8.2 МОНІТОРИ

Семафори забезпечують досить потужний і гнучкий інструмент для здійснення взаємних виключень і координації процесів. Але по суті, **семафори** – це глобальні змінні, які розділяються, що є ознакою поганої структури програми. Тому створити коректно працюючу програму з використанням семафорів не завжди легко. Складність полягає в тому, що операції `wait` і `signal` можуть бути розкидані за усією програмою, і не завжди можна відстежити їх взаємодію на контрольованому ними семафорі. Немає контролю використання семафорів і з боку компілятора або операційної системи, відповідно – немає гарантій, що семафори будуть використані правильно.

Використовувати семафори треба дуже акуратно і обережно, оскільки одна незначна помилка (наприклад, як ми вже знаємо, проста перестановка двох операцій wait) може призвести до зупинки системи. Це нагадує програмування на асемблері, але, насправді, ще складніше.

При використанні семафорів дві проблеми – взаємне виключення і умовна синхронізація – програмуються однією і тією ж парою примітивів, проте це різні поняття і хотілося б мати різні механізми їх реалізації.

Для того щоб спростити і полегшити роботу програмістів при написанні коректних програм, було запропоновано більш високорівневий засіб синхронізації, що називається монітором.

Монітори – це програмні модулі мови програмування, які при тій же ефективності реалізації, що і семафори, забезпечують кращу структуризацію коду. Монітор інкапсулює:

- критичні дані (змінні), що розділяються;
- функції, що реалізовують операції над даними, які розділяються;
- синхронізацію виконання паралельних потоків, що викликають вказані функції.

Потік отримує доступ до змінних у моніторі тільки шляхом виклику процедур цього монітора. Код синхронізації додається компілятором. Два потоки не можуть одночасно виконуватися в одному моніторі, оскільки не можуть одночасно

виконуватися дві процедури монітора. Тобто, тільки один процес може бути активним стосовно монітора (рис. 8.2).

Компілятор обробляє виклики процедур монітора особливим чином. Коли процес викликає процедуру монітора, то перші декілька інструкцій цієї процедури перевіряють, чи не активний який-небудь інший процес стосовно цього монітора. Якщо так, то процес призупиняється, поки інший процес не звільнить монітор. Таким чином, виключення входу декількох процесів в монітор реалізується не програмістом, а компілятором, що робить помилки менш імовірними.

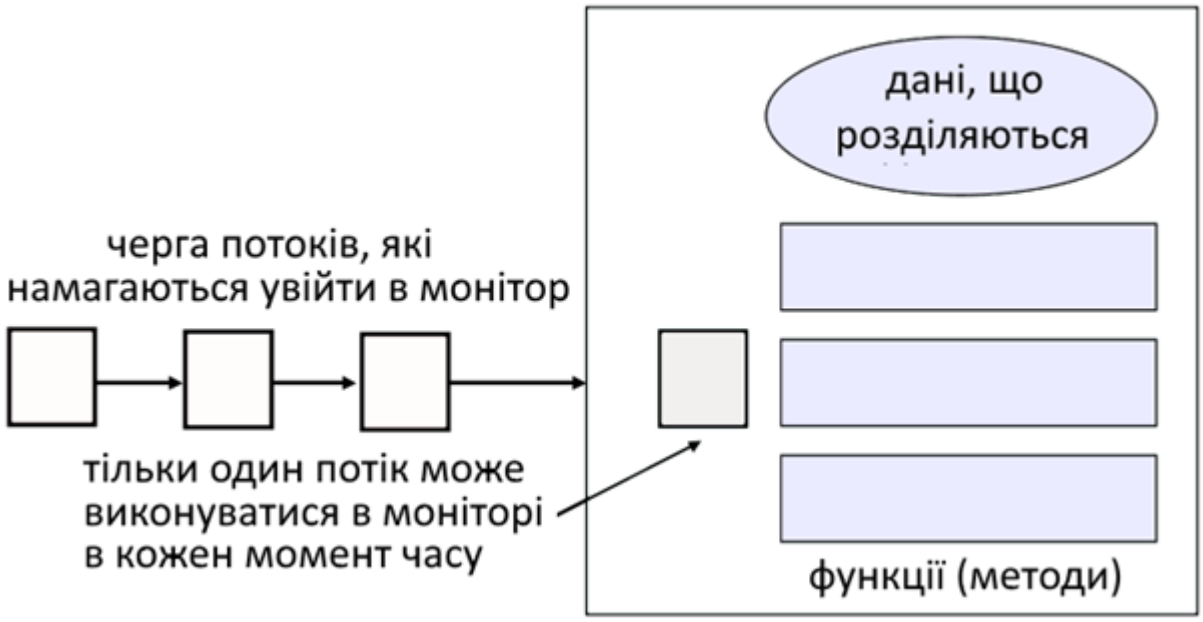

Рисунок 8.2 – Проста модель монітора

Уперше формальне визначення концепції монітора з деякими відмінностями було дане в 1974 році Хоаром (Hoare) і Бринч Хансеном (Brinch Hansen). Розглянемо монітор версії Хоара.

8.2.1 Монітори з сигналами (монітори Хоара)

Монітор являє собою програмний модуль, що складається з ініціюючих послідовностей, однієї або декількох процедур і локальних даних (умовних змінних) Хоча в різних мовах монітори оголошуються і створюються по-різному, будемо вважати монітор статичним об'єктом виду:

```
monitor Monname{
[оголошення постійних змінних]
[оператори ініціалізації]
[процедури]
}
```

Основні його характеристики:

1. Локальні змінні монітора доступні тільки його процедурам.
2. Процес входить в монітор шляхом виклику однієї з його процедур.
3. У моніторі в певний момент часу може виконуватися тільки один процес. Будь-який інший процес, що викликає монітор, буде призупинений в очікуванні доступності монітора.

Перші дві характеристики примушують нас згадати про об'єкти в ООП. Фактично об'єктно-орієнтовані мови програмування можуть легко організувати монітор як об'єкт із спеціальними характеристиками.

Дотримання умови виконання тільки одного процесу в певний момент часу дозволяє монітору забезпечити взаємовиключення. Дані монітора доступні в цей момент тільки одному процесу. Отже, захистити спільно використовувані структури даних можна, просто помістивши їх в монітор. Якщо дані в моніторі представляють деякий ресурс, то монітор забезпечує взаємовиключення при зверненні до ресурсу.

На абстрактному рівні можна описати структуру монітора таким чином:

```
monitor monitor_name {
     <опис внутрішніх змінних>; void m1(...){...
     }
     void m2(...){...
     }
     ...
     void mn(...){...
     }
     {
     < блок ініціалізації внутрішніх змінних>;
     }
}
```

Тут функції `m1,..., mn` є функціями-методами монітора, а блок ініціалізації внутрішніх змінних містить операції, які виконуються один і тільки один раз: при

створенні монітора або при найпершому виклику якої-небудь функції-методу до її виконання.

Для широкого застосування в паралельних обчисленнях монітори повинні включати елементи синхронізації. Припустимо, що процес, знаходячись в моніторі, має бути призупинений до виконання деякої умови. При цьому потрібно мати деякий механізм, який не лише призупиняє процес, але і звільняє монітор для інших процесів. Пізніше, коли умова виявиться виконаною, а монітор доступним, призупинений процес зможе продовжити свою роботу з того місця, де він був призупинений.

Монітор підтримує синхронізацію за допомогою змінних умови, розташованих (і доступних) тільки в моніторі. Працювати з цими змінними можуть дві функції:

1. cwait(c) – призупиняє виконання процесу по умові c.
2. csignal(c) – поновлює виконання деякого процесу, призупиненого викликом cwait(c) з тією ж умовою. Якщо є декілька таких процесів, вибирається один з них. Якщо таких процесів немає, функція не робить нічого.

Зверніть увагу на те, що операції wait/signal монітора відрізняються від відповідних операцій семафора. Якщо процес в моніторі передає сигнал, але при цьому немає жодного очікуючого його процесу, то сигнал втрачається.

Якщо один з процесів увійшов до монітора, то інші процеси, які намагаються увійти до монітора, приєднуються до черги процесів, призупинених в очікуванні доступності монітора. Якщо ж процес в моніторі тимчасово зупиняється, виконавши виклик cwait(x), то процес поміщається в чергу процесів, які очікують повторного входу в монітор при виконанні умови.

Якщо процес, що виконується в моніторі, виявить друге значення змінної умови x, то він виконує операцію csignal(x), яка повідомляє про виявлену зміну відповідної черги. Як приклад використання монітора повернемося до задачі

«Виробник-Споживач» з обмеженим буфером. У лістингу 8.7 показаний розв'язок цієї задачі з використанням монітора. Модуль монітора ProducerConsumer управляє буфером для зберігання і отримання символів. Монітор включає дві змінні умов: notfull істинно, якщо в буфері є місце хоч би для одного символу, а notempty істинно, якщо в буфері є хоч би один символ.

Лістинг 8.7 –. Розв'язок задачі «Виробник-Споживач» з обмеженим буфером з використанням монітора Хоара

```
Monitor ProducerConsumer;
Char buffer [N]     /* Місце для N елементів */
Int nextin, nextout;     /* Покажчиків буфера */
Int count;     /* Кількість елементів у буфері */
Int notfull, notempty;   /* Синхронізація */
Void Append (char x)
     { if (count == N)
     cwait(notfull);     /* Буфер заповнений */
     Buffer [nextin] = x;
     Nextin % N;
     Count++;  /* Додаємо елемент у буфер */
     Csignal(notempty);  /* Відновлення роботи споживача */
     }
Void Take (char x) {
     if (count == 0)
     cwait(notempty); /* Буфер порожній */
     x = Buffer [nextout];
     Nextout % N;
     Count--;  /* Видаляємо елемент з буфер */
     Csignal(notfull); /* Поновлюємо роботу виробників */
     }
{    /* Тіло монітора */
     nextin = 0; /* Спочатку буфер порожній */
     nextout = 0;
     count = 0;
}
Void Producer () {
     char x;
     While (true) {
          Produce (x);
          Append (x);
          }
     }
Void Consumer () {
     char x;
```

```
        While (true) {
              Take (x);
              Consume (x);
              }
        }
Void Main () {
Perbegin (Producer, Consumer);
}
```

Виробник може додавати символи в буфер тільки з монітора за допомогою процедури Append; прямого доступу до буфера у нього немає. Спочатку процедура перевіряє умову notfull, щоб з'ясувати, чи є в буфері порожнє місце. Якщо його немає, процес призупиняється, і до монітора може увійти інший процес (виробник або споживач). Пізніше, коли в буфері з'явиться вільне місце, призупинений процес витягається з черги і поновлює свою роботу. Після того, як процес помістить символ у буфер, він сигналізує про виконання умови notempty, що розблоковує процес споживача, якщо він був призупинений.

Цей приклад ілюструє розділення відповідальності при роботі з монітором і при використанні семафора. Монітор автоматично забезпечує взаємовиключення: одночасне звернення виробника і споживача до буфера неможливе.

Проте програміст повинен коректно розмістити всередині монітора примітиви Cwait і Csignal для того, щоб запобігти розміщенню елемента в заповненому буфері, або вибірці з порожнього буфера. У разі використання семафорів відповідальність як за синхронізацію, так і за взаємовиключення повністю лежить на програмістові.

Слід звернути увагу, що в лістингу 8.7 процес покидає монітор негайно після виконання функції Csignal. Якщо виклик Csignal здійснюється не в кінці процедури, то, за пропозицією Хоара, процес, що викликав цю функцію, призупиняється для того, щоб звільнити монітор для іншого процесу,поміщається в чергу призупинених процесів і залишається там до тих пір, поки монітор знову не звільнитися. Якщо виконання умови x не чекає жоден процес, виклик Csignal(x) не виконує ніяких дій.

Як при роботі з семафорами, так і при роботі з моніторами легко допустити помилку у функції синхронізації. Наприклад, якщо опустити будь-який з викликів Csignal у моніторі, то процес, що потрапив у відповідну чергу, залишиться там

назавжди. Перевага моніторів порівняно з семафорами в тому, що всі синхронізуючі функції знаходяться в моніторі. Таким чином, перевірити коректність синхронізації і відловити можливі помилки виявляється простіше, ніж при використанні семафорів. Крім того, при правильно розробленому моніторі доступ до захищених ресурсів коректний незалежно від запитуваного процесу. При використанні ж семафорів доступ до ресурсу коректний тільки в тому випадку, якщо правильно розроблені всі процеси, що звертаються до ресурсу.

8.2.2 Монітори із сповіщенням і широкомовленням

Визначення монітора, дане Хоаром, вимагає, щоб у разі, якщо черга очікування виконання умови не порожня, то при виконанні яким-небудь процесом операції Csignal для цієї умови був негайно запущений процес, що знаходиться у вказаній черзі. Таким чином, процес, що виконує операцію Csignal, повинен або негайно вийти з монітора, або бути призупиненим. У такого підходу є два недоліки:

1. Якщо, виконуючий операцію Csignal, процес не завершив своє перебування, то потрібно два додаткових перемикання процесів: один для призупинення цього процесу, а другий для відновлення його роботи, коли монітор стане доступний.

2. Планувальник процесів, пов'язаний з сигналом, має бути ідеально надійний. При виконанні Csignal процес з відповідної черги має бути негайно активізований, причому планувальник повинен гарантувати, що до активізації ніякий інший процес не увійде до монітора. Інакше умова, з якою активізується процес, може встигнути змінитися. Так, наприклад, в лістингу 8.7, коли виконується Csignal(notempty), процес з черги notempty має бути активізований до того, як новий споживач увійде до монітора. Якщо станеться збій процесу виробника безпосередньо після того, як він додасть символ, так що операція Csignal не буде виконана, то в результаті процеси в черзі notempty виявляться навіки заблоковані.

Лемпсон (Lampson) і **Ределл** (Redell) розробили інше визначення монітора для мови програмування Mesa, Їх підхід дозволяє долати описані проблеми, а крім того, надає ряд корисних розширень концепції моніторів. Структура монітора Mesa використана і в мові програмування Modula-3 [12].

У мові програмування Mesa примітив Csignal був замінений примітивом Cnotify, який інтерпретується таким чином. Коли процес, що виконується в моніторі, викликає Cnotify(x), про це повідомляється черга умови x, але виконання процесу, що викликав Cnotify, триває. Результат повідомлення полягає в тому, що процес на початку черги умови відновить свою роботу в найближчому майбутньому, коли монітор виявиться вільним. Проте оскільки немає гарантії, що деякий інший процес не ввійде до монітора до згаданого очікуючого процесу, при відновленні роботи наш процес повинен ще раз перевірити, чи виконана умова. У разі використання такого підходу процедури монітора ProducerConsumer матимуть такий вигляд (лістинг 8.8).

Лістинг 8.8 – Код монітора ProducerConsumer

```
Void Append(char x) {
     While(count == N)
     cwait(notfull);     /* Буфер заповнений */
     Buffer [nextin] = x;
     Nextin % N;
     Count++; /* Додаємо елемент у буфер */
     Cnotify(notempty); /* Повідомляємо споживача */
}
Void Take(char x) {
     While(count == 0)
     cwait(notempty); /* Буфер порожній */
     x = Buffer [nextout];
     Nextout % N;
     Count--;  /* Видаляємо елемент з буферу */
     Cnotify(notfull);   /* Повідомляємо виробника */
}
```

Інструкції if замінені циклами while. Таким чином, виконуватиметься як мінімум одне зайве обчислення змінної умови. Проте в цьому випадку відсутні зайві перемикання процесів, і немає обмежень на момент запуску очікуючого процесу після виклику Cnotify.

Однією з корисних особливостей такого роду моніторів може бути пов'язаний з кожним примітивом умови Cnotify граничний час очікування. Процес, який прочекав повідомлення впродовж граничного часу, поміщається в список активних незалежно

від того, було повідомлення про виконання умови чи ні. Така можливість запобігає нескінченному голодуванню процесу в разі збою інших процесів перед повідомленням про виконання умови.

При використанні правила, згідно з яким відбувається повідомлення процесу, а не його насильницька активація, в систему команд можна включити примітив (наприклад, cbroadcast – передача), який викликає активацію всіх очікуючих процесів. Це може бути в ситуаціях, коли процес не обізнаний про кількість очікуючих процесів.

Припустимо, наприклад, що в програмі «виробник-споживач» функції Append і Take можуть працювати з символьними блоками змінної довжини. У цьому випадку, коли виробник додає в буфер блок символів, він не зобов'язаний знати, скільки символів готовий спожити кожен з очікуючих процесів. Він просто виконує інструкцію cbroadcast, і усі очікуючі процеси отримають повідомлення про те, що вони можуть спробувати отримати свою частку символів з буфера (широкомовне повідомлення).

Крім того, широкомовне повідомлення може використовуватися в тому випадку, коли процес не в змозі точно визначити, який саме процес з очікуючих має бути активований. Хорошим прикладом такої ситуації може служити диспетчер пам'яті. Припустимо, що у нас є j байт вільної пам'яті, і деякий процес звільняє додатково k байт. Диспетчерові не відомо, який саме з очікуючих процесів зможе працювати з j+k байт вільної пам'яті; отже, він може використати виклик cbroadcast, і всі очікуючі процеси самі перевірять, чи достатньо їм вільної пам'яті.

Перевага монітора Лемпсона-Ределла порівняно з монітором Хоара є його менша схильність до помилок. Оскільки кожна процедура після отримання сигналу перевіряє змінну монітора з використанням циклу While, то навіть при передачі процесом невірного або широкомовного повідомлення – це не призведе до помилки в програмі, що отримала сигнал. Просто переконавшись, що її даремно активізували, програма знову перейде в стан очікування.

Наведемо приклад програми «Виробник-Споживач» на мові програмування Java (лістинг 8.9). Java – об'єктно-орієнтована мова програмування, що підтримує потоки на рівні користувача, і що дозволяє групувати методи (процедури) в класи.

Додавання в опис методу ключового слова synchronized гарантує, що якщо хоч би один потік почав виконання цього методу, жоден інший потік не зможе виконати інший синхронізований (визначений як synchronized) метод з цього класу.

Лістинг 8.9 – Розв'язок задачі «Виробник-Споживач» на Java [12]

```
public class ProducerConsumer {
     static final int N = 100;     // Розмір буфера
     //створити екземпляр потоку виробника
     static producer p = new producer();
     // створити екземпляр потоку споживача
     static consumer c = new consumer();
     //створити екземпляр монітора
     static our_monitor mon = new our_monitor();
     public static void main(String args[]) {
          p.start();     // запуск потоку виробника
          c.start();     // запуск потоку споживача
          }
     static class producer extends Thread {
     public void run(){  // метод run містить програму потоку
     int item;
     while(true) {  // цикл виробника
          item = produce_item(); mon.insert(item);
          }
        }
     private int produce_item(){.} // виробництво
     }
     static class consumer extends Thread {
     public void run(){ // метод run містить програму потоку
     int item;
     while(true) {  // цикл споживача
          item = mon.remove(); consume_item(item);
          }
     }
     private void consume_item(int item) }
     // споживання
     }
```

```
    static class our_monitor {    // монітор
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0;  // лічильник і індекси
    public synchronized void insert(int val) {
    if(count == N) go_to_sleep(); //якщо буфер повний - чекати
    buffer[hi] = val;   // помістити елемент у буфер
    hi % N; // наступний сегмент для елементу
    count = count+1; // лічильник елементів у буфері
    if(count == 1) notify(); // якщо споживач в стані
    // очікування, то активувати його
    }
    public synchronized int remove() {
    int val;
    if(count == 0) go_to_sleep(); //якщо буфер порожній - чекати
    val = buffer[lo] ; // забрати елемент з буфера
    lo % N; // наступний сегмент для витягання
    count = count - 1; //тепер у буфері на 1 елемент менше
    if(count == N - 1) notify(); //якщо виробник в стані
    // очікування, то активувати його
    return val;
         }
         private void go_to_sleep() { try { wait(); }
         catch(interruptedException exc);
         }
    }
  }
```

У програмі виробника є нескінченний цикл формування даних і розміщення їх у загальному буфері. У коді споживача є нескінченний цикл з вилученням даних із загального буфера. Клас `our_monitor` містить буфер, змінні адміністрування і два методи синхронізації. Коли виробник активний в процедурі `insert`, споживач не може бути активний в процедурі `remove`, що виключає стан змагання. Змінна cont містить кількість елементів у буфері, набуваючи значень від 0 до N-1. Змінна lo є індексом наступного буфера, з якого слід витягнути дані. Змінна hi є індексом наступного буфера, в який слід помістити дані. Розв'зана ситуація, коли lo = hi, що

означає 0 або N елементів у буфері. Відрізняти ці два випадки можна за змінною count.

Синхронізовані методи в мові Java відрізняються від стандартних моніторів відсутністю змінних стану. Натомість пропонується дві процедури wait і notify, які використовуються в синхронізованих методах. Процедура може бути перервана, для чого і служить увесь набір програм, що оточує її. У нашому випадку go_to_sleep описує відхід в стан очікування. Останній приклад написаний на Java, а не на C, як усі інші приклади, оскільки C і багато інших мов не мають моніторів (окрім Modula-3).