ЛЕКЦІЯ 5. БАГАТОЗАДАЧНІСТЬ, ВЗАЄМОДІЯ ПОТОКІВ, МІЖПРОЦЕСОВА ВЗАЄМОДІЯ

5.1. Багатозадачність
5.2. Взаємодія потоків
5.3. Міжпроцесова взаємодія

5.1. Багатозадачність

У більшості сучасних ОС (таких, як лінія Windows, сучасні версії UNIX) може виконуватися одночасно багато процесів, а в адресному просторі кожного процесу – багато потоків. Ці системи підтримують багатопотоковість або реалізують модель потоків. Процес у такій системі називають багатопотоковим процесом.

Максимально можлива кількість процесів (захищених адресних просторів) і потоків, які в них виконуються, може варіюватися в різних системах.

Коли у системі з'являється новий процес, для старого процесу є два основних варіанти дій: продовжити виконання паралельно з новим процесом – такий режим роботи називають *асинхронним виконанням* або призупинити виконання доти, поки новий процес не буде завершений, – такий режим роботи називають *синхронним виконанням* (зазвичай для однопроцесорних комп'ютерів). Використання декількох потоків у застосуванні означає внесення в нього паралелізму (concurrency). ***Паралелізм*** – це одночасне (з погляду прикладного програміста) виконання дій різними фрагментами коду застосування. Така одночасність може бути реалізована на одному процесорі *шляхом перемикання задач* (випадок псевдопаралелізму), а може ґрунтуватися на *паралельному виконанні коду на декількох процесорах* (випадок справжнього паралелізму).

Можна виділити такі основні види паралелізму: багатопроцесорних систем; операцій введення-виведення; взаємодії з користувачем; розподілених систем.

Схеми моделей потоків визначаються співвідношенням між *потоками користувача* та *потоками ядра*:

Схема багатопотоковості M:1 (є найранішою) – кожен процес може містити багато потоків користувача, однак про наявність цих потоків ОС не відомо, вона працює тільки із процесами. За планування потоків і перемикання контексту відповідає бібліотека підтримки потоків. Сьогодні її практично не використовують через суттєві недоліки, що не відповідають ідеології багатопотоковості.

Схема багатопотоковості 1:1 ставить у відповідність кожному потоку користувача один потік ядра. Вона проста і надійна в реалізації і сьогодні є найпоширенішою.

Схема багатопотоковості M:N. У цій схемі присутні як потоки ядра, так і потоки користувача, які відображаються на потоки ядра так, що *один потік ядра може відповідати декільком потокам користувача*. Кількість потоків ядра може бути змінена програмістом для досягнення максимальної продуктивності. Схема є складною в реалізації і сьогодні здає свої позиції схемі 1:1.

5.2. Основні принципи взаємодії потоків

5.2.1. Задачі синхронізації

Потоки, які виконуються в рамках процесу паралельно, можуть бути незалежними або взаємодіяти між собою.

Потік є *незалежним*, якщо він не впливає на виконання інших потоків процесу, не зазнає впливу з їхнього боку, та не має з ними жодних спільних даних. Його виконання однозначно залежить від вхідних даних і називається *детермінованим*.

Усі інші потоки є такими, що взаємодіють. Ці потоки мають дані, спільні з іншими

потоками (вони перебувають в адресному просторі їхнього процесу), їх виконання залежить не тільки від вхідних даних, але й від виконання інших потоків, тобто вони є *недетермінованими* (далі розглянемо докладно приклади такої недетермінованості).

Дані, які є загальними для кількох потоків, називають спільно використовуваними даними (shared data). Це – найважливіша концепція багатопотокового програмування. Усякий потік може в будь-який момент часу змінити такі дані. Механізми забезпечення коректного доступу до спільно використовуваних даних називають механізмами синхронізації потоків. Відразу ж зазначимо, що проблеми синхронізації й організації паралельних обчислень є одними з найскладніших у практичному програмуванні.

Розглянемо ***приклад***, що може відбутися, коли потоки, які взаємодіють, разом використовуватимуть спільні дані без додаткових заходів із забезпечення синхронізації.

Уявімо, що для банківського обслуговування кожного користувача виділяють окремий потік (чим намагаються підвищити продуктивність системи у разі великої кількості одночасних запитів). Припустимо, що поміщення даних на вклад користувача зводиться до збільшення глобальної змінної `total_amount`. У цьому разі кожен потік під час зміни вкладу виконує такий найпростіший оператор:

```
total_amount = total_amount + new_amount;
```

Виникає запитання: чи можна дати гарантію, що внаслідок роботи із вкладом потік, який відповідає кожному користувачу, буде здатний збільшити значення total_amount на потрібну величину?

Насправді цей на перший погляд найпростіший оператор зводиться до послідовності таких дій:

- отримання поточного значення `total_amount` із глобальної пам'яті;
- збільшення його на `new_amount` і збереження результату в глобальній пам'яті.

Розглянемо випадок, коли два користувачі *A* і *B* спільно користуються одним і тим самим рахунком. На рахунку є 100 грошових одиниць. Користувач *A* збирається покласти на рахунок 1000, користувач *B* у цей самий час – 100. Потік T_A відповідає користувачу *A*, потік T_B – користувачу *B*. Розглянемо таку послідовність подій (***варіант 1***).

1. Потік T_A зчитує значення `total_amount`, рівне 100.
2. Потік T_B зчитує значення `total_amount`, теж рівне 100.
3. Потік T_A збільшує зчитане на кроці 1 значення `total_amount` на 1000, отримує 1100 і зберігає його у глобальній пам'яті.
4. Потік T_B збільшує зчитане на кроці 2 значення `total_amount` на 100, отримує 200 і теж зберігає його у глобальній пам'яті, перезаписуючи те, що зберіг потік T_A. У результаті внесок користувача *A* повністю втрачено.

   Тепер розглянемо іншу послідовність подій (***варіант 2***).
5. Потік T_A зчитує `total_amount`, збільшує його на 1000 і записує значення 1100 у глобальну пам'ять.
6. Потік T_B зчитує `total_amount`, рівний 1100, збільшує його на 100 і записує значення 1200 у глобальну пам'ять.

   У результаті обидва внески зареєстровані успішно.

Як бачимо, результат виконання наведеного раніше найпростішого фрагмента коду залежить від послідовності виконання потоків у системі. Це спричиняє до такого: в одній ситуації код може працювати, в іншій – ні, і передбачити появу помилки в загальному випадку неможливо. Таку ситуацію називають *станом гонок* або *змаганням*, що є однією з найбільш важко вловлюваних помилок, з якими зіштовхуються програмісти.

Отже, потрібно передбачити захист зміни спільно використовуваних даних від впливу інших потоків. Це і є ***основним завданням синхронізації***. Перейдемо до аналізу різних підходів до його розв'язання.

5.2.2. Критичні секції та блокування

Розглянемо використання найпростішої ідеї для вирішення проблеми змагань. Неважко помітити, як джерелом нашої помилки є те, що зовні найпростіша операція покладення грошей на рахунок насправді розпадається на кілька операцій, при цьому завжди залишається шанс втручання між ними якогось іншого потоку. У цьому випадку кажуть, що *вихідна операція не є атомарною*.

Звідси випливає, що розв'язанням проблеми змагання є перетворення фрагмента коду, який спричиняє проблему, в атомарну операцію, тобто в таку, котра гарантовано виконуватиметься цілковито без втручання інших потоків. Такий фрагмент коду називають ***критичною секцією*** (critical section):

```
// початок критичної секції
total_amount = total_amount + new_amount;
// кінець критичної секції
```

Тепер, коли два потоки візьмуться виконувати код критичної секції одночасно, той з них, що почав першим, виконає весь її код цілком до того, як другий почне своє виконання (другий потік чекатиме, поки перший не закінчить виконання коду критичної секції). У результаті підсумку гарантовано матимемо в нашій програмі послідовність подій за варіантом 2, і змагання не відбудеться ніколи.

Залишається відповісти на далеко не просте запитання: «Як нам змусити систему сприймати кілька операцій як одну атомарну операцію?»

Найпростішим розв'язанням такої задачі було б *заборонити переривання на час виконання коду критичної секції*. Такий підхід, хоча й розв'язує задачу в принципі, на практиці не може бути застосовуваний, оскільки внаслідок зациклення або аварії програми у критичній секції *вся система може залишитися із заблокованими перериваннями*, а отже, у непрацездатному стані.

Раціональнішим розв'язанням є використання ***блокувань*** (locks).

Блокування – це механізм, який не дозволяє більш як одному потокові виконувати код критичної секції.

Використання блокування зводиться до двох дій: запровадження (блокування, функція `acquire_lock()`) і зняття блокування (розблокування, функція `release_lock()`). У разі заблокування перевіряють, чи не було воно вже зроблене іншим потоком, і якщо це так, цей потік переходить у стан очікування, інакше він запроваджує блокування і входить у критичну секцію. Після виходу із критичної секції потік знімає блокування.

```
acquire_lock(lock);
// критична секція
release_lock(lock);
```

Так реалізовують властивість взаємного виключення, звідси походить інша назва для блокування – ***м'ютекс*** (mutex, скорочення від mutual exclusion).

Є низка алгоритмів, які дають можливість коректно реалізувати блокування, спираючись на атомарність звичайних операцій записування в пам'ять і читання з пам'яті. До них належать ***алгоритми Деккера і Петерсона та алгоритм булочної*** (bakery algorithm). Із цими алгоритмами можна ознайомитися в літературі, тут же обмежимося тими алгоритмами, які спираються на наявну апаратну підтримку.

Для організації блокування в архітектурі IA-32 може бути використана спеціальна інструкція процесора, яку називають *«перевірити і заблокувати»* (Test & Set Lock, TSL). Параметром цієї інструкції є деяка адреса в пам'яті (наприклад, яка визначає місцезнаходження змінної блокування). Розглянемо, що відбувається під час виконання цієї інструкції (для простоти вважатимемо, що в пам'яті, з якою працює ця інструкція, може зберігатися тільки 0 або 1, при цьому 1 означає «блокування є», 0 – «блокування відсутнє»).

```
void acquire_lock(int lock) {
// якщо блокування немає (lock == 0), задати (lock=1)
// і вийти - ми ввійшли у критичну секцію
// інакше чекати, поки блокування не знімуть, і не міняти lock
while (TSL(lock) != 0);
}
void release_lock(int lock) {
// зняти блокування
lock = 0; }
```

Головним недоліком описаної технології є те, що потік повинен постійно перевіряти в циклі, чи не зняли блокування. Таку ситуацію називають *активним очікуванням* (busy waiting), а таке блокування – *спін-блокуванням* (spinlock). Використання циклу під час активного очікування завантажує процесор і допустиме тільки упродовж короткого часу (це справедливо для однопроцесорних систем, у багатопроцесорних використання спін-блокувань може бути виправдане тому, що під час виконання циклу активного очікування одним процесором інші можуть продовжувати свою роботу).

Альтернативою активному очікуванню є операція *призупинення потоку* (thread blocking, sleep). При цьому потік за умови блокування негайно або через якийсь час переходить зі стану виконання у стан очікування, передаючи процесор іншому потокові. Для того щоб реалізувати критичну секцію за допомогою цієї операції, потрібно доповнити її операцією виведення потоку зі стану очікування *(поновленням потоку,* wakeup). Робота із критичною секцією в цьому разі буде мати такий вигляд:

```
int lockguard = 0;
   // блокування для критичних секцій усередині
   // acquire_lock() і release_lock()
void acquire_lock(int lock) {
   // перевірити, чи безпечно виконувати цю дію
  while (TSL(lockguard) != 0);
     // якщо блокування запроваджене - призупинити потік
  if (lock == 1) sleep();
     // інакше заблокувати й увійти у критичну секцію
  else lock = 1;
    // дає можливість іншим потокам виконувати цю дію
    // цей код повинен виконуватися завжди у разі завершення
операції
  lockguard = 0;
}
void release_lock(int lock) {
    // перевірити на безпеку виконання цієї дії
  while (TSL(lockguard) != 0);
    // розбудити один із призупинених потоків, якщо вони є
  if (waiting_threads()) wakeup(some_thread);
```

```
    // якщо призупинені потоки відсутні - зняти блокування
  else lock = 0;
    // дає можливість іншим потокам виконувати цю дію
    // цей код повинен виконуватися завжди у разі завершення операції
    lockguard = 0;
}
```

Призупинення й поновлення потоку спричиняють перемикання контексту, що є досить тривалою операцією. У результаті в найкращому разі вхід у критичну секцію займе набагато менше часу, ніж було б витрачено на перемикання контексту, у гіршому – буде затрачено часу всього удвічі більше, ніж під час негайного призупинення потоку.

Розглянемо набір ***готових механізмів синхронізації***, які надають сучасні ОС.

Синхронізаційні механізми поділяють на такі основні категорії:

- *універсальні низького рівня*, які можна використовувати різними способами (***семафори***);
- *прості низького рівня*, кожен з яких пристосований до розв'язання тільки однієї задачі (***м'ютекси та умовні змінні***);
- *універсальні високого рівня*, виражені через прості; до цієї групи належить ***концепція монітора***, яка може бути виражена через м'ютекси та умовні змінні;
- *високого рівня*, пристосовані до розв'язання конкретної синхронізаційної задачі (***блокування читання-запису і бар'єри***).

5.2.3. Семафори

Концепцію семафорів запропонував у 1965 році Е. Дейкстра – відомий голландський фахівець у галузі комп'ютерних наук. Семафори є найстарішими синхронізаційними примітивами з числа тих, які застосовуються на практиці.

Семафор – це спільно використовуваний невід'ємний цілочисельний лічильник, для якого задано початкове значення і визначено такі атомарні операції:

- *Зменшення семафора* (`down`): якщо значення семафора більше від нуля, його зменшують на одиницю, якщо ж значення дорівнює нулю, цей потік переходить у стан очікування доти, поки воно не стане більше від нуля (кажуть, що потік «очікує на семафорі» або «заблокований на семафорі»). Цю операцію називають також *очікуванням* – `wait`. Ось її псевдокод:

```
void down (semaphore_t sem) {
if (sem > 0) sem--;
else sleep(); }
```

- *Збільшення семафора* (`up`): значення семафора збільшують на одиницю; коли при цьому є потоки, які очікують на семафорі, один із них виходить із очікування і виконує свою операцію `down`. Якщо на семафорі очікують кілька потоків, то внаслідок виконання операції `up` його значення залишається нульовим, але один із потоків продовжує виконання (у більшості реалізацій вибір цього потоку буде випадковим). Цю операцію також називають ***сигналізацією*** – `post`:

```
void up (semaphore_t sem) {
sem++;
if (waiting_threads()) wakeup (some_thread);
}
```

Фактично значення семафора визначає кількість потоків, що може пройти через цей

семафор без блокування. Коли для семафора задане нульове початкове значення, то він блокуватиме всі потоки доти, поки якийсь потік його не «відкриє», виконавши операцію `up`. Операції `up` і `down` можуть бути виконані будь-якими потоками, що мають доступ до семафора.

Дейкстра використовував для операції `down` позначення P (від голландського probieren – перевіряти), а для операції `up` – позначення V (від голландського ver-horen – збільшувати). Ці позначення часто використовують у літературі.

5.2.4. М'ютекси

М'ютексом називають синхронізаційний примітив, що не допускає виконання деякого фрагмента коду більш як одним потоком. Фактично м'ютекс є реалізацією блокування на рівні ОС. Його основне завдання – блокувати всі потоки, які намагаються отримати доступ до коду, коли цей код уже виконує деякий потік.

М'ютекс може перебувати у двох станах: вільному і зайнятому. Початковим станом є «вільний». Над м'ютексом можливі дві атомарні операції.

- *Зайняти м'ютекс* (mutex_lock): якщо м'ютекс був вільний, він стає зайнятим, і потік продовжує своє виконання (входячи у критичну секцію); якщо м'ютекс був зайнятий, потік переходить у стан очікування (кажуть, що потік «очікує на м'ютексі», або «заблокований на м'ютексі»), виконання продовжує інший потік. Потік, який зайняв м'ютекс, називають *власником м'ютекса* (mutex owner).
- *Звільнити м'ютекс* (mutex_unlock): м'ютекс стає вільним; якщо на ньому очікують кілька потоків, з них вибирають один, він починає виконуватися, займає м'ютекс і входить у критичну секцію. У більшості реалізацій вибір потоку буде випадковим. Звільнити м'ютекс може тільки його власник. Ось псевдокод цієї операції:

```
mutex_unlock (mutex_t mutex) {
if (mutex.owner != this_thread) return error;
mutex.state = free;
if (waiting_threads()) wakeup (sone_thread);
}
```

Основною ***відмінністю м'ютексів від семафорів є*** те, що звільнити м'ютекс може тільки його власник, тоді як змінити значення семафора може будь-який потік, котрий має до нього доступ. Ця відмінність досить суттєва і робить реалізацію взаємних виключень за допомогою м'ютексів простішою (з коду завжди ясно, який потік може змінити стан м'ютекса).

5.2.5. Умовні змінні та концепція монітора

Умовною змінною називають синхронізаційний примітив, який дає змогу організувати очікування виконання умови всередині критичної секції, заданої м'ютексом. Умовна змінна завжди пов'язана із конкретним м'ютексом і даними, захищеними цим м'ютексом. Для умовної змінної визначено такі операції.

- *Очікування* (**`wait`**). Додатковим вхідним параметром ця операція приймає м'ютекс, який повинен перебувати в закритому стані. Виклик `wait` відбувається в ситуації, коли не виконується деяка умова, потрібна потоку для продовження роботи. Внаслідок виконання `wait` потік (позначатимемо його *Tw*) припиняється (кажуть, що він «очікує на умовній змінній»), а м'ютекс відкривається (ці дві дії відбуваються атомарно). Так інші потоки отримують можливість увійти в критичну секцію і змінити там дані, які вона захищає, можливо, виконавши умову, потрібну потоку

Tw. На цьому операція `wait` не завершується – її завершить інший потік, викликавши операцію `signal` після того, як умову буде виконано.

- *Сигналізація* (**`signal`**). Цю операцію потік (назвемо його T_S) має виконати після того, як увійде у критичну секцію і завершить роботу з даними (виконавши умову, яку очікував потік, що викликав операцію `wait`). Ця операція перевіряє, чи немає потоків, які очікують на умовній змінній, і якщо такі потоки є, переводить один із них (*Tw*) у стан готовності (цей потік буде поновлено, коли відповідний потік *Ts* вийде із критичної секції). Внаслідок поновлення потік *Tw* завершує виконання операції `wait` – блокує м'ютекс знову (поновлення і блокування теж відбуваються атомарно). Якщо немає жодного потоку, який очікує на умовній змінній, операція `signal` не робить нічого, і інформацію про її виконання в системі не зберігають.
- *Широкомовна сигналізація* (**`broadcast`**) відрізняється від звичайної тим, що переведення у стан готовності і, зрештою, поновлення виконують для всіх потоків, які очікують на цій умовній змінній, а не тільки для одного з них.

Отже, виконання операції `wait` складається з таких етапів: відкриття м'ютекса, очікування (поки інший потік не виконає операцію `signal` або `broadcast`), закриття м'ютекса.

По суті, це перша *неатпомарна* операція, визначена для синхронізаційного примітива, але така відсутність атомарності цілком контрольована (завжди відомо, де потік T_W перейшов у стан очікування і що його з цього стану виведе).

Наведемо принципові ***відмінності умовних змінних від семафорів***, які використовують для організації очікування за умовою.

- *Умовні змінні* можуть бути *використані* лише *всередині критичних секцій*, при цьому інші потоки можуть входити у критичну секцію під час очікування на умовній змінній. Семафори небезпечно використовувати всередині критичних секцій, оскільки це зазвичай призводить до взаємного блокування.
- Умовні змінні не зберігають стану, а семафори – зберігають.

Поняття монітора

Як ми бачимо, умовні змінні не використовують окремо від м'ютексів, причому є кілька правил взаємодії між цими примітивами. Ці правила є підґрунтям поняття монітора – синхронізаційної концепції вищого рівня.

Монітором називають набір функцій, які використовують один загальний м'ютекс і нуль або більше (>= 0) умовних змінних для керування паралельним доступом до спільно використовуваних даних відповідно до певних правил. Функції цього набору *називають функціями монітора*.

Ось правила, яких слід дотримуватися у разі реалізації монітора.

- Під час входу в кожну функцію монітора потрібно займати м'ютекс, під час виходу – звільняти.
- Під час роботи з умовною змінною (і під час очікування, і під час сигналізації) необхідно завжди вказувати відповідний м'ютекс. Не можна працювати з умовними змінними, якщо м'ютекс незайнятий.
- Під час перевірки на виконання умови очікування потрібно використати цикл, а не умовний оператор.

Ідея монітора була вперше запропонована в 1974 році відомим ученим у галузі комп'ютерних наук Ч. А. Хоаром. Монітор часто розуміють як високорівневу конструкцію мови програмування (як приклад такої мови звичайно наводять Java), а саме як набір функцій або методів класу, всередині яких автоматично зберігається неявний загальний

м'ютекс разом із операціями очікування і сигналізації. Насправді, як ми бачимо, концепція монітора може ґрунтуватися на базових примітивах – м'ютексах і умовних змінних – і не повинна бути обмежена якоюсь однією мовою.

Монітори Хоара відрізняються від тих, що були розглянуті тут (ці монітори ще називають *MESA-моніторами* за назвою мови, у якій вони вперше з'явилися). Головна відмінність полягає у реалізації сигналізації.

- У моніторах Хоара після сигналізації потік T_S негайно припиняють, і керування переходить до потоку *Tw*, який при цьому захоплює блокування. Коли потік *Tw* вийде із критичної секції або знову виконає операцію очікування, потік *Ts* буде поновлено.
- У MESA-моніторах, після сигналізації потік *Ts* продовжує своє виконання, а потік *Tw* просто переходить у стан готовності до виконання. Він зможе продовжити своє виконання, коли потік *Ts* вийде з монітора (чекати цього доведеться недовго, тому що звичайно сигналізація відбувається наприкінці функції монітора).

  На практиці використовують переважно MESA-монітори.

Загальна стратегія організації паралельного виконання

Для коректної організації виконання багатопотокових програм особливо важливі два із розглянутих раніше правила.

- М'ютекс захищає не код критичної секції, а спільно використовувані дані всередині цієї секції.
- Виклик `wait` для умовної змінної відбувається тоді, коли не виконується умова, пов'язана зі спільно використовуваними даними всередині критичної секції, виклик `signal` – коли умова, пов'язана з цими даними, починає виконуватися.

  Як бачимо, м'ютексами і умовними змінними керують дані, які вони захищають.

Блокування читання-записування

М'ютекси є засобом, який захищає спільно використовувані дані від будь-якого одночасного доступу з боку кількох потоків – будь то читання чи зміна. Насправді нам не завжди потрібен такий однозначний захист, наприклад, для певного типу задач хотілося б розрізняти читання спільно використовуваних даних та їхню модифікацію (для того, щоб, скажімо, дозволяти читання кільком потокам одночасно, а модифікацію – тільки одному). Для розв'язання такої задачі використовують *блокування читання-записування* (read-write locks).

Блокування читання-запису – це синхронізаційний примітив, для якого визначені два режими використання: відкриття для читання і відкриття для записування. При цьому повинні виконуватися такі умови:

- будь-яка кількість потоків може відкривати таке блокування для читання, коли немає жодного потоку, що відкрив його для записування;
- блокування може відкриватися для записування тільки за відсутності потоку, що відкрив його для читання або для записування.

Простіше кажучи, читати дані може будь-яка кількість потоків одночасно за умови, що ніхто ці дані не змінює; змінювати дані можна тільки тоді, коли їх ніхто не читає і не змінює.

Такі блокування корисні для даних, які зчитуються частіше, ніж модифікуються (наприклад, більшість СУБД реалізує блокування такого роду для забезпечення доступу до бази даних).

Типи блокувань

Розрізняють два типи блокувань читання-записування: *з кращим читанням* і з *кращим записом*. Відмінність між ними виявляється тоді, коли потік намагається відкрити таке блокування для читання за умови, що він робить це не першим і що є призупинені потоки, які очікують можливості відкрити це блокування для записування.

- У разі *кращого читання* потік негайно відкриває блокування для читання і продовжує свою роботу незалежно від того, є потоки-записувачі, що очікують, чи ні. Потоки-записувачі продовжують своє очікування.
- У разі *кращого записування* за наявності потоків-записувачів, що очікують, потік-читач припиняється й не буде поновлений доти, поки всі записувачі не виконають свої дії і не закриють блокування.

Зазначимо, що для обох типів потік-записувач не може відкрити блокування, поки його тримає відкритим хоча б один читач, – перевага надається тільки новим потокам-читачам, які намагаються відкривати додаткові блокування.

Операції блокувань

Розглянемо операції, допустимі для блокувань читання-записування.

- *Відкриття для читання* (`rwlock_rdlock`). Якщо є потік, який відкрив блокування для записування, поточний потік припиняють. Якщо такий потік відсутній:
  - у разі кращого читання блокування відкривають для читання і потік продовжує своє виконання;
  - у разі кращого записування перевіряють, чи немає призупинених потоків, які очікують відкриття цього блокування для записування; якщо вони є – потік припиняють, якщо немає – блокування відкривають для читання і потік продовжує своє виконання.

При цьому необхідно, щоб кілька потоків могли відкрити блокування для читання, тому в разі, коли блокування вже відкрите для читання, для його нового відкриття збільшують внутрішній лічильник потоків-читачів.

- *Відкриття для записування* (`rwlock_wrlock`). Якщо є потік, який відкрив блокування для читання або записування, поточний потік припиняють; коли жодного такого потоку немає, блокування відкривають для записування і потік продовжує своє виконання.
- *Закриття* (`rwlock_unlock`). У разі наявності кількох потоків, які відкрили блокування для читання, воно залишається відкритим для читання, і внутрішній лічильник потоків-читачів зменшують на одиницю. Якщо блокування відкрите для читання тільки одним потоком (лічильник дорівнює одиниці) його знімають, якщо є потоки-записувачі, які очікують на цьому блокуванні, один із них поновлюють. Коли блокування відкрите для записування, його знімають, при цьому за наявності потоків-читачів, що очікують, всі вони поновлюються, а з потоків-записувачів, що очікують, поновлюють тільки один. Якщо очікують і читачі й записувачі, результат залежить від типу блокування (у разі кращого читання або якщо жодного записувача немає, поновлюють усіх читачів, а якщо читачі відсутні у разі кращого записування – поновлюють одного з записувачів.

Блокування читання-записування, як і м'ютекси, мають власника, тому не можна закрити блокування в потоці, який його не відкривав.

Ось псевдокод прикладу використання таких блокувань для синхронізації доступу до банківського рахунку:

```
// блокування для захисту рахунку
rwlock_t account_lock;
// сума на рахунку - спільно використовувані дані
volatile float amount;
// додавання на рахунок
void set_amount (float new_amount) {
  rwlock_wrlock(account_lock);  // відкриття для записування
  amount = amount + new_amount;
  rwlock_unlock(account_lock};
}
// перегляд рахунку
float get_amount () {
  float cur_amount;
  rwlock_rdlock(account_lock);    // відкриття для читання
  cur_amount = amount;
  rwlock_unlock(account_lock);
  return cur_amount;
}
```

Голодування

Під час використання блокувань читання-записування можуть виникати проблеми, пов'язані з типом блокування. Якщо система працює за умов значного навантаження, для блокувань із кращим читанням можлива така послідовність дій потоків.

1. Потік T_{R1} відкриває блокування для читання L.
2. Потік T_{W1} намагається відкрити блокування L для записування і призупиняється.
3. Потік T_{R2} відкриває блокування L для читання.
4. Потік T_{R1} виконує операцію закриття блокування, але блокування L залишається зайнятим T_{R2}.
5. Потік T_{R3} відкриває блокування L для читання.
6. Потік T_{R2} виконує операцію закриття блокування, але блокування L залишається зайнятим T_{R3} (і т. д.).

У результаті додаткові потоки-читачі захоплюватимуть блокування L до того, як його звільнять усі попередні читачі. Це означає, що L ніколи повністю не звільниться, і потік-записувач T_{W1} ніколи не отримає керування. Таку ситуацію, як і під час планування, називають голодуванням.

Для блокувань із кращим записуванням голодування потоку-записувача бути не може. У нашому випадку потік T_{R2} на кроці 3 призупиниться і після закриття блокування потоком T_{R1} на кроці 4 потік T_{W1} дістане можливість виконуватися. З іншого боку, у цьому випадку за певних умов можна отримати голодування потоків-читачів.

Остаточного розв'язання проблеми голодування не запропоновано, рекомендують ретельно зважувати можливість використання різних типів блокувань і розраховувати можливість навантаження, яке може призвести до голодування.

5.2.6. Синхронізація за принципом бар'єра

Іноді буває зручно розбити задачу (наприклад, складний розрахунок) на кілька послідовних етапів, при цьому виконання наступного етапу не може розпочатися без *цілковитого* завершення попереднього. Якщо на кожному етапі роботу розподілити по окремих потоках, які паралельно виконуватимуть кожен свою частину, виникає

запитання: яким чином дочекатися завершення всіх потоків попереднього етапу до початку наступного?

Звичайне використання `thread_join()` тут не підходить, тому що приєднати можна якийсь конкретний потік, а не цілий набір потоків. Для розв'язання цього завдання запропоновано концепцію спеціального синхронізаційного об'єкта – бар'єра.

Бар'єр – це блокування, яке зберігають доти, поки кількість потоків, що очікують, не досягне деякого наперед заданого числа, після чого всі ці потоки продовжують своє виконання.

Так, якщо наприкінці етапу задачі поставити бар'єр, він буде утримувати потоки від продовження, поки вони всі не завершать роботу цього етапу.

Розглянемо дві операцій бар'єра.

- *Ініціалізація бар'єра* (`barrier_init`). Параметром даної операції є *n* – кількість потоків, які ми синхронізуватимемо на цьому бар'єрі. Внутрішній лічильник призупинених потоків покладають рівним нулю. Значення *n* після виклику `barrier_init` міняти не можна.
- *Очікування на бар'єрі* (`barrier_wait`). Внаслідок даної операції лічильник призупинених потоків збільшується на одиницю. Якщо він не досягнув *n*, цей потік призупиняють (кажуть, що він «очікує на бар'єрі»). Якщо ж після збільшення лічильник виявляється рівним *n* (досягнуто максимальної кількості потоків для бар'єра), усі потоки, що очікували на бар'єрі, поновлюються (ситуація переходу бар'єра), при цьому одному з них повертають спеціальне значення (зазвичай -1), а іншим – нуль. Надалі потік, що отримав це спеціальне значення, може зробити деякі підсумкові дії після переходу бар'єра. При переході бар'єра його «скидають» (внутрішній лічильник призупинених потоків знову покладають рівним 0) і він починаючи із наступної операції `barrier_wait` блокуватиме потоки знову.

  Ось псевдокод прикладу використання бар'єра:

```
barrier_t barrier_step;  // бар'єр етапу
void thread_fun() {      // функція потоку, розбита на етапи
  int res;
  step1(); // виконання дій етапу 1
  res = barrier_wait(barrier_step);
       // очікування завершення етапу 1
  if (res == -1) stepl_done();
  step2(); // виконання дій етапу 2
  res = barrier_wait(barrier_step);
       // очікування завершення етапу 2
  if (res == -1) step2_done();
  // тощо
}
// n - кількість потоків для виконання поетапної задачі
void run() {
  thread_t threads[n];
  barrier_init(barrier_step, n):    // бар'єр для n потоків
  for (i=0; i<n; i++)
    threads[i] = thread_create(thread_fun); // створюємо n потоків
  for (i=0; i<n; i++)
    thread_join(threads[i]); // очікуємо завершення n потоків
}
```

Тут виконання функції потоку розбито на етапи. Кожен етап не може бути виконаний без цілковитого завершення попереднього. Створюють *n* потоків, кожен з яких приходить до бар'єра наприкінці етапу. Після переходу бар'єра потоки починають виконувати дії наступного етапу, в кінці якого стоїть новий бар'єр і т. д. Після кожного етапу один із потоків підбиває його результати, викликаючи функцію `done()` для цього етапу.

5.2.7. Взаємодія потоків

Механізми синхронізації потоків у Windows реалізовані на трьох рівнях: синхронізація в ядрі (на рівні потоків ядра), виконавчій системі та в режимі користувача у підсистемі Win32.

Основними механізмами *синхронізації ядра* Windows є спін-блокування. Код *виконавчої системи* також може користуватися спін-блокуваннями, але вони пов'язані з активним очікуванням і їхнє застосування обмежується лише організацією взаємного виключення впродовж короткого часу. Складніші задачі синхронізації розв'язують за допомогою *диспетчерських об'єктів* (dispatcher objects) і *ресурсів виконавчої системи* (executive resources).

Диспетчерські об'єкти – це об'єкти виконавчої системи, що можуть бути використані разом із сервісами очікування менеджера об'єктів.

Ресурси виконавчої системи можуть бути використані тільки в коді ОС, *коду користувача* вони *недоступні.* Вони не є об'єктами виконавчої системи, а реалізовані як структури даних зі спеціальними операціями, які можна виконувати над ними. Такий ресурс допускає різні режими доступу – взаємне виключення (як м'ютекс) і доступ за принципом блокування читання-записування.

На основі диспетчерських об'єктів підсистема Win32 реалізує набір об'єктів синхронізації *режиму користувача*. До них належать, зокрема, ***м'ютекси, семафори і події.***

Потік може синхронізуватися з диспетчерським об'єктом через виконання очікування на його дескрипторі. Для очікування сигналізації одного об'єкта у Win32 використовують функцію `WaitForSingleObject()`:

```
DWORD WaitForSingleObject(HANDLE handle, DWORD timeout);
```

Параметр `handle` визначає дескриптор синхронізаційного об'єкта, а параметр `timeout` задає максимальний час очікування в мілісекундах (значення INFINITE свідчить про нескінченне очікування). Функція `WaitForSingleObject()` повертає такі значення: `WAIT_OBJECT_0` – відбулася сигналізація об'єкта; `WAIT_TIMEOUT` – минув час очікування (якщо `timeout` не дорівнював INFINITE), а об'єкт свого стану так і не змінив.

Можна очікувати сигналізації не одного об'єкта, а кількох одразу (деякий аналог використання кількох умов очікування для умовної змінної). Для цього використовують функцію `WaitForMultipleObjects()`:

```
DWORD WaitForMultipleObjects(
DWORD count,            // довжина масиву дескрипторів
CONST HANDLE *handles, // масив дескрипторів
BOOL waitall,           // прапорець режиму очікування
DWORD timeout        // аналогічно до WaitForSingleObject );
```

Функція приймає масив дескрипторів `handles` завдовжки `count` (максимальна довжина масиву 64 елементи). Режим очікування і повернене значення залежать від прапорця `waitall`.

- Якщо `waitall` дорівнює TRUE, задано режим очікування всіх об'єктів. Очікування завершується у разі здійснення сигналізації всіх об'єктів, функція поверне

WAIT_OBJECT_0.

- Якщо waitall дорівнює FALSE, задано режим очікування одного об'єкта. Очікування завершується у разі здійснення сигналізації хоча б одного з об'єктів, функція поверне WAIT_OBJECT_0 + i, де i – індекс дескриптора цього об'єкта в масиві handles.

Найпростішими синхронізаційними об'єктами із дескрипторами є ***м'ютекси***. Для створення м'ютекса використовують функцію CreateMutex():

```
HANDLE CreateMutex() // повертає дескриптор м'ютекса
//атрибути безпеки, нульовий покажчик, якщо за замовчуванням
LPSECURITY_ATTRIBUTES sec_attrs,
BOOL init_locked, // відразу зайнятий поточним потоком, якщо TRUE
LPCTSTR mutex_name );// ім'я або нульовий покажчик, якщо не потрібно
```

Ім'я м'ютекса задавати не обов'язково, якщо потрібно звертатися до м'ютекса тільки в рамках цього процесу.

Ось приклад створення м'ютекса:

```
HANDLE mutex = CreateMutex (0, FALSE, 0);
```

Щоб ***зайняти м'ютекс***, необхідно викликати функцію очікування:

```
WaitForSingleObject(mutex, INFINITE);
```

Якщо м'ютекс вільний, потік його займає і продовжує виконання. Коли м'ютекс зайнятий (заблокований іншим потоком), то поточний потік очікує, поки його не буде звільнено (або поки не мине час очікування, якщо другий параметр не дорівнює INFINITE). У першому випадку він блокує м'ютекс, у другому – ні, і WaitForSingleObject() повертає WAIT_TIMEOUT.

Звільнення м'ютекса здійснює функція ReleaseMutex(mutex). При цьому відбувається сигналізація м'ютекса і потік, що його очікує, поновлюється. Якщо кілька потоків очікували один і той самий м'ютекс, його буде зайнято одним із потоків, при цьому не можна заздалегідь передбачити яким. Після використання дескриптор потрібно закрити функцією CloseHandle(mutex).

Наведемо приклад роботи із м'ютексом Win32 API.

```
HANDLE mutex = CreateMutex (0, FALSE, 0);
WaitForSingleObject(mutex, INFINITE);
// критична секція
ReleaseMutex(mutex);
CloseHandle(mutex);
```

М'ютекси Win32 API є рекурсивними.

Для роботи з ***умовними змінними*** Win32 API використовує події.

Події – це засоби сигналізації об'єктів для виклику деяких дій. Відповіддю на події звичайно є поновлення одного або кількох потоків. Фактично події можна розуміти як спробу реалізувати функціональність умовних змінних без інтеграції з м'ютексами. Але відмова від такої інтеграції призводить до того, що під час роботи з подіями виникають певні проблеми. Потрібно враховувати можливість «втручання» інших потоків між передачею події та реакцією на неї.

Розрізняють два типи подій – з *автоматичним* (auto reset event) та *ручним скиданням* (manual reset event), а також два засоби їхньої сигналізації – функції PulseEvent() і SetEvent(). Будь-який спосіб сигналізації можна використати з різним типом подій.

Для створення події використовують функцію CreateEvent():

```
HANDLE CreateEvent(  // повертає дескриптор створеної події
  LPSECURITY_ATTRIBUTES sec_attrs, // може бути 0
  BOOL manual_reset,   // тип події
  // якщо TRUE - після створення відразу сигналізувати
  BOOL init_state,
  LPCTSTR evt_name );
  // рядок з ім'ям або нульовий покажчик
```

Коли значення `manual_reset` дорівнює TRUE, то маємо подію із ручним скиданням, а коли воно дорівнює FALSE – з автоматичним.

```
HANDLE event = CreateEventt(), TRUE, FALSE, 0); // ручне скидання
```

Для очікування сигналізації події використовують функцію очікування

```
WaitForSingleObjec(event, INFINITE);
```

Тепер розглянемо різні варіанти зміни стану подій. Насамперед зазначимо, що всі функції сигналізації події та скидання її стану мають однаковий синтаксис:

```
BOOL SetEvent(HANDLE event);       // два види
BOOL PulseEvent(HANDLE event);     // сигналізації подій
BOOL ResetEvent(HANDLE event);     // скидання події
```

Під час виконання функції `SetEvent()` відбувається сигналізація події. Подальші дії залежать від типу події.

Виконання функції `PulseEvent()` подібне до двох послідовних викликів `SetEvent()` і `ResetEvent()`. При цьому подію сигналізує і поновлює один із потоків, які очікують на події (для подій з автоматичним скиданням), або всі потоки, що очікують (для подій з ручним скиданням), після чого стан події скидають. Коли жодного потоку, що очікує немає, подію негайно скидають і факт її сигналізації зникає. Виклик функції `SetEvent()` зберігає стан події, а виклик функції `PulseEvent()` – ні.

Очевидно, що за допомогою функції `PulseEvent()` можна виконувати дії, аналогічні до сигналізації умовних змінних. Виконання цієї функції для події з автоматичним скиданням аналогічне до операції `signal`, а для події із ручним скиданням – до операції `broadcast`.

На жаль, основна проблема, пов'язана з реалізацією умовних змінних на основі подій, полягає не в сигналізації, а в організації очікування. Насправді коректно реалізувати першу частину wait (атомарне звільнення м'ютекса і перехід у стан очікування) на основі стандартних функцій очікування (`WaitForSingleObject()` або `WaitForMultipleObjects()`) неможливо. Якщо спочатку звільняти м'ютекс, а потім починати очікувати подій, отримаємо помилкове поновлення.

Деяким компромісом тут може бути використання функції `SignalObjectAndWait()`, що вперше з'явилася у Windows NT 4.0 і відтоді доступна у системах лінії Windows XP. Ця функція дає змогу атомарно сигналізувати один об'єкт і почати очікування зміни стану іншого.

```
DWORD SignalObjectAndWait()
HANDLE object_to_signal, //дескриптор сигналізованого об'єкта
HANDLE object_to_wait, //дескриптор об'єкта, на який ми очікуємо
DWORD timeout,         // максимальний інтервал очікування
BOOL alertable         // може бути FALSE
);
```

У нашій ситуації така функція може надати можливість просто реалізувати умовні змінні з однією операцією сигналізації – `signal` або `broadcast`, чого може бути

достатньо для багатьох застосувань. Коли ж потрібна умовна змінна, що одночасно має очікувати обидві операції відразу, можна скористатися складнішими вирішеннями.

```
HANDLE cond_signal_init() {
  // подія з автоматичним скиданням
  HANDLE signal_event = CreateEvent(0, FALSE, FALSE, 0);
  return signal_event;
}
HANDLE cond_broadcast_init() {
// подія з ручним скиданням
  HANDLE broadcast_event = CreateEvent(0, TRUE, FALSE, 0);
  return broadcast_event;
}
// виконання signal
void cond_signal_send (HANDLE signal_event) {
  PulseEvent(signal_event);
}
// виконання broadcast
void cond_broadcast_send (HANDLE broadcast_event) {
  PulseEvent(broadcast_event);
}
// wait для signal
void cond_signal_wait (HANDLE signal_event, HANDLE mutex) {
  // звільняємо м'ютекс і чекаємо signal
  SignalObjectAndWait(mutex, signal_event, INFINITE, FALSE);
  // після приходу події знову займаємо м'ютекс
  WaitForSingleObject(mutex, INFINITE);
}
// wait для broadcast
void cond_broadcast_wait (HANDLE broadcast_event, HANDLE mutex) {
  // звільняємо м'ютекс і чекаємо broadcast
  SignalObjectAndWait(mutex, broadcast_event, INFINITE, FALSE);
  // після приходу події знову займаємо м'ютекс
  WaitForSingleObject(mutex, INFINITE);
}
```

За наявності у Win32 API функції `SignalObjectAndWaitMultiple()` з очікуванням на зразок `WaitForMultipieObjects()` можна було б очікувати дві події відразу – `signal` і `broadcast`, що повністю розв'язувало б задачу реалізації умовної змінної.

```
  HANDLE events[] = { signal_event, broadcast_event };
  // такої функції немає, на жаль
  SignalObjectAndWaitMultiple(mutex, events, INFINITE, FALSE);
```

5.3. Міжпроцесова взаємодія

Реалізація міжпроцесової взаємодії здійснюється трьома основними методами: *передавання повідомлень, розподілюваної пам'яті* та *відображуваної пам'яті*.

В основі ***методів передавання повідомлень*** (message passing) лежать різні технології, що дають змогу потокам різних процесів (які, можливо, виконуються на різних комп'ютерах) обмінюватися інформацією у вигляді фрагментів даних фіксованої чи

змінної довжини, котрі називають *повідомленнями* (messages). Процеси можуть приймати і відсилати повідомлення, при цьому автоматично забезпечується їхнє пересилання між адресними просторами процесів одного комп'ютера або через мережу. Важливою особливістю технологій передавання повідомлень є те, що вони не спираються на спільно використовувані дані – процеси можуть обмінюватися повідомленнями, навіть не знаючи один про одного.

Методи ***розподілюваної пам'яті*** (shared memory) дають змогу двом процесам обмінюватися даними через загальний буфер пам'яті. Перед обміном даними кожний із тих процесів має приєднати цей буфер до свого адресного простору з використанням спеціальних системних викликів (перед цим перевіряють права). Жодних засобів синхронізації доступу до цих даних розподілювана пам'ять не забезпечує, програміст, так само, як і при розробці багатопотокових застосувань, має організовувати її сам.

Ще однією категорією засобів міжпроцесової взаємодії є ***відображувана пам'ять*** (mapped memory). Ця технологія зводиться до того, що за допомогою спеціального системного виклику (зазвичай це `mmap()`) певну частину адресного простору процесу однозначно пов'язують із вмістом файла. Після цього будь-яка операція записування в таку пам'ять спричиняє зміну вмісту відображеного файла, яка відразу стає доступною усім застосуванням, що мають доступ до цього файла. Інші застосування теж можуть відобразити той самий файл у свій адресний простір і обмінюватися через нього даними один з одним.

Основною характеристикою міжпроцесової взаємодії є те, що у процесів ***немає спільного адресного простору***, тому тут не можна безпосередньо працювати зі спільно використовуваними даними, як це було можливо для потоків.

Для вирішення проблеми міжпроцесової синхронізації необхідно:

- *по-перше*, організувати спільну пам'ять між процесами (це може бути розподілювана пам'ять або файл, відображений у пам'ять);
- *по-друге*, розмістити в цій пам'яті стандартні синхронізаційні об'єкти (семафори, м'ютекси, умовні змінні);
- *по-третє*, використовуючи ці об'єкти, працювати зі спільно використовуваними даними, як це робилося у разі використання потоків.

Такий підхід широко застосовують на практиці. На жаль, досить складно запропонувати спосіб його реалізації для міжпроцесової синхронізації у більшості систем, оскільки різні системи пропонують різний набір засобів організації спільної пам'яті та засобів сигналізації, які можуть працювати в такій пам'яті.

Основи передавання повідомлень

Як було вже згадано, засоби передавання повідомлень ґрунтуються на обміні повідомленнями – фрагментами даних змінної довжини. Основою такого обміну є не спільна пам'ять, а *канал зв'язку* (communication channel). Він забезпечує взаємодію між процесами (для того, щоб спілкуватися, вони повинні створити канал зв'язку) і є абстрактним відображенням мережі зв'язку.

Виокремлюють такі ***характеристики каналів*** зв'язку:

- спосіб задавання;
- кількість процесів, які можуть бути з'єднані одним каналом;
- кількість каналів, які можуть бути створені між двома процесами;
- пропускна здатність каналу (кількість повідомлень, які можуть одночасно перебувати в системі й бути асоційованими з цим каналом);
- максимальний розмір повідомлення;

- спрямованість зв'язку через канал (двобічний або однобічний зв'язок).

В однобічному зв'язку для конкретного процесу допускають передавання даних тільки в один бік.

Основна особливість передавання повідомлень полягає в тому, що процеси спільно використовують тільки канали. Немає необхідності забезпечувати взаємне виключення процесів під час доступу до спільно використовуваних даних, замість цього досить визначити *примітиви передавання повідомлень* – спеціальні операції обміну даними через канал, які забезпечують не лише обмін даними, але й синхронізацію.

Є два примітиви передавання повідомлень: `send` (для відсилання повідомлення каналом) і `receive` (для отримання повідомлення з каналу).

Зупинимося на основних питаннях синхронізації під час передавання повідомлень. Можна виокремити різні групи методів передавання повідомлень залежно від того, як вони дають можливість відповісти на два запитання.

1.Чи може потік бути призупинений під час виконання операції `send`, якщо повідомлення не було отримане?

2.Чи може потік бути призупинений під час виконання операції `receive`, якщо повідомлення не було відіслане?

У реальних системах відповідь на друге запитання практично завжди буде позитивною – неблокувальне приймання повідомлень спричиняється до того, що вони губляться. Варіанти відповідей на перше запитання визначають два класи передавання повідомлень – *синхронне* і *асинхронне.*

Під час синхронного передавання повідомлень операція `send` призупиняє процес до отримання повідомлення, а під час асинхронного передавання повідомлень вона не призупиняє процес (тобто є *неблокувальною);* після відсилання повідомлення процес продовжує своє виконання, не чекаючи отримання результату. Найзручніше в цьому випадку використати непряму адресацію через поштові скриньки.

Реалізація синхронного й асинхронного передавання повідомлень залежить від низки характеристик каналу й обміну повідомленнями, насамперед від пропускної здатності каналу.

Під час обміну повідомленнями необхідне підтвердження їх отримання. Деякі методи обміну повідомленнями не вимагають підтвердження зовсім, в інших випадках можлива ситуація, коли відправника після виконання операції `send` блокують доти, поки одержувач не надішле йому інше повідомлення із підтвердженням отримання; таку технологію називають *обміном повідомленнями із підтвердженням отримання.*

Найпростіший засіб передавання повідомлень – це *канал.* Він є циклічним буфером, записування у який виконують за допомогою одного процесу, а читання – за допомогою іншого. У конкретний момент часу до каналу має доступ тільки один процес. Операційна система забезпечує синхронізацію згідно з правилом: якщо процес намагається записувати в канал, у якому немає місця, або намагається зчитати більше даних, ніж поміщено в канал, він переходить у стан очікування.

Розрізняють безіменні та поіменовані канали.

До *безіменних каналів* немає доступу за допомогою засобів іменування, тому процес не може відкрити вже наявний безіменний канал без його дескриптора. Це означає, що такий процес має отримати дескриптор каналу від процесу, що його створив, а це можливо тільки для зв'язаних процесів.

До *поіменованих каналів* (named pipes) є доступ за іменем. Такому каналу може відповідати, наприклад, файл у файловій системі, при цьому будь-який процес, який має

доступ до цього файла, може обмінюватися даними через відповідний канал. Поіменовані канали реалізують непрямий обмін даними.

Обмін даними через канал може бути однобічним і двобічним.

Іншою технологією асинхронного непрямого обміну даними є застосування ***черг повідомлень*** (message queues). Для таких черг виділяють спеціальне місце в системній ділянці пам'яті ОС, доступне для застосувань користувача. Процеси можуть створювати нові черги, відсилати повідомлення в конкретну чергу й отримувати їх звідти. Із чергою одночасно може працювати кілька процесів. Для того щоб процеси могли розрізняти адресовані їм повідомлення, кожному з них присвоюють тип. Відіслане повідомлення залишається в черзі доти, поки не буде зчитане. Синхронізація під час роботи з чергами схожа на синхронізацію для каналів.

Найрозповсюдженішим методом обміну повідомленнями є використання ***сокетів*** (sockets). Ця технологія насамперед призначена для організації мережного обміну даними, але може бути використана й для взаємодії між процесами на одному комп'ютері (власне, мережну взаємодію можна розуміти як узагальнення IPC).

Сокет – це абстрактна кінцева точка з'єднання, через яку процес може відсилати або отримувати повідомлення. Обмін даними між двома процесами здійснюють через пару сокетів, по одному на кожен процес. Абстрактність сокету полягає в тому, що він приховує особливості реалізації передавання повідомлень – після того як сокет створений, робота з ним не залежить від технології передавання даних, тому один і той самий код можна без великих змін використовувати для роботи із різними протоколами зв'язку.

Особливості протоколу передавання даних і формування адреси сокету визначає *комунікаційний домен;* його потрібно зазначати під час створення кожного сокету. Прикладами доменів можуть бути *домен Інтернету* (який задає протокол зв'язку на базі TCP/IP) і *локальний домен* або *домен UNIX*, що реалізує зв'язок із використанням імені файла (подібно до поіменованого каналу). Сокет можна використовувати у поєднанні тільки з одним комунікаційним доменом. *Адреса сокету* залежить від домену (наприклад, для сокетів домену UNIX такою адресою буде ім'я файла).

Способи передавання даних через сокет визначаються його *типом*. У конкретному домені можуть підтримуватися або не підтримуватися різні типи сокетів.

Наприклад, і для домену Інтернет, і для домену UNIX підтримуються сокети таких типів:

- *потокові* (stream sockets) – задають надійний двобічний обмін даними суцільним потоком без виділення меж (операція читання даних повертає стільки даних, скільки запитано або скільки було на цей момент передано);
- *дейтаграмні* (datagram sockets) – задають ненадійний двобічний обмін повідомленнями із виділенням меж (операція читання даних повертає розмір того повідомлення, яке було відіслано).

Під час обміну даними із використанням сокетів зазвичай застосовується технологія клієнт-сервер, коли один процес (сервер) очікує з'єднання, а інший (клієнт) з'єднують із ним.

Перед тим як почати працювати з сокетами, будь-який процес (і клієнт, і сервер) має створити сокет за допомогою системного виклику socket(). Параметрами цього виклику задають комунікаційний домен і тип сокету. Цей виклик повертає дескриптор сокету – унікальне значення, за яким можна буде звертатися до цього сокету.

Подальші дії відрізняються для сервера і клієнта. Спочатку розглянемо послідовність кроків, яку потрібно виконати для сервера.

1. Сокет пов'язують з адресою за допомогою системного виклику bind(). Для сокетів

домену UNIX як адресу задають ім'я файла, для сокетів домену Інтернету – необхідні характеристики мережного з'єднання. Далі клієнт для встановлення з'єднання й обміну повідомленнями має буде вказати цю адресу.

2. Сервер дає змогу клієнтам встановлювати з'єднання, виконавши системний виклик `listen()` для дескриптора сокету, створеного раніше.
3. Після виходу із системного виклику `listen()` сервер готовий приймати від клієнтів запити на з'єднання. Ці запити вишиковуються в чергу. Для отримання запиту із цієї черги і створення з'єднання використовують системний виклик `accept()`. Внаслідок його виконання в застосування повертають новий сонет для обміну даними із клієнтом. Старий сокет можна використовувати далі для приймання нових запитів на з'єднання. Якщо під час виклику `accept()` запити на з'єднання в черзі відсутні, сервер переходить у стан очікування.

Для клієнта послідовність дій після створення сокету зовсім інша. Замість трьох кроків досить виконати один – встановити з'єднання із використанням системного виклику `connect()`. Параметрами цього виклику задають дескриптор створеного раніше сокету, а також адресу, подібну до вказаної на сервері для виклику `bind()`.

Після встановлення з'єднання (і на клієнті, і на сервері) з'явиться можливість передавати і приймати дані з використанням цього з'єднання. Для передавання даних застосовують системний виклик `send()`, а для приймання – `recv()`.

Зазначену послідовність кроків використовують для встановлення надійного з'єднання. Якщо все, що нам потрібно, – це відіслати і прийняти конкретне повідомлення фіксованої довжини, то з'єднання можна й не створювати зовсім. Для цього як відправник, так і одержувач повідомлення мають попередньо зв'язати сокети з адресами через виклик `bind()`. Потім можна скористатися викликами прямого передавання даних: `sendto()` – для відправника і `recvfrom()` – для одержувача. Параметрами цих викликів задають адреси одержувача і відправника, а також адреси буферів для даних.

Віддалений виклик процедур

Технологія *віддаленого виклику процедур* (Remote Procedure Call, RFC) є прикладом синхронного обміну повідомленнями із підтвердженням отримання.

Розглянемо послідовність кроків, необхідних для обміну даними в цьому разі.

1. Операцію `send` оформляють як виклик процедури із параметрами.
2. Після виклику такої процедури відправник переходить у стан очікування, а дані (ім'я процедури і параметри) доставляються одержувачеві. Одержувач може перебувати на тому самому комп'ютері, чи на віддаленій машині; технологія RPC приховує це. Класичний віддалений виклик процедур передбачає, що процес-одержувач створено внаслідок запиту.
3. Одержувач виконує операцію `receive` і на підставі даних, що надійшли, виконує відповідні дії (викликає локальну процедуру за іменем, передає їй параметри і обчислює результат).
4. Обчислений результат повертають відправникові як окреме повідомлення.
5. Після отримання цього повідомлення відправник продовжує своє виконання, розглядаючи обчислений результат як наслідок виклику процедури.