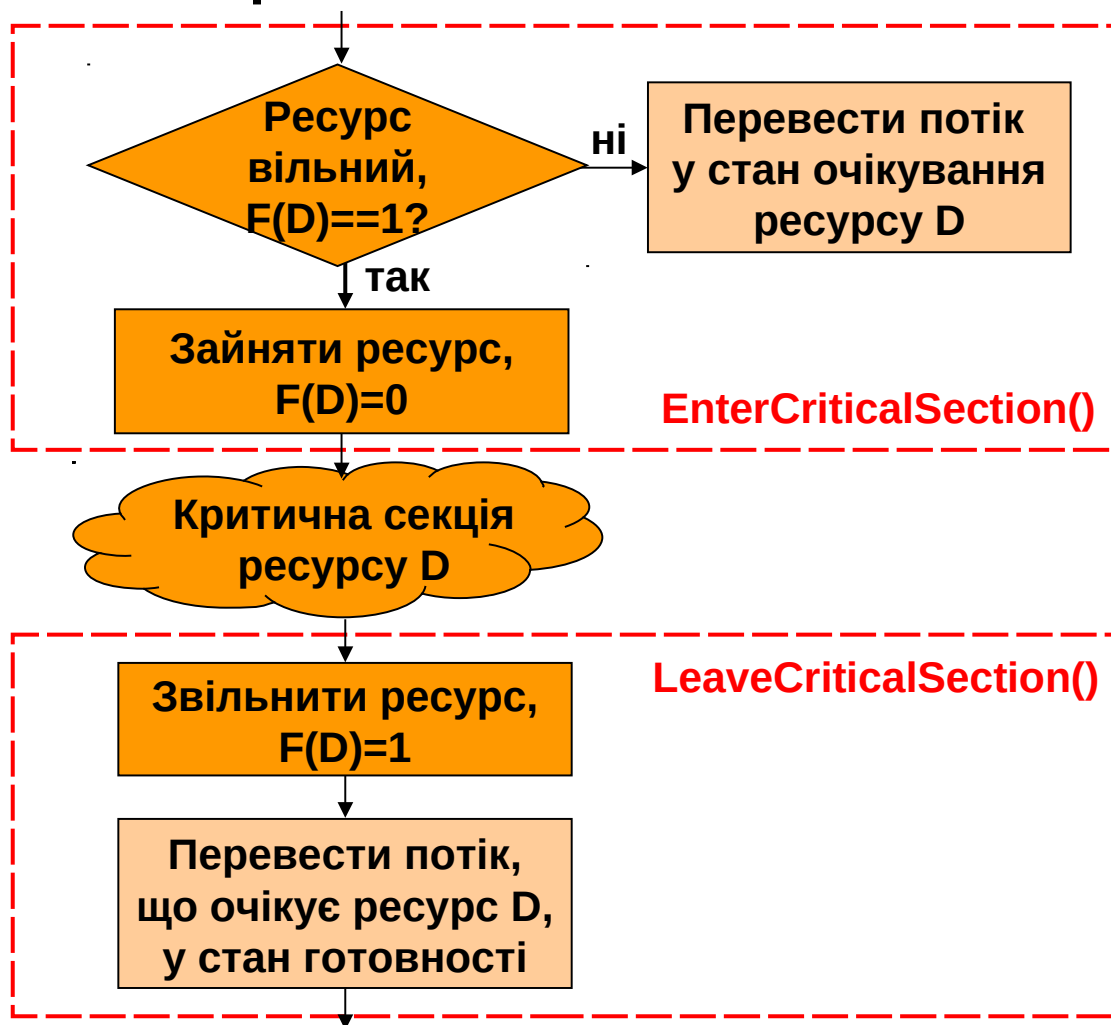
- Ситуація, коли 2 чи більше потоків обробляють спільні поділювані дані, і кінцевий результат залежить від співвідношення швидкостей потоків, називається **гонками** або **змаганням** (*race condition*)
- **Критична секція** (*critical section*) – частина програми, в якій здійснюється доступ до поділюваних 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
 - Застосовується у деяких функціях ядра
 - У загальному випадку 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

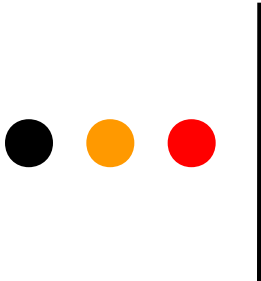


- Функція **WAIT(D)** переводить потік у стан очікування події звільнення ресурсу **D**
- Функція **POST(D)** викликається після звільнення ресурсу **D** і переводить усі потоки, що його очікують, у стан готовності (планувальник обере один з них на виконання)
- У POSIX – виклики **sleep()** і **wakeup()**
- 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)** { *// up()*
 S++;
 if (waiting_threads()) **POST(S);** *// wakeup()*
}
 - **P(S)** { *// down()*
 if (S > 0) S--;
 else **WAIT(S);** *// sleep()*
}
- 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

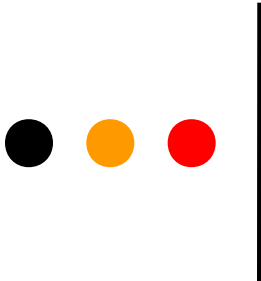
- **Потік-виробник** створює об'єкти і поміщає їх у буфер. Операція розміщення об'єкта не є атомарною
- **Потік-споживач** отримує об'єкти і видаляє їх з буфера. Операція видалення об'єкта не є атомарною
- Буфер має фіксовану довжину **N**
- Вимоги:
 1. Коли виробник або споживач працює з буфером, усі інші потоки повинні чекати завершення цієї операції
 2. Коли виробник має намір помістити об'єкт у буфер, а буфер повний, він має чекати, поки в буфері звільниться місце
 3. 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

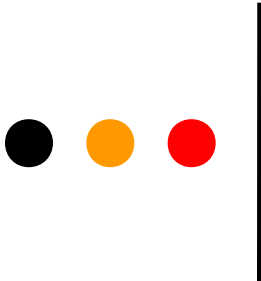
1. Організуємо критичну секцію для роботи з буфером. Для цього використаємо двійковий семафор **lock**

1. 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



Рішення задачі виробник-споживач

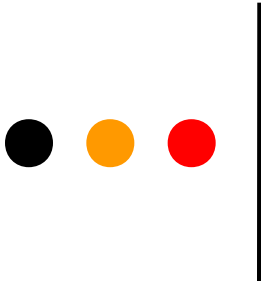
3. Для організації очікування споживача впровадимо семафор **full**, значення якого дорівнює кількості зайнятих місць у буфері
- Споживач перед спробою забрати об'єкт з буфера зменшує цей семафор, а якщо той дорівнював 0 – переходить у стан очікування
 - Виробник після того, як додасть об'єкт до буфера, 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) {  
        item = produce_new_item();  
        P(empty);  
        P(lock);  
        add_to_buffer(item);  
        V(lock);  
        V(full);  
    }  
}  
  
void consumer() {  
    while(1) {  
        P(full);  
        P(lock);  
        item = get_from_buffer();  
        V(lock);  
        V(empty);  
        consume_new_item(item);  
    }  
}
```

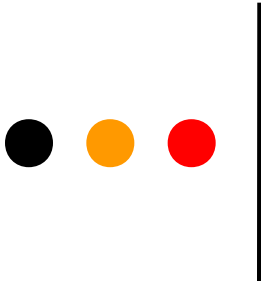


Проблема взаємних блокувань

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

```
void producer() {  
    while(1) {  
        item = produce_new_item();  
        P(lock);  
        P(empty);  
        add_to_buffer(item);  
        V(lock);  
        V(full);  
    }  
}
```

 - // працює постійно*
 - // створюємо об'єкт*
 - // входимо у критичну секцію*
 - // перевіряємо наявність місця*
 - // додаємо об'єкт до буфера*
 - // виходимо з критичної секції*
 - // повідомляємо про новий об'єкт*
- Перевірка умови з можливим очікуванням здійснюється **всередині** критичної секції. Можлива така послідовність дій:
 1. Виробник входить у критичну секцію, закриваючи семафор **lock**
 2. Виробник перевіряє семафор **empty** і очікує на ньому (буфер повний)
 3. 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):

Потік А

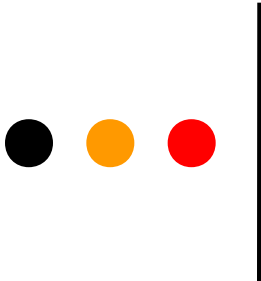
- 1 Захопити диск
- 2 Захопити принтер
- 3 Звільнити диск
- 4 Звільнити принтер

Потік В

- 1 Захопити принтер
- 2 Захопити диск
- 3 Звільнити диск
- 4 Звільнити принтер

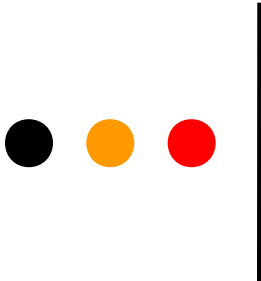
A1→A2→B1 (В очікує)→A3→A4→B1→B2→B3→B4 – **успішно**

A1→B1→B2 (В очікує)→A2 (А очікує) – **взаємне блокування**



Шляхи вирішення проблеми взаємних блокувань

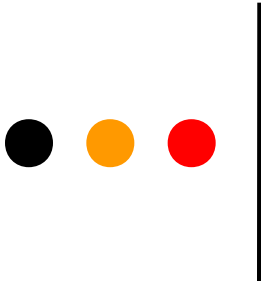
- Запобігання взаємних блокувань
 1. Запит ресурсів здійснюється у певній послідовності, спільній для усіх потоків
 2. Якщо один з потрібних ресурсів зайнятий, то потік має звільнити усі ресурси, що йому необхідні одночасно, і повторити спробу через деякий час
 3. Потік запитує усі ресурси у центрального 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**” (**rollback**) 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**
 - Наприклад, у POSIX: `pthread_mutex_lock()`, `pthread_mutex_unlock()`
- 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**
 - У POSIX: `pthread_mutex_trylock()`, якщо м'ютекс зайняти неможливо, повертає помилку з кодом **EBUSY**
- 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

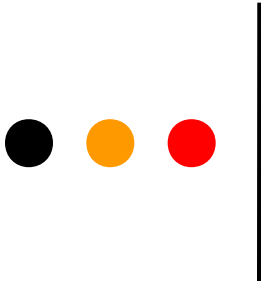
- Умовна змінна пов'язана з певним м'ютексом і даними, які він захищає
- Визначені три операції:
 - **сигналізація** (**signal**) – потік, що виконав дії з даними у критичній секції, перевіряє, чи не очікують на умовній змінній інші потоки, і якщо очікують – переводить один з них у стан готовності (потік 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)



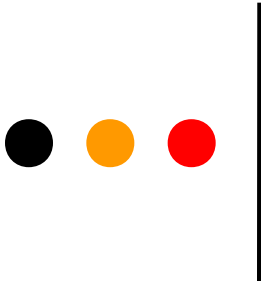
Монітор (засіб синхронізації високого рівня)

- **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 кожен 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
- Хоча концепція монітору не залежить від будь-якої алгоритмічної мови, реалізації моніторів не є специфічними для певних операційних систем, а якраз і є конструкціями алгоритмічних мов високого рівня
 - У різних мовах (Java, C#) реалізації є різними



Взаємодія між процесами

- ◎ **Міжпроцесова взаємодія** (*Interprocess communication, IPC*) реалізується такими основними методами:
 - Сигнали
 - Передавання повідомлень
 - Поділювана пам'ять (*shared memory*)
 - Відображувана пам'ять (*mapped memory*)
- ◎ **Сигнали** (аналог переривань) — найпростіший метод (UNIX)
 - Сигнали бувають:
 - **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!



Методи передавання повідомлень

©Важлива особливість — технології **передавання повідомлень** (*message passing*) не спираються на спільно використовувані дані

- Тому вони можуть застосовуватись як в межах одного комп'ютера, так і в мережах
- Процеси можуть обмінюватись повідомленнями навіть не знаючи про один одного

©Технології передавання повідомлень:

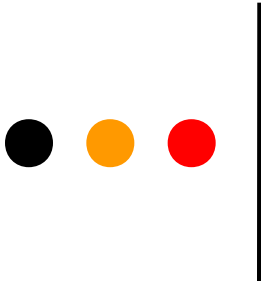
- **Канали**

- **безіменні** та **поіменовані** (*named pipes*)
- З каналом одночасно може працювати лише один процес

- **Черги повідомлень** (*message queues*)

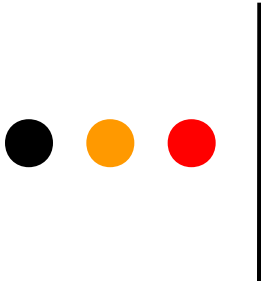
- З чергою одночасно може працювати кілька процесів

- **Сокети** (*sockets*) — насамперед, для взаємодії у мережі



Методи поділюваної пам'яті

- ◎ Поділювана пам'ять дає змогу двом процесам обмінюватись даними через спільний буфер
- ◎ Кожний з процесів має приєднати цей буфер до свого адресного простору
 - Для цього є спеціальні системні виклики
 - Перед приєднанням проводять перевірку прав процесу
- ◎ Буфер доступний за іменем
- ◎ Дані у буфері є спільно використовуваними для цих процесів — як для потоків
 - Як наслідок, виникає проблема синхронізації
- ◎ Поділювана пам'ять не надає засобів синхронізації, програміст має сам організувати їх по аналогії зі синхронізацією потоків



Технологія відображуваної пам'яті

- ©Зазвичай, відображувану пам'ять використовують у поєднанні з інтерфейсом файлової системи
 - Це називають: **файли, відображені у пам'ять** (*memory-mapped files*)
- ©За допомогою спеціального системного виклику певну частину адресного простору процесу однозначно пов'язують із вмістом певного файлу
 - Зазвичай, таким викликом є **mmap()**
- ©Після цього записування у цю ділянку пам'яті спричиняє зміну вмісту файлу, що відображений
 - При цьому змінений вміст стає доступним усім процесам, що мають доступ до цього файлу
 - Інші процеси так само можуть відобразити цей файл у свій адресний простір
- ©У деяких ОС саме відображувана пам'ять є базовим системним механізмом, на якому ґрунтуються інші види міжпроцесової взаємодії