МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«Київський політехнічний інститут»

Факультет прикладної математики

Кафедра спеціалізованих комп'ютерних систем

КОНСПЕКТ ЛЕКЦІЙ
з дисципліни

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Лектор: доцент Марченко О.І.

Київ, 2009–2014

Процеси, потоки, задачі

Поняття процесу з'явилось при розробці перших багатозадачних операційних систем (ОС) в 60-ті роки XX століття. В операційній системі процес пов'язується з кожним видом робіт, які виконуються в комп'ютерній системі під керуванням цієї ОС. Наприклад, до таких видів робіт можна віднести виконання кожної прикладної задачі, різні системні дії такі, як вивід інформації на монітор, на друк, введення інформації з клавіатури, від миші, забезпечення роботи дисків, флеш-пам'яті, тощо.

Процес (process) – абстрактне поняття, що включає опис певних дій, заданих кодом системної або прикладної програми, та ресурсів цієї програми (адресний простір оперативної пам'яті, глобальні дані, відкриті файли, сигнали, що очікують на обробку, потоки цього процесу), пов'язаних з виконанням цієї програми в комп'ютерній системі.

Процеси працюють паралельно, тобто дозволяють виконувати певні дії одночасно. Сама по собі програма не є процесом; процес – це програма, що виконується в багатозадачній операційній системі, плюс набір відповідних їй ресурсів. Одну програму можуть виконувати один або більше процесів та/або потоків.

Потік (thread) – це частини коду процесу, які працюють паралельно (одночасно) і використовують деякі спільні ресурси цього процесу. Концептуально, потоки існують в рамках процесу і є більш дрібними одиницями керування програмою. У російськомовній літературі термін «thread» перекладається також як «нить» (рос.).

Під час роботи програми керування постійно передається від одного процесу/потоку до іншого процесу/потоку. Причому передача керування від процесу до процесу в більшості операційних систем є дією більш складною і вимагає більше часу, ніж передача керування від потоку до потоку. Тому **процеси іноді називають важкими процесами (heavyweight process), а потоки – легкими процесами (lightweight process)**, і використовуються важкі процеси для вирішення задач, які не потребують або мало потребують обміну даними з іншими задачами, а легкі процеси – для підзадач, які є частинами однієї задачі і потребують частого обміну даними. Звідси походить ще одна назва важких та легких процесів: **важкі процеси називають також слабко зв'язаними процесами (loosely coupled processes), а легкі процеси – тісно зв'язаними процесами (tightly coupled processes)**.

В більшості багатозадачних операційних систем (зокрема Windows, Solaris, деякі версії Unix) потоки реалізовані саме згідно вказаної концепції, тобто потоки існують тільки в рамках якогось процесу. В ОС Linux реалізація потоків відрізняється від загальноприйнятої концепції. В цій ОС потоки (легкі процеси) фактично є такими ж процесами як і важкі процеси. Відміною потоків від процесів в ОС Linux є використання потоками спільних ресурсів. Так зроблено тому, що в ОС Linux процеси (важкі процеси) перемикаються так само швидко, як в інших ОС перемикаються потоки (легкі процеси) (див. рис.1).

У сучасних багатозадачних ОС передбачається наявність у процесів і потоків двох основних віртуальних ресурсів: віртуального процесора та віртуальної пам'яті. Процесу призначається один або декілька віртуальних процесорів і віртуальна пам'ять комп'ютерної системи. Кожен потік однієї програми (або процесу) отримує окремий віртуальний процесор, але віртуальну пам'ять потоки однієї програми використовують одну й ту саму.

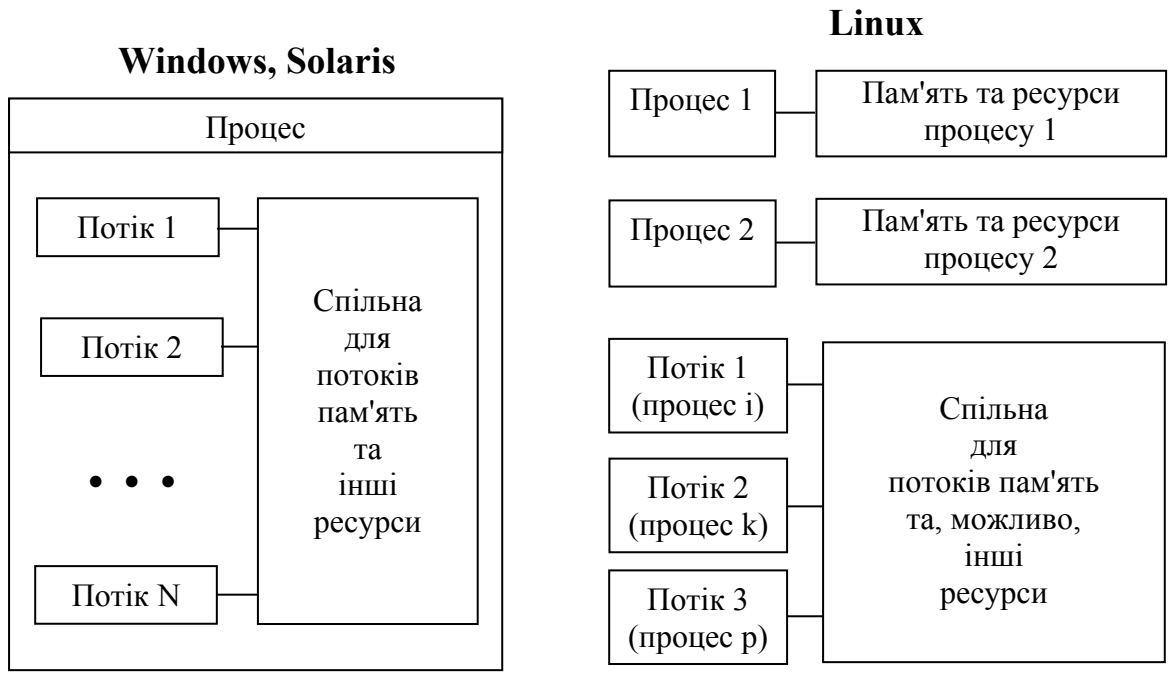


Рис.1. Реалізація потоків в операційних системах.

Історично, спочатку виникло поняття процесу, яке було реалізоване в перших багатозадачних операційних системах. Пізніше засоби роботи з паралельними процесами були реалізовані як спеціальні конструкції деяких мов програмування високого рівня (МВР), таких як Modula-2, Ada, Java, C#, паралельні версії мови Fortran, та у вигляді спеціалізованих бібліотек процедур та функцій, таких, як PVM, MPI, Win32. Процедури/функції цих бібліотек можна викликати як із системних програм ОС, так й з програм, написаних на МВР. Поняття потоку було введено значно пізніше з розвитком теорії операційних систем та мов програмування високого рівня (МВР).

Таким чином, засоби роботи з паралельними процесами та потоками можна поділити на:

1) вбудовані в ОС;
2) мовні конструкції МВР;
3) бібліотечні.

Термін «**задача**» («**task**») раніше використовувся як синонім терміна «**процес**», оскільки виконання одного процесу пов’язувалося з рішенням якоїсь системної або прикладної задачі, яку можна було виконувати паралельно з іншими. Звідси фактично походить і термін «багато**задачна** операційна система». Пізніше, після появи поняття потоку, термін «**задача**» став, в основному (але не обов’язково), використовуватись як синонім терміна «**потік**», як більш дрібної одиниці роботи комп’ютерної системи, для якої треба планувати паралельну роботу. Відповідно до цього виник ще один термін «багато**поточна** програма», тобто програма, виконання якої забезпечується двома або більше потоками.

Дескриптор процесу/потоку

Дескриптор процесу/потоку – це спеціальна структура даних, що вміщує всю інформацію про цей процес/потік, яка потрібна операційній системі для планування та керування всіма процесами і потоками, що виконуються в комп’ютерній системі.

Дескрипторам процесу/потоку в різних ОС, МВР та бібліотеках відповідають різні структури даних, іноді досить складні і об’ємні, наприклад, як в Linux, але всі вони включають, як правило, таку основну інформацію:

1) ідентифікатор процесу/потоку (PID – Process IDentifier);
2) ділянка оперативної пам’яті для стеку процесу/потоку;
3) пріоритет процесу/потоку;
4) поточний стан процесу/потоку;
5) інші ресурси.

При використанні паралельних конструкцій МВР та бібліотек для програмування прикладних паралельних програм, дескриптор має більш просту структуру, і пряме звертання до його полів, як правило, не використовується; значення полів дескриптора в цих випадках змінюються при виконанні відповідних процедур та функцій. При розробці системних програм з використанням паралельних засобів, вбудованих в операційну систему, наприклад програм в просторі ядра ОС Linux, робота напряму з дескрипторами процесів/потоків є звичайним явищем.

Стани процесів/потоків

Кожен процес/потік може знаходитись в одному з допустимих для нього станів. Кількість станів, їх назви та точне призначення відрізняються для різних реалізацій паралельних процесів, але стани, що відповідають нижчепереліченим характеристикам є практично в усіх реалізаціях (див. рис.2.):

1) ініціалізований (породжений, але ще не готовий до виконання) стан процесу/потоку;
2) готовий до виконання стан процесу/потоку;
3) активний стан (стан виконання) процесу/потоку
4) призупинений (затриманий) на певний час стан процесу/потоку;
5) призупинений в очікуванні на певний сигнал або певну подію стан процесу/потоку;
6) стан «зомбі» процесу/потоку (завершений, але не видалений з пам’яті процес/потік);
7) завершений або зупинений стан процесу/потоку (повністю завершений і видалений з пам’яті процес/потік).

В деяких реалізаціях паралельних процесів певні стани можуть бути відсутні (наприклад, ініціалізований стан або стан «зомбі») або об’єднуватись в один стан (наприклад, затриманий на певний час стан і стан очікування на певний сигнал).

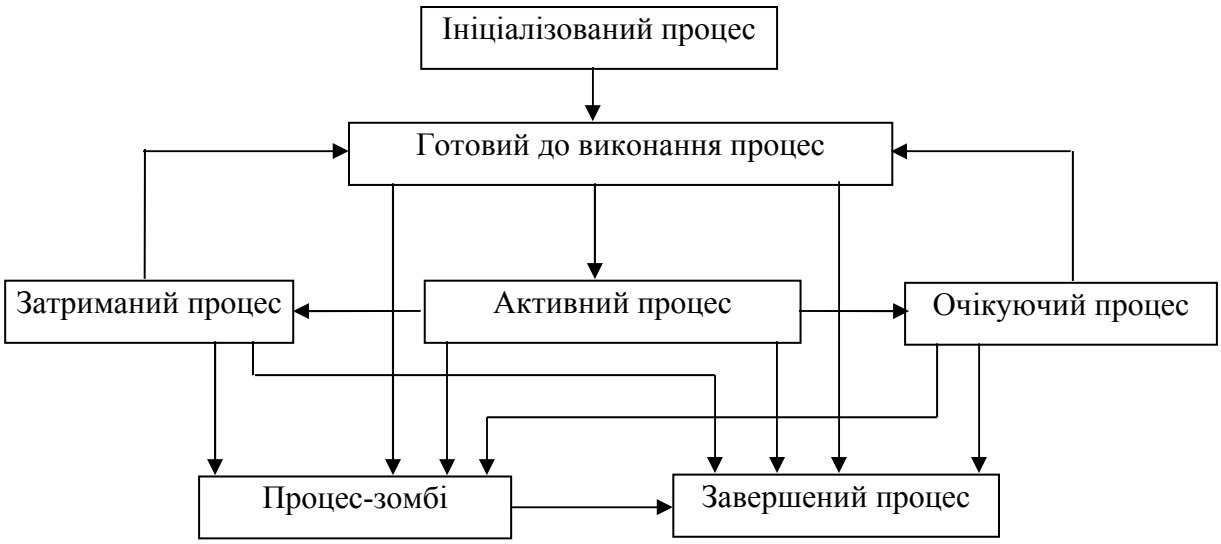


Рис.2. Стани процесів/потоків

Процеси-зомбі виникають, як правило, при алгоритмічних помилках при програмуванні паралельних процесів, і, в якійсь мірі, стан «зомбі» можна вважати різновидом стану очікування, бо в стані «зомбі» процес очікує на сигнал для видалення з пам’яті, який ніколи не з’явиться.

Ще однією характерною помилковою ситуацією (іноді її також називають станом) є ситуація, коли процес/потік А очікує на сигнал S1, а процес/потік В – на сигнал S2, які посилаються відповідно в процесах/потоках В та А, але після того як обидва процеси/потоки зупинились в очікуванні сигналів S1 та S2 (див. рис.3.). Інакше кажучи, два процеси/потоки взаємно очікують на дві події, які повинні відбутися в іншому процесі/потоці, але насправді

ніколи не відбудуться. Така ситуація називається **тупиком**, або **взаємоблокуванням**, або **дедлоком (deadlock)**. Виникнення тупиків є однією з найбільших загроз при програмуванні паралельних процесів, яку треба передбачати і не допускати.

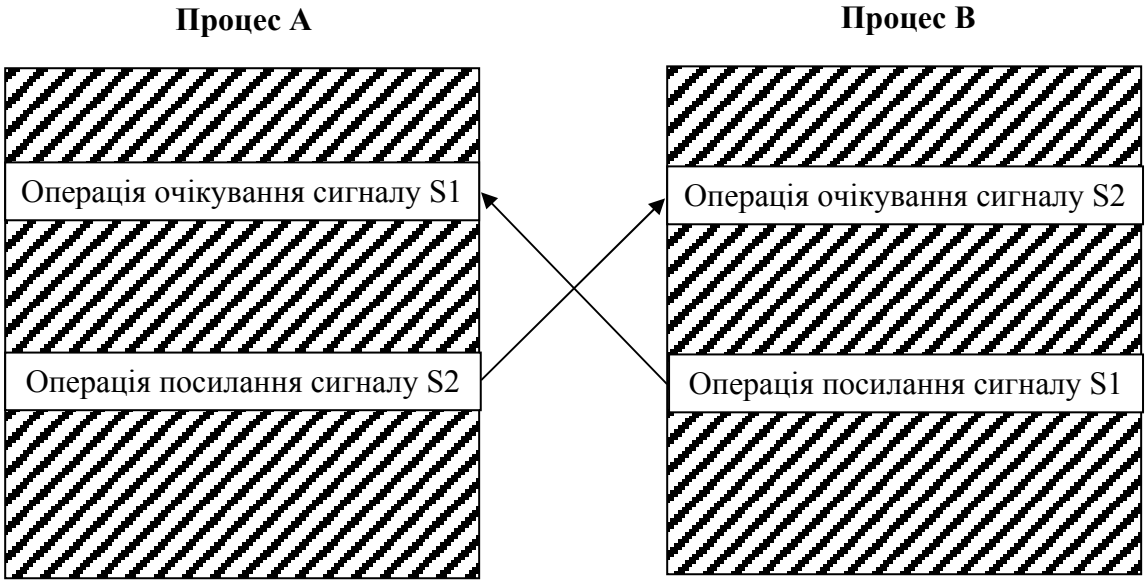


Рис.3. Ситуація тупика або взаємоблокування (deadlock)

Принципи організації паралельних процесів/потоків та основні операції керування ними

Серед різних реалізацій можна виділити два принципи організації взаємозалежності паралельних процесів/потоків:

1) плаский чи площинний (flat) принцип, коли всі процеси після створення є рівноправними і не залежать один від одного; цей принцип, зокрема, використовується в мові Modula-2;
2) ієрархічний принцип, коли створений процес залежить від свого предка, тобто процесу, в якому він був створений, і, в результаті, утворюється ієрархія взаємозалежності процесів; цей принцип, зокрема, використовується в операційній системі Linux та мові Ada.

Але незалежно від використаного принципу організації взаємозалежності між процесами/потоками та кількістю і призначенням їх станів, кожна реалізація паралельних процесів/потоків повинна забезпечити рішення наступних основних підзадач керування процесами/потоками:

1) оголошення процесів/потоків та пов’язаних з ними даних;
2) ініціалізацію та запуск процесів/потоків;
3) операції для забезпечення комунікації (обміну даними) процесів/потоків;
4) операції для забезпечення синхронізації процесів/потоків;
5) завершення процесів/потоків.

Рішення цих підзадач забезпечуються спеціальними конструкціями мов програмування та/або спеціальними процедурами і функціями.

Комунікація та синхронізація процесів/потоків

Комунікація та синхронізація процесів/потоків є основними підзадачами, які вирішує програміст при організації взаємодії паралельних процесів/потоків.

При рішенні підзадачі комунікації вирішується питання передачі інформаційних та керуючих (сигналів, значень прапорців, тощо) даних від одного довільного процесу/потоку до іншого довільного процесу/потоку.

При рішенні підзадачі синхронізації вирішуються наступні питання:

1) виключення одночасного звертання двох або більше процесів/потоків до одного і того ж ресурсу, як правило, спільних (глобальних) даних;
2) своєчасний запуск і своєчасне завершення кожного процесу/потоку по відношенню до інших;
3) своєчасне призупинення (блокування) і відновлення роботи кожного процесу/потоку по відношенню до інших;
4) забезпечення одночасної «зустрічі» процесів/потоків в потрібних точках їх коду (як правило, для обміну даними).

В залежності від архітектури комп'ютерної системи підзадачі комунікації та синхронізації процесів/потоків можуть вирішуватись різними засобами. Для комп'ютерних систем зі спільною пам'яттю застосовується, як правило, модель комунікації та синхронізації, яка базується на використанні спільних (глобальних) змінних – **модель спільних змінних**, а для систем з розподіленою пам'яттю – **модель передачі повідомлень**. Щоправда слід зазначити, що модель передачі повідомлень успішно використовується і для систем зі спільною пам'яттю.

Модель спільних даних

При використанні моделі спільних даних комунікація (передача даних) і синхронізація процесів/потоків відбувається за допомогою спільних (глобальних) даних.

Якщо в програмі процесів/потоків небагато і вони не потребують активної комунікації, а також, згідно алгоритму, ці процеси/потоки гарантовано не можуть звертатись до одних і тих же даних, то проблем із синхронізацією процесів/потоків у такій програмі практично не виникає. Але така програма є виключенням, а не правилом, і тому підзадача синхронізації процесів/потоків є ключовою при написанні паралельних програм.

Наведений в попередньому параграфі перелік питань підзадачі синхронізації можна звести до двох головних проблем:

1) проблеми одночасного доступу (звертання) до спільних даних;
2) проблеми несинхронної роботи процесів/потоків без обміну інформацією (наприклад, процеси/потоки повинні вивести дані на екран або на друк в строго визначеному порядку).

Одночасний доступ (звертання) до одних і тих же даних під час комунікації одного або більше процесів/потоків призводить до некоректної поведінки процесів/потоків та/або отримання некоректних результатів. Тому вирішення цієї проблеми є першим ключовим завданням синхронізації паралельних процесів/потоків, яке називається **завданням взаємного виключення**.

Для вирішення завдання взаємного виключення можна запропонувати два принципово різних підходи:

1) підхід, що ґрунтується на ***контролі процесів/потоків***;
2) підхід, що ґрунтується на ***контролі спільного ресурсу***.

Фрагменти коду процесів/потоків, в яких вони звертаються до спільного ресурсу і, в результаті, можуть конфліктувати, називаються **критичними секціями (critical section),** або **критичними областями/зонами** (**critical region**), або **критичними ділянками** (**critical area**). Надалі будемо використовувати термін «критична секція».

Зауваження. Термін «критична секція (область, зона, ділянка)» є двозначним. Цим терміном називається як фрагмент коду процесу/потоку, так і один із засобів (механізмів) взаємодії процесів/потоків (див. розділ «Критичні секції»), призначений для обмеження таких фрагментів.

Підхід, що ґрунтується на контролі процесів/потоків, полягає в розташуванні точок синхронізації перед критичними секціями та після них і використанні в цих точках таких команд синхронізації, щоб не допустити одночасного виконання коду критичних секцій в різних процесах/потоках (див. рис.4.).

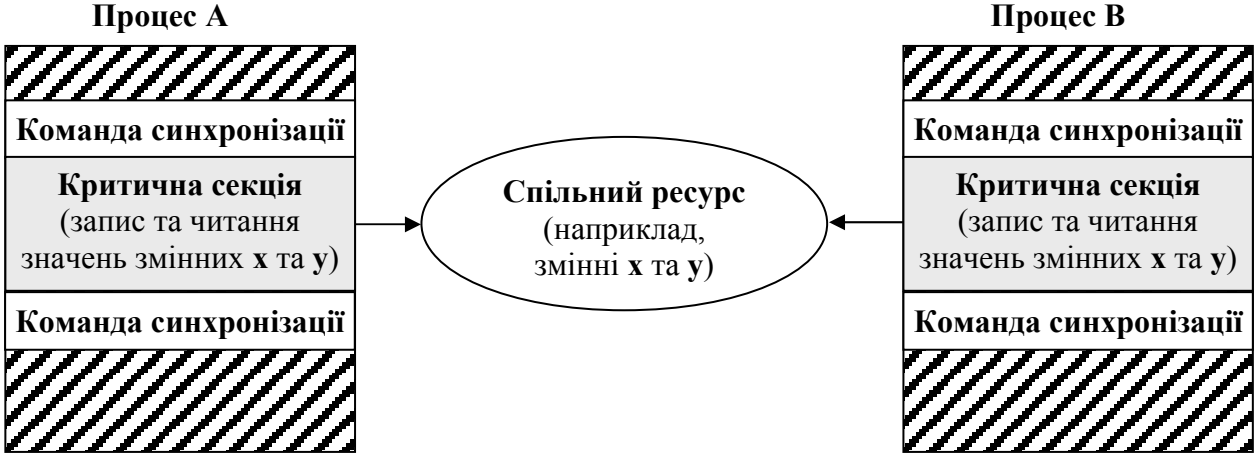


Рис.4. Рішення завдання взаємного виключення, що ґрунтується на ***контролі процесів/потоків***

Підхід, що ґрунтується на контролі спільного ресурсу, полягає в використанні спеціальних конструкцій мов програмування високого рівня, які «захищають» цей спільний ресурс, не допускаючи одночасного доступу до нього (див. рис.5.).

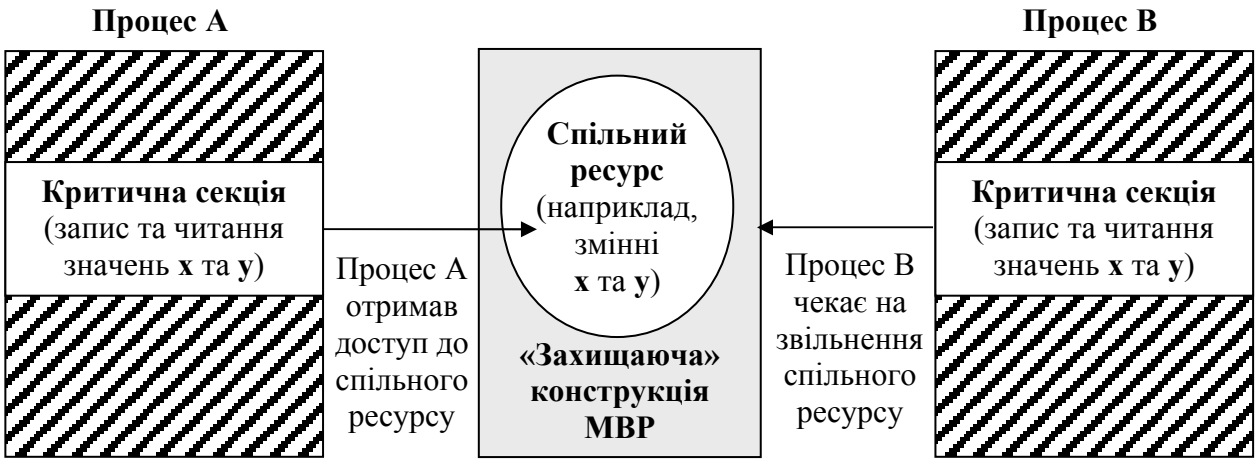


Рис.5. Рішення завдання взаємного виключення, що ґрунтується на ***контролі спільного ресурсу***

Проблема несинхронної роботи без обміну інформації процесів/потоків полягає в тому, що виконання дій та/або доступу до даних відбувається в цих процесах/потоках не в тому порядку, якому потрібно. Вирішення цієї проблеми є другим ключовим завданням синхронізації паралельних процесів/потоків. Оскільки це завдання вирішується, як правило за допомогою механізму «очікування події – посилання сигналу про подію», то його називають **завданням синхронізації за подіями**.

Механізм «очікування події – посилання сигналу про подію» полягає в тому, що один процес/потік А призупиняється, виконавши команду Wait(S1) для очікування сигналу (події) S1, а другий процес/потік В виконує команду Send(S1), тобто посилає сигнал (сигналізує про настання події). Якщо процес В виконає команду Send(S1) раніше, ніж процес А команду Wait(S1), то процес А не зупиняється, а продовжує свою роботу, бо сигнал вже був посланий (подія вже відбулася) (див. рис.6.). Точки процесів А і В, в яких вони виконують відповідно команди Wait(S1) і Send(S1) називаються **точками синхронізації**.

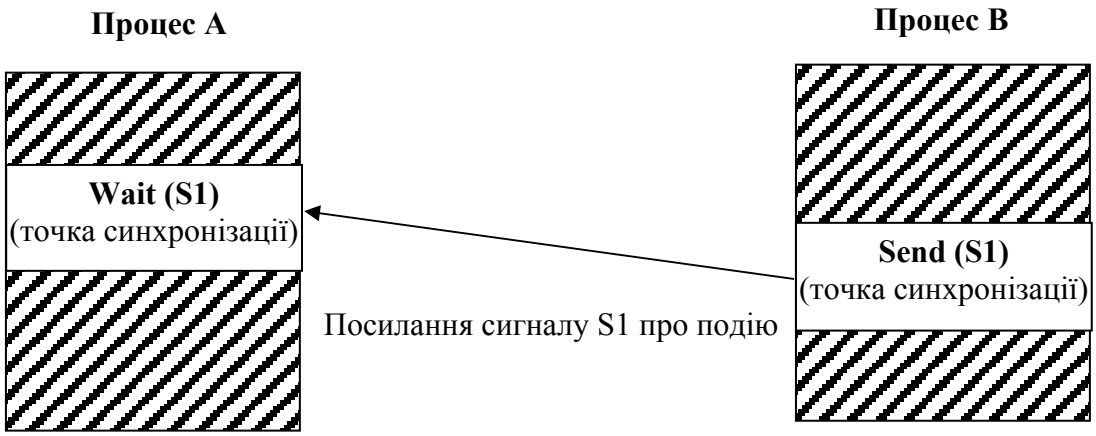


Рис.6. Механізм «очікування події – посилання сигналу про подію»

Для рішення загальної задачі синхронізації процесів/потоків були запропоновані різні засоби, такі як семафори, м’ютекси, сигнальні змінні, критичні секції, монітори та інші.

Засоби взаємного виключення процесів/потоків, що ґрунтуються на контролі спільного ресурсу

Засоби, що розглядаються в цьому розділі (монітори і атомарні дані та операції), призначені для рішення тільки однієї з підзадач синхронізації процесів/потоків (але ключової!), а саме для контролю за коректним використанням спільних даних. Історично ці засоби були запропоновані для рішення даної підзадачі не першими, а є результатом узагальнення характерних помилок, що часто виникали при використанні запропонованих раніше універсальних засобів синхронізації, таких, як семафори, критичні секції та сигнальні змінні (розглядаються в наступному розділі).

Монітори

Ідея монітора була запропонована вченими Хансеном (Hansen P. Brinch) та Хоаром (Hoare C.A.R.) у 1973-1974 роках [Hansen P. Brinch. Operating System Principles, Prentice Hall, N.Y., 1973; Hoare C.A.R. Monitors: An Operating System Structuring Concept, Communication of ACM, Vol.17, #10, Oct.1974, pp.549-557.] і реалізована в нових мовах програмування Concurrent Pascal (Б.Хансен) та Modula (Н.Вірт) у 1975 році.

Образно кажучи, **монітор –** *це пристосоване до вирішення задачі взаємного виключення процесів/потоків об'єднання двох мовних конструкцій, запропонованих теоретиками раніше, – програмного модуля і класу*.

Узагальнена структура монітора має такий вигляд:

monitor Ідентифікатор_монітору;

Опис спільних для процесів даних, які потрібно захищати монітором
(вони доступні тільки через виклик процедур монітора)
Опис процедур доступу до спільних даних, які потрібно захищати монітором

begin

Оператори ініціалізації спільних для процесів даних (якщо потрібні)

end Ідентифікатор_монітору;

Від модуля взята структура конструкції монітора і принцип приховування інформації, а від класу – принцип інкапсуляції (об'єднання даних і процедур їх обробки в єдиній конструкції), і до цього доданий новий власний **принцип монітора*, який полягає в тому, що процедури монітора можуть виконуватись тільки зі взаємним виключенням***, тобто якщо процес/потік А викликав одну з процедур монітора Р1, то жоден інший процес/потік не може викликати жодну іншу процедуру монітора доки не закінчиться робота процедури Р1 для процесу/потоку А.

Якщо на виконання процедур монітора претендують відразу декілька інших процесів/потоків, то з них формується окрема черга доступу до монітора.

Прямий доступ до структур даних монітора забороняється.

Така заборона в конкретних МВР реалізується або директивами доступу (private, public), або просто семантикою реалізації самої конструкції монітора, тобто дозволяється тільки виклик процедур та функцій.

У конкретних реалізаціях монітор може називатися як *монітором* (Concurrent Pascal, Modula, C#), так може мати й інші назви, наприклад, *«захищений модуль – protected unit»* (Ada95) або *«клас із синхронізованими методами – class with synchronized methods»* (Java).

Наведемо приклад монітора.

```
monitor CommonBuffer;

    const n = 10;
    type TBuffer = array [1..n] of integer;
    var Buffer : TBuffer;
           i : integer;

    function ReadBuffer (i : integer) : integer;
    begin
         Result := Buffer[i];
    end;

    procedure WriteBuffer (i : integer; Value : integer);
    begin
         Buffer[i] := Value;
    end;

begin
    for i := 1 to n do Buffer[i] := 0;
end CommonBuffer;
```

Конструкція монітора, як спеціалізована конструкція для рішення задачі безпечного взаємного виключення, є більш простою і надійною у використанні, ніж пара конструкцій «м'ютекс + сигнальна змінна» (див. відповідні підрозділи), які дозволяють вирішити цю задачу так само безпечно, але більш складним і менш надійним чином. Фактично, використання пари «м'ютекс + сигнальна змінна» для рішення задачі безпечного взаємного виключення є детальною моделлю внутрішньої роботи (що забезпечується реалізацією) монітора.

Атомарні структури даних та атомарні операції

Атомарні (їх ще називають неподільними) **структури даних і атомарні операції** – це, свого роду, міні-монітори, опис яких «розпорошений» по розділах загальних описів.

Спосіб реалізації атомарних даних та операцій може бути різним (прагми, спеціальні атомарні типи даних або атрибути/директиви для опису таких даних, тощо), але їх суть залишається однаковою, а саме: **атомарні операції над атомарними даними виконуються за принципом монітора (взаємного виключення)**, тобто якщо процес/потік А виконує одну атомарну операцію Р1 над структурою даних D1, то жоден інший процес/потік не може виконати жодну іншу атомарну операцію над тією ж структурою даних D1 доки не закінчиться виконання операції Р1 для процесу/потоку А. Не атомарні операції над атомарними даними, як правило, не допускаються.

Відмінність атомарних даних та операцій від монітора полягає в двох особливостях:

1) атомарні дані та операції реалізовані тільки для відносно простих окремих структур даних і тільки для простих операцій таких, як читання значення змінної, присвоєння, арифметичні операції. Монітор же може «захищати» одночасно будь-яку кількість структур даних і виконувати над ними складні «атомарні» алгоритми, які описуються процедурами та функціями монітора;
2) із процесів/потоків, що конкурують за доступ до монітора, формується окрема черга, а виконання атомарних операцій над атомарними даними регулюється операційною системою і порядок їх виконання конкуруючими процесами/потоками, як правило, є непередбаченим.

Універсальні засоби синхронізації процесів/потоків

Семафори

Історично, семафори є першим засобом синхронізації паралельних процесів/потоків. Концепція семафора була запропонована видатним голландським теоретиком програмування **Едсгером Вайбом Дейкстрою (Edsger Wybe Dijkstra)** у 60-х роках XX-го сторіччя при розробці ним першої багатозадачної операційної системи, побудованої у вигляді множини паралельно виконуваних взаємодіючих процесів. Саме під час цієї роботи з'явились поняття синхронізації процесів та ідея семафору.

Семафор – це змінна (лічильник) S спеціального типу Semaphore, над якою допустимі дві атомарні операції P(S) і V(S). Назви цих операцій походять від голландських слів ***Proben (перевірити/спробувати)*** та ***Verhogen (виконати інкремент, тобто збільшення на одиницю).***

Пізніше операції ***P(S)*** і ***V(S)*** почали називати відповідно ***Down(S)*** або ***Lock(S)*** або ***Acquire(S) – закрити/заблокувати/захопити семафор S,*** і ***Up(S)*** або ***Unlock(S)*** або ***Release(S) – відкрити/розблокувати/звільнити семафор S,*** а для **сигнальних змінних** їх стали називати відповідно ***Wait(S) – чекати на сигнал S і Send(S) – послати сигнал S***. *Сигнальні змінні є, фактично, спеціальним видом семафорів* (див. розділ «Сигнальні (умовні) змінні»).

Семафори бувають двійкові (бінарні) та багатозначні. Змінні (лічильники) двійкових семафорів можуть приймати тільки два значення 0 і 1 (або false і true), а множинні – значення від 0 до заданого N.

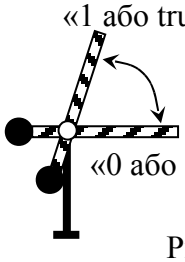
У різних ОС, МВР та бібліотеках реалізація семафорів має свої особливості. Ці особливості описуються в подальших розділах, а зараз *розглянемо класичну реалізацію семафорів*, як її визначив Дейкстра.

Двійкові (бінарні) семафори

Термін **«семафор»** походить від аналогії із залізничним семафором, який може бути закритим/заблокованим і відкритим/розблокованим.

Закритому/заблокованому стану семафора відповідає значення 0 (false), а відкритому/розблокованому стану семафора – значення 1 (true).

«1 або true» - семафор відкритий/розблокований



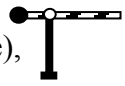
«0 або false» - семафор закритий/заблокований

Рис.7. Аналогія двійкового семафору із залізничним семафором

Розглянемо виконання класичних операцій P(S) та V(S) для двійкового семафору.

Операція P(S): **Перевірити значення змінної S (спробувати пройти за семафор)**

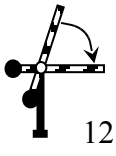
якщо семафор S закритий/заблокований (S=0 або S=false),



то призупинити процес/потік («поїзд») і поставити його у чергу очікування семафору S;
інакше

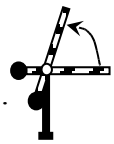
дозволити входження процесу/потоку («поїзду») в критичну ділянку, бо семафор відкритий/розблокований (S=1 або S=true),

і після цього закрити/заблокувати семафор для інших



процесів/ потоків («поїздів»), тобто виконати дію S:=0 або S:=false.

Операція V(S):
Відкрити/розблокувати семафор S, тобто виконати дію S:=1 або S:=true.



Як зазначено в заголовку розділу, семафори є універсальним засобом синхронізації процесів/потоків і можуть використовуватись як для рішення підзадачі взаємного виключення (на основі контролю процесів/потоків), так і для рішення загальних проблем синхронізації. На рисунках 8 – 10 показані схеми деяких задач за допомогою двійкових семафорів.

На цих рисунках разом з назвами операцій P(S) та V(S) наведені назви Wait(S) та Send(S). Це зроблено для того, щоб показати подібність двійкових семафорів та сигнальних змінних.

Рисунок 8 демонструє застосування двійкового семафору для рішення підзадачі взаємного виключення, що ґрунтується на контролі процесів/потоків.

Критичні секції коду процесів A та B, в яких виконується звертання до спільного ресурсу, обрамлюються операціями P(S) і V(S), а семафор S в початковому стані є відкритим/розблокованим, тобто має значення 1.

Першим до семафору S (до виконання команди P(S)) підходить процес/потік A і, оскільки семафор відкритий, то процес/потік A входить в свою критичну секцію, захоплюючи спільний ресурс, а семафор S змінює своє значення на 0, сигналізуючи, що спільний ресурс вже зайнятий. Процес B, який підійшов до семафору S другим, вимушений чекати доки семафор S відкриється. Після того, як процес/потік A виконає свою критичну секцію, виконується операція V(S), яка змінює стан семафора S на відкритий/розблокований, записуючи до нього значення 1 і сигналізуючи, що спільний ресурс вже вільний. В результаті наступний процес/потік B входить в свою критичну секцію, захоплюючи спільний ресурс, а семафор S знову змінює своє значення на 0, і так далі.

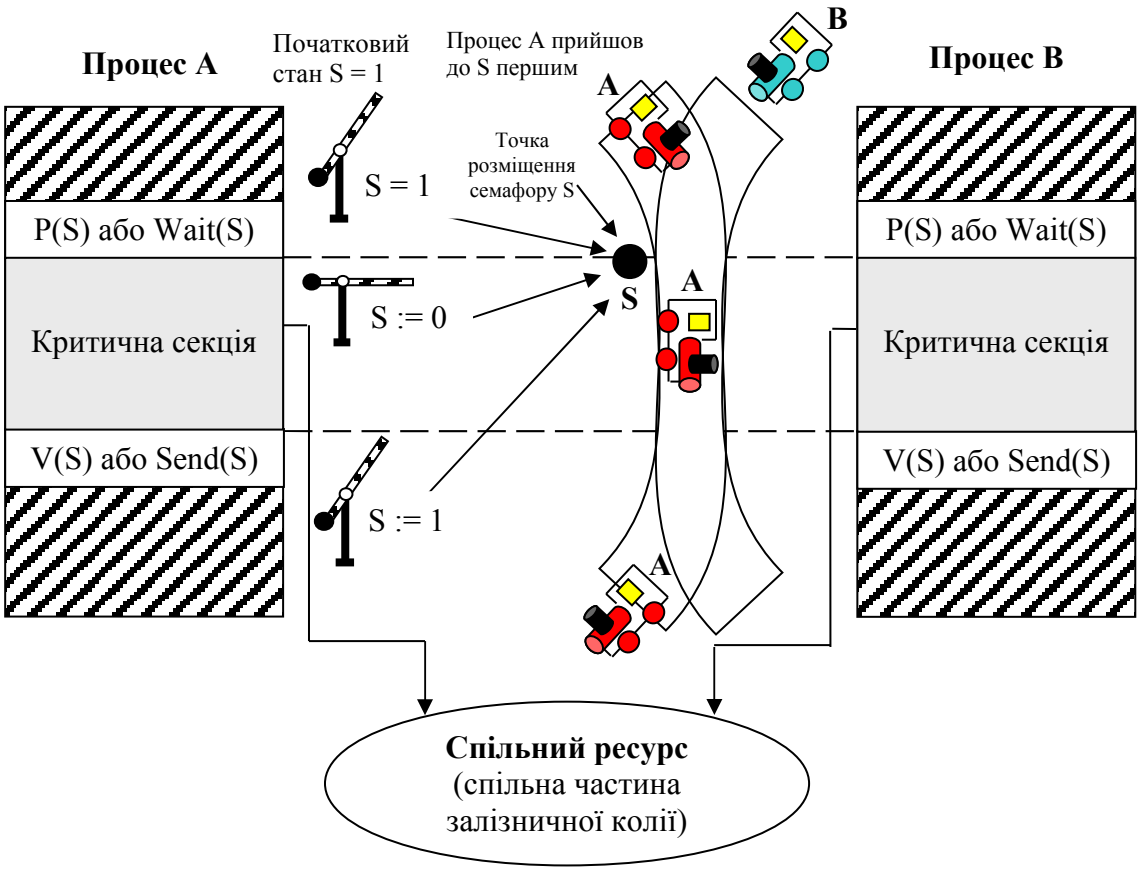


Рис.8. Застосування двійкового семафору для рішення завдання взаємного виключення

Рисунок 9 демонструє ситуацію, коли треба два процеси/потоки A і B синхронізувати так, щоб частина коду процесу/потоку A **після** точки очікування (вертикальна штриховка) завжди виконувалась **не раніше** (пізніше можна), ніж процес/потік B виконає частину свого

коду **до** точки події (горизонтальна штриховка). В точці очікування ставиться операція P(S), а точці сигналу про подію ставиться операція V(S). В початковому стані семафор S закритий/заблокований (S=0).

На цьому рисунку показаний випадок, коли процес А приходить до своєї точки синхронізації першим і, виконавши команду P(S), зупиняється (оскільки семафор закритий) в очікуванні на подію, яка повинна відбутися в процесі/потоці В. Коли процес В приходить до своєї точки синхронізації, він виконує команду V(S), змінюючи значення семафору S на 1 і відкриваючи його для процесу А. В результаті процеси/потоки А і В виконуються далі одночасно (умова «не раніше» виконана). Якщо ж першим до своєї точки синхронізації прийде процес/потік В (цей випадок на рисунку не показаний), то він відразу відкриє семафор (S:=1) і продовжить свою роботу, бо він не повинен чекати на процес/потік А. Коли ж процес/потік А підійде до своєї точки очікування, він зможе рухатись далі не зупиняючись, оскільки семафор вже відкритий. В результаті процес/потік А почне виконання частини свого коду після точки синхронізації пізніше, ніж процес/потік В закінчить частину свого коду до точки синхронізації (умова «не раніше» знову виконана).

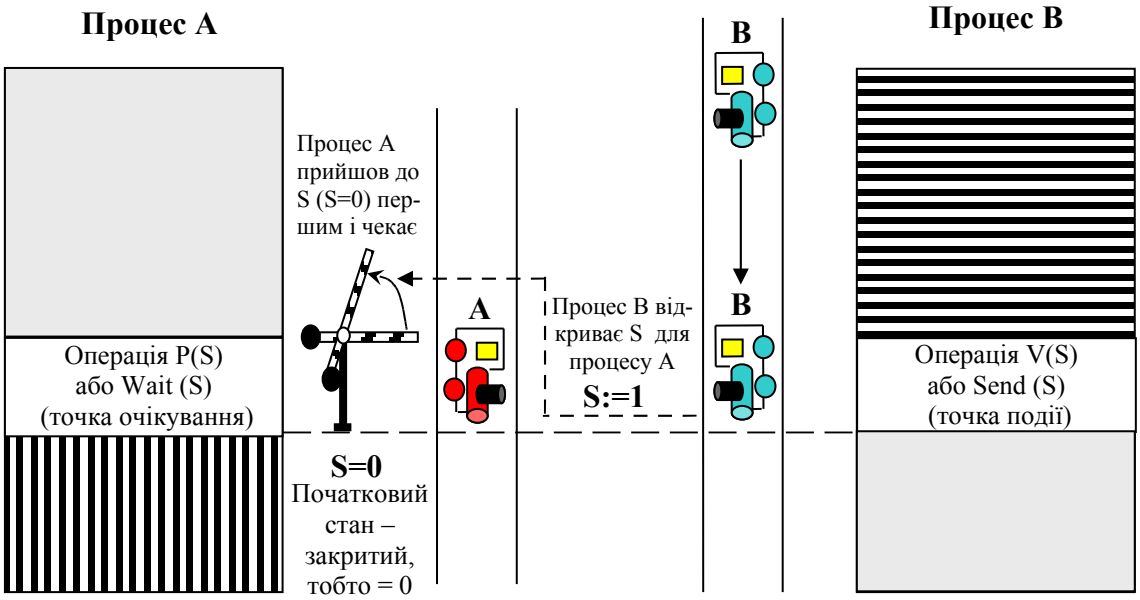


Рис.9. Застосування двійкового семафору для рішення простої задачі синхронізації

Рисунок 10 демонструє ситуацію повної синхронізації двох процесів/потоків А і В у визначених точках, тобто обидва процеси **обов'язково** повинні зустрітися у точках синхронізації, який би з них не прийшов до своєї точки синхронізації першим, і продовжити своє виконання далі завжди одночасно. Для рішення такої задачі потрібно використати два двійкових семафори: перший семафор S1 слугує для зупинки в точці синхронізації процесу А, а другий семафор S2 – для зупинки в точці синхронізації процесу В. Початковий стан обох семафорів S1=0 і S2=0, тобто обидва семафори закриті.

Обидва процеси А і В в своїх точках синхронізації виконують дві команди: спочатку виконується відкриття семафора **іншого** процесу командою V(S2) в процесі А і командою V(S1) в процесі В, а потім виконується перевірка **свого** семафора командою P(S1) в процесі А і командою P(S2) в процесі В, і, якщо він ще закритий, то процес очікує доки інший процес також прийде в точку синхронізації.

В результаті, якщо до точки синхронізації першим прийшов процес А, то він спочатку відкриває дорогу процесу В, піднімаючи його семафор S2, а потім переходить в стан очікування, коли відкриється свій семафор S1. Коли до точки синхронізації підходить процес В, то він також спочатку відкриває дорогу процесу А, піднімаючи його семафор S1, **і обидва процеси А і В опиняються в точках синхронізації одночасно і, оскільки обидва семафори вже відкриті, то виконання частин коду обох процесів нижче точок синхронізації (вертикальна штриховка) починається також одночасно**. Якщо до точки синхронізації першим прийде процес В, то дії будуть виконані так само з точністю до навпаки, але, головне,

що знову **обидва процеси A і B опиняються в точках синхронізації одночасно при відкритих семафорах і рухаються далі також одночасно.**

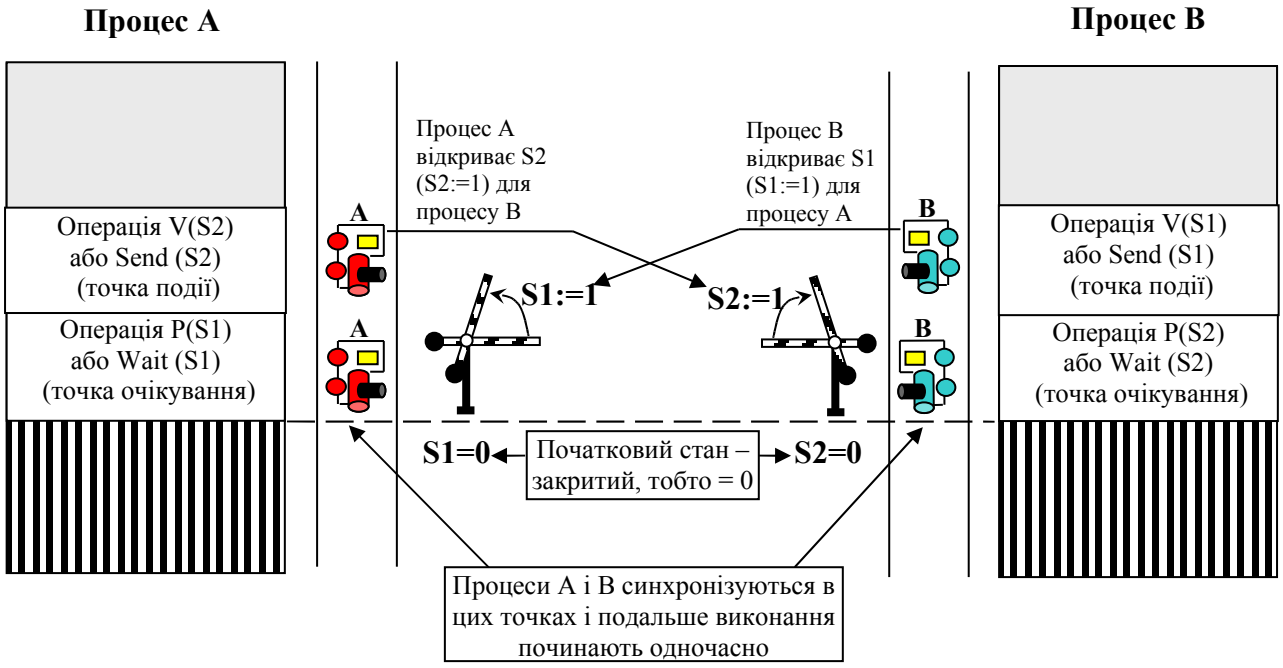
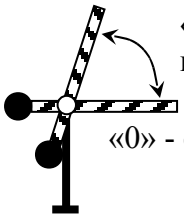


Рис.10. Застосування двійкових семафорів для рішення задачі повної синхронізації двох процесів

Багатозначні (множинні) семафори

Багатозначний (множинний) семафор, на відміну від двійкового, може приймати множину значень від нуля до заданого цілого додатного числа N (від'ємні значення не допускаються). ***Закритому/заблокованому стану багатозначного семафора відповідає тільки одне значення, а саме нуль. Всі інші значення від 1 до N відповідають відкритому/розблокованому стану семафора***.



«від 1 до N» - семафор відкритий/розблокований

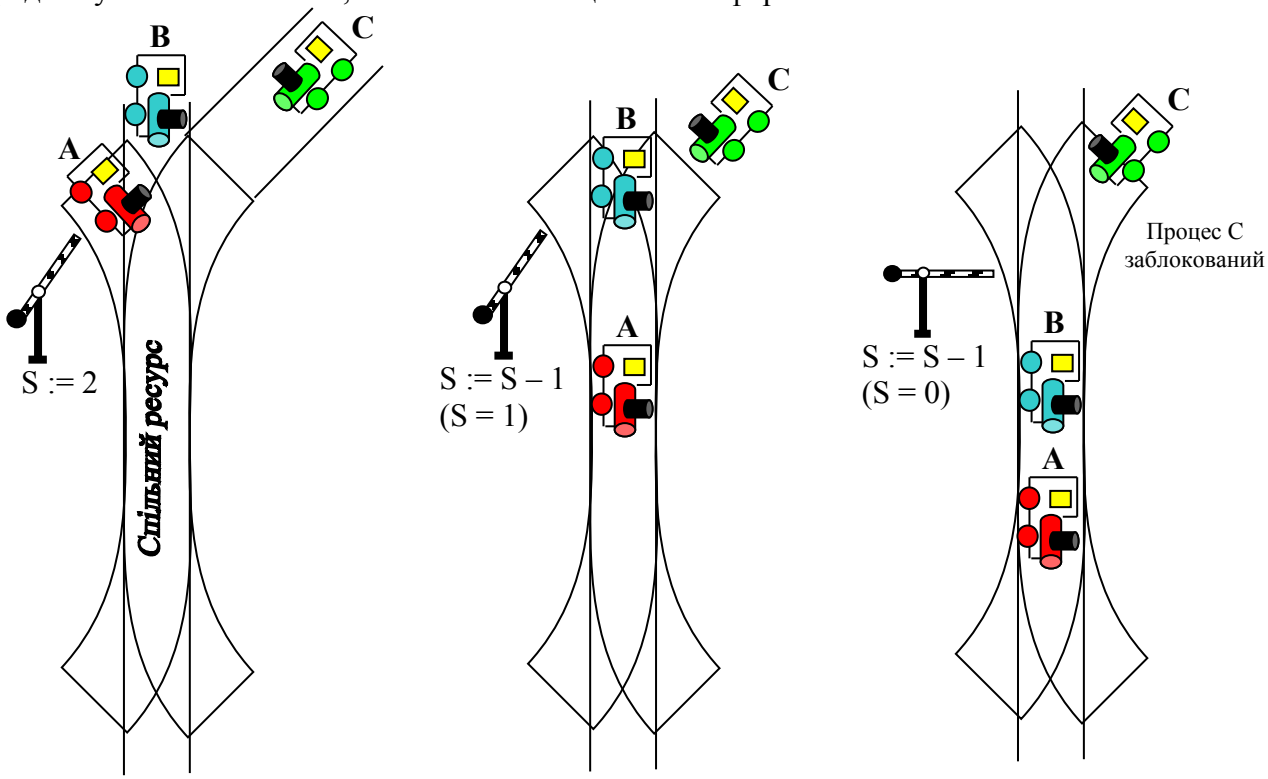
«0» - семафор закритий/заблокований

Рис.11. Аналогія багатозначного семафору із залізничним семафором

Багатозначні семафори використовуються в ситуаціях, коли спільним ресурсом дозволяється користуватись одразу декільком процесам/потокам, кількість яких визначається наперед заданим значенням N. Початкове значення семафору ставиться рівним N, кожен процес/потік, який входить в критичну секцію, зменшує значення семафору на одиницю, а кожен процес/потік, який виходить з критичної секції, збільшує значення семафору на одиницю. В результаті, якщо в критичній секції одночасно знаходиться вже N процесів/потоків, то семафор має значення 0, тобто є закритим/заблокованим, і вхід до критичної секції (N+1)-му процесу/потоку буде заблокований (див. рис.12.) доки один з попередніх процесів/потоків не вийде з критичної секції і не збільшить значення семафору.

У деяких системах паралельного програмування на додаток до окремих багатозначних семафорів, реалізовані ще й **масиви багатозначних семафорів**, які дозволяють виконати управління безпечним використанням багатьох спільних ресурсів одночасно.

Крім того, у окремих реалізаціях семафорів є можливість також задавати **період чекання на звільнення заданого семафора**, після закінчення якого процес/потік, що чекав, продовжує своє виконання, не захоплюючи цього семафора.



а) Початкове значення семафору встановлюється **S:=2**. Процес А прийшов до семафору S першим і перевіряє його значення, процес В прийде другим, а процес С – третім.

б) Процес А зайшов до критичної секції і зменшив значення **S:=S-1**, процес В підійшов до семафору S і перевіряє його значення, а процес С ще не дійшов до семафору. Поточне значення семафору **S=1**.

в) Процес В зайшов до критичної секції і зменшив значення **S:=S-1 (значення S стало 0)**, а процес А також ще знаходиться в критичній секції. Процес С перевірив значення закритого семафору **S=0** і перейшов в стан очікування.

Рис.12. Робота багатозначного семафору при N=2

Розглянемо виконання операцій P(S) та V(S) для багатозначного семафору.

Операція P(S): **Перевірити значення змінної S (спробувати пройти за семафор)**

якщо семафор S закритий/заблокований (S=0), 

то призупинити процес/потік («поїзд») і поставити його у чергу очікування семафору S;
інакше

дозволити входження процесу/потоку («поїзду») в критичну ділянку, бо семафор відкритий/розблокований (S>0),

і після цього зменшити значення семафору на одиницю S:=S–1, тобто наблизити семафор S до закритого/заблокованого стану або взагалі закрити/заблокувати його, якщо після цього значення семафору стало рівним нулю.

Операція V(S):
Збільшити значення семафору на одиницю S:=S+1, тобто відкрити/розблокувати семафор S, або (якщо він вже відкритий) наблизити його до повністю відкритого стану (S=N).

У деяких реалізаціях (наприклад, у мові Modula-2) використовується різновид багатозначних семафорів, який називається **багатозначними сигнальними змінними** (див. розділ «Сигнальні (умовні) змінні»).

М'ютекси

Термін «м'ютекс» походить від англійського терміну «mutex» (MUTual Exclusion, тобто взаємне виключення).

М'ютекс є засобом організації взаємного виключення доступу процесів до спільного ресурсу. Фактично, **м'ютекс є спрощеним видом двійкового семафору, який на відміну від семафору не має зв'язаного з ним лічильника (змінної).**

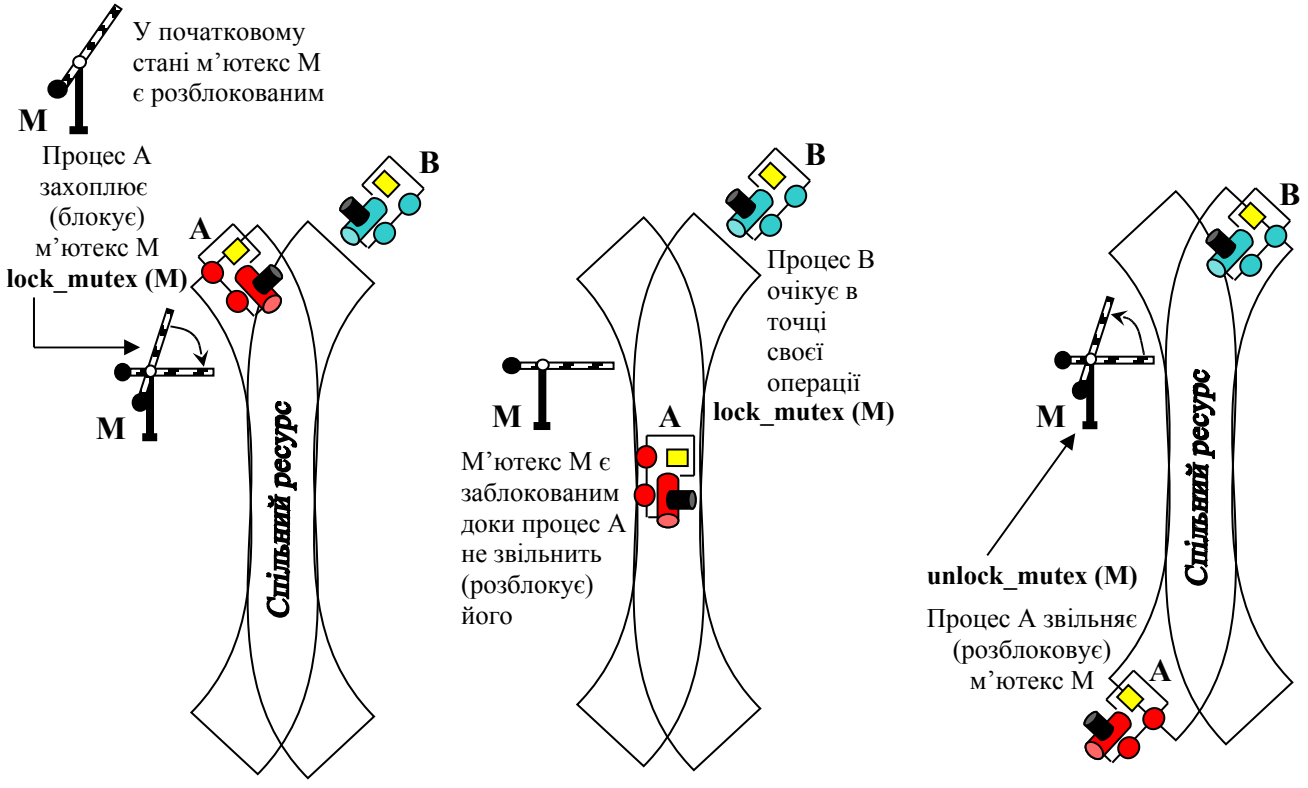
Над м'ютексом допустимі дві основні операції:

1) блокувати (закрити) м'ютекс (lock a mutex) або захопити (взяти) м'ютекс (acquire a mutex або get a mutex або take a mutex);
2) розблокувати (відкрити) м'ютекс (unlock a mutex) або звільнити (віддати) м'ютекс (release a mutex або put a mutex).

Причому, **на відміну від двійкового семафору, операцію звільнення м'ютекса може виконати тільки той же процес, який його захопив.** Тобто, якщо процес А захопив м'ютекс М у свою «тимчасову власність», то жоден інший процес В не може виконувати ніяких дій з м'ютексом М (ні повторного захоплення, ні звільнення) доки процес А сам не звільнить його і «не віддасть у загальне користування». Саме тому, **на відміну від двійкового семафору, м'ютекс може бути використаний тільки для рішення завдання взаємного виключення, але не може бути використаний для рішення завдання синхронізації**.

В конкретних реалізаціях м'ютексів, як правило, вимагається, щоб перед використанням м'ютексу, для нього були виконані операції створення м'ютекса та ініціалізації м'ютекса.

Приклад використання м'ютекса показаний на рисунку 13.



а) У початковому стані м'ютекс М є розблокованим. Процес А прийшов до м'ютекса М першим і захоплює (блокує) його разом зі спільним ресурсом за допомогою операції **lock_mutex (M)**.

б) Доки процес А володіє м'ютексом М і спільним ресурсом, процес В не має права змінювати стан м'ютекса М і знаходиться в стані очікування в точці виконання операції **lock_mutex (M)** в своєму коді.

в) Коли процес А закінчує виконання своєї критичної секції з використанням спільного ресурсу, він звільняє (розблоковує) м'ютекс М за допомогою операції **unlock_mutex (M)**, дозволяючи процесу В захопити його.

Рис.13. Застосування м'ютекса для рішення завдання взаємного виключення

Підсумовуючи, можна сказати, що **м'ютекс – це спеціалізований і спрощений вид двійкового семафору, призначений для рішення тільки задачі взаємного виключення**.

Крім того, слід зауважити, що аналогічно семафорам, у окремих реалізаціях м'ютексів є можливість також задавати **період чекання на звільнення заданого м'ютекса**, після закінчення якого процес/потік, що чекав, продовжує своє виконання, не захоплюючи цього м'ютекса.

Сигнальні (умовні) змінні

Сигнальні або умовні змінні (signal variable, conditional variable, completion variable) є ще одним засобом синхронізації процесів, який дуже схожий на семафори.

Сигнальні змінні, аналогічно семафорам, **бувають простими (двійковими) та багатозначними**. Сигнальні змінні також називають просто сигналами, що вносить деяку плутанину із сигналами-повідомленнями, що посилаються до цих сигнальних змінних.

Сигнальні (умовні) змінні є спеціалізованим видом семафорів, що *використовуються в основному для рішення задачі синхронізації*, подібно до того, як показано на рисунках 9 та 10.

Хоча сигнальні змінні можна застосувати також і для рішення задачі взаємного виключення (як показано на рисунку 8), але це не є їх характерним використанням, якщо мова програмування чи бібліотека функцій включають, крім сигнальних змінних, також стандартні засоби взаємного виключення процесів (м'ютекси, монітори, семафори). Можливість використання сигнальних змінних для рішення задачі взаємного виключення витікає із семантики операцій Wait(S) та Send(S), які виконуються майже так само, як і операції семафорів P(S) та V(S) відповідно, але, з точки зору логіки алгоритмів, посилання сигналу (повідомлення) самому собі або чекання на сигнал (повідомлення) від себе не є логічним. Тому, сигнальні змінні в основному використовуються для синхронізації процесів/потоків, коли операції Wait(S) та Send(S) виконуються в різних процесах, наприклад, Wait(S) у процесі A, а Send(S) у процесі B.

Сигнальна змінна – це спеціальна змінна, з якою зв'язані дві структури даних:

1) **лічильник надісланих у цю змінну сигналів (повідомлень);**
2) **черга процесів/потоків, які чекають на отримання сигналу (повідомлення), надісланого до цієї змінної.**

Крім операцій Wait(S) та Send(S), для сигнальних змінних, як правило, реалізована операція Init(S) для початкової ініціалізації (створення) сигнальної змінної, а також деякі додаткові операції, наприклад, Awaited(S) для перевірки, чи є її черга непустою, та GetValue(S) для взяття поточного значення лічильника сигнальної змінної.

Загальновизнаної класичної семантики багатозначних сигнальних змінних не існує, і тому конкретні реалізації таких змінних можуть досить суттєво відрізнятись. З іншого боку семантика операцій Wait(S) та Send(S) для двійкових сигнальних змінних досить стабільна і є практично еквівалентною двійковим семафорам.

Двійкові сигнальні змінні

Розглянемо семантику операцій Wait(S) та Send(S) для двійкових сигнальних змінних. Лічильник двійкової сигнальної змінної S приймає значення 1 (True) тільки якщо сигнал S був посланий і його не очікує жоден інший процес.

Операція Wait(S) виконується практично так само, як операція P(S) двійкового семафору.

Операція Wait(S): Перевірити значення лічильника сигнальної змінної S
якщо S=0 (False) (тобто сигнал S ще не був посланий)
то
призупинити процес, що виконав Wait(S), і помістити його у чергу сигнальної змінної S;
інакше (тобто якщо сигнал S вже був посланий і S=1 (True))
використати (забрати собі) цей сигнал, скинувши лічильник у значення S=0 (False), і продовжити виконання без затримки.

Операція Send(S) на відміну від операції V(S) двійкового семафору не просто безумовно встановлює значення лічильника у стан S=1, а виконує перевірку поточного стану черги сигнальної змінної S і деякі інші дії над цією чергою.

Тобто семантика сигнальних змінних включає деякі дії, які при реалізації класичних двійкових семафорів покладаються на диспетчер паралельних процесів, що слідкує за станом усіх черг.

Операція Send(S): Перевірити поточний стан черги сигнальної змінної S
якщо Черга(S) є пустою (тобто якщо жоден процес ще не чекає на цей сигнал, **незалежно** від стану лічильника S=0 чи S=1)
то
виконати S:=1, тобто встановити ознаку наявності невикористаного ще сигналу;

інакше (тобто якщо в Черзі(S) є як мінімум один процес, що чекає на цей сигнал)
активувати процес, що стоїть у голові Черги(S), і залишити значення S=0 (False), бо сигнал вже був використаний, як тільки з'явився.

Оскільки м'ютекси призначені для рішення тільки задачі взаємного виключення, а логічне використання сигнальних змінних – це рішення задачі синхронізації, то **на практиці м'ютекси та сигнальні змінні використовуються разом: сигнальна змінна синхронізує доступ групи процесів до спільного ресурсу, а м'ютекс, що використовується в парі з нею, забезпечує неможливість одночасного доступу до цього ресурсу**.

Використання пари «м'ютекс + сигнальна змінна» замість використання двох двійкових семафорів для рішення такої подвійної задачі одночасної синхронізації та взаємного виключення *має перевагу у тому, що в тексті програми* (завдяки використанню операцій з різними назвами Lock-Unlock та Wait-Send замість однакових операцій P та V для семафорів з різним призначенням) *неможливо помилково відкрити семафор, призначений для взаємного виключення, у іншому процесі (тобто використати як для задачі синхронізації) і навпаки, а також набагато чіткіше видно роботу алгоритму.* Крім того, можуть бути інші причини доцільності використання семафорів або сигнальних змінних та м'ютексів, які визначаються особливостями їх реалізації у конкретних ОС чи мовах програмування.

Наведемо приклад рішення класичної задачі «постачальник/споживач» за допомогою одного м'ютекса та двох сигнальних змінних. Основними проблемами цієї здачі є:
- недопущення одночасного звертання двох або більше процесів до спільного буфера;
- недопущення запису до вже повного буфера;
- недопущення читання з пустого буфера.

```
program Producer_Consumer;
var  Buf : buffer;
```

```
     M : mutex;
     Not_Full, Not_Empty : signal;
function IsFull : boolean;
{повертає True, якщо буфер є повним}
begin . . .
end;

function IsEmpty : boolean;
{повертає True, якщо буфер є пустим}
begin . . .
end;

function ProduceNew : data;
{повертає нове значення для запису у буфер}
begin . . .
end;

procedure WriteBuffer (in X : data);
{записує значення X до буфера}
begin . . .
end;

procedure ReadBuffer (out X : data);
{читає значення X з буферу}
begin . . .
end;

process Producer;
var X : data;
begin
   loop
       X := ProduceNew;
       Lock(M);
       if IsFull then
           Unlock(M);
           Wait(Not_Full);
           Lock(M);
      endif;
      WriteBuffer(X);
      Unlock(M);
      Send(Not_Empty);
   endloop;
end;

process Consumer;
var X : data;
begin
   loop
       Lock(M);
       if IsEmpty then
           Unlock(M);
           Wait(Not_Empty);
           Lock(M);
      endif;
      ReadBuffer(X);
      Unlock(M);
      Send(Not_Full);
```

```
   endloop;
end;

Begin
      . . .
End.
```

Аналогічно спільному використанню з м'ютексами сигнальні змінні використовуються також і в парі з моніторами.

Багатозначні сигнальні змінні

Як зазначалося вище, **багатозначні сигнальні змінні**, фактично, є різновидом багатозначних семафорів, і реалізація таких змінних досить суттєво відрізняється в різних реалізаціях.

Розглянемо реалізацію багатозначних сигнальних змінних у мові Modula-2, яка є цікавою з теоретичної точки зору.

Примітка. Мова Modula-2 є другою популярною мовою програмування (після мови Pascal), створеною професором Ніклаусом Віртом, яка націлена на потреби системного програмування і включає засоби паралельного програмування.

У мові Modula-2 реалізація багатозначних сигнальних змінних є *розширеним* видом багатозначних семафорів. Це визначається тим, що лічильник сигнальної змінної мови Modula-2 може змінюватись у діапазоні від **–N до +N** (у багатозначних семафорів від 0 до +N):

– N ÷ –1 – черга не пуста, тобто процесів, що виконали команду чекання WAIT було більше, ніж процесів, які виконали команду посилання сигналу SEND.

0 – черга пуста і кількість процесів, що виконали команду чекання WAIT дорівнює кількості процесів, які виконали команду посилання сигналу SEND.

+1 ÷ +N – черга пуста і накопичена певна кількість надісланих сигналів, тобто процесів, що виконали команду чекання WAIT було менше, ніж процесів, які виконали команду посилання сигналу SEND.

Такі властивості багатозначних сигнальних змінних мови Modula-2 надають можливість ефективно використовувати їх як для рішення довільних задач синхронізації, так і задач взаємного виключення. Враховуючи універсальність їх реалізації, сигнальні змінні є єдиним засобом синхронізації та взаємного виключення у мові Modula-2.

Розглянемо семантику операцій над сигнальними змінними, які реалізовані у мові Modula-2.

Процедура Init (S) ініціалізує сигнальну змінну S, що визначається у прив'язуванні до цієї змінної пустої черги процесів та обнуленні лічильника сигналів цієї змінної. Використовувати неініціалізовані сигнальні змінні заборонено.

Процедура WAIT (S) зменшує значення лічильника сигналів змінної S на 1, і, якщо після цього його значення стане від'ємним, то переміщує поточний процес із черги готових процесів до черги сигнальної змінної S та переводить у активний стан процес, що знаходиться у голові черги готових процесів. Процеси, що знаходяться у черзі сигнальної змінної, впорядковані згідно їх пріоритетів.

Функція Awaited (S) повертає значення TRUE, якщо лічильник сигнальної змінної S має від'ємне значення, тобто якщо черга сигнальної змінної S не пуста.

Процедура SEND (S) збільшує лічильник сигналів сигнальної змінної S на 1. Якщо черга сигнальної змінної S не пуста, то процес, що знаходиться в голові цієї черги, переміщується до

черги готових процесів. Якщо пріоритет переміщеного процесу більше, ніж у процеса, що стоїть в голові черги готових процесів, то переміщений процес стане активним процесом.

Процедура Notify (S). Якщо черга сигнальної змінної S не пуста, то процес, що знаходиться в голові черги, переміщується до черги чекаючих процесів і збільшує лічильник сигналів цієї сигнальної змінної на 1. Тобто, якщо використовувати тільки Notify і не використовувати SEND, то лічильник буде працювати тільки в діапазоні від –N до 0. Переведення процесу з черги чекаючих процесів до черги готових процесів відбувається тільки при найближчому такті роботи диспетчера процесів. Якщо черга сигнальної змінної S пуста, то виконання процедури Notify не призводить ні до яких дій, на відміну від SEND, виконання якої завжди збільшує лічильник.

В інших реалізаціях сигнальні змінні є фактично множинними семафорами або з якимись не дуже суттєвими відмінностями, або в точності еквівалентні множинним семафорам, які мають назву «сигнальні змінні».

Також відзначимо, що аналогічно семафорам та м'ютексам, у деяких реалізаціях сигнальних змінних є можливість задавати **період чекання на сигнал заданої сигнальної змінної**.

Спінлоки

Спінлок (англ. Spinlock) є ще одним засобом організації взаємного виключення доступу процесів до спільного ресурсу.

Фактично, **спінлок є спеціальним видом м'ютекса, який на відміну від нього має такі особливості:**

- Спінлок є змінною, до якої застосовуються **атомарні операції** блокування та розблокування, які повинні бути в системі команд процесора.
- Процес/потік, що використовує спінлок, **не звільняє процесор** переходячи у стан чекання.
- Спінлоки є сенс використовувати **тільки в багатопроцесорних** обчислювальних системах.

Кожен процесор, що бажає отримати доступ до спільного ресурсу, атомарно записує умовне значення «зайнято» до змінної спінлока, використовуючи команду swap або її аналог (в архітектурі x86 — xchg). Якщо попереднє значення змінної, що повертається цією командою, було «спінлок вільний/розблокований», то даний процес блокує спінлок (записує до змінної спінлока значення «спінлок захоплений/заблокований») і отримує доступ до ресурсу, в протилежному випадку, процесор повертається до операції swap і **крутиться у циклі чекання не звільняючи процесора**, доки спінлок буде звільнений. Після роботи зі спільним ресурсом процесор-володілець спінлока повинен записати до нього умовне значення «спінлок вільний/розблокований».

В деяких реалізаціях (наприклад C#) операція Wait спінлока не «відмовляючись» від часу процесора, завантажує його в циклі на задану кількість ітерацій (як правило, 50 ітерацій еквівалентні паузі приблизно в одну мікросекунду, хоча це залежить від швидкості і завантаженості процесора).

У прикладних паралельних програмах спінлоки використовуються рідко, бо головне їх застосування – це чекання ресурсу, який повинен звільнитися дуже скоро (протягом мікросекунд), і тому використання м'ютекса буде менш ефективним, оскільки він витрачає процесорний час на перемикання потоків. І навпаки, у системних програмах, що працюють з ядром операційної системи, спінлоки використовуються досить часто, бо в таких задачах ситуація короткотермінового чекання є характерною.

Слід зауважити, що використання спінлоків є вигідним тільки на багатопроцесорних комп'ютерах, бо на однопроцесорному комп'ютері у спільного ресурсу нема жодного шансу звільнитися, доки потік, що виконав операцію Wait спінлока і чекає, не витратить залишок свого кванту часу, а значить, мета спінлоку (витратити на чекання менше часу, ніж при

використанні м'ютекса) буде недосяжною з самого початку. Тому часті або довготривалі виклики операції Wait спінлока даремно витрачають час процесора.

Критичні секції

Критичні секції (англ. critical sections, critical regions) є найпростішим засобом організації взаємного виключення доступу процесів до спільного ресурсу.

Критичні секції вирішують задачу взаємного виключення на основі підходу, що ґрунтується на контролі процесів/потоків, як показано на рис.4, де в початкових точках синхронізації перед критичною секцією ставиться операція блокування (Lock) критичної секції, а в точках синхронізації після коду критичної секції ставиться операція розблокування (Unlock) критичної секції. **Доки процес/потік виконує код своєї критичної секції час процесора належить тільки йому, і він не може бути перерваним іншим процесом/потоком.** Очевидно, що операції Lock та Unlock повинні використовуватись попарно.

Критична секція в класичному розумінні – це синтаксична конструкція, яка, на відміну від семафорів та м'ютексів, об'єднує команди входу до критичної секції (у сенсі «фрагмент програми») **та виходу з неї у *єдину конструкцію*.**

Інакше кажучи, класична критична секція має вигляд, подібний до оператора блоку в мовах програмування (**begin … end** у мові Pascal або **{ … }** у мовах C/C++/C#).

Перевага конструкції «критична секція» *полягає в тому, що ця конструкція дозволяє компілятору легко виявити помилки пропущеного розблокування критичної секції (виходу з критичної секції).*

Такі помилки є характерними при використанні семафорів та м'ютексів, тобто, коли при вході до критичної секції (фрагменту програми) блокується один семафор/м'ютекс (наприклад S1), а при виході з неї помилково розблоковується інший (наприклад S2), або ж взагалі операція розблокування пропущена. Конструкції семафорів та м'ютексів не дозволяють компілятору виявити такі помилки.

Критичні секції бувають двох видів:

1) **неіменовані** критичні секції (Modula-2);
2) **іменовані** критичні секції (Win32, Linux).

Неіменовані критичні секції у мові Modula-2 реалізовані двома процедурами Lock та Unlock, які не мають параметрів, і повинні використовуватись парами (аналогічно до **begin … end**). У результаті компілятор може легко виявити не закриті критичні секції.

Недоліком неіменованих критичних секцій *є їх відносна «негнучкість» у використанні, бо вхід будь-якого процесу/потоку до своєї критичної секції забороняє вхід до своїх критичних секцій* ***усім*** *іншим процесам/потокам даної програми.*

Класична реалізація **іменованої критичної секції** має такий вигляд:

```
      ...
var
      CS1, CS2 : CriticalSection;
           . . .
Begin
           . . .
      ┌ LockCriticalSection (CS1);
      │    . . .
      └ UnlockCriticalSection;
           . . .
      ┌ LockCriticalSection (CS2);
      │    . . .
```

```
    └ UnlockCriticalSection;
          . . .
End.
```

Як видно з прикладу, процедура/оператор закінчення критичної секції UnlockCriticalSection працює як ключове слово **end** у операторі блоку **begin … end**, тобто закриває найближчу критичну секцію.

***Іменовані критичні секції позбавлені недоліку неіменованих критичних секцій**, бо входження якогось потоку до критичної секції CS1 не блокує входження іншого процесу до критичної секції CS2.*

На жаль, у деяких реалізаціях критичними секціями називають засоби взаємодії паралельних процесів/потоків, які по своїй суті еквівалентні м'ютексам або двійковим семафорам, оскільки у процедурі закінчення критичної секції, так само, як у двійкових семафорах та м'ютексах, потрібно вказувати ім'я критичної секції, що розблоковується, тобто

```
          . . .
┌ LockCriticalSection (CS1);
│         . . .
└ UnlockCriticalSection (CS1);
          . . .
```

Відмінність такої реалізації від класичної нібито зовсім невелика, але вона змінює суть конструкції іменованої критичної секції, оскільки, в результаті, знову з'являється можливість таких же помилок, як при використанні двійкових семафорів та м'ютексів.

Така практично еквівалентна двійковим семафорам та м'ютексам реалізація критичних секцій вносить додаткову плутанину до термінології паралельного програмування.

Про термінологію

На жаль, в сфері паралельного програмування (процесів/потоків та засобів їх синхронізації) устаткувалась не зовсім чітка україномовна і російськомовна термінологія. Причин цьому декілька. По-перше, з розвитком теорії паралельних процесів з'являлись нові поняття і концепції, яким попередня термінологія відповідала не зовсім точно. По-друге, навіть в оригінальній англомовній термінології не завжди застосовували найкращі терміни. По-третє, в різних реалізаціях паралельних процесів історично використовуються не завжди однакові терміни для однакових понять. По-четверте (і найвпливовіше), при перекладі на російську, а пізніше і на українську мову, перекладачі часто використовували дослівний переклад, вибираючи не завжди найкращі з можливих синонімів для англомовних термінів (lock - блокировка). В результаті з'явились такі «терміни-шедеври», як «захватить блокировку» російською мовою і «захопити блокування» українською, які з точки зору «чистої» російської та української мов взагалі не мають сенсу, не кажучи вже про те, що навіть з технічної точки зору ці терміни не зовсім точно відображують суть того, що відбувається при синхронізації процесів. Або, наприклад, як при такому підході до переводу перекласти словосполучення "a blocked lock"? "Заблокированная блокировка" (рос.), "заблоковане блокування" (укр.)? Інший приклад з документації: "Once the spinlock is locked...". Переклад у вищенаведеному силі: "Як тільки спін-блокування буде заблоковане..."

Тому у викладеному матеріалі автор намагався уникати подібних термінологічних «шедеврів».

Модель передачі повідомлень

Рандеву

Довгий час багато-процесове та багато-потокове програмне забезпечення реалізовувалося на машинах з одним процесором, і тому передача даних між процесами забезпечувалась через області спільної пам'яті (**модель спільних даних** для організації взаємодії паралельних процесів/потоків).

Але для програмування комп'ютерних систем з розподіленою пам'яттю така модель не може бути використана. Тому для організації паралельних процесів/потоків в програмах для таких систем треба використовувати **модель передачі повідомлень**.

Чарльз Хоар та Брінч Хансен у 1978 році показали, що можна піти більш природним шляхом при рішенні проблеми міжпроцесового зв'язку, ніж використання спільних даних, якщо **вважати передачу даних та синхронізацію єдиною нерозривною діяльністю**. Причому **запропонований ними метод є універсальним**, тобто може бути застосований як для одно-, так і для багатопроцесорних систем, а також як для систем з розподіленою пам'яттю, так і для систем із спільною пам'яттю.

!!!

Чарльз Хоар та Брінч Хансен запропонували новий **метод взаємодії процесів,** який був названий ***рандеву.***

Коли процес А має намір передати дані процесу В, обидва процеси повинні заявити про свою готовність встановити з'єднання, видавши заявки відповідно на передачу та отримання. Якщо виявиться, що процес А видасть заявку на передачу першим, то його виконання призупиняється до тих пір, поки процес В не видасть заявку на отримання. Аналогічно, якщо черговість буде протилежною. Коли обидва процеси таким чином синхронізуються, виконується передача даних та кожний з процесів продовжує свою роботу.

!!!

Варто відмітити, що фактично передача даних між процесами при взаємодії шляхом рандеву, виконується без використання буфера. Тому передача даних між асинхронними процесами, для запобігання непотрібної затримки процесу, що досяг точки рандеву першим, повинна програмуватись з використанням процесу-посередника, який буде виконувати роль буфера.

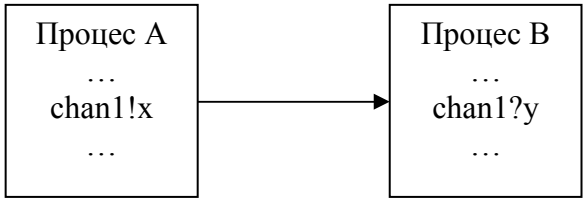
Хоар та Хансен запропонували дві моделі рандеву:

1. Хоар запропонував **симетричну модель рандеву**, що реалізована в мовах Occam та Occam-2.
2. Хансен запропонував **асиметричну модель рандеву**, що була реалізована в мові Ада.

Симетрична модель рандеву

Модель, запропонована Хоаром, реалізує взаємодію процесів симетрично в тому сенсі, що обидва взаємодіючих процеси оброблюються однаково та повинні бути повністю синхронізованими на операторах передачі та отримання даних.

В наступному прикладі процес А передає дані процесу В.



Такий механізм є симетричним та синхронним тому, що при виконанні рандеву обидва процеси називають ім’я одного й того ж каналу та очікують один одного в точках рандеву.

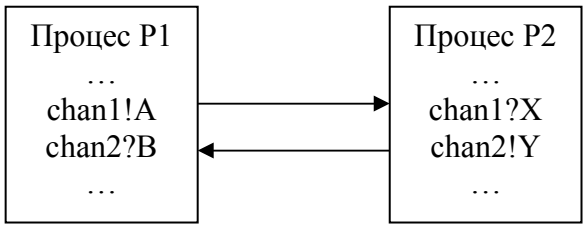
Взагалі, Occam-модель будується на основі посередника, що зветься каналом. Кожний канал має унікальне ім’я та передає інформацію тільки в одному напрямку. Крім того, з кожним каналом може бути зв’язаний тільки один процес-передавач та один процес-отримувач.

Команди для читання та запису в канал мають наступний вигляд:

1. Для передачі значення виразу через канал:
   ІмяКаналу ! Вираз

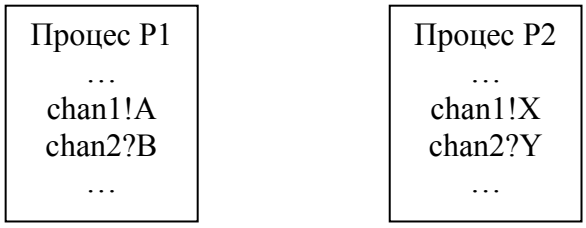
2. Для приймання значення з каналу в деяку змінну:
   ІмяКаналу ? Змінна

Розглянемо ще один приклад:



Процес, що досяг подвійного рандеву першим, затримується на першій команді та очікує на другий процес. Як тільки другий процес також досягає відповідної команди, обидва процеси стають активними і знову зупиняються для виконання другого рандеву. Після цього P1 та P2 відновлюють роботу.

Розглянемо модифікацію цього прикладу:



В цьому прикладі процес P1 не може оброблюватись, поки процес P2 не зчитає з каналу chan1 значення виразу A, а процес P2 не може цього зробити, поки не виконає операцію запису значення виразу Y в канал chan2. Але читання значення з каналу chan2 до змінної B у процесі P1 може виконатись тільки після операції запису виразу A в канал chan1.

Цей приклад є класичним прикладом дедлоку в паралельній програмі.

Хоар розробив теорію взаємодії паралельних процесів, яку назвав *CSP (Communicating Sequential Processes)* і яка базується на використанні моделі симетричного рандеву. Він показав, що формальна семантика CSP дозволяє доводити відсутність у заданій програмі дедлоків.

Асиметрична модель рандеву

Альтернативою симетричному підходу є асиметрична модель рандеву, в якій при виконанні рандеву тільки один процес (master - хазяїн) називає ім'я другого процесу (slave - підлеглий). Цей підхід взяв за основу Хансен у проекті системи взаємодії процесів у мові Ada (Ада).

В мові Ada **задача (процес)**, так само як і пакет (модуль) складається з двох частин: специфікації та тіла задачі.

Специфікація задачі (процесу) може мати одну з двох форм:

1. Форма 1:
```
task Ид-р;
```
2. Форма 2:
```
task Ид-р is
    описи_входів рандеву
end Ид-р;
```

Форма 1 визначає задачу без входів. Таку задачу не можна викликати іншими задачами для виконання рандеву. Вона використовується для процесів-хазяїв.

Тіло задачі має наступну форму:

```
task body Ид-р is
        описи
begin
        послідовність операторів
        [exception обробка виключень]
end Ид-р;
```

Приклад. В якості прикладу розглянемо взаємодію двох задач (процесів) при рішенні класичної задачі ПОСТАЧАЛЬНИК–СПОЖИВАЧ:
Задача PRODUCER (постачальник) зчитує деякий текст зі стандартного вхідного файлу та передає цей текст іншій задачі – CONSUMER (споживач), яка заміняє всі маленькі букви великими та записує перетворений символ до стандартного вихідного файлу.

Вхідний файл → PRODUCER → CONSUMER → Вихідний файл

Специфікація задачі PRODUCER має вигляд:

```
task PRODUCER;
```

Специфікація задачі CONSUMER має наступний вигляд:

```
task CONSUMER is
        entry RECEIVE (C: in CHARACTER);
        -- entry – це опис точки входу рандеву
        -- C – це формальний параметр
end CONSUMER;
```

Задача PRODUCER зчитує по одному символу зі стандартного вхідного файлу та передає їх задачі CONSUMER. Тіло задачі PRODUCER має такий вигляд:

```
        task body PRODUCER is
-- PRODUCER – це задача-хазяїн з точки зору виконання рандеву
                C: CHARACTER;
        begin
                while not END_OF_FILE (STANDART_INPUT) loop
                        GET (C);
                        CONSUMER.RECEIVE (C);
                        -- оператор виклику точки входу рандеву і
                        -- передачі до неї параметра C
                end loop;
        end PRODUCER;
```

Виклики входів рандеву аналогічні викликам процедур, за виключенням, що при виклику входу повинно вказуватись ім'я задачі, що містить цей вхід. Виклики входів можна розглядати як виклики процедур, що відбуваються в момент виконання рандеву між задачами.
Задача PRODUCER завершиться, коли будуть вичерпані дані у вхідному файлі.

Тіло задачі CONSUMER має наступний вигляд:

```
        task body CONSUMER is
-- CONSUMER – це задача-підлеглий з точки зору виконання рандеву

                X: CHARACTER;
        begin
                loop
-- Оператор accept – це оператор приймання інформації, що передається
-- через рандеву.
-- Імена задач, які викликають точку входу RECEIVE, не вказуються!
-- Тобто точку входу процесу-підлеглого може викликати будь-який процес-хазяїн
                        accept RECEIVE (C: in CHARACTER) do
                                X := C;
                                -- переприсвоєння необхідне, бо C є локальною
                                -- для оператора accept і використовувати C за
                                -- межами accept не дозволяється
                        end RECEIVE;

                        PUT (UPPER(X));
                        PUT (Character'Pos(X));
                end loop;

        end CONSUMER;
```

Рандеву виконується у момент, коли задача PRODUCER викликала вхід RECEIVE, а задача CONSUMER готова прийняти цей виклик. Задачі синхронізуються на вході RECEIVE. Виконання задачі PRODUCER призупиняється до тих пір, поки задача CONSUMER не досягне кінця оператора accept, що зв'язаний зі входом RECEIVE.

Задача PRODUCER закінчиться коли буде досягнутий кінець її тіла. Задача CONSUMER не завершиться ніколи, бо містить нескінчений цикл.

Повний текст головної програми матиме такий вигляд:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure CONVERT_TO_UPPER_CASE is
        task PRODUCER;
        task CONSUMER is
                entry RECEIVE (C: in CHARACTER);
        end CONSUMER;
        function UPPER (C: in CHARACTER) return CHARACTER is
        begin
                if c >= 'a' and c <= 'z' then
                        return CHARACTER'VAL (CHARACTER'POS(C) –
                        – CHARACTER'POS('a') + CHARACTER'POS('A');
                else
                        return C;
                enf if
        end UPPER;
        task body PRODUCER is
                …
        end PRODUCER;
        task body CONSUMER is
                …
        end CONSUMER;
begin
        -- тіло головної програми
        -- задачі стають активними автоматично
    null;
        -- за синтаксисом тіло підпрограми має містити хоча б один
        -- оператор. Тому, якщо тіло не містить жодного оператора,
        -- то ставиться оператор null.
end CONVERT_TO_UPPER_CASE;
```

При досягненні кінця файлу виконання процедури GET викликає виключну ситуацію, яка призводить до завершення задачі, бо обробник виключних ситуацій не був заданий.

Оператор відбору (select)

Оператор відбору select дозволяє організувати кілька варіантів взаємодії процесів за допомогою рандеву.

Варіанти взаємодії процесів за допомогою рандеву (види використання оператора відбору select):

1. **Відбір з очікуванням (selective wait).**

Оператор відбору select з очікуванням забезпечує недетермінований прийом викликів входу рандеву від однієї чи декількох задач.

2. **Умовний виклик входу (conditional entry call).**

Умовний виклик входу рандеву на відміну від звичайного є неблокуючим, тобто процес, що викликав вхід рандеву, не чекає, поки викликаний процес буде готовий прийняти виклик цього входу, а продовжує своє виконання (виклик пропускається).

3. **Часовий виклик входу (timed entry call).**

Часовий виклик входу рандеву аналогічний умовному, але перед тим, як продовжити своє виконання, процес, що викликав вхід рандеву, очікує протягом заданого періоду часу. Якщо викликаний процес не буде готовий прийняти цей виклик за вказаний час, то виклик рандеву також пропускається.