2 Взаємодія процесів

2.2.1 Проблема взаємного виключення і способи її вирішення

Два процеси називаються *паралельними (concurrent)*, якщо їх виконання може перекриватися в часі, тобто, наприклад, другий процес починається раніше, ніж завершується перший. Два процеси називаються *послідовними (serial***),** якщо вони не є паралельними.

мультипроцесорній системі досягається *фізична паралельність.* Якщо виконання декількох процесів чергується в одному-єдиному процесорі, то досягається *логічна паралельність.*

Можна виділити три види взаємодії процесів:

• *обмін інформацією* – якщо процеси безпосередньо проінформовані про наявність один одного;

• *конкуренція за доступ до ресурсів* – якщо процеси не проінформовані про наявність один одного; ОС повинна регулювати такі звернення;

• *узгодження дій процесів* – якщо процеси не прямо обізнані один про одний (наприклад, процес А генерує дані, а процес В виводить їх на друк).

При цьому паралельні процеси взаємодіють за допомогою або *механізму поділюваних змінних*, або *механізму передачі повідомлень*.

При роботі паралельних процесів, як в однопроцесорних, так і в багатопроцесорних системах неможливо передбачити відносну швидкість їх роботи, і тому виникають проблеми при взаємодії таких процесів.

1. *Стан гонок* (змагань), коли два процеси використовують одну і ту ж глобальну змінну і обидва виконують читання і запис цієї змінної. Критичним при цьому виявляється порядок читання і запис цієї змінної різними процесами.

2. *Блокування та взаємоблокування при доступі до ресурсів*[1]. Процес А може зажадати і одержати контроль над деякими пристроєм

[1] Питання взаємоблокуваня винесено в окремий підрозділ 2.4. Взаємне блокування.

введення/виведення (ПВВ), після чого тимчасово призупинити роботу. При цьому небажано, щоб ОС блокувала даний ПВВ і не дозволяла іншим процесам використовувати його, так як це може призвести до взаємоблокування.

Програмні помилки, які важко виявити, так як результат роботи програми перестає бути детермінованим і відтворюваним.

Приклад стану гонок. Нехай у нас є процедура, що виконує читання символу з клавіатури і його відображення на екрані, яка знаходиться в пам'яті, що розділяється і використовується всіма процесами:

```
void echo ()
{
chin = getchar (); chout = chin; putchar (chout);
}
```

Розглянемо таку послідовність подій:

процес А виконує оператор chin = getchar () – введено символ «*x*»;

процес В виконує процедуру echo до кінця – введено і надруковано символ «*y*»;

процес А продовжує своє виконання, але при цьому буде виведений вже символ «*y*», а не «*x*», тобто «*y*» буде виведений двічі, а «*x*» – втрачений.

Така проблема не буде виникати, якщо в кожен момент часу тільки один процес буде входити в процедуру *echo*, і ця процедура обов'язково повинна бути повністю виконана процесом, що увійшов до неї до того, як стане можливим її виконання іншим процесом.

Та ж проблема залишається і в багатопроцесорній системі.

Для уникнення проблем типу гонок, блокувань та взаємоблокувань в концепції процесів існує *механізм взаємного виключення* (mutual exclusion).

Взаємне виключення – механізм, який гарантує, що в будь-який момент часу тільки один процес виконує деяку визначену послідовність дій, і тим самим виключає можливість роботи іншого процесу.

Критичний ресурс (КР) – ресурс, до якого в кожний момент часу можливий доступ тільки одного процесу. Доступ до такого ресурсу здійснюється в ***критичній секції (КС)*** – частини коду, яка в будь-який момент часу може виконуватися тільки одним процесом.

Але взаємне виключення може привести до двох проблем:

взаємного блокування (*deadlock*).

голодування (*starvation*).

Розглянемо як виникає проблема голодування. Нехай є процеси *A, B* і *C*, кожному з яких періодично потрібен доступ до ресурсу *P*. Нехай процес *A* використовує ресурс *P*, а процеси *B* і *C* – очікують. Потім процес *B* використовує ресурс *P*, потім знову процес *A* використовує ресурс *P* і т.д. Тобто, процес *C* може ніколи не отримати доступ до ресурсу *P*, незважаючи на те, що ніякого взаємного блокування немає. Така ситуація і називається *голодуванням*.

Отже, щоб уникнути гонок, ми повинні уникати знаходження двох процесів одночасно в критичних секціях. Але цієї вимоги ще недостатньо! Для правильної спільної роботи паралельних процесів необхідно виконання шести умов для досягнення взаємного виключення:

два процеси не повинні одночасно перебувати в критичних секціях;

у програмі не повинно бути закладено ніяких початкових припущень про швидкість або кількість процесорів;

процес, що перебуває поза критичною секцією, не повинен блокувати інший процес;

повинна бути неможлива ситуація, коли процес вічно чекає попадання критичну секцію (тобто не повинні з'являтися взаємоблокування і голодування);

коли в КС немає жодного процесу, будь-який процес, що запитав можливість входу в цю КС, повинен негайно отримати доступ до неї;

процес залишається в КС лише протягом обмеженого часу.

Існують такі підходи до досягнення взаємного виключення:

програмний – в цьому випадку сам процес є «відповідальним» за досягнення взаємного виключення, тобто процес (як системний, так і користувацький) повинен координувати свої дії з іншими процесами для досягнення взаємного виключення без підтримки з боку ОС або середовища програмування;

апаратний – з використанням спеціальних машинних команд;

примітиви синхронізації – семафори, монітори та інші подібні засоби.

2.2.2 Апаратні способи досягнення взаємного виключення

Існують такі апаратні способи досягнення взаємного виключення:

Заборона переривань.

Спеціальні команди, що виконуються атомарно.

При *забороні переривань* процесор перемикається на інший процес тільки по перериванню. Відповідно, процес забороняє переривання перед входом у критичну секцію і дозволяє переривання відразу після виходу з критичної секції. Взаємне виключення при цьому досягається, тому що процес не переривається всередині критичної секції, і тому інші процеси не можуть увійти в свої критичні секції.

такого підходу є певні недоліки: по-перше, він може бути застосований тільки в однопроцесорних системах, по-друге, важливі події введення/виведення можуть бути не оброблені і, по-третє, вразі краху процесу всередині критичної секції може відбутися порушення роботи ОС.

Спеціальні команди, що виконуються атомарно. Під атомарними операціями маються на увазі такі команди зміни даних в оперативній пам'яті, які виконуються процесором з повним блокуванням доступу до пам'яті, тобто жоден інший процесор або зовнішній пристрій не може отримати доступ до комірок пам'яті, поки процесор-монополіст не завершить операцію з нею.

Розглянемо реалізацію взаємного виключення за допомогою атомарної команди «перевірити і встановити» (TestAndSet). Ця команда виконує атомарно дві операції: читання з комірки пам'яті і запис в неї значення «зайнято».

Нехай нульове значення змінної *lock* відповідає умові, що критична секція вільна, а значення *lock*, відмінне від нуля, відповідає зайнятості критичної секції. Тоді перед входом у критичну секцію процес повинен в циклі перевіряти, чи звільнилася критична секція:

```
while (TestAndSet (& lock));
/ * Критична секція * /
```

Відповідно, після виходу з критичної секції процес повинен встановити змінну в значення «вільно»:

```
lock = 0;
```

2.2.3 Програмні способи досягнення взаємного виключення

Розрізняють наступні програмні способи досягнення взаємного виключення:

Змінні блокування.

Алгоритм Деккера.

Алгоритм Петерсона.

Змінні блокування. Процес для здійснення входу в критичну секцію безперервно перевіряє значення деякої змінної, щоб визначити, чи є вільним ресурс, що розділяється. Ця змінна має назву «*змінна блокування*» і зберігає стан критичної секції. Початкове значення змінної блокування дорівнює *0*. Якщо процес хоче увійти в критичну секцію, то він попередньо зчитує значення змінної блокування, і якщо воно дорівнює 0, то процес змінює його на 1 і входить в критичну секцію. Якщо ж значення цієї змінної вже дорівнює 1, то процес чекає, поки це значення не зміниться на 0. Таким чином, 0 означає, що жодного процесу в критичній секції немає, а 1 – що певний процес вже перебуває в критичній секції.

```
while (busy);
busy = 1;
/ * Критична секція * /
.....
busy = 0;
/ * Решта коду * /
.....
```

При такому підході виникають певні проблеми, а саме:

ті ж проблеми, що і при організації доступу до загальних змінних;

якщо після виходу з критичної секції процес з якоїсь причини не встановить значення змінної блокування рівним 0, то інші процеси не зможуть увійти в критичну секцію, порушується умова про те, що процес, який знаходиться поза критичною секцією не повинен блокувати інший процес;

проблема непродуктивної витрати часу процесору.

Активне очікування (*busy waiting*) – постійна перевірка змінної в очікуванні деякого значення. *Спін-блокування* – блокування, що використовує активне очікування.

Алгоритм Деккера. Голландський математик Т. Деккер був першим, хто розробив програмне рішення проблеми взаємного виключення (1970). Коли процес *P0* намагається увійти в критичну секцію, він встановлює свій прапор *flag [0]* таким, що рівний *true*, а потім перевіряє стан прапора іншого процесу *P1*. Якщо його прапор дорівнює *false*, то процес *P0* може негайно увійти в критичну секцію, інакше *P0* звертається до змінної *turn*.

Якщо *turn = 0*, то це означає, що зараз – черга процесу *P0* на вхід в критичну секцію, і тоді *P0* періодично перевіряє стан прапора процесу *P1* Цей процес, в свою чергу, в певний момент часу виявляє, що зараз не його черга для входу в критичну секцію, і встановлює свій прапор рівним *false*, даючи можливість процесу *P0* увійти в критичну секцію.

після того як *P0* вийде з критичної секції, він встановить свій прапор рівним *false* для звільнення критичної секції і присвоїть змінній *turn* значення *1* для передачі прав на вхід в критичну секцію процесу *P1*:

P0	*P1*
flag[0] = true; *while (flag[l])* *{* *if(turn == 1)* *{* *flag[0] = 1 false;* *while(turn == 1)* */* очікувати */;* *flag[0] = true;* *}* *}* */* критична секція */* *turn = 1;* *flag[0] = false;*	*flag[l] = true;* *while (flag[0])* *{* *if (turn == 2)* *{* *flag[0] = false;* *while(turn щ 1)* */* очікувати */;* *flag [1] = true;* *}* *}* */* критична секція */* *turn =0;* *flag[l] = false;*

Алгоритм Петерсена. Г. Петерсон розробив набагато більш простий алгоритм взаємного виключення (1981 р.). З цього моменту алгоритм Деккера вважається застарілим.

```
int turn; / * 0 або 1 - чия зараз черга? * /
int interested [2];
/ * Всі змінні спочатку рівні 0 * /

void enter_cs (int  proc)
{
      int other = 1 - proc;
      interested [proc] = true;
      turn = proc;
      while (turn == proc && interested [other]);
}
void leave_cs (int proc)
{

interested [proc] = false;
}
```

Перед входом у критичну секцію процес викликає функцію *enter_cs* зі своїм номером (0 або 1) в якості параметра. Після виходу з КС процес викликає функцію *leave_cs* , щоб позначити свій вихід і тим самим дозволити іншому процесу вхід в КС.

Початково обидва процеси знаходяться поза КС. Коли процес 0 викликає функцію *enter_cs*, ця функція повертає керування тому, що процес 1 не зацікавлений в попаданні в КС. Тепер, якщо процес 1 теж викличе функцію *enter_cs* , то йому доведеться чекати, коли *interested [0]* стане рівним *false*, це відбудеться тільки в той момент, коли процес *0* викличе функцію *leave_cs*.

Нехай тепер обидва процеси намагаються викликати функцію *enter_cs* практично одночасно. Тоді обидва вони збережуть свої номери в змінну *turn*. При цьому в ній збережеться номер того процесу, який був другим, а попередній номер буде загублений. Нехай *другим* був процес 1, і відповідно, *turn = 1*. У цьому випадку, коли обидва процеси дійдуть до оператора *while*, процес *0* увійде в КС, а процес *1* залишиться в циклі і буде чекати, поки процес *0* вийде з КС.

2.3 Примітиви синхронізації

2.3.1 Семафори

Семафор – це спеціальна змінна, яка має ціле значення і пов'язану з ним чергу. Над семафором визначено три операції:

семафор може бути ініціалізований ненегативним значенням;

операція *wait* зменшує значення семафора; якщо це значення стає негативним, то процес, що виконує операцію *wait*, блокується;

операція *signal* збільшує значення семафора; якщо це значення не позитивне, то заблокований операцією *wait* процес розблокується.

Існує також більш обмежена версія семафора – *бінарний семафор*, що приймає тільки значення 0 або 1.

Для зберігання процесів, що очікують семафори, використовується *черга*. Якщо вона обслуговується за принципом FIFO, то такий семафор називається *сильним семафором*, інакше – *слабким семафором*.

Можна сказати, що семафори забезпечують рішення для всіх видів проблем синхронізації, але їм також властиві недоліки:

процес, що використовує семафор, повинен бути обізнаний про інші процеси, що використовують цей же семафор, так як операції над семафорами у всіх взаємодіючих процесах повинні бути чітко скоординовані;

операції над семафорами у всіх процесах повинні бути ретельно налагоджені, так як пропуск однієї з таких операцій може призвести до неспроможності (порушення цілісності ресурсу, що розділяється) або до взаємного блокування;

програми, що використовують семафори, дуже важко перевіряти на коректність.

Наведемо приклад застосування семафорів при взаємному виключенні. Розглянемо рішення задачі взаємовиключення з використанням семафора *s*. Нехай є *n* процесів. У кожному з цих процесів перед входом у критичну секцію виконується виклик *wait (s)*. Якщо значення *s* при цьому стає негативним, то відповідний процес призупиняється. Якщо ж це значення дорівнює 1, то воно зменшується до нуля, а процес негайно входить у критичну секцію. І оскільки *s* вже більше не є позитивним, жоден інший процес не може увійти в критичну секцію.

```
semaphore s = 1:
.....
wait (s);
/ * Критична секція * /
signal (s);
/ * Решта коду * /
```

Семафор ініціалізується значенням 1. Отже, перший процес, що виконує виклик *wait (s)*, може негайно потрапити в критичну секцію, встановлюючи при цьому значення семафора рівним *0*. Будь-який інший процес при спробі увійти в критичну секцію виявить, що вона зайнята. Відповідно, відбудеться блокування такого процесу, а значення семафора буде зменшено до *-1*. Намагатися увійти в критичну секцію може будь-яка кількість процесів, і кожна така неуспішна спроба зменшує значення семафору. Після того як процес, який увійшов в критичну секцію першим, покине його, значення *s* збільшується, а один із заблокованих процесів (якщо такі є) віддаляється з черги семафора і активізується. Таким чином, як тільки планувальник ОС надасть цьому процесу можливість виконання, процес тут же зможе увійти в критичну секцію.

Сигналізуючий семафор – це семафор з нульовим початковим значенням. Процес сигналізує про подію, виконуючи операцію *signal (s)*, а всі інші процеси очікують події, виконуючи операцію *wait (s)*.

2.3.1 М’ютекси

М’ютекс - це спрощена версія семафора: змінна, яка може знаходитися в одному з двох станів – блокованому і неблокованому.

Значення м’ютекса встановлюється за допомогою двох процедур. Якщо процес (або потік) збирається увійти в критичну секцію, то він викликає процедуру *mutex_lock*. Якщо м’ютекс не заблокований, тобто вхід в критичну секцію дозволений, то запит виконується, а процес, що викликає входить в критичну секцію. Якщо ж м’ютекс заблокований, то процес блокується до тих пір, поки інший процес, що перебуває в критичній секції, не вийде з неї, викликавши процедуру *mutex_unlock*. Якщо при цьому м’ютекс блокував кілька процесів; то з них вибирається один.

2.3.3 Монітори

Щоб спростити написання програм, був запропонований механізм, синхронізації більш високого рівня.

Монітор – це набір процедур, змінних і інших структур даних, об'єднаних в особливий модуль. Він являє собою високорівневу конструкцію мови програмування, яка забезпечує функціональність, еквівалентну до функціональності семафорів, але при цьому ним набагато легше управляти. Монітори реалізовані в мовах програмування Concurrent Pascal, Pascal-Plus, Modula-2, -3, Java.

Монітор являє собою програмний модуль, який складається з послідовності, що ініціалізується однією або декількома процедурами і локальних даних. *Основні характеристики монітора*:

локальні змінні монітора доступні тільки його процедурам; зовнішні процедури доступу до локальних даних монітора не мають;

процес входить в монітор шляхом виклику однієї з його процедур;

в моніторі в певний момент часу може виконуватися тільки один процес; будь-який інший процес, що викликав монітор, буде припинений в очікуванні доступності монітора.

Перші дві характеристики нагадують нам об'єкти в об'єктно-орієнтованому програмуванні; дотримання ж третьої умови дозволяє монітору забезпечити взаємовиключення. Дані монітора доступні в деякий конкретний момент тільки одному процесу, - отже, захистити спільно використовувані дані можна, помістивши їх в монітор. Якщо дані в моніторі представляють якийсь ресурс, то монітор забезпечує взаємовиключення при зверненні до цього ресурсу.

Для широкого застосування в паралельних обчисленнях монітори повинні включати в себе інструмент синхронізації. Припустимо, наприклад, що якийсь процес використовує монітор і, перебуваючи в цьому моніторі, він повинен бути припинений до виконання певної умови. При цьому потрібно механізм, який не тільки призупиняє процес, але і звільняє монітор, дозволяючи увійти в нього іншому процесу. Пізніше, коли умова виявиться виконаною, а монітор доступним, припинений процес зможе продовжити свою роботу з того місця, де він був призупинений.

Монітор підтримує синхронізацію за допомогою *змінних умови*, що розташовуються в моніторі і доступних тільки в моніторі. Працювати з цими змінними можуть дві функції:

swait (c) – призупиняє виконання процесу, що викликає за умовою *c*; монітор при цьому доступний для використання іншим процесом;

csignal (c) – відновлює виконання процесу, припиненого викликом *cwait (c)*; якщо є декілька таких процесів, то вибирається один з них, а якщо таких процесів немає, то ця функція нічого не робить.

2.4 Взаємні блокування

Взаємне блокування – це блокування групи процесів, що очікують події, яка може ніколи не настати, оскільки дану подію може викликати тільки інший процес з тієї ж групи. Взаємоблокування зазвичай виникають при доступі процесів до ресурсів.

Ресурс – це будь-який об'єкт, який може запитуватися і очікуватися процесом; ресурс може складатися з будь-якої кількості ідентичних одиниць, і процес може запитувати будь-яку кількість одиниць ресурсу.

Існують дві *основні категорії ресурсів*:

Повторно використовувані ресурси (reusable) не виснажуються при використанні (можуть використовуватися знову). Вони мають фіксовану кількість одиниць, що не може створюватися або знищуватися. Процес, що запитує такий ресурс, утримує його при використанні, потім звільняє, а потім ресурс може бути виділений іншим процесам. Приклади повторно використовуваних ресурсів: процесор, пристрій вводу/виводу, основна та вторинна пам'ять, структури даних (файли).

Ресурси, що витрачаються (consumable) можуть бути створені (вироблені) або знищені (спожиті). Якщо процес використав такий ресурс, то останній припиняє своє існування («витрачається»). Зазвичай обмеження на кількість витрачених ресурсів певного типу відсутні, і не заблокований процес-виробник може створити будь-яку кількість таких ресурсів. Приклади ресурсів, що витрачаються: переривання, сигнали, повідомлення, інформація в буферах введення/виведення.

два режими доступу до ресурсу:

Монопольний (exclusive).

Розділюваний (shared).

При *монопольному (exclusive) режимі* доступ до ресурсу в кожний момент часу може отримати тільки один процес. При *розділюваному (shared) режимі* доступ до ресурсу одночасно може отримати будь-яка кількість процесів.

Ці режими несумісні, тобто до ресурсу не може бути реалізований доступ одночасно в монопольному і розділюваному режимі. Якщо ж існує багато одиниць (копій) ресурсу, то доступ до одних з них може здійснюватися в розділюваному режимі для декількох процесів, а доступ до інших – в монопольному режимі. Очевидно, що два цих режиму доступу мають сенс тільки для повторно використовуваних ресурсів, так як споживані ресурси знищуються відразу після використання.

Приклад 1. Взаємне блокування з повторно використовуваними ресурсами.

Процес P1
Запит ресурсу A
Запит ресурсу B

Процес P2
Запит ресурсу B
Запит ресурсу A

Коли обидва ці процеси дійдуть до другого запиту, виникне взаємне блокування . Один зі способів уникнути його – це накласти системні обмеження на порядок запиту ресурсів.

Приклад 2. Взаємне блокування з повторно використовуваними ресурсами.

Нехай нам доступно 200 Кб ОП і виконується наступна послідовність запитів:

Процес P1
Запит 80 Кб
Запит 60 Кб

Процес P2
Запит 70 Кб
Запит 80 Кб

Якщо обидва ці процеси дійдуть до другого запиту, то виникне взаємне блокування – незважаючи на те, що в системі досить пам'яті для виконання кожного з процесів окремо.

Приклад 3. Взаємне блокування з ресурсами, що витрачаються.

Нехай два процеси намагаються одержати повідомлення від іншого процесу, а потім відправити йому повідомлення.

Процес P1
ml = Receive (P2);
Send (P2, m2);

Процес P2
ml = Receive (P1)
Send (P1, m2);

Взаємне блокування при цьому виникне, якщо операція *Receive* є блокуючою.

Умови виникнення взаємного блокування :

взаємне виключення – одночасно використовувати ресурс може тільки один процес;

утримання та очікування – процес може утримувати виділені ресурси під час очікування інших ресурсів;

відсутність перерозподілу – ресурс не може бути примусово забраний процесу, що його утримує;

циклічне очікування – існує замкнутий ланцюг процесів, кожний з яких утримує як мінімум один ресурс, необхідний процесу, наступного у ланцюзі після даного процесу.

Моделювання взаємних блокувань.

Узагальнений ресурсний граф – це двочастковий спрямований граф з непересічними множинами вузлів процесів і ресурсів.

Вектор доступності – невід'ємний цілий вектор (r_1 r_2, ..., r_m), що позначає кількість одиниць ресурсів, доступних в будь-якому стані.

Кожен процес на узагальненому ресурсному графі позначається кружечком, а ресурс – прямокутником, де ресурс, що використовується – це прямокутник з подвійною рамкою. Кружечки всередині прямокутників позначають кількості одиниць даного типу ресурсу. Дуга, спрямована від процесу P до ресурсу R, називається *дугою запиту*. Вона означає, що процесу P призначено одну одиницю ресурсу R. Дуга, спрямована від повторно використовуваного ресурсу R до процесу P, називається дугою призначення. Вона означає, що процесу P призначена одна одиниця ресурсу R. Дуга, спрямована від ресурсу R, що використовується до процесу P, називається дугою виробника. Вона означає, що процес P є виробником ресурсу R.

Приклад (рисунок 2.8). Процеси P_1 і P_2 утримують по одній одиниці ресурсу R_1; при цьому процес P_2 є виробником споживаного ресурсу R_2. Доступно по одній одиниці кожного ресурсу. Процес P_1 активний, а P_2 – заблокований і очікує призначення одиниці ресурсу R_1. Процес P_i є заблокованим в якомусь стані тоді і тільки тоді, коли для деякого ресурсу R_j кількість дуг запиту більше r_j

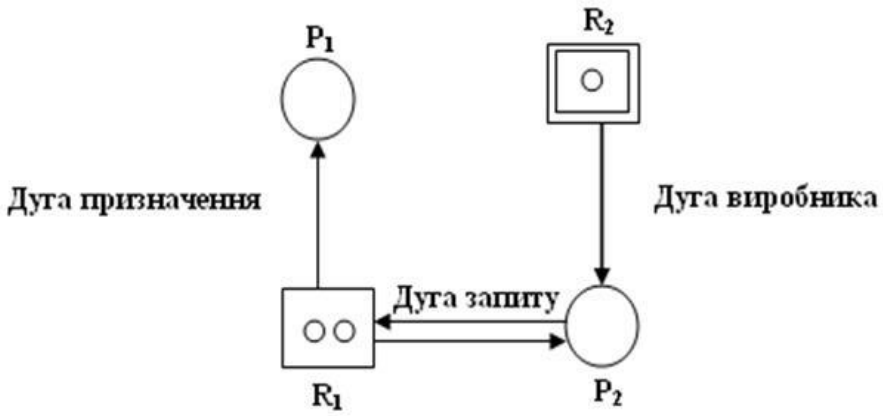

Рисунок 2.8 – Приклад графа ресурсів

Цикл в графі означає наявність взаємоблокування, циклічно включає в себе процеси і ресурси (припускаємо, що в системі є по одному ресурсу кожного виду).

Нехай існує три процеси: A, B, C і три ресурси: R, S, T, а також є наступна послідовність запитів і повернень ресурсу:

Процес A
запит R
запит S
звільнення R
звільнення S

Процес B
запит S
запит T
звільнення S
звільнення T

Процес C
запит T
запит R
звільнення T
звільнення R

ОС може запустити будь-який незаблокований процес в будь-який момент часу. Нехай запити ресурсів відбуваються у такому порядку:

процес A запрошує R;

B запрошує S;

C запрошує E;

A запрошує S;

B запрошує T;

C запрошує R.

Якщо ці шість запитів будуть здійснюватися в зазначеній послідовності, то ми отримаємо граф, показаний на рисунку 2.9, тобто відбудеться взаємне блокування.

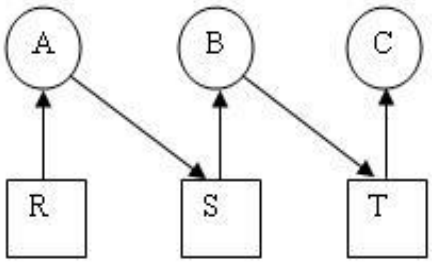

Рис. 2.9. Взаємне блокування.

Однак ОС не зобов'язана запускати процеси в якомусь особливому порядку. Зокрема, якщо виконання якого-небудь запиту спричиняє глухий кут, ОС може призупинити відповідний процес без задоволення запиту (тобто не виконуючи план цього процесу) до тих пір, поки це безпечно.

нашому випадку ОС могла б призупинити процес B, якби вона «знала» про виникнення взаємоблокування, і тим самим виключити це взаємоблокування.

Графи ресурсів є інструментом, що дозволяє побачити, чи може задана послідовність дій стати причиною взаємоблокування. Для цього після кожного кроку граф перевіряється на наявність у ньому циклів.

Важливим питанням є *виявлення взаємних блокувань*. Це можливо за наявності або одного ресурсу кожного типу, або наявності декількох ресурсів кожного типу.

Виявлення взаємних блокувань при наявності одного ресурсу кожного типу. У такій ситуації необхідною і достатньою умовою взаємоблокування є наявність циклу в графі ресурсів.

Виявлення взаємних блокувань при наявності декількох ресурсів кожного типу. Нехай n – кількість процесів P_1, P_2,…,

P_m; m – кількість класів ресурсів R_1, R_2 ,…, R_m;

E_i – кількість одиниць ресурсу класу;

– вектор існуючих ресурсів, що містить загальну кількість наявних в системі одиниць кожного ресурсу;

A – вектор доступних ресурсів, елемент якого A_i є кількість одиниць ресурсу R_i, що доступні в даний момент;

C – матриця поточного розподілу, елемент якої C_{ij} є кількість одиниць ресурсу R_j використовуване процесом P_i;

– матриця запитів, елемент якої R_{ij} є кількість одиниць ресурсу R_j, яку запитує процес P_j.

Останні чотири структури показані на рисунку 2.10.

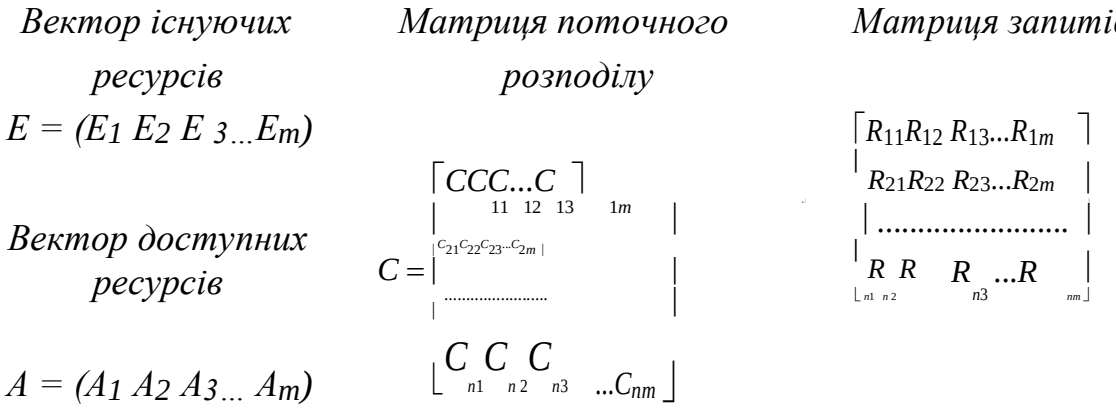

Рисунок 2.10 – Структури даних, що необхідні для алгоритму виявлення взаємних блокувань

Нехай у нас є алгоритм, який ставить на процесах відмітку, і вона є ознакою того, що ці процеси можуть закінчити роботу і, отже, не знаходяться у ситуації взаємного блокування. Тоді після завершення такого алгоритму будь-який немаркований процес знаходиться в тупіку.

Подібний алгоритм виявлення таких станів буде складатися з наступних кроків:

шукаємо немаркований процес P_i ,для якого *i*-й рядок матриці R поелементно не перевищує вектор A, тобто для всіх $1 \le k \le m$ виконується умова: $R_{ik} \le A_k$,

якщо такий процес знайдений, то додаємо *i*-й рядок матриці C до вектора A, маркуємо процес і повертаємося до кроку 1;

якщо таких процесів не існує, то робота алгоритму закінчується. Таким чином, на першому кроці цей алгоритм шукає процес, який може бути виконаний до кінця, тобто всі необхідні для нього ресурси доступні, виконує його і повертає зайняті таким процесом ресурси в загальний фонд ресурсів, а сам процес – маркує як завершений. Якщо ж виявиться, що всі процеси можуть працювати, то жоден з них в даний момент не заблокований.

Для ілюстрації роботи описаного алгоритму розглянемо рисунку 2.11.

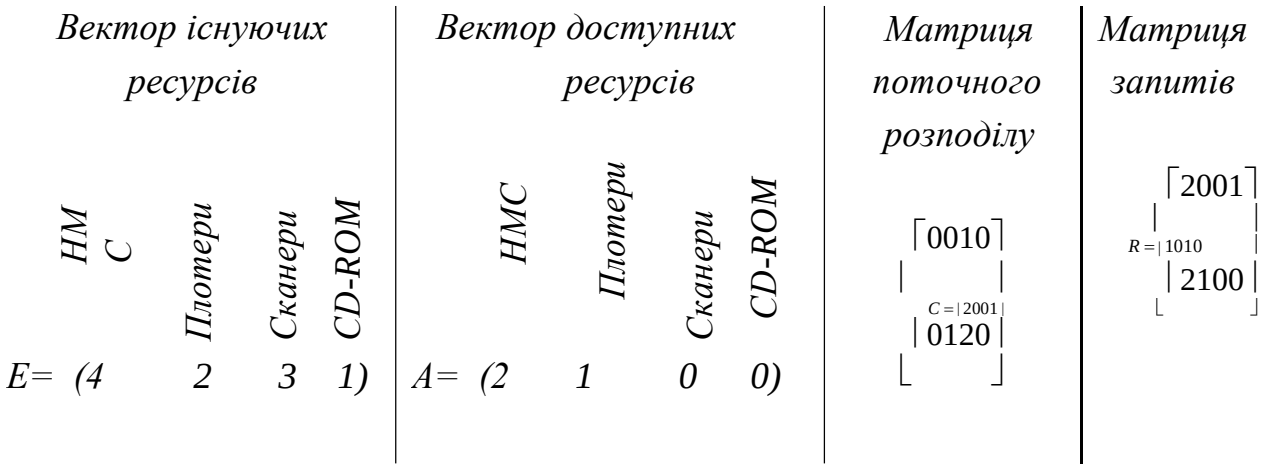

Рисунок 2.11 – Приклад використання алгоритму визначення взаємних блокувань

Тут є три процесу та чотири класи ресурсів (наприклад, накопичувачі на магнітній стрічці – НМС, плотери, сканери і пристрій для читання компакт-дисків). Процес 1 використовує один сканер. Процес 2 займає два НМС і пристрій для читання компакт-дисків. Процес 3 займає плотер і два сканера.

кожен з цих процесів потребує додаткового пристрою, як показує матриця R.

Працюючи з описаним вище алгоритмом виявлення взаємних блокувань, ми шукаємо процес, для якого запит ресурсів може бути виконаний. Вимоги першого процесу не можна виконати, тому що в системі немає доступного пристрою для читання компакт-дисків. Запити другого процесу також не можна задовольнити, так як немає вільних сканерів. Але третій процес може отримати все необхідне, тому він працює, завершується і повертає всі свої ресурси. Отже,

=(2220).

цього моменту може почати виконуватися процес 2, який по завершенні повертає свої ресурси в систему. Отримаємо: A = (4 2 2 1).

Тепер може працювати і процес, який залишився. Таким чином, у розглянутій системі не виникає взаємного блокування.

Далі змінимо варіант, що зображений на рис.2.11. Нехай процесу 3, окрім двох НМС і плотеру знадобиться також і пристрій для читання компакт-дисків. Тоді виникає ситуація, коли вже жоден із запитів не може бути виконаним. Тобто вся система опиниться заблокованою.

Для попередження взаємних блокувань необхідно визначати *безпечні* та *небезпечні стани* взаємодії процесів.

Стан безпечний, якщо він не перебуває в тупіку і якщо існує такий порядок планування, при якому кожен процес може виконатися до завершення, навіть якщо всі процеси захочуть отримати максимальну кількість необхідних їм ресурсів.

Небезпечний стан сам по собі ще не є блокованим. Однак тільки в безпечному стані система може гарантувати, що всі процеси успішно завершать свою роботу, а в небезпечному стані такої гарантії дати не можна.

Нідерландським вченим Е. Дейкстрою був розроблений **«алгоритм банкіра»,** який є розширенням алгоритму виявлення взаємних блокувань. Він перевіряє, чи веде виконання кожного з наявних запитів до небезпечного стану, і якщо так, то такий запит відхиляється.

«Алгоритм банкіра» для одного виду ресурсу. Щоб зрозуміти, чи є стан безпечним, перевіряється, чи можна надати достатньо ресурсів для завершення роботи процесу. Якщо так, то ці запити вважаються «погашеними», після чого перевіряється наступний, найближчий до межі запиту, процес. Якщо в результаті всі запити можуть бути виконані, то такий стан визнається безпечним, а початковий запит можна задовольнити (виконати).

На рисунок 2.12-а представлено чотири процеси, кожен з яких отримав певну кількість одиниць ресурсу. Усього є 10 одиниць ресурсу. В останній колонці міститься значення максимального запиту ресурсу кожним процесом.

Нехай в деякий момент виникає ситуація, що показана на рисунку 2.12-б. Цей стан є безпечним тому, що залишилися дві одиниці вільного ресурсу, і ОС може затримати всі запити, крім запитів процесу С, дозволяючи процесу С завершитися і повернути всі чотири одиниці ресурсу.

Розглянемо тепер, що могло б статися, якби в ситуації на рисунку 2.12-б був виконаний запит ще однієї одиниці ресурсу для процесу В. Ми потрапили б стан, що показаний на рисунку 2.12-в, який є небезпечним. Тоді, якщо б всі процеси запросили свою максимальну кількість одиниць ресурсу, то ОС не змогла б їх забезпечити, і ми потрапили б у глухий кут, тобто відбулося взаємне блокування.

Процес	Отримав	Мах
A	0	6
B	0	5
C	0	4
D	0	7

Вільно: 10

а)

Процес	Отримав	Мах
A	1	6
B	1	5
C	2	4
D	4	7

Вільно: 2

б)

Процес	Отримав	Мах
A	1	6
B	2	5
C	2	4
D	4	7

Вільно: 1

в)

Рисунок 2.12 – Три стани розподілу ресурсів:
а) – початкове, б) – безпечне, в) – небезпечне

«Алгоритм банкіра» для декількох видів ресурсів. Розглянутий вище «алгоритм банкіра» можна узагальнити і для управління системою з декількома видами ресурсів. Для цього:

Шукаємо в матриці R рядок, що відповідає процесу, у якого невиконані потреби ресурсів менше або дорівнюють вектору A. Якщо такого рядка не існує, то система в результаті потрапить в глухий кут, оскільки жоден процес не може пропрацювати до успішного завершення.

Припускаємо, що процес, рядок якого ми обрали в пункті 1, запросив всі необхідні ресурси і закінчив роботу. Відзначаємо цей процес як «завершений» і додаємо всі його ресурси до вектору A.

Повторюємо кроки 1 і 2 до тих пір, поки або всі процеси будуть помічені як «завершені», і стан в цьому випадку буде вважатися безпечним, або відбудеться взаємне блокування, і тоді стан буде вважатися небезпечним.

2.5 Класичні проблеми міжпроцесної взаємодії

Проблема виробника і споживача. «Проблема виробника і споживача», також відома як «проблема обмеженого буфера», полягає в наступному. Нехай два процеси спільно використовують буфер обмеженого розміру. Один з них («виробник»), поміщає дані в цей буфер, а другий, («споживач») – зчитує їх звідти.

Умови синхронізації:

«виробник» повинен чекати, якщо буфер повний;

«споживач» має чекати, якщо буфер порожній;

операції з буфером – це критичні секції, тобто працювати з буфером може тільки один процес.

Нижче наведено вирішення «проблеми виробника і споживача» з використанням семафорів:

```
   define N 100 / * кількість сегментів в буфері *
 / typedef int semaphore;
semaphore mutex = 1; / * контроль доступу в критичну секцію *
/ semaphore empty = N;
                    / * Кількість порожніх сегментів буфера *
/ semaphore full = 0;
                   / * Кількість повних сегментів буфера * /
void producer (void)
{
int item;
while (TRUE) {
item = produce__item ();
                  / * Створити дані, що поміщуються в буфер *
/ wait (& empty);
                 / * Зменшити лічильник порожніх сегментів буферу * /
wait (& mutex); / * вхід у критичну область * /
 insert___itern (item);
                 / * Помістити в буфер новий елемент *
/ signal (& mutex);
                / * Вихід з критичної області * /
signal (full);
               / * Збільшити лічильник повних сегментів буфера * /
       }
}
void consumer (void);
{
      int item;
while (TRUE) {
      wait (& full);
           / * Зменшити кількість повних сегментів буфера * /
    wait (& mutex); / * вхід у критичну область * /
    item = remove__item ();
                       / * Видалити елемент з буфера * /
signal (& mutex);
                         / * Вихід з критичної області * /
signal (& empty);
                      / * Збільшити лічильник порожніх сегментів буфера *
/ consume_item (item); / * обробка елементу * /
         }
}
```

Тут семафор *mutex* використовується для реалізації взаємного виключення, тобто для виключення одночасного звернення до буферу двох процесів. Решта семафорів використані для синхронізації: семафори *full* і *empty* необхідні, щоб гарантувати, що «виробник» припиняє роботу, коли буфер повний, а «споживач» припиняє роботу, коли буфер порожній.

Можна узагальнити цю задачу і на випадок *m* виробників і *n* споживачів.

Задача про «читачів» і «письменників». Це одна із задач синхронізації, що описує доступ процесів до поділюваної бази даних. Процеси при цьому розділяються на дві категорії: «читачі» – ніколи не модифікують базу даних, і «письменники» – можуть читати і модифікувати базу даних.

Правила синхронізації тут наступні:

«читачі» можуть працювати паралельно, якщо ніхто з «письменників» не модифікує базу даних;

модифікувати базу може тільки один «письменник», коли ніхто не читає інформацію з бази даних;

змінні стану може переглядати та змінювати тільки один процес.

Нижче наведена схема вирішення даної задачі:

«Читач»	**«Письменник»**
Чекати, поки закінчить роботу «письменник» (якщо хтось пише)	Чекати, поки закінчать роботу всі «читачі» (якщо хтось читає)
Читати з бази даних	Модифікувати базу даних
За необхідністю – «розбудити» «письменника», що очікує	За необхідністю – «розбудити» «читачів», що очікують

Задача про «філософів, що обідають». Ця задача моделює використання загальних ресурсів декількома процесами. Суть її така. П'ять філософів сидять за круглим столом. На столі – п'ять тарілок з макаронами і п'ять виделок, по одній між тарілками. Передбачається, що треба їсти двома виделками відразу. Поведінка кожного з філософів наступна: думає – хоче їсти (намагається взяти дві виделки) – їсть.

Обмеження:

кожен філософ повинен чекати, поки не звільняться дві виделки поруч з ним;

модифікація змінних стану повинна виконуватися в критичній секції.

Задача про «сплячого перукаря». Ця задача моделює обслуговування потоку запитів по черзі. Клієнти приходять у перукарню і встають в чергу. Перукар запрошує і обслуговує клієнтів по одному.

Обмеження:

клієнт повинен чекати, якщо є черга і / або перукар зайнятий обслуговуванням іншого клієнта;

перукар «спить», якщо немає клієнтів;

перший клієнт «будить» перукаря.