ДВНЗ «Ужгородський національний університет»
Факультет інформаційних технологій
Кафедра інформаційних управляючих систем та технологій

В. М. Коцовський

Теорія паралельних обчислень

Конспект лекцій

Частина II

Ужгород – 2019

ЗМІСТ

6. ОСНОВИ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ

6.1. Основні поняття паралельного програмування

Паралельна програма містить кілька процесів, які працюють паралельно над виконанням деякої задачі [10]. Кожний процес — це послідовна програма, тобто послідовність операторів, які виконуються один за одним. Послідовна програма має *один потік керування*, паралельна — *декілька*.

Сумісна робота процесів паралельної програми здійснюється за допомогою їх *взаємодії* (communication). Взаємодія реалізується за рахунок використання *спільних змінних* (shared variables) або *передачі повідомлень*. У першому випадку один процес може змінювати значення змінних, які використовуються у інших процесах. У другому випадку процес надсилає повідомлення, яке отримують інші процеси.

При будь-якій взаємодії між процесами необхідною є взаємна *синхронізація* (synchronization). Існує два основних види синхронізації — взаємне виключення (mutual exclusion) та умовна синхронізація (condition synchronization). *Взаємне виключення* забезпечує, щоб критичні секції операторів не виконувалися одночасно. *Умовна синхронізація* затримує процес до тих пір, поки не виконається певна умова. Наприклад, взаємодія процесів виробника (producer) та споживача (consumer) часто забезпечується з використанням буфера у спільній пам'яті. Виробник записує у буфер, споживач читає з нього. Для уникнення одночасного використання буфера виробником і споживачем (що може призвести до зчитування не повністю записаного повідомлення) використовується взаємне виключення. Умовна синхронізація використовується для перевірки того, чи було зчитано споживачем останнє записане у буфер повідомлення.

6.2. Нотація паралельних операторів та процесів

За замовчуванням оператори виконуються послідовно, тобто один за другим. Оператор `co` (concurrent — паралельний або такий, що відбувається одночасно) вказує на те, що кілька операторів можуть виконуватися паралельно. Розглядають дві форми оператора `co`. В першій формі оператор `co` має кілька гілок (arms):

```
co оператор1,
// ...
// операторn,
```

```
oc
```

Кожна гілка містить оператор (або список операторів). Гілки відокремлюються *символом паралелізму* "//". Оператор, наведений вище, означає наступне:

почати паралельне виконання усіх операторів та очікувати їх завершення.

Оператор co, таким чином, завершується після виконання усіх операторів, які містяться у його гілках.

У другій формі оператор co використовує один або кілька квантифікаторів, які вказують на те, що набір операторів має виконуватися паралельно для кожної комбінації значень параметрів циклу. Наприклад, наступний тривіальний оператор заповнює масиви a та b нулями:

```
co[i = 0 to n-1] {
    a[i] = 0;
    b[i] = 0;
}
```

Цей оператор створює *n* процесів, по одному для кожного значення змінної i. Область видимості лічильника циклу — опис процесу, і у кожного процесу своє, відмінне від інших, значення змінної i. Дві форми оператора co можна поєднувати. Наприклад, одна гілка може мати квантифікатор в квадратних дужках, а друга — ні.

Декларація *процесу* є, по суті, скороченою формою оператора co з одною гілкою. Вона починається з ключового слова process та назви процесу. Тіло процесу містить визначення локальних змінних та список операторів.

В наступному простому прикладі визначається процес foo, який підсумовує числа від 1 до 10, записуючи результат в глобальну зміну x.

```
process foo{
    int sum = 0;
    for [i = 1 to 10]
        sum += i;
    x = sum;
}
```

Опис process записується на тому самому рівні програми, що й опис функцій та процедур, тобто процес не є оператором, на відміну від co. Крім того, оголошені процеси виконується у фоновому режимі, тоді як виконання оператора, який йде за опера-

тором co, починається після завершення усіх операторів, які містяться у гілках розпаралелювання co. Декларації процесів, на відміну від оператору co, не можуть бути вкладені у інші декларації або оператори.

Ще один приклад: запис значень від 1 до n у стандартний потік виведення:

```
process bar1 {
    for [i = 1 to n]
        write(i); # або "printf("%d\n",i);"
}
```

Масив процесів оголошується шляхом додавання квантифікатора (в квадратних дужках) до назви процесу:

```
process bar2[i = 1 to n] {
    write(i);
}
```

bar1 та bar2 записують в стандартний вивід значення від 1 до n. Проте порядок, у якому їх записує масив процесів bar2, є *недетермінованим*, оскільки масив bar2 складається з n окремих процесів, які виконується у довільному порядку. Існує n! різних можливих порядків виводу чисел для цього масиву процесів.

Для запису однорядкових коментарів використовується символ "#" (використання традиційного коментаря "//" є неможливим, оскільки "//" використовується для відокремлення гілок оператору co).

6.3. Парадигми паралельного програмування

Не зважаючи на існування великої кількості паралельних програмних комплексів, в них використовується лише невелика кількість патернів або парадигм паралельного програмування. В роботі [10] виокремлено п’ять основних парадигм:

1) ітеративний паралелізм;

2) рекурсивний паралелізм;

3) «виробники та споживачі» (конвеєри);

4) «клієнти та сервери»;

5) «взаємодіючі рівні» (interacting peers).

З використанням однієї або кількох із цих парадигм і програмуються додатки.

Ітеративний паралелізм використовується, коли у програмі є кілька процесів (часто ідентичних), кожний із яких містить один або кілька циклів. Таким чином, кожний процес є ітеративною програмою. Процеси програми працюють сумісно над однією задачею; вони взаємодіють та синхронізуються з допомогою спільних змінних або передачі повідомлень. Ітеративний паралелізм частіше за все зустрічається в наукових обчисленнях, які виконуються на кількох процесорах.

Рекурсивний паралелізм може використовуватися у випадку наявності у програмі однієї або кількох рекурсивних процедур, виклики яких незалежні, тобто кожний із них опрацьовує свою частини загальних даних. Рекурсія часто застосовується у імперативних мовах програмування, особливо при реалізації алгоритмів типу "поділяй та пануй" або "перебір з поверненням" (backtracking). Рекурсія є однією з фундаментальних парадигм також і в символьних, логічних, та функціональних мовах програмування. Рекурсивний паралелізм використовується для розв’язування таких комбінаторних проблем, як сортування, планування (задача комівояжера) та ігри (шахи та інші).

Виробники та споживачі — це взаємодіючі процеси. Вони часто організовані у конвеєр, через який проходить інформація. Кожний процес конвеєра є *фільтром*, який споживає вихідні дані свого попередника та продукує вхідні дані для свого користувача. Фільтри зустрічаються на рівні додатків в операційних системах типу ОС Unix, всередині самих операційних систем, прикладних програм, якщо один процес виробляє вихідні дані, які споживає (читає) інший процес.

Клієнти та сервери — найбільш поширена модель взаємодії в розподілених системах, від локальних мереж до World Wide Web. Клієнтський процес робить запит до сервісу та очікує відповіді. Сервер очікує запити від клієнтів, а потім діє відповідно до цих запитів. Сервер може бути реалізований у вигляді одиночного процесу, який не може обробляти одночасно кілька запитів клієнтів, або (при необхідності паралельного обслуговування запитів) як багатопотокова програма. Клієнти та сервери — паралельне програмне узагальнення процедур та їх викликів: сервер виконує роль процедури, а клієнт її викликає. Але якщо код клієнта та код сервера розташовані на різних машинах, звичайний механізм виклику процедур недоступний. Замість цього необхідно використовувати віддалений виклик процедури або *рандеву*.

Взаємодіючі рівні — остання парадигма взаємодії. Вона зустрічається в розподілених програмах, в яких кілька процесів для розв'язування задачі виконують один і той самий код та обмінюються повідомленнями. Взаємодіючі рівні використовуються для реалізації розподілених паралельних програм, особливо при ітеративному паралелізмі та децентралізованому прийняті рішень в розподілених системах.

6.3.1. Ітеративний паралелізм: множення матриць

Ітеративна послідовна програма використовує для обробки даних та обчислення результатів цикли типу `for` та `while`. Ітеративна паралельна програма містить кілька ітеративних процесів. Кожний процес обчислює результати для підмножини даних, а потім ці результати збираються разом.

В якості прикладу розглянемо задачу із галузі наукових обчислень. Припустимо, що задані дві $n \times n$-матриці `a` та `b`. Мета — обчислити $c = a \cdot b$.

Матриці `a, b, c` є спільними глобальними змінними, індекси рядків та стовпців змінюються від `0` до `n-1`, елементи матриці є дійсними числами з плаваючою крапкою.

Добуток матриць можна знайти з використанням послідовної програми, яка має наступний псевдокод:

```
for [i = 0 to n-1] {
    for [j = 0 to n-1] {
        c[i,j] = 0;
        for [к = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

Множення матриць — приклад *задачі з масовим паралелізмом*, оскільки програма містить велику кількість операцій, які можуть виконуватися паралельно. Дві операції можуть виконуватися паралельно, якщо вони незалежні. Припустимо, що *множина зчитування* операції містить змінні, які вона читає, але не змінює, а *множина запису* — змінні, які вона змінює (і, можливо, зчитує). Дві операції є *незалежними*, якщо їх множини запису не перетинаються. Неформально кажучи, процеси завжди можуть безпечно читати змінні, які не змінюються. Однак двом процесам взагалі кажучи небезпечно змінювати значення однієї і тієї самої змінної або одному процесу зчитувати змінну, яка записується іншим процесом.

При обчислені добутку матриць обчислення проміжкових добутків є незалежними операціями. У рядках 3–5 попередньої програми виконується ініціалізація та обчислення елемента матриці `c`. Внутрішній цикл програми зчитує рядок матриці `a` та стовпець матриці `b`, а потім зчитує та записує один елемент матриці `c`. Множина зчитування для внутрішнього добутку — це рядок матриці `a` та стовпець матриці `b`, множина запису — елемент матриці `c`.

Оскільки множини запису внутрішніх добутків не перетинаються, їх можна виконувати паралельно. Можливими є варіанти, коли паралельно обчислюються результуючі рядки, результуючі стовпці чи групи рядків та стовпців.

Для початку розглянемо паралельне обчислення рядків матриці `c`.

```
co [i = 0 to n-1] { # паралельне обчислення рядків
    for [j = 0 to n-1] {
        c[i,j] = 0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

Ця програма відрізняється від послідовного варіанта лише тим, що у зовнішньому циклі оператор `for` замінений оператором `co`. Але семантична різниця велика: оператор `co` вказує, що його тіло для кожного значення індексу `i` буде виконуватися паралельно (принаймні теоретично, в залежності від кількості наявних процесорів).

Другий спосіб паралельного множення матриць — паралельне обчислення стовпців матриці `c`:

```
co [j = 0 to n-1] { # паралельне обчислення стовпців
    for [i = 0 to n-1] {
        c[i,j] = 0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

У цій версії два зовнішні цикли (за i та за j) помінялися місцями (якщо тіла двох циклів незалежні, то їх можна безпечно міняти місцями).

Дві отримані версії програми по суті мають ідентичні обчислювальні властивості. Проте у випадку множення $m \times n$ та $n \times p$ прямокутних матриць числа m та p можуть

сильно відрізнятися, а тому з огляду на параметри обчислювальної системи обчислення з паралелізмом за рядками у випадку $m < p$ можуть виявитися значно ефективнішими, ніж обчислення з паралелізмом за стовпцями.

Програму для паралельного обчислення усіх проміжних добутків можна кількома способами. Можна використати один оператор `co` для двох індексів.

```
co [i = 0 to n-1, j = 0 to n-1] { # усі рядки та усі стовпці
    c[i,j] = 0;
    for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
}
```

Тіло оператора `co` виконується паралельно для кожної комбінації значень індексів `i` та `j`, тому програма визначає n^2 процесів. (Чи будуть вони всі виконуватися паралельно, залежить від конкретної реалізації). Другий спосіб паралельного обчислення проміжних добутків полягає у використанні вкладених операторів `co`.

```
co [i = 0 to n-1] { # рядки паралельно, потім
    co [j = 0 to n-1] { # стовпці паралельно
        c[i,j] = 0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

Для кожного рядка (зовнішній оператор `co`) і потім для кожного стовпця (внутрішній оператор `co`) задається по одному процесу. Третій спосіб написати цю програму — поміняти місцями два рядки останньої програми. Результат усіх трьох програм однаковий: виконання внутрішнього циклу для усі n^2 комбінацій значень `i` та `j`. Різниця між ними — у визначенні процесів, а отже, і у часі їх створення.

Слід зазначити, що усі попередні паралельні програми були отримані заміною оператора `for` на `co`, причому це було зроблено тільки для індексів `i` та `j`. Виникає питання, як бути з внутрішнім циклом за індексом `k`? Чи можна цей оператор замінить оператором `co`? Відповідь — «ні», оскільки тіло внутрішнього циклу як зчитує, так і змінює значення змінної `c[i,j]`. Можна обчислити суму у циклі `for` за змінною `k`, використовуючи каскадну схему здвоєння, але як зазначено в [10], це не може забезпечити суттєвого прискорення для більшості машин.

Інший спосіб розпаралелити обчислення — використати ключове слово `process` замість оператора `co` (тобто використати декларацію процесу). По суті, `process` — це оператор `co`, який виконується у фоновому режимі. Наприклад, перша паралельна програма з наведених вище (з паралелізмом по рядкам результату) може бути записана наступним чином:

```
process row[i = 0 to n-1] { # рядки паралельно
       for [j = 0 to n-1] {
       c[i,j] = 0;
       for [k = 0 to n-1]
          c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

В програмі визначений масив процесів — `row[1], row[2]` і т.д. — по одному для кожного значення індексу `i`. Ці *n* процесів створюються і починають виконуватися, коли зустрічається рядок декларації процесу. Якщо за декларацією процесу йдуть оператори, то вони виконуються паралельно з процесом, тоді як оператори, записані після оператора `co`, не виконуються до його завершення.

В програмах, наведених вище, для кожного елементу, рядка або стовпця результуючої матриці використано по одному процесу. Припустимо, що число процесорів в системі менше за `n` (так зазвичай і буває, коли `n` велике). У цьому випадку є очевидний спосіб повного використання усіх процесорів: поділити матрицю на смуги (рядків чи стовпців) і для кожної смуги створити робочий процес, який обчислює результати для елементів своєї смуги. Припустимо, що є `P` процесорів і `n` кратне `P`. Тоді у випадку смуг рядків робочі процеси можна запрограмувати так:

```
process worker[w = 1 to P] {
    int first = (w-1) * n/P;     # перший рядок смуги
    int last = first + n/P - 1; # останній рядок смуги
    for [i = first to last] {
       for [j = 0 to n-1] {
          c[i,j] = 0;
          for [k = 0 to n-1]
             c[i,j] = c[i,j] + a[i,k]*b[k,j];
       }
    }
}
```

В програму додані оператори, необхідні для визначення першого та останнього рядка кожної смуги. Потім рядки смуги вказуються у циклі (за індексом `i`) для обчислення елементів матриці `c`.

Таким чином, суттєвою умовою розпаралелювання програм є наявність незалежних обчислень, тобто обчислень з множинами запису, що не перетинаються. Для добутку матриць незалежними обчисленнями є скалярні добутки рядків на стовпці, оскільки кожний із них записує (та читає) свій елемент `c[i,j]`. Тому можна паралельно обчислювати усі скалярні добутки, рядки, стовпці або смуги з використанням оператора `co` або декларації процесу.

6.3.2. Рекурсивний паралелізм: адаптивна квадратура

Програма є рекурсивною, якщо вона містить процедури, які викликають самі себе — прямо або опосередковано. Рекурсія дуальна до ітерації, тобто рекурсивні програми можна перетворити у ітеративні і навпаки. У тілі багатьох рекурсивних процедур звертання до самої себе зустрічається більше одного разу. Наприклад, алгоритм `quicksort` розбиває масив на дві частини, а потім двічі викликає себе для окремого сортування лівої та правої частин. Значна кількість алгоритмів обробки дерев та графів мають подібну структуру.

Рекурсивну програму можна реалізувати за допомогою паралелізму, якщо вона містить кілька незалежних рекурсивних викликів. Два виклики процедури (або функції) є незалежними, якщо їх множини запису не перетинаються. Ця умова виконується, якщо:

1) процедура не звертається до глобальних змінних або тільки читає їх;

2) аргументи и результуючі змінні процедур — різні змінні.

Наприклад, якщо процедура не звертається до глобальних змінних і має тільки параметри-значення (за механізмом їх передачі), то будь-який її виклик буде незалежним (процедура читає та змінює тільки локальні змінні, і кожний екземпляр процедури має свою локальну копію змінних).

Розглянемо *задачу квадратури*, яка полягає у наближеному обчисленні інтеграла неперервної функції. Припустимо, що це функція $f(x)$. Як показано на рис. 6.1, інтеграл функції $f(x)$ від a до b — це площа фігури, обмеженої графіком $f(x)$, віссю абсцис та прямими $x = a$ і $x = b$.

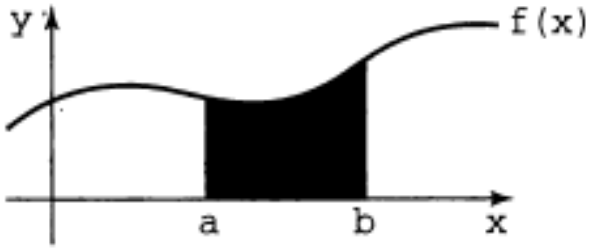

Рис. 6.1. Задача квадратури

Існує два основних способи апроксимації значення інтеграла. Перший — розбити інтервал від *a* до *b* на фіксоване число відрізків, а потім апроксимувати площу на кожному з них за правилом трапецій або за правилом Сімпсона.

```
double fleft = f(a), fright, area = 0.
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width]{
    fright = f(x);
    area = area + (fleft + fright) * width / 2;
    fleft = fright;
}
```

Кожна ітерація обчислює площа малої фігури за правилом трапецій і додає її до загального значення площі. Змінна `width` — ширина кожної трапеції, відрізки перебираються зліва направо, тому праве значення кожної ітерації стає лівим значенням наступної ітерації.

Другий спосіб апроксимації інтеграла — використовувати парадигму «поділяй і пануй» і змінне число інтервалів. Зокрема, спочатку обчислюють значення *m* — середину відрізка[*a*, *b*]. Потім апроксимують площу трьох областей під кривою *f*(*x*): від а до *m*, від *m* до *b* та від *a* до *b*. Якщо сума менших площ дорівнює більшій площі з деякою заданою точністю ε, то апроксимацію можна вважати достатньою [10]. Якщо ні, то задача ділиться на дві підзадачі: від *a* до *m* та від *m* до *b*, і процес повторюється. Цей спосіб називається *адаптивною квадратурою*, оскільки алгоритм «адаптується» до форми кривої. Його можна запрограмувати так.

```
double quad(double left,right,fleft,fright,lrarea){
    double mid = (left + right) / 2;
    double fmid = f(mid);
    double larea = (fleft+fmid) * (mid-left) / 2;
```

```
        double rarea = (fmid+fright) * (right-mid) / 2;
        if(abs((larea+rarea) - lrarea) > EPSILON){
            larea = quad(left, mid, fleft, fmid, larea);
            rarea = quad(mid, right, fmid, fright, rarea);
        }
        return (larea + rarea);
    }
```

Інтеграл функції $f(x)$ від a до b апроксимується таким викликом функції:

```
area = quad (a, b, f (a), f (b), (f (a)+f (b)) * (b-a) / 2);
```

У функції знову використовується правило трапеції. Значення функції f у крайніх точках відрізка і наближена площа цього інтервалу передаються в кожен виклик функції `quad`, щоб не обчислювати їх більше одного разу.

Ітеративну програму не можна розпаралелити (у наведеній формі), оскільки тіло циклу і зчитує, і записує значення змінної `area`. Проте в рекурсивної програмі виклики функції `quad` незалежні за умови, що обчислення функції $f(x)$ не дає побічних ефектів. Зокрема, аргументи функції `quad` передаються за значенням, і в її тілі немає присвоювання глобальних змінних. Таким чином, для завдання паралельного виконання рекурсивних викликів функції можна використовувати оператор `co`.

```
co larea = quad (left, mid, fleft, fmid, larea);
// rarea = quad (mid, right, fmid, fright, rarea);
oc
```

Це єдина зміна, необхідне для того, щоб зробити цю програму паралельною. Оскільки оператор `co` не закінчується до тих пір, поки не будуть завершені обидва виклику функцій, значення змінних `larea` і `rarea` обчислюються до того, як функція `quad` поверне їх суму.

Отже, програму з декількома рекурсивними викликами функцій можна легко перетворити в паралельну рекурсивну програму, якщо виклики незалежні. Проте існує чисто практична проблема: паралельно виконуваних операцій може стати занадто багато. Кожен оператор `co` в наведеній вище програмі створює два процеси, по одному для кожного виклику функції. Якщо глибина рекурсії велика, то виникне занадто велика для паралельного виконання кількість процесів. Вирішення цієї проблеми полягає *в скороченні, або відтинанні*, дерева рекурсії при занадто великій кількості процесів, тобто перехід з паралельних рекурсивних викликів на послідовні.

6.3.3. Виробники і споживачі: канали ОС Unix

Процес-виробник виконує обчислення і виводить потік результатів. Процес-споживач вводить і аналізує потік значень. Багато програм в тій чи іншій формі є виробниками та / або споживачами. Поєднання стає особливо цікавим, якщо виробники і споживачі об'єднані в конвеєр — послідовність процесів, в якій кожен з них споживає дані виходу попередника і виробляє дані для подальшого процесу. Класичним прикладом є конвеєри в операційній системі Unix. Однією з найбільш потужних функцій, запропонованих в ОС Unix, була можливість прив’язки стандартних "пристроїв" введення-виведення до різних типів файлів. Зокрема, файли `stdin` або `stdout` можуть бути пов’язані з файлом даних або з "файлом" особливого типу, який називається каналом.

Канал — це буфер (черга типу FIFO) між процесом-виробником і процесом-споживачем, який містить зв’язану послідовність символів, до якої виробник може дописувати нові символи. Символи видаляються, коли процес-споживач зчитує їх з каналу. Процес-виробник очікує (при необхідності), поки в буфері з’явиться вільне місце, потім додає в кінець буфера новий рядок. Процес-споживач очікує (при необхідності), поки в буфері не з’явиться рядок даних, потім зчитує його з буфера. Такі буфери реалізовують з використанням спільних змінних і різних примітивів синхронізації (прапорців, семафорів та моніторів).

6.3.4. Клієнти і сервери: файлові системи

Між виробником і споживачем існує односпрямований потік інформації. Цей вид взаємодії між процесами часто зустрічається в паралельних програмах і не має аналогів в послідовних, оскільки в послідовній програмі тільки один потік управління, тоді як виробники і споживачі — незалежні процеси з власними потоками управління і власними швидкостями виконання.

Ще однією типовою схемою в паралельних програмах є взаємозв’язок типу клієнт-сервер. Процес-клієнт запитує сервіс, потім очікує обробки запиту. Процес-сервер багаторазово очікує запит, обробляє його, потім посилає відповідь. Як показано на рис. 6.2, існує двонапрямлений потік інформації: від клієнта до сервера і назад.

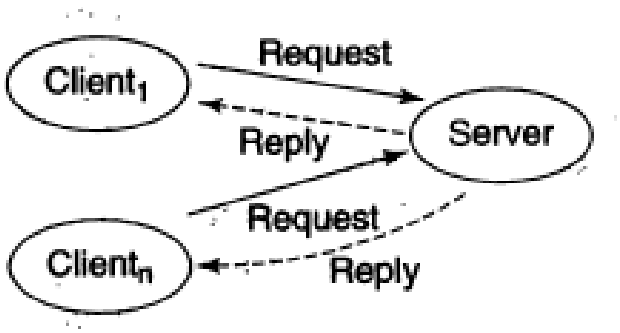

Рис. 6.2. Клієнти та сервери

Зв’язки між клієнтом і сервером в паралельному програмуванні аналогічні відносинам між програмою, що викликає підпрограму, і самої підпрограмою в послідовному програмуванні. Більш того, як підпрограма може бути викликана з кількох місць програми, так і у сервера зазвичай є багато клієнтів. Запити кожного клієнта повинні оброблятися незалежно, проте паралельно може оброблятися декілька запитів, подібно до того, як одночасно можуть бути активні кілька викликів однієї і тієї ж процедури.

Взаємодія типу клієнт-сервер зустрічається в операційних системах, об’єктно-орієнтованих системах, мережах, базах даних і багатьох інших програмах. Типовий приклад — читання і запис файлу. Для визначеності припустимо, що є модуль файлового сервера, що забезпечує дві операції з файлом: `read` (читати) і `write` (писати). Коли процес-клієнт хоче отримати доступ до файлу, він викликає операцію читання або запису у відповідному модулі файлового сервера.

На однопроцесорній машині або в іншій системі з пам’яттю файловий сервер зазвичай реалізується набором підпрограм (для операцій `read`, `write` і т. д.) і структурами даних, що зображають файли (наприклад, дескрипторами файлів). Отже, взаємодія між процесом-клієнтом і файлом зазвичай реалізується викликом відповідної процедури. Однак, якщо файл розділяється, важливо, щоб запис в нього вівся одночасно тільки одним процесом, а зчитуватися він може одночасно кількома. Цей різновид задачі — приклад так званої задачі про "читачів і письменників", класичної задачі паралельного програмування.

6.3.5. Взаємодіючі рівні: розподілене множення матриць

Розглянемо два способи вирішення цієї задачі з використанням процесів, взаємодіючих за допомогою пересилання повідомлень. Перша програма використовує керуючий процес і масив незалежних робочих процесів. У другій програмі робочі процеси рівні і їх взаємодія забезпечується круговим конвеєром. Мал. 6.3 ілюструє схему взаємодії цих процесів.

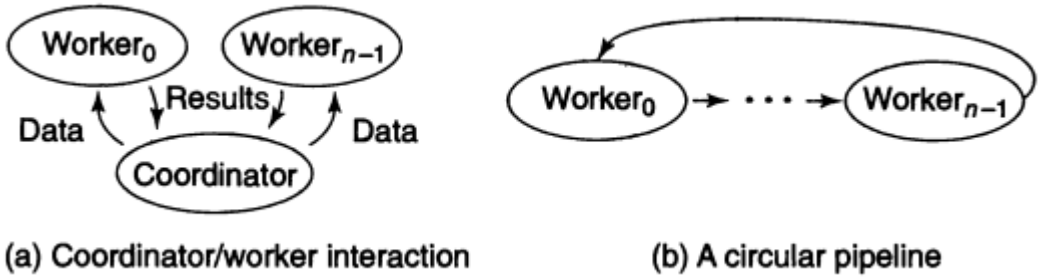

Рис. 6.3. Множення матриць з використанням передачі повідомлень

На машинах з розподіленою пам'яттю кожен процесор має доступ тільки до власної локальної пам'яті. Це означає, що програма не може використовувати глобальні змінні, тому будь-яка змінна повинна бути локальною для деякого процесу і може бути доступною тільки цьому процесу або процедури. Отже, для взаємодії процеси повинні використовувати передачу повідомлень.

Нехай нам необхідно знайти добуток квадратних $n \times n$ матриць a і b, а результат помістити в матрицю c. Припустимо, що у системі є n процесорів. Можна використовувати масив з n робочих процесів, змусивши кожен робочий процес обчислювати один рядок результуючої матриці c:

```
process worker [i = 0 to n-1] {
    receive початкові значення вектора a та матриці b;
    # вектори a, c — i-ті рядки відповідних матриць
    for [j = 0 to n-1] {
        c[j] = 0;
        for [k = 0 to n-1]
            c[j] = c[j] + a[k] * b[k,j];
    }
    send вектор-результат c керуючому процесу;
}
```

Робочий процес i обчислює *i*-й рядок результуючої матриці *c*. Щоб це зробити, він повинен отримати рядок *i* вихідної матриці *a* і всю вихідну матрицю *b*. Кожен робочий процес спочатку отримує ці значення від окремого керуючого процесу. Потім робочий процес обчислює свій рядок результатів і відсилає її назад керуючому.

Керуючий процес ініціює обчислення, збирає і виводить їх результати. Зокрема, спочатку керуючий процес посилає кожному робочому відповідну рядок матриці *a* і всю матрицю *b*. Потім керуючий процес очікує отримання рядків матриці від кожного робочого процесу. Схема керуючого процесу така.

```
process coordinator {
    double a[n, n]; # Вихідна матриця a
    double b[n, n]; # Вихідна матриця b
    double c[n, n]; # Результуюча матриця c
    # ініціалізувати a та b;
    for [i = 0 to n-1] {
        send рядок i матриці a процесу worker[i];
        send всю матрицю b процесу worker[i];
    }
    for [i = 0 to n-1]
        receive рядок i матриці c від процесу worker[i];
    вивести результат, який тепер в матриці c;
}
```

Оператори send та receive — це *примітиви* передачі повідомлень. Операція send надсилає повідомлення іншому процесу; операція receive чекає повідомлення від іншого процесу, а потім зберігає його в локальних змінних.

Тепер припустимо, що у кожного процесу є тільки один стовпець, а не вся матриця *b*. Отже, в початковому стані робочий процес i має стовпець i матриці *b*. Маючи лише ці вихідні дані, робочий процес може обчислити тільки значення c[i,i]. Для того щоб робочий процес i міг вирахувати всю рядок матриці c, він повинен отримати всі стовпці матриці *b*. Для цього можна використовувати кругової конвеєр (див. рис. 6.3). Кожен робочий процес виконує послідовність раундів; в кожному раунді він відсилає свій стовпець матриці *b* наступному процесу і отримує інший її стовпець від попереднього. Програма має наступний вигляд:

```
process worker [i = 0 to n-1] {
    double a[n];      # Рядок i матриці a
    double b[n];      # Один стовпець матриці b
    double c[n];      # Рядок i матриці c
```

```
        double sum = 0;  # Для проміжних добутків
        int nextCol = i; # Наступний стовпець результатів
        receive рядок i матриці a та стовпець i матриці b;
        # Обчислити c[i, i] = a[i,*] b[*,i]
        for [k = 0 to n-1]
            sum = sum + a[k] * b[k];
        c[nextCol] = sum;
        # Пустити по колу стовпчики та обчислити інші c[i,*]
        for [j = 1 to n-1] {
            send мій стовпець матриці b наступному процесу;
            receive новий стовпець матриці b від попереднього;
            sum = 0;
            for [k = 0 to n-1]
                sum = sum + a[k] * b[k];
            if (nextCol == 0)
                nextCol = n-1;
            else
                nextCol = nextCol-1;
            c[nextCol] = sum;
        }
        send вектор-результат c керуючому процесу;
    }
```

Робочі процеси впорядковані відповідно до їх індексів. (Для процесу $n - 1$ наступним є процес 0, а попереднім для 0 — процес $n - 1$.) Стовпці матриці b передаються по колу між робочими процесами, тому кожен процес врешті-решт отримає кожен стовпець. Змінна `nextCol` відстежує, куди у векторі `c` помістити черговий проміжний добуток. Як і в першому обчисленні, передбачається, що керуючий процес надсилає рядки матриці a і стовпці матриці b робочим процесам, а потім отримує від них рядки матриці c.

У попередній програмі використано відношення між процесорами, яке називається *взаємодіючі рівні*. Кожен робочий процес виконує той самий алгоритм і взаємодіє з іншими робочими процесами, щоб обчислити свою частину необхідного результату.

У першій з наведених програм матриця b *дублюється* в кожному процесі. У другій програмі в будь-який момент часу у кожного процесу є один рядок матриці a і тільки один стовпець матриці b. Це знижує витрати пам’яті для кожного процесу, але друга програма виконується довше першої, оскільки на кожній її ітерації кожен робочий процес повинен відіслати повідомлення одному сусідові і отримати повідомлення від іншого. Дані програми ілюструють класичне протиріччя між часом і простором в обчисленнях.

7. ПРОГРАМУВАННЯ ІЗ СПІЛЬНИМИ ЗМІННИМИ

У послідовних програмах часто використовуються спільні змінні, наприклад, для зберігання глобальних структур даних, але вважається, що краще обходитися без них. Разом з тим, паралельні програми цілком залежать від спільних компонентів, оскільки процеси можуть працювати над однією задачею, тільки взаємодіючи. А єдиний спосіб взаємодії — можливість для одного процесу записувати в *щось*, звідки інший процес читає. Цим чимось може бути колективна змінна або канал зв'язку. Тому взаємодія програмується як запис і читання спільних змінних або як передача і прийом повідомлень.

Взаємодія підвищує необхідність синхронізації. Існує два основних її типи: взаємне виключення і умовна синхронізація. Взаємне виключення зустрічається, коли два процеси повинні по черзі звертатися до таких спільних об'єктів, як, наприклад, записи в системі замовлення авіаквитків. Умовна синхронізація умов відбувається, коли одному процесу доводиться чекати інший процес, наприклад, коли процес-споживач очікує дані від процесу-виробника.

7.1. Стан, дії, історія та властивості

Стан паралельної програми складається зі значень змінних програми в деякий момент часу. Змінні можуть бути описані явно певними програмістом або неявними (на кшталт програмного лічильника кожного процесу), що зберігають приховану інформацію про стан. Паралельна програма починає виконання в деякому вихідному стані. Кожен процес програми виконується незалежно, і в міру виконання він перевіряє і змінює стан програми.

Процес виконує послідовність операторів. Оператор, в свою чергу, реалізується послідовністю *неподільних дій*. Ці дії перевіряють чи змінюють стан програми *неподільним чином*. Прикладами неподільних дій є неперервні машинні інструкції, які завантажують і зберігають слова пам'яті.

Виконання паралельної програми призводить до чергування послідовностей неподільних дій, вироблених кожним процесом. Конкретне виконання кожної програми може бути розглянуто як *історія* $s_0 \to s_1 \to \ldots \to s_n$, де s_0 — початковий стан. Переходи між станами здійснюються неподільними діями, що змінюють стан. Навіть паралельне виконання можна подати у вигляді лінійної історії, оскільки паралельна реалізація на-

бору неподільних дій еквівалентна їх виконання в деякому послідовному порядку. Мета синхронізації — виключити небажані історії паралельної програми.

Властивістю програми називається атрибут, який є істинним за будь-якої можливої історії програми і, отже, при всіх її виконаннях. Є два типи властивостей: безпека та живучість. *Властивість безпеки* полягає в тому, що програма ніколи не потрапляє в «поганий» стан (при якому деякі змінні можуть мати небажані значення). *Властивість живучості* означає, що програма в кінці кінців завжди потрапляє в «гарний» стан, тобто стан, в якому всі змінні мають бажані значення.

Взаємне виключення — це приклад властивості безпеки в паралельній програмі. При поганому стані два процеси такої програми одночасно виконують дії в різних критичних секціях. Можливість врешті-решт увійти в критичну секцію — приклад властивості живучості в паралельній програмі. В хорошому стані кожен процес виконується в своїй критичній секції.

Постає питання: як перевірити, що дана програма має бажану властивість? Звичайний підхід полягає в тестуванні. Його можна охарактеризувати фразою «запусти програму і подивися, що вийде». Це відповідає перебору деяких можливих історій програми і перевірці їх прийнятності. Недолік такої перевірки полягає в тому, що кожен тест стосується тільки однієї історії виконання, а в загальному випадку кожний запуск програми приводить до нової історії.

Другий підхід — використання *операторних міркувань*, які можна назвати "вичерпним аналізом випадків" (перебираються всі можливі історії виконання програми). Для цього аналізуються способи чергування неподільних дій процесів. На жаль, в паралельній програмі число можливих історій зазвичай дуже велике. Припустимо, що програма містить n процесів і кожен з них виконує послідовність з m неподільних дій. Тоді число різних історій програми складе $\frac{(n \cdot m)!}{(m!)^n}$. Для програми з трьох процесів, кожен з яких виконує лише дві неподільні операції, можливі 90 різних історій.

Третій підхід — використання стверджувальних міркувань (assertional reasoning). Цей підхід можна назвати "абстрактним аналізом" [10].

7.2. Розпаралелювання: пошук зразка в файлі

Розглянемо одну просту задачу і вивчимо способи її розпаралелювання. Розглянемо задачу пошуку всіх входжень шаблону `pattern` у файлі. Послідовна програма:

```
string line;
прочитати рядок в line;
while(! EOF){
    шукати pattern в line;
    if(pattern є в line)
        вивести line;
    прочитати наступний рядок;
}
```

Тепер бажано з'ясувати, чи можна розпаралелити цю програму? Основна вимога для можливості розпаралелювання будь-якої програми полягає в тому, що вона повинна містити незалежні частини. Дві частини взаємно залежні, якщо кожна з них породжує результати, необхідні для іншої; це можливо, тільки якщо вони зчитують і записують спільні змінні. Отже, дві частини програми незалежні, якщо вони не виконують читання і запис одних і тих же змінних. Більш точне визначення таке:

Незалежність паралельних процесів. Дві частини програми є незалежними, якщо перетин множини запису однієї та множини зчитування іншої — порожній.

При цьому вважаємо, що зчитування або запис будь-якої змінної неподільні. Іноді дві частини програми можуть безпечно виконуватися паралельно, навіть змінюючи одні і ті самі змінні. Це можливо, якщо не важливий порядок, в якому відбувається зміна. Наприклад, якщо кілька процесів періодично оновлюють графічний екран, і будь-який порядок виконання оновлень не псує вигляду екрана.

Повернемося до задачі пошуку шаблону в файлі. Які частини програми є незалежними і, отже, можуть бути виконані паралельно? Програма починається читанням першого рядку введення; це *повинно* бути зроблено перед усіма іншими діями. Після цього програма входить в цикл пошуку шаблону, виводить рядок, якщо шаблон був знайдений, а потім зчитує новий рядок. Вивести рядок до того, як в ній був виконаний пошук шаблону, не можна, тому перші два рядки циклу виконати паралельно неможливо. Розглянемо іншу, паралельну, версію попередньої програми,

```
string line;
прочитати вхідний рядок в line;
```

```
while(!EOF){
    co
        шукати pattern у line;
        if(pattern є у line)
            вивести line;
    // Прочитати наступний рядок та записати його в line;
    oc;
}
```

Відзначимо, що перша гілка оператора co є послідовністю операторів. Але два процеси не є незалежними, оскільки перший читає line, а інший записує в неї. Тому, якщо другий процес виконується швидше першого, він буде перезаписувати рядок до того, як її встигне перевірити перший процес.

Частини програми можуть виконуватися паралельно тільки в тому випадку, якщо вони читають і записують різні змінні. Припустимо, що другий процес записує не в ту змінну, яку перевіряє перший процес, і розглянемо наступну програму.

```
string line1, line2;
прочитати рядок введення в line1;
while(!EOF){
    co
        шукати pattern в line1;
        if(pattern є в line1)
            вивести line1;
    // прочитати наступний рядок та записати його в line2;
    oc;
    line1 = line2;
}
```

Процеси всередині оператора co незалежні, але їх дії пов’язані останнім оператором у циклі, який копіює line2 в line1. Паралельна програма, наведена вище, вірна, але абсолютно неефективна. По-перше, в останньому рядку циклу вміст змінної line2 копіюється в змінну line1. Це вимагає копіювання великої кількості символів, а це — накладні витрати. По-друге, в тілі циклу міститься оператор co, а це означає, що при кожному повторенні циклу while будуть створюватися, виконуватися і знищуватися по два процеси.

Вирішення проблеми у тому, що замість використання оператора co всередині циклу while, можна помістити цикли while в кожну гілку оператора co. Отримана програма є прикладом схеми типу "виробник-споживач". Тут другий процес є виробни-

ком, а перший — споживачем. Вони взаємодіють з допомогою змінної `buffer`. Відзначимо, що оголошення змінних `line1` і `line2` тепер стали локальними для процесів. Перевага стилю "`while` всередині `co`" полягає в тому, що процеси створюються тільки одного разу, а не при кожному повторенні циклу. Недоліком є необхідність використовувати два буфера і програмувати синхронізацію. Оператори, що передують зверненню до буфера `buffer` і наступні за ним, вказують тип необхідної синхронізації.

```
string buffer;       # Містить один рядок введення
bool done = false;   # Сигнал про завершення
co # процес 1: знайти шаблони
   string line1;
   while (true) {
       очікувати заповнення буфера або значення true змінної done;
       if(done) break;
       line1 = buffer;
       сигналізувати, що буфер порожній;
       шукати pattern в line1;
       if (pattern є в line1)
          надрукувати line1;
   }
   // # процес 2: прочитати нові рядки
   string line2;
   while (true) {
       прочитати наступний рядок введення в line2;
       if (EOF) {done = true; break;}
       очікувати спустошення буфера
       buffer = line2;
       сигналізувати про заповнення буфера;
   }
oc
```

7.3. Синхронізація: пошук максимального елемента масиву

Розглянемо іншу задачу, яка вимагає синхронізації процесів. Вона полягає у пошуку максимального елемента масиву `a[n]`. Припустимо, що `n` додатне і всі елементи масиву — додатні цілі числа. Для знаходження розв’язку можна використовувати таку послідовну програму:

```
int m = 0;
for [i = 0 to n-1]
   if (a[i] > m)
      m = a[i];
```

Розглянемо способи розпаралелювання наведеної програми. Припустимо, що цикл повністю розпаралелений за допомогою паралельної перевірки всіх елементів:

```
int m = 0;
co [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

Ця програма є некоректною, оскільки процеси не є незалежними: кожен з них і читає, і записує змінну `m`. Зокрема, припустимо, що всі процеси виконуються з однаковою швидкістю, і, отже, кожен з них порівнює свій елемент масиву `a[i]` зі змінною `m` в один і той же час. Всі процеси визначать, що нерівність виконується (оскільки всі елементи масиву більші за початкове значення змінної `m`). Тому всі процеси спробують оновити значення `m`. Апаратне забезпечення пам'яті буде виконувати оновлення в порядку деякої черги. Кінцевим значенням `m` буде значення `a[i]`, присвоєне їй останнім процесом.

Розглянемо версію програми з використанням неподільних операцій:

```
int m = 0;
co [i = 0 to n-1]
    ⟨if (a[i] > m)
        m = a[i];⟩
```

Кутові дужки вказують, що кожен оператор `if` виконується як неподільна операція, тобто перевіряє поточне значення `m` і оновлює його в одній, неподільний дії.

На жаль, остання програма — це майже те ж саме, що і послідовна. У послідовній програмі елементи масиву `a` перевіряються в установленому порядку — від `a[0]` до `a[n-1]`. У останній програмі елементи масиву `a` перевіряються в порядку виконання процесів. Але через синхронізації перевірки все ще виконуються по одній.

Спробуємо забезпечити, щоб оновлення змінної `m` були неподільними операціями, а значення `m` було дійсно максимальним. Припустимо, що порівняння виконуються паралельно, але поновлення виробляються по одному, як в наступній програмі.

```
int m = 0;
co [i = 0 to n-1]
    if (a[i] > m)
        ⟨m = a[i];⟩
```

Ця версія програми некоректна, оскільки ця програма насправді є тим же, що і перша паралельна програма: кожен процес може порівняти своє значення елемента масиву `a` із змінною `m` і потім оновити значення змінної `m`.

Вихід полягає в поєднанні останніх двох програм. Можна безпечно виконувати паралельні порівняння, оскільки це дії, які тільки читають змінні. Але необхідно забезпечити, щоб при завершенні програми значення `m` дійсно було максимальним. Це досягається таким чином:

```
int m = 0;
co [i = 0 to n-1]
    if (a[i] > m)        # перевірка значення m
       ⟨if (a[i] > m)    # повторна перевірка значення m
           m = a[i];⟩
```

Ідея полягає в тому, щоб спочатку перевірити нерівність, а потім, якщо воно виконується, провести ще одну перевірку перед оновленням значення змінної. Вона може здатися зайвою, але це не так. Наприклад, якщо певний процес оновив значення `m`, то частина інших процесів визначить, що їх значення `a[i]` менше нового значення `m`, і не буде виконувати тіло оператора `if`. Після подальших оновлень ще менше процесів визначать, що умова в першій перевірці істинна.

В розглянутих прикладах є три ключових моменти:

- синхронізація необхідна для отримання правильних результатів, якщо процеси і зчитують, і записують спільні змінні;
- неподільність дій позначається у псевдокоді кутовими дужками;
- метод подвійної перевірки перед оновленням спільної змінної часто є корисним.

7.4. Неподільні дії

Неподільна дія виконує неподільне перетворення стану. Тобто будь-який проміжний стан, який може виникнути у процесі виконання дії, є невидимим для інших процесів.

У паралельних програмах оператор присвоювання не є неподільним. Розглянемо наступний фрагмент коду:

```
int y = 0, z = 0;
co
    x = y+z;
    // y = 1; z = 2;
oc;
```

Якщо вираз `x = y+z` реалізовується за допомогою завантаження значень змінних у регістри та виконанням операції додаванням, то змінна `x` може набувати значення 0, 1, 2, 3. Це відбувається тому, що додавання може проводитися як із початковими значеннями змінних, так і з кінцевими значеннями або якоюсь їх комбінацією в залежності від того, до якої міри виконано 2-й процес. Ще однією особливістю програми є те, що не можна зупинити програму і побачити її стан, у якому сума `y+z` приймає значення 2.

Передбачається, що машини мають наступні характеристики.

- Значення базових типів (наприклад int) зберігаються в елементах пам'яті (наприклад словах), які зчитуються і записуються неподільними операціями.
- Значення обробляються так: їх поміщають в регістри, там до них застосовують операції і потім записують результати назад в пам'ять.
- Кожен процес має власний набір регістрів. Це реалізується або шляхом надання кожному процесу окремого набору регістрів, або шляхом збереження і відновлення значень регістрів при виконанні різних процесів.
- Будь-які проміжні результати, що з'являються при обчисленні складних виразів, зберігаються в регістрах або областях пам'яті, що належать процесу, що виконується, наприклад, в його стеку.

У цій моделі машини, якщо вираз в одному процесі не звертається до змінної, зміненої іншим процесом, обчислення виразу завжди буде неподільною операцією, навіть якщо для цього необхідно виконати кілька дрібномодульних дій. За тих самих умов присвоювання буде неподільною операцією.

На жаль, багато операторів в паралельних програмах, що посилаються на колективні змінні, не задовольняють цим умовам. Однак часто виконуються більш м'які умови.

Умова "не більше одного". *Критичним посиланням* у виразі називається посилання на змінну, яка змінюється іншим процесом. Оператор присвоювання `x = e` задовольняє умову "не більше одного", якщо або вираз `e` містить не більше одного критичного посилання, а змінна `x` не зчитується іншим процесом, або вираз `e` не містить критичних посилань, а інші процеси можуть зчитувати змінну `x`.

Ця умова називається "не більше одного", оскільки в такому випадку можлива лише одна спільна змінна, і на неї посилаються не більше одного разу. Аналогічне ви-

значення застосовується до виразів, які не є операторами присвоювання. Такий вираз задовольняє умові "не більше одного", якщо він містить не більше одного критичного посилання.

Якщо оператор присвоювання задовольняє вимогам умови "не більше одного", то виконання оператора присвоювання *буде здаватися неподільної операцією*, оскільки єдина спільна змінна у виразі буде записуватися або зчитуватися тільки один раз. Наприклад, якщо вираз `e` не містить критичних посилань, а змінна `x` — проста змінна, що читається іншими процесами, то вони не можуть розпізнати, чи обчислюється вираз неподільним чином. Аналогічно, якщо `e` містить одне критичне посилання, то процес, що виконує присвоювання, не зможе розрізнити, яким чином змінюється значення змінної; він побачить тільки деякий кінцеве значення.

Наведемо кілька прикладів. У наступній програмі обидва присвоювання задовольняють умові "не більше одного":

```
int x = 0, y = 0;
co x = x + 1; // y = y + 1; oc;
```

Тут немає критичних посилань, тому кінцевим значенням і `x`, і `y` буде 1.

Обидва присвоювання в наступній програмі також задовольняють умові:

```
int x = 0, y = 0;
co x = y + 1; // y = y + 1; oc;
```

Перший процес посилається на `y` (одне критичне посилання), але змінна `x` не читається другим процесом, і в другому процесі немає критичних посилань. Кінцевим значенням змінної `x` буде або 1, або 2, а кінцевим значенням `y` — 1. Перший процес побачить змінну `y` або перед її збільшенням, або після, але в паралельній програмі він ніколи не знає, яке значення є він бачить, оскільки порядок виконання програми недетермінований.

У наступному прикладі жодне присвоювання не відповідає вимозі "не більше одного":

```
int x = 0, y = 0;
co x = y + 1; // y = x + 1; oc;
```

Вираз в кожному процесі містить критичне посилання, і кожен процес присвоює значення змінної, що зчитується іншим процесом. Кінцевими значеннями змінних `x` і `y` можуть бути 1 і 2, 2 і 1, або навіть 1 і 1 (якщо процеси зчитують значення змінних `x` і

y до присвоєння їм значень). Однак, оскільки кожне присвоювання посилається тільки один раз і тільки на одну змінну, яку змінює інший процес, кінцевими будуть ті значення, які дійсно існували в деякому стані. Це відрізняється від прикладу, наведеного вище, в якому вираз y + z посилався на дві змінні, що змінюються іншим процесом.

Задача. Розглянемо програму:

```
int x = 1, y = 1;
co
    ⟨x = x + y;⟩
    // y = 0;
    // x = x - y;
oc
```

а) поясніть, чи задовольняє ця програма властивості "не більше одного";

б) вкажіть можливі кінцеві значення змінних x та y.

7.5. Задача синхронізації: оператор очікування

Можливо, вираз або оператор присвоювання не задовольняє умові "не більше одного", проте необхідно виконати його неподільним чином. Тоді потрібен механізм синхронізації, що дозволяє задати крупномодульну неподільну дію, — послідовність дрібномодульних неподільних операцій, яка виглядає як неподільна.

Як конкретний приклад уявимо, що база даних містить два значення x і y, які завжди повинні бути однакові в тому сенсі, що жоден процес, використовує базу даних, не повинен бачити стану, в якому x і y розрізняються. Отже, якщо процес змінює x, він повинен змінити і y в тій самій неподільній дії.

Ще один приклад: нехай один процес вставляє елементи в чергу, реалізовану у вигляді зв'язаного списку. Інший процес видаляє елементи зі списку за умови, що вони там є. Вставка і видалення повинні бути неподільними діями. До того ж, якщо список порожній, необхідно відкласти виконання операції видалення до того, як в список буде вставлений елемент.

Запис ⟨e⟩ вказує, що вираз e має бути обчислений неподільним чином.

Синхронізація визначається за допомогою оператора await:

```
⟨await (B) S;⟩
```

Булевий вираз `B` задає *умову затримки* (delay condition), а `S` — це список послідовних операторів. Оператор `await` розташований у кутових дужках для вказівки, що він виконується як неподільна дія. Зокрема, вираз `B` гарантовано має значення "істина", коли починається виконання `S`, і жодний проміжний стан в `S` не видно іншим процесам. Наприклад, виконання коду

```
⟨await (s > 0) s = s-1;⟩
```

відкладається до моменту, коли значення `s` стане позитивним, а потім воно зменшується на 1. Оператор `await` є дуже потужним, оскільки може бути використаний для визначення будь-яких крупномодульних неподільних дій.

Загальна форма оператора `await` визначає як взаємне виключення, так і синхронізацію за умовою. Для визначення тільки взаємного виключення можна використовувати скорочену форму оператора `await`:

```
⟨S;⟩
```

Наприклад, в наступному операторі значення x і y збільшуються в неподільній дії:

```
⟨x = x + 1; y = y + 1;⟩
```

Проміжний стан, в якому змінна `x` була збільшена на одиницю, а `y` — ще ні, за визначенням не буде видимим для інших процесів, що посилаються на змінні `x` або `y`.

Якщо потрібно виконати тільки умовну синхронізацію використовують скорочену форму оператора `await`:

```
⟨await (B);⟩
```

Наприклад, наступний оператор відкладає виконання процесу до моменту, коли значення змінної `count` стане позитивним:

```
⟨await (count > 0);⟩
```

Якщо вираз `B` задовольняє умову "не більше одного", то `⟨await (B);⟩` може бути реалізовано як

```
while (!B);
```

Це приклад так званого *циклу очікування* (spin loop). Тіло оператора `while` порожнє, тому він просто зациклюється до тих пір, коли значенням B стане `false`.

Безумовна неподільна дія — це дія, яке не містить в тілі умови затримки `B`. Така дія може бути виконаною негайно. *Умовна неподільна дія* — це оператор `await` з умовою `B`. Така дія не може бути виконаною, поки умова `B` не стане істинною. Якщо `B` хибне, воно може стати істинним тільки в результаті дій інших процесів. Таким чином, процес, який чекає виконання умовного неподільної дії, може виявитися затриманим непередбачувано довго.

Задача. Розглянемо наступну програму:

```
int x = 2, y = 3;
co
    ⟨x = x + y;⟩
    // ⟨y = x * y;⟩
oc
```

а) вкажіть можливі кінцеві значення змінних `x` та `y`;

б) припустимо, що кутові дужки прибрані, і кожний оператор присвоювання реалізується трьома неподільними діями: зчитування змінної, додавання або множення та запис змінної. Якими тепер можуть бути кінцеві значення змінних `x` та `y`?

7.6. Синхронізація типу "виробник-споживач"

Дано два процеси: `Producer` і `Consumer`. Процес `Producer` має локальний масив цілих чисел `a[n]`; `Consumer` — `b[n]`. Мета — скопіювати вміст масиву `a` в масив `b`. Нехай змінна `buf` — це спільна цілочислова змінна, яка використовується у якості буфера взаємодії. Процеси `Producer` і `Consumer` повинні отримувати доступ до змінної `buf` по черзі. Нехай спільні змінні `p` та `c` — лічильники числа поміщених і вибраних елементів, відповідно. Їх початкові значення — 0. Тоді умови синхронізації процесів `Producer` і `Consumer` можуть бути записані в наступному вигляді:

```
PC: c <= p <= c + 1
```

Процеси `Producer` і `Consumer` використовують змінні `p` та `c` для синхронізації доступу до буфера `buf`. Оператори `await` застосовуються для припинення процесів до тих пір, поки буфер не стане повним або порожнім. Якщо істинна умова `p == c`, то буфер порожній (останній поміщений в нього елемент був вибраний), а якщо `p > c` —

заповнений. Якщо синхронізація реалізується описаним способом, кажуть, що процес знаходиться в *стані активного очікування*, або *зациклений*, оскільки він зайнятий перевіркою умови в операторі `await`, але все, що він робить, — це повторення циклу до тих пір, поки умова не виконається. Це звичайний тип синхронізації, необхідний на найнижчих рівнях програмних систем, наприклад, в операційних системах та мережевих протоколах

```
int buf, p = 0, c = 0;
process Producer {
    int a[n];
    while (p < n) {
        ⟨await (p == c);⟩
        buf = a[p];
        p = p + 1;
    }
}
process Consumer {
    int b[n];
    while (c < n) {
        ⟨await (p > c);⟩
        b[c] = buf;
        c = c + 1;
    }
}
```

7.7. Стратегії планування і справедливість

Більшість властивостей живучості залежить від *справедливості* (fairness), пов'язаної з гарантіями того, що кожен процес отримує шанс на продовження виконання незалежно від дій інших процесів. *Стратегія планування* (scheduling policy) визначає, яку неподільну дію буде виконано наступною. Далі буде розглянуто три ступеня справедливості, які можуть бути забезпечені стратегією планування.

7.7.1. Безумовна справедливість

Розглянемо наступну просту програму:

```
bool continue = true;
co while (continue);
// continue = false;
oc
```

Припустимо, що стратегія планування призначає процесор для процесу до тих пір, поки той не завершиться, чи не буде припинений. У випадку одного процесора дана

програма не завершиться, якщо спочатку буде виконуватися перший процес. Однак, якщо 2-й процес врешті-решт отримає право на виконання, програма буде завершена. Дана ситуація відображена в наступному визначенні:

Стратегія планування *безумовно справедлива*, якщо *будь-яка безумовна неподільна дія врешті-решт виконується*.

Для програми, наведеної вище, безумовно справедливою стратегією планування на одному процесорі було б *циклічне* (round-robin) виконання, а на мультипроцесорі — *синхронне*.

7.7.2. Слабка справедливість

Якщо програма містить умовну неподільну дію (оператор `await` з логічною умовою `B`), необхідно робити більш сильні припущення, щоб гарантувати просування процесу. Причина в тому, що умовна неподільна дія не почне виконуватися, поки умова `B` не стане істинною.

Стратегія планування *справедлива в слабкому сенсі*, якщо:

1) вона безумовно справедлива;
2) кожна умовна неподільна дія врешті-решт виконується, якщо її умова стає і потім залишається істинною, поки її бачить процес, який виконує умовну неподільну дію.

Таким чином, якщо для дії ⟨`await (B) S;`⟩ умова `B` стає істинною, то `B` залишається істинною принаймні до закінчення виконання умовної неподільної дії. Циклічна стратегія і тактика квантування часу є справедливими в слабкому сенсі, якщо кожному процесу дається можливість виконання. Причина в тому, що будь-який призупинений процес врешті-решт побачить, що умова закінчення його затримки істинна.

7.7.3. Сильна справедливість

Справедливості в слабкому сенсі, однак, недостатньо для гарантії, що будь-який оператор `await` в кінці кінців виконається. Це пов’язано з тим, що умова може змінити своє значення (від хибного до істинного і навпаки), поки процес припинений. У цьому випадку необхідна сильніша стратегія планування.

Стратегія планування *справедлива в сильному сенсі*, якщо:

1) вона безумовно справедлива;

2) будь-яка умовна неподільна дія в кінці кінців виконується в припущенні, що її умова буває істинною нескінченно часто.

Умова буває істинною нескінченно часто, якщо вона істинна нескінченне число разів у кожній історії програми (що не закінчується). Щоб стратегія планування була справедливої в сильному сенсі, дія повинна вибиратися для виконання не тільки тоді, коли її умова хибна, але і тоді, коли вона істинна.

Щоб побачити відмінності між справедливістю в слабкому і сильному сенсі, розглянемо наступну програму.

```
bool continue = true, try = false;
co
    while (continue){
        try = true;
        try = false;
    }
// ⟨await (try) continue = false;⟩
oc
```

При стратегії, справедливої в сильному сенсі, ця програма в кінці кінців завершиться, оскільки значення змінної `try` нескінченно часто істинно. Однак при стратегії планування, справедливої в слабкому сенсі, програма може не завершитися, оскільки значення змінної `try` також нескінченно часто є хибним.

На жаль, *неможливо розробити стратегію планування процесора, яка була б і практичною, і справедливою в сильному сенсі* [10]. Ще раз розглянемо програму, наведену вище. На одному процесорі диспетчер, чергуючий дії двох процесів, буде справедливим в сильному сенсі, оскільки другий процес буде бачити стан, в якому значення змінної `try` істинно. Циклічне планування і планування з квантуванням часу практичні, але в загальному випадку не є справедливими в сильному сенсі, оскільки процеси виконуються в непередбачуваному порядку. Диспетчер мультипроцессора, виконуючий процеси паралельно, також практичний, але не є справедливим в сильному сенсі. Причина в тому, що другий процес може перевіряти значення змінної `try` тільки тоді, коли воно хибне. Це, звичайно, малоймовірно, але теоретично можливо.

Повернемося до програми копіювання масиву, наведеної у підрозділі 5.6. (Синхронізація типу "виробник-споживач"). Ця програма вільна від блокувань. Таким чином, програма буде завершена, оскільки кожен процес регулярно отримує можли-

вість просунутися в своєму виконанні. Процес буде просуватися, оскільки стратегія справедлива в слабкому сенсі. Справа в тому, що, коли один процес робить істинним умову закінчення затримки іншого процесу, ця умова залишається істинною, поки інший процес не буде продовжений і не змінить спільні змінні.

Задача. Розглянемо наступну програму:

```
int x;
co
   ⟨await (x >= 3) x = x - 2;⟩
   // ⟨await (x >= 2) x = x - 3;⟩
   // ⟨await (x == 1) x = x + 5;⟩
oc
```

Для яких початкових значень змінної `x` програма завершиться, якщо застосовується справедлива у слабкому сенсі стратегія планування? Якими будуть кінцеві значення `x`?

8. БЛОКУВАННЯ ТА БАР'ЄРИ

8.1. Задача критичної секції

Задача критичної секції — це одна з класичних задач паралельного програмування. Вона була першою всебічно вивченою задачею, але інтерес до неї не згасає, оскільки критичні секції коду є в більшості паралельних програм. Крім того, розв'язок цієї задачі можна використовувати для реалізації операторів `await`.

У задачі критичної секції `n` процесів багаторазово виконують спочатку критичну, а потім некритичну секцію коду. Критичній секції передує протокол входу, а за нею йде протокол виходу. Таким чином, передбачається, що процес має наступний вигляд:

```
process CS [i = 1 to n] {
    while (true) {
        протокол входу;
        критична секція;
        протокол виходу;
        некритична секція;
    }
}
```

Кожна критична секція є послідовністю операторів, що мають доступ до деякого спільного об'єкта. Кожна некритична секція — це ще одна послідовність операторів. Передбачається, що процес, який увійшов в критичну секцію, обов'язково коли-небудь з неї вийде; таким чином, процес може завершитися тільки поза критичною секцією. Завдання — розробити протоколи входу і виходу, які задовольняють наступним властивостям:

(1) **Взаємне виключення**. У будь-який момент часу тільки один процес може виконувати свою критичну секцію.

(2) **Відсутність взаємного блокування** (**живого блокування**). Якщо кілька процесів намагаються увійти в свої критичні секції, хоча б один це здійснить.

(3) **Відсутність зайвих затримок**. Якщо один процес намагається увійти в свою критичну секцію, а інші виконують свої некритичні секції або завершені, першому процесу дозволяється вхід в критичну секцію.

(4) **Можливість входу**. Процес, який намагається увійти в критичну секцію, коли-небудь це зробить.

Перші три властивості є властивостями безпеки, четверта — властивістю живучості.

Тривіальний спосіб вирішення завдання критичної секції полягає в обмеженні кожної критичної секції кутовими дужками, тобто у використанні безумовних операторів `await`. З семантики кутових дужок відразу випливає умова (1) — взаємне виключення. Інші три властивості задовольнятимуться при безумовно справедливій стратегії планування, оскільки вона гарантує, що процес, який намагається виконати неподільну дію, відповідну його критичної секції, в кінці кінців це зробить, незалежно від дій інших процесів. Однак при такому "розв'язку" виникає питання про те, як реалізувати кутові дужки.

Всі чотири зазначених властивості важливі, однак найбільш істотним є взаємне виключення. Спочатку ми зосередимося на ньому, а потім дізнаємося, як забезпечити виконання інших властивостей. Для опису властивості взаємного виключення необхідно визначити, чи знаходиться процес у своїй критичній секції. Щоб спростити запис, розглянемо розв'язок задачі у випадку двох процесів, `CS1` і `CS2`; він легко узагальнюється для `n` процесів.

Нехай `in1` та `in2` — логічні змінні з початковим значенням `false`. Якщо процес `CS1 (CS2)` знаходиться в своїй критичній секції, змінній `in1 (in2)` присвоюється значення `true`. Поганий стан, якого ми будемо намагатися уникнути, — якщо обидві змінні `in1` та `in2` є істинними. Таким чином, нам потрібно, щоб для будь-якого стану виконувалося заперечення умови поганого стану:

```
MUTEX: !(in1 && in2)
```

Предикат `MUTEX` має бути глобальним інваріантом. Для цього він повинен виконуватися в початковому стані і після кожного присвоювання змінним `in1` та `in2`. Зокрема, перед тим, як процес `CS1` увійде в критичну секцію, зробивши тим самим `in1` істинною, він повинен переконатися, що `in2` хибна. Це можна реалізувати так:

```
⟨await (!in2) in1 = true;⟩
```

Вхідний протокол процесу `CS2` аналогічний. При виході з критичної секції затримуватися ні до чого, тому захищати оператори, які надають змінним `in1` та `in2` значення `false`, немає необхідності.

Розв'язок наведено у наступній програмі:

```
bool in1 = false, in2 = false;
process CS1 {
   while (true) {
      ⟨await (!in2) in1 = true;⟩  # вхід
```

```
            критична секція;
            in1 = false;                # вихід
            некритична секція;
        }
    }
    process CS2 {
        while (true) {
            ⟨await (!in1) in2 = true;⟩  # вхід
            критична секція;
            in2 = false;                # вихід
            некритична секція;
        }
    }
```

За побудовою програма задовольняє умові взаємного виключення. Взаємне блокування тут не виникне: якби кожен процес був заблокований в своєму протоколі входу, то обидві змінні, і `in1`, і `in2`, були б істинними, а це суперечить тому, що в даній точці коду обидві вони хибні. Зайвих затримок також немає, оскільки один процес блокується, тільки якщо інший перебуває в критичній секції, тому небажані паузи при виконанні програми не виникають.

Нарешті, розглянемо властивість живучості: процес, який намагається увійти в критичну секцію, в кінці кінців зможе це зробити. Якщо процес `CS1` намагається увійти, але не може, то змінна `in2` істинна, і процес `CS2` знаходиться в критичній секції. За припущенням процес врешті-решт виходить з критичної секції, тому змінна `in2` коли-небудь стане хибною, а змінна захисту входу процесу `CS1` — істинної.

Якщо процесу `CS1` вхід все ще не дозволено, це означає, що або диспетчер несправедливий, або процес `CS2` знову досяг входу в критичну секцію. В останньому випадку описаний вище сценарій повторюється, так що коли-небудь змінна `in2` стане хибною. Таким чином, або змінна `in2` стає хибною нескінченно часто, або процес `CS2` завершується, і змінна `in2` приймає значення `false` і залишається в такому стані. Для того щоб процес `CS1` в будь-якому випадку входив в критичну секцію, потрібно забезпечити справедливу в сильному сенсі стратегію планування.

8.2. Критичні секції: активні блокування

У крупномодульному розв’язку, наведеному в попередній програмі, використовуються дві змінні. При узагальненні для випадку `n` процесів знадобиться `n` змінних. Однак існує тільки два цікаві для нас стани: або певний процес знаходиться в своїй

критичній секції, або жодного там немає. Незалежно від числа процесів, для того, щоб розрізнити ці два стани, досить однієї змінної.

Нехай `lock` — логічна змінна, яка показує, чи знаходиться процес в критичній секції, тобто

```
lock == (in1 || in2)
```

Використовуючи `lock`, можна реалізувати протоколи входу і виходу так:

```
# критичні секції на основі блокування
bool lock = false;
process CS1 {
    while (true) {
        ⟨await (!lock) lock = true;⟩    # вхід
        критична секція;
        lock = false;                   # вихід
        некритична секція;
    }
}
process CS2 {
while (true) {
    while (true) {
        ⟨await (!lock) lock = true;⟩    # вхід
        критична секція;
        lock = false;                   # вихід
        некритична секція;
    }
}
```

Перевага останньої програми полягає в тому, що її можна використовувати для вирішення задачі критичної секції при будь-якому числі процесів. Всі вони будуть спільно використовувати змінну `lock` і виконувати одні і ті ж протоколи.

Використання змінної `lock` замість `in1` та `in2` дуже важливе, оскільки майже у всіх машин, особливо у мультипроцессоров, є спеціальна інструкція для реалізації умовних неподільних дій. Тут застосовується інструкція, яка називається "перевірити-встановити" (test and set — `TS`), в якості аргументу отримує змінну `lock` і повертає логічне значення. У неподільний дії інструкція `TS` зчитує і зберігає значення змінної `lock`, присвоює їй значення `true`, а потім повертає збережене попереднє значення змінної `lock`. Результат дії інструкції `TS` описується наступною функцією:

```
bool TS(bool lock) {
```

```
    ⟨bool initial = lock;   # зберегти початкове значення
     lock = true;           # встановити lock
     return initial; ⟩      # повернути початкове значення
}
```

Використовуючи інструкцію `TS`, можна реалізувати крупномодульний варіант програми для задачі критичної секції. Умовні неподільні дії замінюються циклами. Цикли не закінчуються, поки змінна `lock` не стане хибною, тобто інструкція `TS` повертає значення "фальш". Наведений розв'язок працює при будь-якому числі процесів. Використання блокуючої змінної, зазвичай називається *циклічним блокуванням* (spin lock), оскільки процес постійно повторює цикл, очікуючи зняття блокування.

```
# критичні секції на основі інструкції "перевірити-встановити"
bool lock = false; # колективна змінна
process CS [i = 1 to n] {
    while (true) {
        while (TS(lock)) skip;      # протокол входу
        критична секція;
        lock = false;               # протокол виходу
        некритична секція;
    }
}
```

Взаємне виключення (1) забезпечено: якщо кілька процесів намагаються увійти в критичну секцію, тільки один з них першим змінить значення змінної `lock` з хибного на істинне, отже, тільки один з процесів успішно завершить свій вхідний протокол і увійде в критичну секцію. Відсутність взаємного блокування (2) випливає з того, що, якщо обидва процеси знаходяться в своїх вхідних протоколах, то `lock` хибна, і, отже, один з процесів увійде в свою критичну секцію. Небажані затримки (3) не виникають, оскільки, якщо обидва процеси виходять за межею своїх критичних секцій, `lock` хибна, і, отже, один з процесів може успішно увійти в критичну секцію, якщо інший виконує некритичну секцію або був завершений.

З іншого боку, виконання властивості можливості входу (4) не гарантовано. Якщо використовується справедлива у сильному сенсі стратегія планування, то спроби процесу увійти в критичну секцію завершаться успіхом, оскільки змінна `lock` нескінченно часто буде приймати значення "фальш". При справедливою в слабкому сенсі стратегії планування, яка зустрічається найчастіше, процес може назавжди зациклитися в протоколі входу. Однак це може статися, тільки якщо інші процеси весь час успішно

входять в свої критичні секції, чого на практиці бути не повинно. Отже, наведений розв'язок з великою вірогідністю задовольняє умові справедливої стратегії планування.

Побудований розв'язок з циклічної блокуванням має суттєву властивість:

у розв'язку задачі критичної секції з циклічної блокуванням протокол виходу повинен присвоювати спільним змінним їх початкові значення.

У початковому стані обидві змінні `in1` та `in2` хибні, як і змінна `lock` (див. 6.1, 6.2).

Хоча остання наведена програма логічно вірна, експерименти на мультипроцесорних машинах показують її низьку продуктивність, якщо кілька процесів змагаються за доступ до критичної секції. Причина в тому, що кожен призупинений процес безперервно звертається до змінної `lock`. Ця "гаряча точка" *викликає конфлікт при зверненні до пам'яті*, який знижує продуктивність модулів пам'яті і шин, що зв'язують процесор і пам'ять.

До того ж інструкція `TS` при кожному виклику *записує значення в* `lock`, *навіть якщо воно не змінюється*. Оскільки в мультипроцесорних машинах для зменшення числа звернень до основної пам'яті використовуються кеші, `TS` виконується набагато довше, ніж просте читання значення спільної змінної. (Коли змінна записується одним з процесорів, її копії потрібно оновити або зробити недійсними в кешах інших процесорів.)

Витрати на оновлення вмісту кеш-пам'яті і конфлікти при зверненні до пам'яті можна скоротити. Для цього можна використовувати наступний протокол входу:

```
# Критичні секції на основі протоколу "перевірити-перевірити-встановити"
    bool lock = false
    process CS [i = 1 to n] {
        while (lock) skip;    # поки lock встановлена, повторювати цикл
        while (TS(lock)) {    # спробувати захопити lock
            while (lock) skip;# повторювати цикл, якщо не вдалося
        }
        критична секція
        lock = false; / * Протокол виходу * /
        некритична секція;
        }
    }
```

Цей протокол називається "*перевірити-перевірити-встановити*", оскільки процес просто перевіряє `lock` до тих пір, поки не з'явиться можливість виконання `TS`. У двох додаткових циклах `lock` просто перевіряється, так що її значення можна прочитати з

кеш-пам'яті, не впливаючи на інші процесори. Таким чином, конфлікти при зверненні до пам'яті скорочуються (але не зникають). Якщо прапорець блокування `lock` не встановлений, то як мінімум один, а можливо, і всі призупинені процеси можуть виконати інструкцію `TS`, хоча продовжувати роботу буде тільки один з них.

Задача. Припустимо, що машина підтримує неподільну інструкцію.

```
flip(lock)
⟨ lock = (lock+1) % 2;  # змінити зміну блокування
return (lock); ⟩        # повернути нове значення
```

Розглянемо такий розв'язок задачі критичної секції для двох процесів:

```
int lock = 0;
process CS[i = 1 to 2] {
    while (true) {
        while (flip(lock) != 1)
            while (lock != 0) skip;
        критична секція;
        lock = 0;
        некритична секція;
    }
}
```

а) поясніть, чому цей підхід не коректний, тобто вкажіть порядок виконання, у результаті якого обидва процеси одночасно опиняться у своїх критичних секціях;

б) припустимо, що перший рядок тіла оператора `flip` змінили — тепер у ній виконується додавання за модулем 3. Чи буде відповідна версія програми коректною?

8.3. Реалізація операторів await

Будь-який розв'язок задачі критичної секції можна використовувати для реалізації безумовної неподільної дії ⟨`S;`⟩. Нехай `CSenter` — вхідний протокол критичної секції, а `CSexit` — вихідний. Тоді дію ⟨`S;`⟩ можна реалізувати так:

```
CSenter;
S;
CSexit;
```

Тут передбачається, що всі секції коду процесів, які змінюють або посилаються на змінні, що змінюються в `S` (або змінюють змінні, на які посилається `S`), захищені аналогічними вхідними та вихідними протоколами. По суті, кутові дужки замінені процедурами `CSenter` та `CSexit`.

Наведений підхід можна використовувати для реалізації оператора ⟨`await (B) S;`⟩. Щоб забезпечити неподільність усієї дії, можна використовувати протокол критичної секції, приховуючи проміжні стани у `S`. Для циклічної перевірки умови `B`, поки вона не стане істинною, можна використовувати наступний цикл:

```
CSenter;
while (!B) {???}
S;
CSexit;
```

Тут передбачається, що критичні секції всіх процесів, що змінюють змінні, які використовуються у `B` або `S`, або використовують змінні, що змінюються в `S`, захищені такими ж протоколами входу і виходу.

Залишається з'ясувати, як реалізувати тіло циклу, зазначеного вище. Якщо тіло виконується, значить, умова `B` була хибною. Отже, єдиний спосіб зробити умову `B` істинною — змінити в іншому процесі значення змінних, що входять в цю умову. Передбачається, що всі оператори, що змінюють ці змінні, знаходяться в критичних секціях, тому, чекаючи, поки умова `B` виконається, потрібно вийти з критичної секції. Але для забезпечення неподільності обчислення `B` і виконання `S` перед повторним обчисленням умови `B` необхідно знову увійти в критичну секцію. Можливим уточненням зазначеного вище протоколу може бути:

```
CSenter;
while (!B){
    CSexit;
    CSenter;
}
S;
CSexit;
```

Дана реалізація зберігає семантику умовних неподільних дій за умови, що протоколи критичних секцій гарантують взаємне виключення.

Попередня програма вірна, але не ефективна, оскільки процес, що виконує її, повторює "жорсткий" цикл, постійно виходячи з критичної секції і входячи в неї, але не може просунутися далі, поки який-небудь інший процес не змінить змінних в умові `B`. Це призводить до конфлікту звернення до пам'яті, оскільки кожен призупинений процес постійно звертається до змінних, що використовуються в протоколах критичної секції і умови `B`.

Щоб скоротити кількість конфліктів звернення до пам’яті, процес перед повторною спробою увійти в критичну секцію повинен робити паузу. Нехай `Delay` — деякий код, уповільнюючий виконання процесу. Тоді програму можна замінити наступним протоколом, який реалізує умовну неподільну дію.

```
CSenter;
while (!B){
    CSexit;
    Delay;
    CSenter;
}
S;
CSexit;
```

Кодом `Delay` може бути, наприклад, порожній цикл, який виконується випадкове число разів. (Щоб уникнути конфліктів пам’яті в коді `Delay` слід використовувати лише локальні змінні.) Цей тип протоколу "відходу" ("back-off") корисний і в самих протоколах `CSenter`, наприклад, його можна використовувати замість `skip` в циклі затримки простого протоколу "перевірити-встановити".

Синхронізація з активним очікуванням часто застосовується в апаратному забезпеченні. Фактично протокол, аналогічний до наведеного у останній програмі, використовується для синхронізації доступу в локальних мережах Ethernet. Щоб передати повідомлення, контролер Ethernet надсилає його в мережу і стежить, не виник при передачі конфлікт з повідомленнями, надісланим приблизно в цей же час іншими контролерами. Якщо конфлікт не виявлено, то вважається, що передача завершилася успішно. В іншому випадку контролер робить невелику паузу, а потім повторює передачу повідомлення. Щоб уникнути стану гонитви, в якому два контролера постійно конфліктують через те, що роблять однакові паузи, їх тривалість вибирається випадковим чином з інтервалу, який подвоюється при кожному виникненні конфлікту. Такий протокол називається *двійковим експоненціальним протоколом відходу.*

8.4. Критичні секції: розв’язок із справедливою стратегією

Розв’язок задачі критичної секції з циклічним блокуванням забезпечують взаємне виключення, відсутність взаємних блокувань, активних тупиків і небажаних пауз. Проте для забезпечення властивості можливості входу (4) їм необхідна справедлива в сильному сенсі стратегія планування. Стратегії планування, які застосовуються на

практиці, є справедливими лише в слабкому сенсі. Малоймовірно, що процес, який намагається увійти в критичну секцію, ніколи цього не зробить, однак може статися, що кілька процесів будуть без кінця змагатися за вхід. Зокрема, розв'язки з циклічним блокуванням не керують порядком, в якому кілька призупинених процесів намагаються увійти в критичні секції.

Нижче наведено три розв'язки задачі критичної секції із справедливою стратегією планування: алгоритми розриву вузла, поліклініки та квитка. Вони залежать тільки від справедливої в слабкому сенсі стратегії планування при якому кожен процес періодично отримує можливість виконання, а умови затримки, ставши справжніми, залишаються такими. Алгоритм розриву вузла досить простий для двох процесів і не залежить від спеціальних машинних інструкцій, але складний для *n* процесів. Алгоритм квитка простий для будь-якого числа процесів, але вимагає спеціальної інструкції "отримати і додати". Алгоритм поліклініки — це варіант алгоритму квитка, для якого не потрібні спеціальні машинні інструкції.

8.4.1. Алгоритм розриву вузла

Розглянемо розв'язок задачі критичної секції для двох процесів. Його недолік в тому, що у ньому не вирішено, який із процесів, які намагаються увійти в критичну секцію, туди дійсно потрапить. Наприклад, один процес може увійти в критичну секцію, виконати її, потім повернутися до протоколу входу і знову успішно увійти в критичну секцію. Щоб розв'язок був справедливим, треба дотримуватися черговості входу в критичну секцію, якщо кілька процесорів намагаються туди увійти.

Алгоритм розриву вузла (також відомий як алгоритм Пітерсона) — це варіант протоколу критичної секції, який "розриває вузол", коли два процеси намагаються увійти в критичну секцію. У алгоритмі використовується додаткова змінна `last` — цілочислова змінна, яка показує, який із процесів `CS1` і `CS2` почав виконувати протокол входу останнім. Якщо обидва процеси намагаються увійти в критичні секції, тобто `in1` і `in2` істинні, виконання останнього з них призупиняється.

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
   while (true) {
      last = 1; in1 = true;          # вхід
```

```
        ⟨await (!in2 || last == 2);⟩
        критична секція;
        in1 = false;                    # вихід
        некритична секція;
    }
}
process CS2 {
    while (true) {
        last = 2; in2 = true;           # вхід
        ⟨await (!in1 || last == 1);⟩
        критична секція;
        in1 = false;                    # вихід
        некритична секція;
    }
}
```

Алгоритм програми дуже близький до дрібномодульних розв'язків, для якого не потрібні оператори `await`. Зокрема, якщо всі оператори `await` задовольняють умові "не більше одного" то їх можна реалізувати у вигляді циклів активного очікування. На жаль, оператори `await` звертаються до двох змінним, кожну з яких змінює інший процес. Однак в даному випадку немає необхідності в неподільному обчисленні умов затримки [10], а тому кожний оператор `await` можна замінити циклом `while`, який повторюється, поки умова закінчення затримки хибна. Таким чином, отримуємо дрібномодульний алгоритм розриву вузла:

```
# дрібномодульний алгоритм розриву вузла
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
    while (true) {
        last = 1; in1 = true;           # вхід
        while (in2 && last == 1) skip
        критична секція;
        in1 = false;                    # вихід
        некритична секція;
    }
}
process CS2 {
    while (true) {
        last = 2; in2 = true;           # вхід
        while (in1 && last == 1) skip
        критична секція;
        in1 = false;                    # вихід
        некритична секція;
```

```
        }
    }
```

У цій програмі вирішується проблема критичних секцій для двох процесів. Таку ж ідею можна використовувати при будь-якому числі процесів. Зокрема, для кожного з `n` процесів протокол входу повинен складатися з циклу, який проходить `n-1` етапів. На кожному етапі використовуються екземпляри алгоритму розриву вузла для двох процесів, щоб визначити, які процеси проходять на наступний етап. Якщо гарантується, що всі `n-1` етапів може пройти не більше, ніж один процес, то в критичній секції одночасно буде знаходитися не більше одного процесу.

Нехай `in[l: n]` та `last[l: n]` — цілочислові масиви. Значення елемента `in[i]` показує, який етап виконує процес `CS[i]`. Значення `last[j]` показує, який процес останнім почав виконувати етап `j`. Внутрішній цикл по `j` процесу `CS[i]` перевіряє всі інші процеси. Процес `CS[i]` чекає, якщо деякий інший процес знаходиться на етапі з рівним або більшим номером етапу, а процес `CS[i]` був останнім процесом, що досягли етапу `j`. Як тільки етапу `j` досягне ще один процес, або всі процеси "перед" процесом `CS[i]` вийдуть зі своїх критичних секцій, процес `CS[i]` отримає можливість виконуватися на наступному етапі.

```
int in[l: n]; int last[l: n]; # масиви з нульовими елементами
process CS[i = 1 to n] {
    while (true) {
        for [j = 1 to n] # протокол входу
            # процес i знаходиться на етапі j та є там останнім
            last[j] = i; in[i] = j;
            for [k = 1 to n && i != k] {
            /* Чекати, якщо процес k знаходиться на етапі з
            більшим номером та процес i був останнім з
            минулих на етап j* /
                while (in[k] > = in[i] && last[j] == i) skip;
            }
        }
        критична секція;
        in[i] = 0; # Протокол виходу
        некритична секція;
    }
}
```

Таким чином, не більше n-1 процесів можуть пройти перший етап, n-2 — другий і так далі. Це гарантує, що пройти всі n етапів і виконувати свою критичну секцію процеси можуть тільки по одному.

Розв'язок для n процесів уникає стан активного тупика, непотрібні затримки і гарантує можливість входу. Ці властивості випливають з того, що даний процес затримується, тільки якщо деякий інший процес знаходиться в протоколі входу попереду даного, і з припущення, що кожен процес врешті-решт виходить зі своєї критичної секції.

8.4.2. Алгоритм квитка

Алгоритм розриву вузла для *n* процесів достатньо складний та малозрозумілий. Алгоритм квітка пропонує більш прозорий розв'язок задачі. Назва пов'язана із тим, що він заснований на витягуванні квитків (номерів) та наступному очікуванні черги.

Нехай number та next — цілі змінні з початковими значеннями 1, а turn[1: n] — масив цілих чисел, початкові значення яких рівні нулю. Щоб увійти в критичну секцію, процес CS[i] спочатку присвоює елементу turn[i] поточне значення number та збільшує number на 1. Для того, щоб процеси отримували унікальні номери, ці дії мають бути неподільні. Після цього CS[i] очікує, поки значення next не стане рівним отриманому ним номеру. При завершенні критичної секції CS[i] неподільною дією збільшує на 1 значення next.

```
# Алгоритм квитка — крупномодульний розв'язок
int number = 1, next = 1, turn[l:n];
process CS[i = 1 to n] {
    while (true) {
        ⟨turn[i] = number; number = number + 1;⟩
        ⟨await (turn[i] == next);⟩
        критична секція;
        ⟨next = next + 1;⟩
        некритична секція;
    }
}
```

У попередній програмі значення елементів масиву turn унікальні. Оператор затримки

```
⟨await (turn[i] == next);⟩
```

гарантує, що тільки один turn[i] рівний next, тобто тільки один процес може знаходитися у своїй критичній секції.

Алгоритм квитка має потенційний недолік: значення `number` та `next` не обмежені, що може призвести до арифметичного переповнення.

У програмі використовуються три крупномодульні дії. Деякі процесори мають інструкції, які повертають попереднє значення змінної та збільшують (зменшують) її в одній неподільній дії. Прикладом є інструкція "отримати та збільшити" `FA(var, incr)` (Fetch-and-Add), з використанням якої можна отримати наступну версію програми:

```
# Алгоритм квитка — дрібномодульний розв'язок
int number = 1, next = 1, turn[l:n];
process CS[i = 1 to n] {
    while (true) {
        turn[i] = FA(number,1);        # протокол входу
        while (turn[i] != next) skip;
        критична секція;
        next = next + 1;               # протокол виходу
        некритична секція;
    }
}
```

Оператор `next = next + 1` без кутових дужок, що є допустимим, оскільки значення `next` змінюється тільки тим єдиним процесом, який увійшов у критичну секцію.

8.4.3. Алгоритм поліклініки

Алгоритм квитка можна безпосередньо реалізовувати лише на машинах із інструкцією "отримати та збільшити". *Алгоритм поліклініки*, схожий на алгоритм квитка. Він забезпечує справедливість планування і не вимагає спеціальних машинних інструкцій. За алгоритмом квитка кожен відвідувач отримує унікальний номер і чекає, поки значення `next` не стане рівним цьому номеру. Алгоритм поліклініки використовує інший підхід. Входячи, відвідувач дивиться на всіх інших і вибирає номер, більший за будь-який інший. Всі відвідувачі повинні чекати, поки назвуть їх номер. Як і в алгоритмі квитка, наступним обслуговується відвідувач з найменшим номером. Відмінність полягає в тому, що для визначення черговості обслуговування відвідувачі порівнюють номери один одного та не використовують загальний лічильник.

Як і в алгоритмі квитка, нехай `turn[1: n]` — цілочисловий масив з початковими значеннями 0. Щоб увійти в критичну секцію, процес `CS[i]` спочатку присвоює змінної `turn[i]` значення, яке на 1 більше, ніж максимальне серед поточних значень елементів масиву `turn`. Потім `CS[i]` очікує, поки значення `turn[i]` не стане най-

меншим серед ненульових елементів масиву turn. Виходячи з критичної секції, процес CS[i] присвоює turn[i] значення 0.

```
# Алгоритм поліклініки — крупномодульний розв'язок
int turn[l:n];
process CS[i = 1 to n] {
    while (true) {
        ⟨turn[i] = max(turn[l:n]) + 1;⟩
        for [j = 1 to n && j != i]
            ⟨await (turn[j] == 0 || turn[i] < turn[j]);⟩
        критична секція;
        turn[i] = 0;
        некритична секція;
    }
}
```

Перша неподільна дія забезпечує унікальність всіх ненульових значень у масиві turn.

Ненульові значення елементів масиву turn унікальні і, як зазвичай, передбачається, що кожен процес врешті-решт виходить зі своєї критичної секції, тому взаємних блокувань немає. Відсутні також зайві затримки процесів, оскільки відразу після виходу процесу CS[i] з критичної секції turn[i] набуває значення 0. Нарешті, алгоритм поліклініки гарантує можливість входу в критичну секцію, якщо планування справедливе в слабкому сенсі. Значення елементів масиву turn продовжують зростати, тільки якщо завжди є хоча б один процес, який намагається увійти в критичну секцію. Алгоритм поліклініки можна безпосередньо реалізувати на сучасних машинах. Щоб присвоїти значення turn[i], необхідно знайти максимальне з n значень, а оператор await двічі звертається до спільної змінної turn[j]. Ці операції можна було б реалізувати неподільним чином, використовуючи ще один протокол критичної секції, наприклад, алгоритм розриву вузла, але це занадто неефективно. До щастя, є більш простий вихід. Якщо необхідно синхронізувати n процесів, корисно спочатку розробити розв'язок для двох процесів, а потім узагальнити його (так робилося у випадку алгоритму розриву вузла).

Розглянемо наступний протокол входу для процесу CS1:

```
turn1 = turn2 + 1;
while (turn2 ! = 0 and turn1 > turn2) skip;
```

Аналогічний і наступний протокол входу для процесу CS2.

```
turn2 = turn1 + 1;
```

```
while (turn 1! = 0 and turn2 > turn1) skip;
```

Процеси можуть почати виконання своїх протоколів входу приблизно одночасно, і обидва присвоять змінним turn1 і turn2 значення 1. Якщо це трапиться, обидва процеси виявляться в своїх критичних секціях в один і той же час. Частково вирішити цю проблему можна за аналогією з дрібномодульним алгоритмом розриву вузла для двох процесів: якщо обидві змінні turn1 та turn2 мають значення 1, то один з процесів повинен виконуватися, а інший — припинятися.

На жаль, обидва процеси все ще можуть одночасно виявитися в критичній секції. Припустимо, що процес CS1 зчитує значення turn2 і отримує 0. Процес CS2 починає виконувати свій протокол входу, визначає, що змінна turn1 все ще має значення 0, присвоює turn2 значення 1 і входить в критичну секцію. У цей момент CS1 може продовжити виконання свого протоколу входу, присвоїти turn1 значення 1 і потім увійти в критичну секцію, оскільки змінні turn1 і turn2 мають значення 1, і процес CS1 в цьому випадку отримує перевагу. Така ситуація називається *станом гонитви*, оскільки процес CS1 "обганяє" CS2 і не враховує, що процес CS2 змінив змінну turn2. Щоб уникнути стану гонитви, необхідно, щоб кожен процес присвоював своїй змінній turn значення 1 (або будь-яке відмінне від нуля) на самому початку протоколу входу. Після цього процес повинен перевірити значення змінної turn інших процесів і переприсвоїти значення своєї змінної, тобто протокол входу процесу CS1 виглядає наступним чином.

```
turn1 = 1; turn1 = turn2 + 1;
while (turn2 ! = 0 && turn1 > turn2) skip;
```

Протокол входу процесу CS2 аналогічний.

```
turn2 = 1; turn2 = turn1 + 1;
while (turn1 ! = 0 && turn2 >= turn1) skip;
```

Тепер один процес не може вийти з циклу while, поки інший не закінчить розпочате раніше присвоювання turn. У цьому розв’язку процесу CS1 віддається перевага перед CS2, коли у обох процесів ненульові значення змінної turn.

Протоколи входу процесів несиметричні, оскільки умова затримки другого циклу трохи відрізняється. Однак їх можна записати і в симетричному вигляді. Нехай (a, b) та (c, d) — пари цілих чисел. Визначимо відношення порівняння для них таким чином:

```
(a,b) > (c,d) == true, якщо a > c або (a == c та b > d)
```

Тепер можна переписати умову `turn1 > turn2` процесу `CS1` у вигляді `(turn1,1) > > (turn2,2)`, а умову `turn2 > = turn1` у процесі `CS2` — `(turn2,2) > (turn1,1)`. Перевага симетричного запису в тому, що тепер алгоритм поліклініки для двох процесів легко узагальнити на випадок `n` процесів.

Кожен з процесів спочатку показує, що він збирається увійти в критичну секцію, присвоюючи своїй змінній `turn` значення 1. Потім він знаходить максимальне значення з усіх `turn[i]` і додає до нього 1. Нарешті, процес запускає цикл `for` і, як в крупномодульному розв'язку, очікує своєї черги. Відзначимо, що максимальне значення масиву визначається зчитуванням всіх його елементів і вибором найбільшого. Ці дії не є неподільними, тому точний результат не гарантується. Однак, якщо кілька процесів отримують одне й те саме значення, вони упорядковуються відповідно до правила, описаного вище.

```
# Алгоритм поліклініки — дрібномодульний розв'язок
int turn[l:n];
process CS[i = 1 to n] {
    while (true) {
        turn[i] = max(turn[l:n]) + 1;
        for [j = 1 to n && j != i]
            while(turn[j] == 0 && (turn[i],i) > (turn[j],j)
            skip
        критична секція;
        turn[i] = 0;
        некритична секція;
    }
}
```

8.5. Бар'єрна синхронізація

Основною властивістю більшості паралельних ітераційних алгоритмів є залежність результатів кожної ітерації від результатів попередньої. Один із способів побудувати такий алгоритм — реалізувати тіло кожної ітерації, використовуючи оператори `co`. Якщо вважати, що на кожній ітерації виконується `n` задач, отримаємо такий загальний вигляд алгоритму:

```
while (true) {
    co [i = 1 to n]
        код розв'язку задачі i;
    oc
}
```

На жаль, цей підхід досить неефективний, оскільки оператор `co` породжує `n` процесів на кожній ітерації. Створювати і знищувати процеси набагато дорожче, ніж реалізувати їх синхронізацію. Тому альтернативна структура алгоритму робить його набагато ефективнішим — процеси створюються один раз на початку обчислень, а потім синхронізуються в кінці кожної ітерації.

```
process Worker[i = 1 to n] {
    while (true) {
        код розв'язку задачі i;
        очікування завершення усіх задач;
    }
}
```

Точка затримки в кінці кожної ітерації є *бар'єром*, якого для продовження роботи повинні досягти всі процеси, тому цей механізм називається *бар'єрною синхронізацією*. Бар'єри можуть знадобитися в кінці циклів або на проміжних стадіях.

Нижче розглянуто кілька реалізацій бар'єрної синхронізації, що використовують різні способи взаємодії процесів.

8.5.1. Спільний лічильник

Найпростіший спосіб описати вимоги до бар'єра — використовувати лічильник `count` з нульовим початковим значенням. Припустимо, що є `n` робочих процесів, які повинні зібратися біля бар'єру. Коли процес доходить до бар'єру, він збільшує значення `count`. Коли значення `count` стане рівним `n`, всі процеси зможуть продовжити роботу:

```
process Worker[i = 1 to n] {
    while (true) {
        код розв'язку задачі i;
        ⟨count = count + 1;⟩
        ⟨await (count == n);⟩
    }
}
```

Однак даний код не повною мірою відповідає поставленому завданню. Складність полягає в тому, що значенням `count` повинен бути 0 на початку кожної ітерації, тобто `count` потрібно обнуляти кожного разу, коли всі процеси пройдуть бар'єр. Більш того, вона повинна мати значення 0 перед тим, як будь-який з процесів знову спробує її збільшити. Цю проблему можна вирішити за допомогою двох лічильників, один з яких збільшується до `n`, а інший зменшується до 0. Їх ролі міняються місцями після кожної

стадії. Однак використання спільних лічильників призводить до чисто практичних труднощів. По-перше, збільшувати і зменшувати їх значення потрібно неподільним чином. По-друге, коли процес призупиняється, він безперервно перевіряє значення змінної `count`. У гіршому випадку `n-1` процесів чекатимуть, поки останній процес досягне бар'єру. В результаті виникне серйозний конфлікт звернення до пам'яті, якщо тільки програма не виконується на мультипроцесорній машині з узгодженою кеш-пам'яттю. Крім того, число `n` повинно бути відносно малим.

8.5.2. Керуючі процеси

Один із способів уникнути конфліктів звернення до пам'яті — реалізувати лічильник `count` за допомогою `n` змінних, значення яких додаються до одного і того ж значення. Нехай є масив цілих чисел `arrive[1: n]` з нульовими початковими значеннями. Замінимо операцію збільшення лічильника `count` в програмі так: `arrive[i] = 1`. Якщо елементи масиву `arrive` зберігаються в різних рядках кеш-пам'яті, то конфліктів звернення до пам'яті не буде. Залишилося реалізувати оператор `await` і обнулити елементи масиву `arrive` в кінці кожної ітерації. Оператор `await` можна записати в такому вигляді.

```
⟨await ((arrive[l] + ... + arrive[n]) == n);⟩
```

Але в такому випадку знову виникають конфлікти звернення до пам'яті, причому це рішення також неефективне, оскільки суму елементів `arrive[i]` тепер постійно обчислює кожний процес `Worker`, який очікує продовження.

Обидві проблеми можна вирішити, використовуючи додатковий набір спільних значень і ще один процес, `Coordinator`. Нехай кожен процес `Worker` замість того, щоб додати усі елементи масиву `arrive`, чекає, поки не стане істинним логічне значення. Нехай `continue[1: n]` — додатковий масив цілих з нульовими початковими значеннями. Після того як `Worker[i]` присвоїть 1 елементу `arrive[i]`, він повинен чекати, поки значенням змінної `continue[i]` не стане 1.

```
⟨arrive[i] = 1; (await (continue[i] == 1);⟩
```

Процес `Coordinator` очікує, поки всі елементи масиву `arrive` не стануть рівні 1, потім присвоює значення 1 всім елементам масиву `continue`.

```
for [i = 1 to n] ⟨await (arrive[i] == 1);⟩
```

```
for [i = 1 to n] continue[i] = 1;
```

Оскільки для продовження процесів `Worker` повинні бути встановлені всі елементи `arrive`, процес `Coordinator` може перевіряти їх в будь-якому порядку. Конфліктів звернення до пам'яті тепер не буде, оскільки процеси очікують зміни різних змінних, кожна з яких може зберігатися в своєму рядку кеш-пам'яті.

Змінні `arrive` та `continue` є прикладами так званого "*прапорця*". Його встановлює один процес, щоб повідомити інші про виконання умови синхронізації. Використовуються два основних правила *синхронізації за допомогою прапорів*:

а) прапор синхронізації скидається тільки процесом, що очікує його установки;

б) прапор не можна встановлювати до тих пір, поки невідомо точно, що він скинутий.

Перше правило гарантує, що прапор не буде скинутий, поки процес не визначить, що він встановлений. Відповідно до цього правила прапор `continue[i]` має скидатися процесом `Worker[i]`, а обнуляти всі елементи масиву `arrive` має `Coordinator`. Згідно з другим правилом один процес не встановлює прапор, поки він не скинутий іншим. В іншому випадку, якщо інший синхронізований процес надалі очікує повторного встановлення прапорця, можливе взаємне блокування. Це означає, що `Coordinator` повинен скинути `arrive[i]` перед установкою `continue[i]`. `Coordinator` може також скинути `arrive[i]` відразу після того, як дочекався його установки. Додавши код скидання прапорів, отримаємо бар'єр з керуючим процесом:

```
# Бар’єрна синхронізация з керуючим процессом
int arrive[l:n], continue[l:n];
process Worker[i = 1 to n] {
    while (true) {
        код розв’язку задачі i;
        arrive[i] = 1;
        ⟨await (continue[i] == 1);⟩
        continue[i] = 0;
    }
}

process Coordinator {
    while (true) {
        for [i = 1 to n] {
            ⟨await (arrive[i] == 1);⟩
            arrive[i] = 0;
        }
```

```
            for [i = 1 to n]
                continue[i] = 1;
        }
    }
```

Хоча в програмі бар'єрна синхронізація реалізована так, що конфлікти звернення до пам'яті виключаються, у даного розв’язку є дві небажані властивості. По перше, потрібен додатковий процес. Синхронізація з активним очікуванням ефективна, якщо кожний процес виконується на окремому процесорі, так що процесу `Coordinator` потрібен свій власний процесор. Але, можливо, було б краще використовувати цей процесор для іншого робочого процесу. Другий недолік використання керуючого процесу полягає в тому, що час виконання кожної ітерації процесу `Coordinator`, і, отже, кожного екземпляра бар’єрної синхронізації пропорційно числу процесів `Worker`. В ітераційних алгоритмах часто все робочі процеси мають ідентичний код. Це означає, що якщо кожний робочий процес виконується на окремому процесорі, то всі вони підійдуть до бар'єра приблизно в один час. Таким чином, всі прапори `arrive` будуть встановлені практично одночасно. Однак процес `Coordinator` перевіряє прапори в циклі, по черзі чекаючи, коли кожен з них буде встановлено.

Обидві проблеми можна подолати, об'єднавши дії керуючого і робочих процесів так, щоб кожний робочий процес був одночасно і керуючим. Організуємо робочі процеси в дерево (рис. 8.1). Сигнал про те, що процес підійшов до бар'єра (прапор `arrive[i]`), відсилається вгору по дереву, а сигнал про дозвіл продовження виконання (прапор `continue[i]`) — вниз. Вузол робочого процесу чекає, коли до бар'єра підійдуть його сини, після чого повідомляє батьківський вузол про те, що він теж підійшов до бар'єра. Коли все сини кореневого вузла підійшли до бар'єра, це означає, що все інші робочі вузли теж підійшли до бар'єра. Тоді кореневої вузол може повідомити нащадкам, що вони можуть продовжити виконання. Ті, в свою чергу, дозволяють продовжити виконання своїх си-

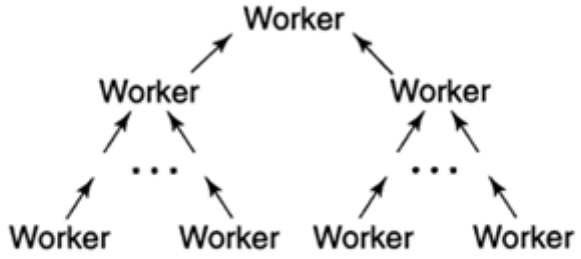

Рис. 8.1. Бар’єр з деревоподібною структурою

нів, і так далі.

```
# Бар'єрна синхронізація за допомогою об'єднуючого дерева
лист L: arrive [L] = 1;
          ⟨await (continue[L] == 1);⟩
          continue[L] = 0;
проміжковий вузол I: ⟨await (arrive[left] == 1);⟩
                           arrive[left] = 0;
                           ⟨await (arrive[right] == 1)⟩
                           arrive[right] = 0;
                           arrive[I] = 1;
                           ⟨await (continue[I] == 1);⟩
                           continue[I] = 0;
                           continue[left] = 1;
                           continue[right] = 1;
кореневий вузол R:   ⟨await (arrive[left] == 1)⟩
                           arrive[left] = 0;
                           ⟨await (arrive[right] == 1);⟩
                           arrive[right] = 0;
                           continue[left] = 1;
                           continue[right] = 1;
```

Отримана реалізація називається *бар'єром з об'єднуючим деревом*, оскільки кожен процес об'єднує результати роботи своїх дочірніх процесів і відправляє батьківському. Цей бар'єр використовує тільки ж змінних, скільки і "централізована" версія з керуючим процесом, але він набагато ефективніше при великих `n`, оскільки висота дерева пропорційна $\log_2 n$. Оператори `await` в даному випадку можна реалізувати у вигляді циклів активного очікування.

8.6. Алгоритми, паралельні за даними

8.6.1. Паралельні префіксні обчислення

Часто буває потрібно застосувати деяку операцію до всіх елементів масиву. Наприклад, щоб обчислити середнє значення числового масиву `a[n]`, потрібно спочатку додати всі елементи масиву, а потім поділити суму на `n`. Іноді потрібно отримати середні значення для всіх префіксів `a[0: i]` масиву. Паралельні префіксні обчислення використовуються в багатьох задачах, включаючи обробку зображень, матричні обчислення і аналіз регулярних мов.

Розглянемо, як паралельно обчислюються суми всіх префіксів масиву. Ця операція називається паралельним префіксним обчисленням. Базовий алгоритм може бути

використаний з будь-яким асоціативним бінарним оператором (додавання, множення, логічні оператори, обчислення максимуму та інші). Нехай дано масив `a[n]` і потрібно обчислити `sum[n]`, де `sum[i]` означає суму перших `i` елементів масиву `a`. Очевидний послідовний розв'язок:

```
sum[0] = a[0];
for [i = 1 to n-1]
    sum[i] = sum[i-1] + a[i];
```

Тепер подивимося, як цей алгоритм можна розпаралелити. Спочатку присвоїмо всім елементам `sum[i]` значення `a[i]`. Потім паралельно додамо `sum[i-1]` та `sum[i]` для всіх, тобто додамо всі елементи, які знаходяться на відстані 1. Тепер подвоїмо відстань і додамо елементи `sum[i-2]` з `sum[i]`. Якщо продовжувати подвоювати відстань, то після $\lceil \log_2 n \rceil$ кроків будуть обчислені всі часткові суми. Наступна таблиця ілюструє кроки алгоритму для масиву з шести елементів.

Початкові значення елементів `a`	1	2	3	4	5	6
Значення `sum` на відстані 1	1	3	5	7	9	11
Значення `sum` на відстані 2	1	3	6	10	14	18
Значення `sum` на відстані 4	1	3	6	10	15	21

Нижче наведена реалізація цього алгоритму. Кожен процес спочатку ініціалізує один елемент масиву `sum`, а потім циклічно обчислює часткові суми. Процедура `barrier(i)`, що викликається в програмі, реалізує точку бар'єрної синхронізації, аргумент `i` — ідентифікатор процедури процесу. Вихід з процедури відбувається, коли усі `n` процесів виконають команду `barrier`. У тілі процедури може бути використаний один з алгоритмів, описаних в попередньому підрозділі. (Бар'єри можна оптимізувати, оскільки на кожному кроці синхронізуються тільки два процеси.)

```
Обчислення часткових сум елементів масиву
process Sum[i = 0 to n-1] {
    int d = 1;
    sum[i] = a[i]; /* ініціалізація елементів sum */
    barrier(i);
    while (d < n) {
        old[i] = sum[i]; # зберегти попереднє значення
        barrier(i);
        if (i-d >= 0)
            sum[i] = old[i-d] + sum[i];
        barrier(i);
```

```
            d = d+d; # подвоїти відстань
        }
    }
```

Точки бар'єрів потрібні для усунення взаємного впливу процесів. Наприклад, всі елементи масиву `sum` повинні бути проініціалізовані до того, як який-небудь процес звернеться до них. Цей алгоритм можна використовувати з будь-яким асоціативним бінарним оператором. Програму можна адаптувати для числа процесів меншого за `n`; тоді кожен процес буде відповідати за об'єднання часткових сум смуги масиву.

8.6.2. Операції зі зв'язаними списками

При роботі з структурами даних типу дерев для пошуку і вставки елементів за логарифмічний час часто використовуються збалансовані бінарні дерева. Проте при використанні алгоритмів, паралельних за даними, багато операцій навіть з лінійними списками можна реалізувати за логарифмічний час. Покажемо, як знайти кінець зв’язаного списку. Цей же алгоритм можна використовувати і для інших операцій над зв’язаними списками, наприклад, вставки елемента в список пріоритетів або поелементного порівняння двох списків.

Припустимо, що є зв’язаний список (див. рис. 8.2), що містить не більше `n` елементів. Зв’язки зберігаються в масиві `link[n]`, а дані — в масиві `data[n]`. На початок списку вказує змінна `head`. Якщо елемент `i` є частиною списку, то `head == i`, або `link[j] == i` для деякого `j` від `0` до `n-1`. Поле `link` останнього елемента списку рівне `null`. Припустимо, що список вже ініціаллізований.

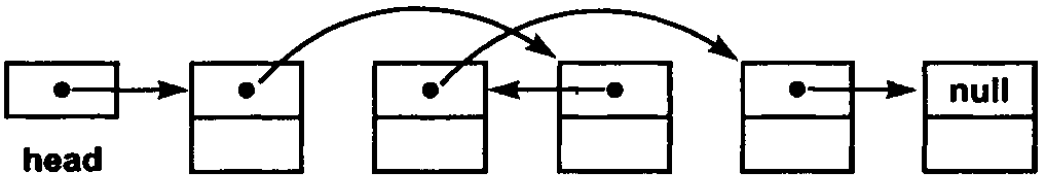

Рис. 8.2. Будова списку

Задача полягає в тому, щоб знайти кінець списку. Стандартний послідовний алгоритм починає роботу з початку списку `head` і рухається за посиланнями, поки не знайде порожній покажчик. Час роботи цього алгоритму пропорційно довжині списку.

Проте пошук кінця списку можна виконати за логарифмічний час, якщо використовувати алгоритм, паралельний за даними, і метод подвоєння з попереднього пункту.

Кожному елементу списку призначається процес `Find`. Нехай `end[n]` спільний масив цілих чисел. Якщо елемент `i` є частиною списку, то завдання процесу `Find[i]` — присвоїти змінній `end[i]` значення, рівне індексу останнього елемента списку, інакше процес `Find[i]` повинен присвоїти `end[i]` значення `null`. Припустимо, що список містить хоча б два елементи. На початку роботи кожен процес записує у `end[i]` значення `link[i]`, тобто індекс наступного елемента списку (якщо він є). Потім процеси виконують ряд етапів. На кожному етапі процес розглядає елемент з індексом `end[end[i]]`. Якщо елементи `end[i]` та `end[end[i]]` — не порожні вказівники, то процес присвоює елементу `end[i]` значення `end[end[i]]`. Далі процес повторюються. Таким чином, після *d*-го циклу змінна `end[i]` буде вказувати на елемент списку, що знаходиться на відстані в 2^{d-1} від *i*-го (якщо такий є). Після $\lceil \log_2 n \rceil$ циклів кожен процес знайде кінець списку.

```
# Пошук кінця послідовного звязаннго списку
process Find[i = 0 to n-1] {
    int new, d = 1;
    end[i] = link[i]; # ініціалізація елементів
    barrier(i);
    while (d < n) {
        new = null; # перевірити, чи треба обновити end[i]
        if (end[i] != null && end[end[i]] != null)
            new = end[end[i]];
        barrier(i);
        if (new != null)
            end[i] = new;
        barrier(i);
        d = d≪1;
    }
}
```

Для ілюстрації роботи алгоритму у випадку списку із 6 елементів розглянемо рис. 8.3. Після третьої ітерації кожний елемент вказує на кінець списку.

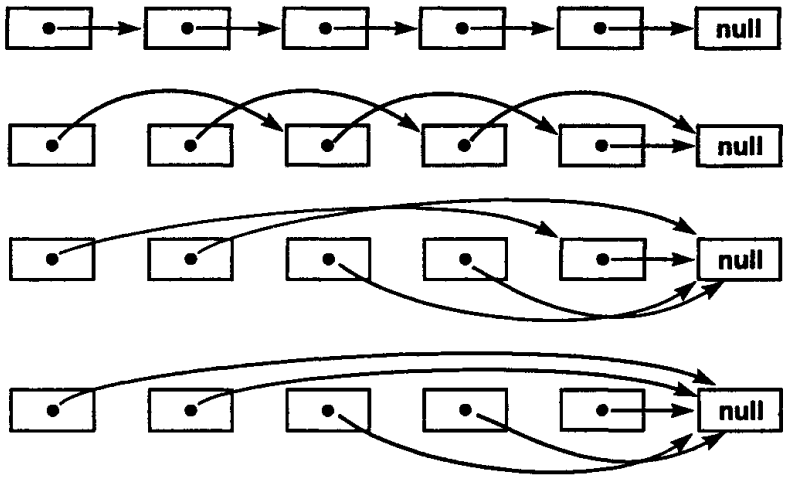

Рис. 8.3. Паралельний пошук кінця списку

8.7. Паралельні обчислення з портфелем задач

Розглянемо ще один спосіб реалізації паралельних обчислень, в якому використовується так званий *портфель задач*. Задача є незалежною одиницею роботи. Задачі поміщаються в портфель, спільний для кількох робочих процесів. Кожен робочий процес виконує наступний основний код.

```
while (true) {
    отримати задачу з портфеля;
    if (задач більше нема)
        break;
    виконати задачу (можливо, породжуючи нові задачі);
}
```

Цей підхід можна використовувати для реалізації рекурсивного паралелізму; тоді задачі будуть являти собою рекурсивні виклики. Його також можна використовувати для ітеративних задач з фіксованим числом незалежних завдань.

Парадигма портфеля завдань має кілька корисних властивостей. По-перше, вона дуже проста у використанні. Досить визначити зображення задачі, реалізувати портфель, запрограмувати виконання задачі і з’ясувати, як розпізнається завершення роботи алгоритму. По друге, програми, що використовують портфель задач, є масштабованими в тому сенсі, що їх можна використовувати з будь-яким числом процесорів; для цього достатньо просто змінити кількість робочих процесів. (Однак продуктивність програми при цьому може і не змінюватися.) І, нарешті, ця парадигма спрощує реалізацію балансування навантаження. Якщо тривалості виконання задач різні,

то, ймовірно, деякі із задачі будуть виконуватися довше інших. Але поки задач більше, ніж робочих процесів (в два-три рази), загальні обсяги обчислень, здійснюваних робочими процесорами, будуть приблизно однаковими.

Покажемо, як за допомогою портфеля задач реалізується множення матриць. При множенні матриць використовується фіксоване число задач. Для захисту доступу до портфелю застосовуються критичні секції, а для виявлення закінчення — бар'єроподібна синхронізація.

Припустимо, що програма буде виконуватися на машині з числом процесорів `PR`. Тоді бажано використовувати `PR` робочих процесів, по одному на кожен процесор. Щоб збалансувати обчислювальну завантаження, процеси повинні обчислювати приблизно порівну проміжних добутків. Кожний робочий процес буде захоплювати завдання при необхідності. Якщо число PR набагато менше, ніж `n`, то відповідний для задачі обсяг роботи — один або кілька рядків результуючої матриці `c`. Для простоти використовуємо окремі рядки. У початковому стані портфель містить `n` задач, по одному на рядок. Робочий процес отримує задачу з портфеля, виконуючи неподільну дію

```
⟨row = nextRow; nextRow++;⟩
```

Тут `row` — локальна змінна. Портфель порожній, коли значення `row` не менш за `n`. У програмі передбачається, що матриці ініціалізовані. Програма завершується, коли всі робочі процеси вийдуть з циклу `while`. Для визначення цього моменту можна скористатися лічильником `done` з нульовим початковим значенням. Перед тим як робочий процес виконає оператор `break`, він повинен збільшити значення лічильника в неподільній дії. Змінна `done` використовується в якості бар'єру-лічильника.

```
# Множення матриць за допомогою портфеля задач
int nextRow = 0; # портфель задач
process Worker[w = 1 to RP] {
    int row;
    double sum; # для проміжних добутків
    while (true) {
        # отримати задачу
        ⟨ row = nextRow; nextRow++; ⟩
        if (row >= n)
            break;
        обчислити внутрішні добутки для c[row, * ];
    }
}
```

9. СЕМАФОРИ

9.1. Синтаксис та семантика

Семафор — це особливий тип спільної змінної, яка обробляється тільки двома *неподільними* операціями `P` і `V`. Семафор можна вважати екземпляром класу семафор, його операції — методами цього класу з додатковим атрибутом неподільності.

Значення семафора є *невід'ємним цілим числом*. Операція `V` використовується для сигналізації про те, що подія відбулася, тому вона збільшує значення семафора. Операція `P` призупиняє процес до моменту, коли відбудеться деяка подія, тому вона, дочекавшись, коли значення семафора стане додатним, зменшує його. Сила семафорів обумовлена тим, що виконання операції `P` може бути призупинено.

Семафор оголошується так:

```
sem s;
```

За замовчуванням початковим значенням є 0, але семафор можна ініціалізувати будь-яким додатним значенням, наприклад:

```
sem lock = 1;
```

Масиви семафорів можна оголошувати і при необхідності ініціалізувати таким чином:

```
sem forks[5] = ([5] 1);
```

Якби в цій декларації не було ініціалізації, то початковим значенням кожного семафора в масиві `forks` був би 0.

Після оголошення та ініціалізації семафор можна обробляти тільки за допомогою операцій `P` і `V`. Кожна з них є неподільним дією з одним аргументом. Нехай `s` — семафор. Тоді операції `P(s)` та `V(s)` визначаються наступним чином.

```
P(s): ⟨await (s > 0) s = s - 1;⟩
V(s): ⟨s = s + 1;⟩
```

Припустимо, що `s` — семафор з поточним значенням 1. Якщо два процеси намагаються одночасно виконати операцію `P(s)`, то це вдасться зробити тільки одному з них. Але якщо один процес намагається виконати операцію `P(s)`, а інший — `V(s)`, то обидві операції будуть успішно виконані в непередбачуваному порядку, а кінцевим значенням семафора `s` стане 1.

Звичайний семафор може приймати будь-які невід'ємні значення, *двійковий семафор* — тільки значення 1 або 0. Це означає, що операція `V` для двійкового семафора може бути виконана, тільки коли його значення 0. Властивості справедливості для операцій з семафором випливають з їх визначення з допомогою оператора `await`. Якщо умова `s > 0` стає і надалі залишається істинною, виконання операції `P(s)` завершиться при справедливій в слабкому сенсі стратегії планування. Якщо умова `s > 0` стає істинною нескінченно часто, то виконання операції `P(s)` завершиться при справедливій в сильному сенсі стратегії планування. Операція `V` для звичайного семафора є безумовною неподільною дією, тому вона завершиться, якщо стратегія планування безумовно справедлива.

9.2. Основні задачі та методи

Семафори безпосередньо підтримують реалізацію взаємного виключення. Крім того, вони забезпечують підтримку простих форм умовної синхронізації, де вони використовуються для сигналізації про події. Для вирішення складніших задач ці два способи застосування семафорів можна комбінувати.

9.2.1. Критичні секції: взаємне виключення

Семафори були придумані в томі числі і для того, щоб полегшити вирішення задачі критичної секції. Саме ці операції і підтримуються семафором. Нехай `mutex` — семафор з початковим значенням 1. Виконання операції `P(mutex)` — це те саме, що і очікування, поки значення змінної `lock` (яка використовувалася у розв’язку задачі критичної секції у попередньому розділі) не стане рівним 1, і подальше присвоєння їй значення 0. Аналогічно виконання операції `V(mutex)` — це те саме, що присвоєння `lock` значення 1 (за умови, що це можна зробити, тільки коли вона має значення 0).

```
# Розв’язок задачі критичної секції з використанням семафорів
sem mutex = 1;
process CS[i = 1 to n] {
    while (true) {
        P(mutex);
        критична секція;
        V(mutex);
        некритична секція;
    }
}
```

9.2.2. Бар'єри: сигналізація подій

Семафори полегшують реалізацію бар'єрної синхронізації. Основна ідея — використовувати семафор в якості прапора синхронізації. Виконуючи операцію `V`, процес встановлює прапор, а при операції `P` — скидає його.

Спочатку розглянемо задачу реалізації бар'єру для двох процесів. Нагадаємо, що необхідно виконати дві вимоги. По-перше, жоден процес не повинен перейти бар'єр, поки до нього не підійшли обидва процеси. По-друге, бар'єр повинен допускати багаторазове використання, оскільки зазвичай одні й ті ж процеси синхронізуються після кожного етапу обчислень. Для вирішення задачі критичної секції досить лише одного семафора для блокування, оскільки потрібно просто визначити, чи знаходиться процес в критичній секції. Але при бар'єрній синхронізації необхідні два семафора в якості сигналів, щоб знати, приходить процес до бар'єра або йде від нього.

Сигнальний семафор `s` — це семафор з нульовим (як правило) початковим значенням. Процес сигналізує про подію, виконуючи операцію `V(s)`; інші процеси очікують події, виконуючи `P(s)`. Для двохпроцесного бар'єру дві істотних події полягають у тому, що процеси прибувають до бар'єра. Отже, задачу можна вирішити за допомогою двох семафорів `arrive1` і `arrive2`. Кожен процес повідомляє про своє прибуття до бар'єра, виконуючи операцію `V` для свого семафора, і потім очікує прибуття іншого процесу, виконуючи для його семафора операцію `P`.

```
# Бар'єрна синхронізація за допомогою семафорів
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1); / * Сигнал про прибуття * /
    P(arrive2); / * Очікування іншого процесу * /
    ...
}
process Worker2 {
    ...
    V(arrive2); / * Сигнал про прибуття * /
    P(arrive1); / * Очікування іншого процесу * /
    ...
}
```

За допомогою аналогічного прийому можна реалізувати бар'єр для `n` процесів. Для цього знадобиться масив семафорів `arrive`. На кожному етапі процес і спочатку пові-

домляє про своє прибуття, виконуючи операцію `V(arrive[i])`, а потім очікує прибуття інших процесів, виконуючи `P` для їх елементів масиву `arrive`. На відміну від ситуації з змінними-прапорами тут потрібен тільки один масив семафорів `arrive`, оскільки дія операції `V` "запам'ятовується", тоді як значення прапора змінної може бути перезаписане. Семафори можна використовувати і в якості сигнальних прапорів в реалізації бар'єрної синхронізації для `n` процесів з керуючим процесом або деревом. Операції `V` запам'ятовуються, тому використовується менше семафорів, ніж прапорців змінних. У керуючому процесі `Coordinator`, наприклад, потрібен всього один семафор.

9.2.3. Виробники і споживачі: розділені семафори

В даному пункті знову розглядається задача про виробників і споживачів. Там передбачалося, що є тільки один виробник і один споживач. Тут розглядається загальний випадок, коли є кілька виробників і кілька споживачів.

У задачі про виробників і споживачів виробники посилають повідомлення споживачам. Процеси спілкуються за допомогою буфера та операцій `deposit` та `fetch`. Виконуючи операцію `deposit`, виробники поміщають повідомлення в буфер; споживачі отримують повідомлення за допомогою операції `fetch`. Щоб повідомлення не перезаписувалися і кожне з них отрималося тільки один раз, виконання операцій `deposit` і `fetch` має чергуватися, причому першою повинна бути `deposit`.

Запрограмувати необхідне чергування операцій можна за допомогою семафорів. Нехай `empty` (порожній) і `full` (повний) — два семафора, що відображають стан буфера. У початковому стані буфер порожній, семафор `empty` має значення 1 (тобто відбулася подія "спорожнити буфер"), а `full` — 0. Перед виконанням операції `deposit` виробник спочатку очікує спорожнення буфера. Коли виробник поміщає в буфер повідомлення, буфер стає заповненим. І, навпаки, перед виконанням операції `fetch` споживач спочатку очікує заповнення буфера, а потім спорожнює його. Процес очікує події, виконуючи операцію `P` для відповідного семафора, і повідомляє про подію, виконуючи `V`. Змінні `empty` і `full` є двійковими семафора. Разом вони утворюють так званий розділений двійковий семафор, оскільки в будь-який момент часу тільки один з них може мати значення 1. Термін "розділений двійковий семафор" пояснюється тим, що змінні `empty` і `full` можуть розглядатися як єдиний двійковий семафор, поділе-

ний на дві частини. У загальному випадку розділений двійковий семафор може бути утворений будь-яким числом двійкових семафорів.

```
# Виробники і споживачі, що використовують семафори
typeT buf; / * Буфер деякого типу T * /
sem empty = 1, full = 0;
process Producer [i = 1 to M] {
    while (true) {
        / * Помістити дані в буфер * /
        P(empty);
        buf = data;
        V(full);
    }
}
process Consumer[j = 1 to N] {
    while (true) {
        / * Одержати дані * /
        P(full);
        result = buf;
        V(empty);
    }
}
```

Кожен процес `Producer` позмінно виконує операції `P(empty)` та `V(full)`, а кожен процес-споживач `Consumer` — `P(full)` та `V(empty)`.

9.2.4. Кільцеві буфери: облік ресурсів

З останнього прикладу видно, як синхронізувати доступ до одного буферу обміну. Якщо дані виробництво і споживання яких приблизно з однаковою частотою, то процесу не доводиться довго чекати доступу до буферу. Однак зазвичай споживач і виробник працюють нерівномірно. Наприклад, виробник може швидко створити відразу кілька елементів, а потім довго обчислювати наступну серію елементів. У таких випадках збільшення ємності буфера може істотно підвищити продуктивність програми, зменшуючи число блокувань процесів.

Припустимо поки, що є тільки один виробник і тільки один споживач. Виробник поміщає повідомлення в спільний буфер, споживач отримує їх звідти. Буфер містить чергу уже розміщених, але ще не прочитаних повідомлень. Ця чергу може бути зображена зв'язаним списком або масивом. Реалізуємо буфер масивом `buf[n]`, де `n > 1`. Нехай змінна `front` є індексом першого повідомлення черзі, а `rear` — індексом пер-

шої порожньої комірки після повідомлення в кінці черги. Спочатку змінні `front` і `rear` мають однакові значення, скажімо, 0. Виробник поміщає в буфер повідомлення з значенням `data`, виконавши такі дії:

```
buf[rear] = data;
rear = (rear + 1)% n;
```

Аналогічно споживач отримує повідомлення в свою локальну змінну `result`, виконуючи дії:

```
result = buf[front];
front = (front + 1)% n;
```

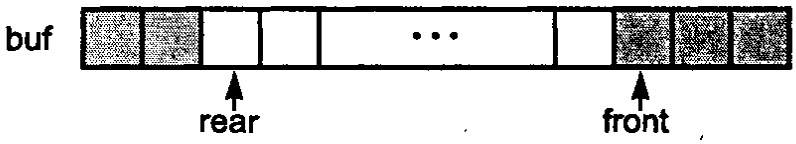

Рис. 9.1. Схема кільцевого буфера

Черга буферизованих повідомлень зберігається в комірках від `buf[front]` до `buf[rear]` (не включно). Змінна `buf` інтерпретується як кільцевий масив, в якому за `buf[n-1]` йде `buf[0]`.

Якщо використовується тільки один буфер (як в схемі "виробник-споживач"), то виконання операцій `depos` та `fetch` має чергуватися. При наявності кількох буферів операцію `deposit` можна виконати, якщо є порожня комірка, а `fetch` — якщо збережено хоча б одне повідомлення. Вимоги синхронізації для одноелементного і кільцевого буфера *однакові*. Зокрема, операції `P` і `V` застосовуються одним і тим же чином. Єдина відмінність полягає в тому, що семафор `empty` ініціалізується значенням `n`, а не 1, оскільки в початковому стані є `n` порожніх комірок.

```
# Кільцевій буфер з використанням семафорів
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0; /* n-2 <= empty + full <= n */
process Producer {
    while (true) {
        створити повідомлення data;
        / * Помістити data в буфер * /
```

```
            P(empty);
            buf[rear] = data; rear = (rear + 1)% n;
            V(full);
        }
    }
    process Consumer {
        while (true) {
            / * Отримати повідомлення result * /
            P(full);
            result = buf[front]; front = (front + 1)% n;
            V(empty);
        }
    }
```

Семафори грають роль *лічильників ресурсів*: кожен враховує кількість елементів ресурсу: `empty` — число порожніх комірок буфера, а `full` — заповнених. Коли жодний процес не виконує `deposit` або `fetch`, сума значень обох семафорів дорівнює загальному числу комірок `n`.

У попередній програмі передбачалося, що є тільки один виробник і один споживач. Це гарантувало неподільне виконання операцій `deposit` та `fetch`. Припустимо, що є кілька процесів-виробників. При наявності хоча б двох вільних комірок два з них могли б виконати операцію `deposit` одночасно. Але тоді обидва спробували б помістити свої повідомлення в одну і ту ж комірку! Аналогічно, якщо є кілька споживачів, два з них одночасно можуть виконати `fetch` і отримати одне і те ж повідомлення. Таким чином, `deposit` і `fetch` стають критичними секціями. Однакові операції повинні виконуватися з взаємним виключенням, але різні можуть виконуватися одночасно, оскільки при роботі семафорів `empty` та `full` виробники і споживачі звертаються до різних комірок буфера.

```
# Кілька виробників і споживачів, що використовують семафори
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0;    # n-2 <= empty + full <= n
sem mutexD = 1, mutexF = 1; # для взаємного виключення
process Producer [i = 1 to M] {
    while (true) {
        створити повідомлення data;
        / * Помістити data в буфер * /
        P(empty);
        P(mutexD);
```

```
        buf[rear] = data; rear = (rear + 1)% n;
        V(mutexD);
        V(full);
    }
}
process Consumer [j = 1 to N] {
    while (true) {
        / * Прочитати повідомлення result * /
        P(full);
        P(mutexF);
        result = buf[front]; front = (front + 1)% n;
        V(mutexF);
        V(empty);
    }
}
```

9.3. Задача про обід філософів

У попередньому підрозділі було показано, як використовувати семафори для вирішення задачі критичної секції. На основі цього розв’язку будується реалізація вибіркового взаємного виключення для двох класичних задач: про філософів, що обідають і про читачів та письменників. Розв’язок задачі про філософів ілюструє реалізацію взаємного виключення між процесами, що конкурують за доступ до множин спільних змінних, задача про читачів та письменників — реалізацію комбінації паралельного та виняткового доступу до спільних змінним.

Хоча задача про філософів скоріше цікава, ніж практична, вона аналогічна до реальних проблем, в яких процесу необхідний одночасний доступ більш, ніж до одного ресурсу. Тому її часто використовують для ілюстрації та порівняння різних механізмів синхронізації.

П'ять філософів сидять біля круглого столу. Вони проводять життя, чергуючи прийоми їжі та роздуми. У центрі столу знаходиться велике блюдо спагеті. У процесі їжі філософи повинні користуватися двома виделками. На жаль, їм дали всього п’ять виделок. Між кожною парою філософів лежить одна виделка і вони домовилися, що кожен буде користуватися тільки тими виделками, які лежать поруч з ним (зліва і справа). Завдання — написати програму, що моделює поведінку філософів. Програма повинна уникати ситуації, в якій всі філософи голодні, але жоден з них не може взяти обидві виделки — наприклад, коли кожен з них тримає по одній вилці і не хоче віддавати її.

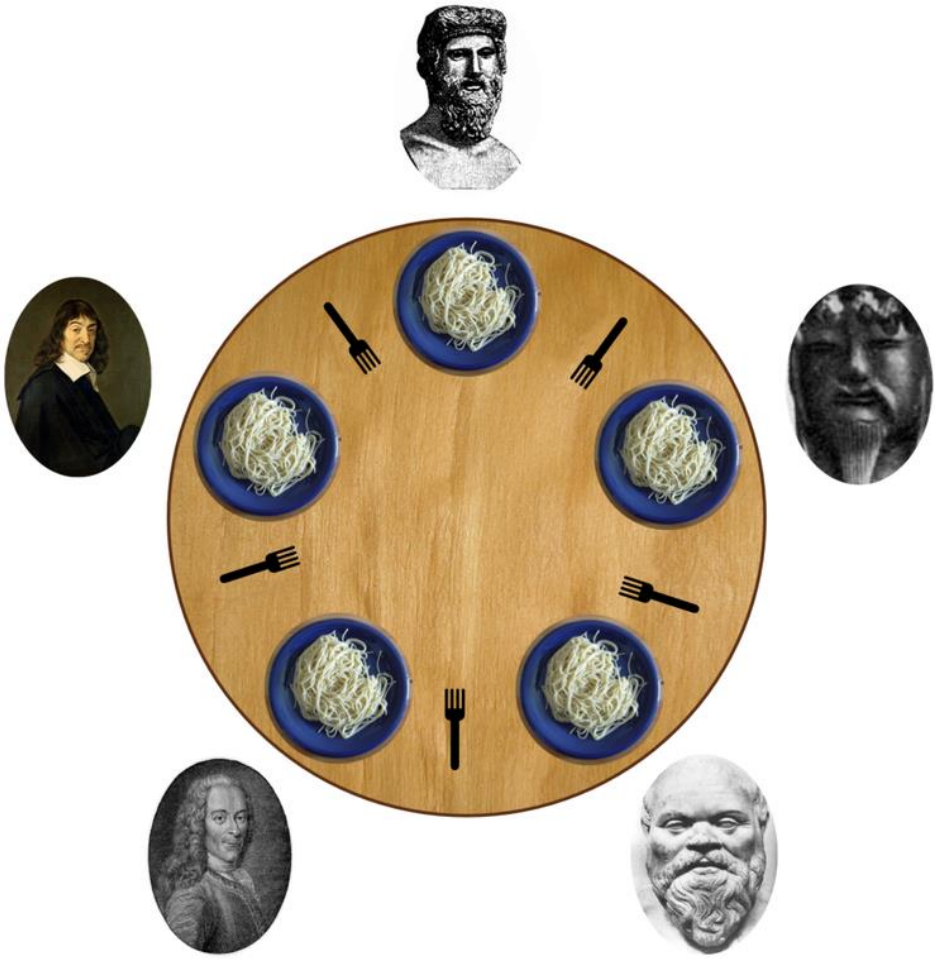

Рис. 9.2. Ілюстрація до задачі про обід філософів

Задача проілюстрована на рис. 9.2. Ясно, що два філософа, які сидять поруч, не можуть їсти одночасно. Крім того, оскільки виделок всього п’ять, одночасно можуть їсти не більше, ніж двоє філософів.

Припустимо, що періоди роздумів і прийомів їжі різні — для їх імітації в програмі можна використовувати генератор випадкових чисел. Змоделюємо поведінку філософів наступним чином.

```
process Philosopher [i = 0 to 4] {
    while (true) {
        поміркувати;
        взяти виделки;
        поїсти;
        віддати виделки;
    }
}
```

Для вирішення задачі потрібно запрограмувати операції "взяти виделки" та "віддати (звільнити) виделки". Виделки є ресурсом. Зосередимося на їх отриманні та звільненні.

Кожна виделка схожа на блокування критичної секції: в будь-який момент часу володіти нею може тільки один філософ. Отже, виделки можна уявити масивом семафорів, ініціалізованих значенням 1. Взяття виделки імітується операцією `P` для відповідного семафора, а звільнення — операцією `V`. Процеси, по суті, ідентичні, тому природно припускати, що вони виконують однакові дії. Наприклад, кожен процес може спочатку взяти ліву виделку, потім праву. Однак це може привести до взаємного блокування процесів. Наприклад, якщо все філософи візьмуть свої ліві виделки, то вони назавжди залишаться в очікуванні можливості взяти праву виделку.

Необхідна умова взаємного блокування — можливість кругового очікування, тобто коли один процес чекає ресурс, зайнятий другим процесом, який чекає ресурс, зайнятий третім, і так далі до деякого процесу, що очікує ресурс, зайнятий першим процесом. Таким чином, щоб уникнути взаємного блокування, досить забезпечити неможливість виникнення кругового очікування. Для цього можна змусити один з процесів, скажімо, `Philosopher[4]`, спочатку взяти праву виделку. Можливий також варіант рішення, в якому філософи з парним номером беруть виделки в одному порядку, а з непарним — у іншому.

```
# Розв'язок задачі про обід філософів з використанням семафорів
sem fork [5] = {1,1,1,1,1};
process Philosopher[i = 0 to 3] {
    while (true) {
        P(fork[i]);   # Взяти ліву виделку
        P(fork[i+1]); # потім праву
        поїсти;
        V(fork[i]); V(fork[i+1]);
        поміркувати;
    }
}
process Philosopher[4] {
    while (true) {
        P(fork[0]);   # Взяти праву виделку
        P(fork[4]);   # потім ліву
        поїсти;
        V(fork[0]); V(fork[4]);
        поміркувати;
    }
}
```

9.4. Задача про читачів та письменників

Задача про читачів та письменників — ще одна класична задача синхронізації. Її часто використовують для порівняння механізмів синхронізації. Вона також дуже важлива для практичного застосування. Формулювання: базу даних поділяють два типи процесів – читачі та письменники. Читачі виконують транзакції, які переглядають записи бази даних, а транзакції письменників і переглядають, і змінюють записи. Передбачається, що спочатку база даних знаходиться в несуперечливому стані. Кожна окрема транзакція переводить базу даних з одного несуперечливого стану в інший. Для запобігання взаємного впливу транзакцій процес-письменник повинен мати винятковий доступ до бази даних. Якщо до бази даних не звертається ні один з процесів-письменників, то виконувати транзакції можуть одночасно скільки завгодно читачів.

Задача про читачів і письменників — це приклад задачі вибіркового взаємного виключення. Класи процесів конкурують за доступ до бази даних. Процеси-читачі конкурують з письменниками, а окремі процеси-письменники — між собою. Це також приклад задачі загальної умовної синхронізації: процеси-читачі повинні чекати, поки до бази даних має доступ хоча б один процес-письменник; процеси-письменники повинні чекати, поки до бази даних мають доступ процеси-читачі або інший процес-письменник.

У цьому підрозділі дається два різних розв’язки задачі про читачів і письменників. У першому вона розглядається як задача взаємного виключення. Цей розв’язок є коротким, і його легко реалізувати. Однак в ньому читачі отримують перевагу перед письменниками і його важко модифікувати. У другому розв’язку задача розглядається як задача умовної синхронізації. Він здається більш складним, але насправді його теж легко реалізувати. Більш того, він легко змінюється для того, щоб реалізувати для читачів і письменників різні стратегії планування. В другому підході використовується потужний метод програмування, який називається передачею естафети та може застосовуватися для розв’язування довільної задачі умовної синхронізації.

9.4.1. Задача про читачів і письменників як задача виключення

Процесам-письменникам потрібен взаємовиключний доступ до бази даних. Доступ процесів-читачів як групи також повинен бути взаємовиключним по відношенню до будь-якого процесу-письменника. Корисний для будь-якої задачі вибіркового взаємного виключення підхід — спочатку ввести додаткові обмеження, реалізувавши

більше винятків, ніж потрібно, а потім послабити обмеження. Очевидне додаткове обмеження — забезпечити винятковий доступ до бази даних кожному читачеві і письменнику. Нехай змінна `rw` — це семафор взаємного виключення з початковим значенням `1`. В результаті отримаємо наступний розв'язок з додатковим обмеженням:

```
# Розв'язок задачі про читачів і письменників з додатковим обмеженням
sem rw = 1;
process Reader [i = 1 to M] {
    while (true) {
        P(rw); # Захопити блокування виключного доступу
        читати базу даних;
        V(rw); # Звільнити блокування
    }
}
process Writer [i = 1 to N] {
    while (true) {
        P(rw); # Захопити блокування виключного доступу
        записати в базу даних;
        V(rw); # Звільнити блокування
    }
}
```

Розглянемо, як послабити обмеження в програмі, щоб процеси-читачів могли працювати паралельно. Читачі як група повинні блокувати роботу письменників, але тільки перший читач повинен захопити блокування взаємного виключення, виконавши операцію `P(rw)`. Решта читачі можуть відразу звертатися до бази даних. Читач, закінчуючи роботу, повинен знімати блокування, тільки якщо є останнім активним процесом-читачем. Отримуємо наступний розв'язок:

```
# Схема розв'язку задачі про читачів і письменників
int nr = 0;   # Кількість активних читачів
sem rw = 1;   # Блокування доступу до бази даних
process Reader [i = 1 to M] {
    while (true) {
        ⟨nr = nr + 1;
        if (nr == 1) P(rw);⟩ # Отримати блокування, якщо перший
        читати базу даних;
        ⟨nr = nr-1;
        if (nr == 0) V(rw);⟩ # Зняти блокування, якщо останній
    }
}
```

Змінна nr слугує для підрахунку числа активних читачів. Щоб уникнути взаємного впливу процесів-читачів, додавання та перевірка повинні виконуватися як критична секція, тому для забезпечення неподільного виконання протоколу входу використані кутові дужки. Процеси Writer описуються без змін.

Тепер реалізуємо неподільні дії за допомогою семафорів. Нехай mutexR — семафор, що забезпечує взаємне виключення процесів-читачів.

```
# Розв'язок задачі про читачів і письменників за допомогою
семафорів
int nr = 0;       # Число активних читачів
sem rw = 1;       # Блокування доступу до бази даних
sem mutexR = 1;   # Блокування доступу читачів до nr
process Reader [i = 1 to m] {
    while (true) {
        P(mutexR);
        nr = nr + 1;
        if (nr == 1) P(rw); # Отримати блокування, якщо перший
        V(mutexR);
        читати базу даних;
        P(mutexR);
        nr = nr-1;
        if (nr == 0) V(rw); # Зняти блокування, якщо останній
        V(mutexR);
    }
}
process Writer [i = 1 to n] {
    while (true) {
        P(rw);
        записати в базу даних;
        V(rw);
    }
}
```

Алгоритм реалізує розв'язок задачі з перевагою читачів. Якщо деякий процес-читач звертається до бази даних, а інший читач і письменник досягають протоколів входу, то новий читач отримує перевагу перед письменником. Отже, це рішення не є справедливим, оскільки нескінченний потік процесів-читачів може постійно блокувати доступ письменників до бази даних.

9.4.2. Розв’язок задачі про читачів і письменників з використанням умовної синхронізації

Побудуємо інший розв’язок, почавши з більш простого визначення необхідної синхронізації. В цьому розв’язку буде використовуватися загальний метод програмування, який називається *передачею естафети* і використовує розділені двійкові семафори як для виключення, так і для сигналізації призупиненим процесам. Метод передачі естафети можна застосувати для реалізації будь-яких операторів типу `await` і, таким чином, для реалізації довільної умовної синхронізації.

Для збереження несуперечності (цілісності) бази даних письменникам необхідний винятковий доступ, але процеси-читачі можуть працювати паралельно в будь-якій кількості. Простий спосіб опису такої синхронізації полягає в підрахунку процесів кожного типу, які звертаються до бази даних, і обмеження значень лічильників. Наприклад, нехай `nr` і `nw` — змінні з невід'ємними цілими значеннями, що зберігають відповідно число процесів-читачів і процесів-письменників, які отримали доступ до бази даних. Схема основної частини процесу-читача виглядає так:

```
⟨nr = nr + 1;⟩
читати базу даних;
⟨nr = nr - 1;⟩
```

Відповідна схема процесу-письменника така:

```
⟨nw = nw + 1;⟩
записати в базу даних;
⟨nw = nw - 1;⟩
```

Щоб уточнити ці схеми, потрібно захистити операції присвоювання. У процесах-читачів для цього необхідно захистити збільшення `nr` умовою `(nw == 0)`, оскільки при збільшенні змінної `nr` значенням `nw` повинно бути 0. У процесах-письменниках необхідно дотримуватися умови `(nr == 0 && nw == 0)`. Однак в захисту операцій віднімання немає необхідності, оскільки *ніколи не потрібно затримувати процес, який звільняє ресурс*. Після врахування необхідних для захисту умов отримаємо крупномодульний розв’язок:

```
#Крупномодульний розв’язок задачі про читачів та письменників
int nr = 0, nw = 0;
process Reader [i = 1 to M] {
```

```
    while (true) {
        ⟨await (nw == 0) nr = nr+1;⟩
        читати базу даних;
        ⟨nr = nr-1;⟩
    }
}
process Writer [i = 1 to N] {
    while (true) {
        ⟨await (nr == 0 && nw == 0) nw = nw+1;⟩
        записати в базу даних;
        ⟨nw = nw-1;⟩
    }
}
```

9.4.3. Метод передачі естафети

Іноді оператори `await` можна реалізувати шляхом прямого використання семафорів або інших елементарних операцій, але в загальному випадку це неможливо. Розглянемо дві умови захисту операторів `await` у програмі з попереднього пункту. Ці умови перекриваються: умова захисту в протоколі входу письменника вимагає, щоб і `nw`, і `nr` дорівнювали 0, а в протоколі входу читача — щоб `nw` дорівнювала 0. Жоден семафор не може розрізнити ці умови, тому для реалізації таких операторів `await`, як зазначений тут, потрібен загальний метод. Викладений далі метод називається *передачею естафети*. Цей метод досить потужний, щоб реалізувати будь-який оператор `await`.

Для реалізації операторів `await` можна використовувати розділені двійкові семафори. Нехай `e` — двійковий семафор з початковим значенням 1. Він буде застосовуватися для управління входом в будь-яку неподільну дію.

З кожною умовою захисту зв'яжемо один семафор і один лічильник з нульовими початковими значеннями. Семафор будемо використовувати для припинення процесу до моменту, коли умова захисту стане істинною. У лічильнику буде зберігатися число припинених процесів. Оскільки є два різні умови захисту, по одному в протоколах входу письменників і читачів, то потрібні два семафора і два лічильника. Нехай `r` — семафор, пов'язаний з умовою захисту в процесі-читачі, а `dr` — відповідний йому лічильник призупинених процесів-читачів. Аналогічно нехай з умовою захисту в процесі письменника пов'язані семафор `w` і лічильник призупинених процесів-письменників `dw`. Для виходу з неподільних дій може використовуватися наступний код:

```
SIGNAL{
    if (nw == 0 && dr > 0) {
        dr = dr-1; V(r); # відновити процес-читач
    }
    else if (nr == 0 && nw == 0 && dw > 0) {
        dw = dw-1; V(w); # відновити процес-письменник
    }
    else
        V(e); # звільнити блокування входу
}
```

Роль коду `SIGNAL` — сигналізувати тільки одному з трьох семафорів. Це означає, що якщо немає активних письменників, але є призупинений читач, то він може бути продовжений за допомогою операції `V(r)`. Якщо немає активних читачів або письменників, але є призупинений письменник, то він може бути продовжений завдяки операції `V(w)`. Інакше, якщо немає відкладених процесів, які можна безпечно продовжити, то вхідний семафор отримає сигнал за допомогою операції `V(e)`.

```
# Схема читачів і письменників з передачею естафети
int nr = 0, nw = 0;
sem e = 1,        # керує входом в критичні секції
r = 0,            # використовується для призупинки читачів
w = 0;            # використовується для призупинки письменників
                  # завжди 0 <= (e + r + w) <= 1
int dr = 0;       # число призупинених читачів
int dw = 0;       # число призупинених письменників
process Reader [i = 1 to M] {
    while (true) {
        # реалізація ⟨await (nw == 0) nr = nr + 1;⟩
        P(e);
        if (nw > 0){
            dr = dr + 1;
            V(e);
            P(r);
        }
        nr = nr + 1;
        SIGNAL;
        читати базу даних;
        # реалізація ⟨nr = nr-1;⟩
        P(e);
        nr = nr-1;
        SIGNAL;
    }
}
```

```
process Writer [i = 1 to N] {
    while (true) {
        # реалізація ⟨await (nr == 0 and nw == 0) nw = nw + 1;⟩
        P(e);
        if (nr > 0 || nw > 0) {
            dw = dw + 1;
            V(e);
            P(w);
        }
        nw = nw + 1;
        SIGNAL;
        записати в базу даних;
        # реалізація ⟨nw = nw-1;⟩
        P(e);
        nw = nw-1;
        SIGNAL;
    }
}
```

Три семафора утворюють розділений двійковий семафор, оскільки в будь-який момент часу тільки один з них може мати значення 1, а всі гілки коду починаються операціями `P` і закінчуються операціями `V`. Отже, оператори між кожною парою `P` та `V` виконуються із взаємним виключенням. Перетворення коду не приводить до взаємного блокування, оскільки семафор затримки отримує сигнал, тільки якщо певний процес знаходиться в стані очікування або повинен в нього перейти. (Процес може збільшити лічильник очікуючих процесів і виконати операцію `V(e)`, але не може виконати операцію `P` для семафора затримки.)

Описаний метод програмування називається *передачею естафети* у зв'язку із способом вироблення сигналів семафора. Коли процес виконується всередині критичної секції, вважається, що він отримав естафету, яка підтверджує його право на виконання. Передача естафети відбувається, коли процес доходить до фрагмента програми `SIGNAL`. Якщо деякий процес очікує умову, яка тепер стала істинною, естафета передається одному з тих процесів, який в свою чергу виконує критичну секцію і передає естафету наступному процесу. Якщо жоден з процесів не очікує умови, естафета передається наступному процесу, який вперше намагається увійти в критичну секцію, тобто наступному процесу, що виконує `P(e)`.

9.5. Розподіл ресурсів та планування

Розподіл ресурсів — це задача визначення того, коли процес може отримати доступ до ресурсу. У паралельних програмах ресурсом є все те, при спробі отримання чого робота процесу може бути припинена. Сюди включається вхід в критичну секцію, доступ до бази даних, доступ до пам'яті, використання принтера і тому подібне. У більшості попередніх програм використовувалася найпростіша стратегія розподілу ресурсів: якщо певний процес чекає і ресурс вільний, то ресурс розподіляється. Наприклад, в задачі критичної секції дозвіл на вхід дається *якомусь* з очікуючих процесів; спроба визначити, *який саме* процес отримає дозвіл на вхід, не робиться. Лише в задачі про читачів та письменників розглядалася більш складна стратегія планування. Але там було надано перевагу класу процесів, а не окремим процесам.

9.5.1. Постановка задачі та загальна схема розв'язку

У будь-якій задачі розподілу ресурсів процеси конкурують за використання елементів ресурсу. Процес запитує один або кілька елементів, виконуючи операцію `request`, часто реалізовану у вигляді процедури. Параметри операції `request` вказують необхідну кількість елементів ресурсу, визначають особливі характеристики (наприклад, розмір блоку пам'яті) і ідентифікують процес, який зробив запит на ресурс. Запит може бути задоволений, тільки коли всі необхідні елементи вільні. Таким чином, процедура `request` очікує звільнення достатньої кількості елементів ресурсу, а потім повертає потрібні елементи. Після використання елементів ресурсу процес звільняє їх операцією `release`.

Загальна спрощена схема операцій `request` та `release` така:

```
request(параметри):
    ⟨await (запит може бути задоволений) отримати
    елементи;⟩
release(параметри):
    ⟨повернути елементи;⟩
```

Операції повинні бути неподільними, оскільки кожної з них необхідний доступ до подання елементів ресурсу.

Цю загальну схему рішення можна реалізувати за допомогою методу передачі естафети. Операція `request` має вигляд звичайного оператора `await`, тому реалізується наступним фрагментом програми.

```
request(параметри):
    P(e);
    if (запит не може бути задоволений) DELAY;
    отримати елементи;
    SIGNAL;
```

Операція `release` теж має вигляд простої неподільної дії і реалізується таким чином:

```
release(параметри):
    P(e);
    повернути елементи;
    SIGNAL;
```

Як і раніше, семафор `e` керує входом в критичну секцію, а `SIGNAL` запускає на виконання призупинені процеси (якщо очікуючий запит може бути задоволений) або знімає блокування семафора входу за допомогою операції `V(e)`. Код `DELAY` операції `request` аналогічний фрагментами коду на початку "Схема читачів і письменників з передачею естафети". Він запам'ятовує, що з'явився новий запит, який повинен бути призупинений, знімає блокування з семафора входу за допомогою операції `V(e)` і блокує процес на семафорі затримки.

9.5.2. Розподіл ресурсів за схемою "найкоротше завдання"

"Найкоротше завдання" (НЗ, Shortest-Job-Next) — це стратегія розподілу ресурсів, яка зустрічається у багатьох різновидах і використовується для різних типів ресурсів. Припустимо, що ресурс складається з одного елемента. Тоді стратегія *"найкоротше завдання"* визначається наступним чином:

Кілька процесів змагаються у використанні одного ресурсу. Процес запитує ресурс, виконуючи операцію `request(time, id)`, де цілочисловий параметр `time` визначає тривалість використання ресурсу цим процесом, а ціле число `id` ідентифікує процес. Якщо в момент виконання операції `request` ресурс вільний, він виділяється для процесу негайно, інакше процес призупиняється. Після використання ресурсу процес звільняє його, виконуючи операцію `release`. Звільнений ресурс розподіляється призупиненому процесу (якщо такий є) з найменшим значенням параметра `time`. Якщо у декількох процесів значення `time` рівні, то ресурс віддається тому з них, хто найдовше чекав.

Стратегію НЗ можна використовувати, наприклад, для розподілу процесорів (параметр `time` означатиме час виконання), для виведення файлів на принтер (`time` — час друку) або для обслуговування віддаленої передачі файлів по протоколу `ftp` (`time` — передбачуваний час передачі файлу).

Стратегія НЗ приваблива, оскільки мінімізує загальні витрати часу на виконання. Разом з тим, їй притаманне несправедливе планування: процес може бути припинений назавжди, якщо існує безперервний потік запитів з меншим часом використання ресурсу. Якщо несправедливість небажана, то можна трохи змінити стратегію НЗ, щоб перевага віддавалася процесам, які чекають занадто довго. Цей метод називається *витримкою*, або *старінням* (*aging*).

Ресурс один, тому для зберігання відомостей про його доступності досить однієї змінної. Нехай `free` – логічна змінна, яка істинна, коли ресурс доступний, і хибна, коли він зайнятий. Для реалізації стратегії НЗ потрібно запам'ятовувати і впорядковувати очікують запити. Нехай `pairs` — набір записів вигляду `(time, id)`, упорядкованих за значеннями поля `time`. Якщо два записи мають однакові значення поля `time`, то вони залишаються у множині `pairs` в порядку їх появи.

Нижче наведено крупномодульний розв'язок:

```
request(time, id):
    P(e);
    if (!free) DELAY;
    free = false;
    SIGNAL;
release():
    P(e);
    free = true;
    SIGNAL;
```

В операції `request` передбачається, що операції `P` над семафором входу `e` обслуговуються в порядку їх появи, тобто за правилом "першим прийшов — першим обслужений". Якщо цього немає, то порядок обробки запитів може не відповідати стратегії НЗ.

Залишилося реалізувати стратегію розподілу ресурсів НЗ. Для цього використовуємо впорядковану множину `pairs` і семафори, що реалізують фрагменти коду `SIGNAL` і `DELAY`. Якщо запит не може бути задоволений, його слід зберегти, щоб до

нього можна було повернутися після звільнення ресурсу. Таким чином, у фрагменті коду DELAY процес повинен:

- вставити параметри запиту в упорядковану множину pairs;
- звільнити управління критичної секцією, виконавши операцію V(e);
- зупинитися на семафорі до задоволення запиту.

Якщо після звільнення ресурсу pairs не є порожньою, то у відповідності зі стратегією НЗ тільки один процес повинен отримати ресурс. Таким чином, якщо є призупинений процес, який тепер може продовжити роботу, він повинен отримати сигнал за допомогою операції V для семафора затримки.

У кожного процесу є своє умова затримки, яка залежить від його позиції в наборі pairs: перший в pairs процес повинен бути запущений перед другим і так далі. Таким чином, кожний процес повинен очікувати на своєму семафорі затримки.

Припустимо, що ресурс використовують n процесів. Нехай b[n] — масив семафорів, кожний елемент якого має початкове значення 0. Будемо вважати, що значення ідентифікаторів процесів id унікальні і знаходяться в межах від 0 до n-1. Тоді процес id призупиняється на семафорі b[id]. Доповнивши операції request і release відповідної обробкою pairs і масиву b, отримаємо:

```
# Розв'язок задачі розподілу ресурсів за схемою НЗ на базі семафорів
bool free = true;
sem e = 1, b[n] = ([n] 0);        # для входу і затримки
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
request (time, id):
    P(e);
    if (!free) {
        вставити (time, id) в pairs;
        V(e);                     # зняти блокування входу
        P(b[id]);                 # очікувати відновлення
    }
    free = false;
    V(e);
release ():
    P(e);
    free = true;
    if (pairs != ∅) {
        видалити першу пару (time, id) з pairs;
```

```
        V(b[id]);                          # Передати
        естафету процесу id
    }
    else V(e);
```

Елементи масиву семафорів `b` є прикладом так званих *приватних семафорів*. Семафор s називається приватним, якщо операцію `P` над ним виконує тільки один процес.

Коли процес повинен бути призупинений, він виконує операцію `P(b[id])` для блокування на власному елементі масиву `b`.

Задача. Узагальнити попередню програму для використання ресурсів, що складаються з кількох елементів. (У цій ситуації кожен елемент може бути вільний або зайнятий, а операції `request` і `release` повинні використовувати параметр `amount`, що визначає необхідну кількість елементів ресурсу.)

10. МОНІТОРИ

Семафори є фундаментальним механізмом синхронізації. Їх використання полегшує програмування взаємного виключення і сигналізації. Однак семафори — низькорівневий механізм; користуючись ними, легко наробити помилок. Програміст повинен стежити за тим, щоб випадково не пропустити виклики операцій `P` і `V` або задати їх більше, ніж потрібно. Семафори глобальні по відношенню до всіх процесів, тому, щоб розібратися, як використовується семафор або інша колективна змінна, необхідно переглянути всю програму. Нарешті, при використанні семафорів взаємне виключення і умовна синхронізація програмуються однієї і тієї ж парою примітивів. Через це важко зрозуміти, для чого призначені конкретні `P` і `V`, не подивившись на інші операції з даними семафорами. Взаємне виключення і умовна синхронізація — це різні поняття, тому і програмувати їх краще різними способами.

Монітори — це програмні модулі, які забезпечують більшу структурованість, ніж семафори, хоча реалізуються так само ефективно. В першу чергу, монітори є механізмом абстракції даних. Монітор інкапсулює поняття абстрактного об’єкту і забезпечує набір операцій, тільки за допомогою яких він обробляється. Монітор містить змінні, що зберігають стан об'єкта, і процедури, що реалізують операції над ним. Процес отримує доступ до змінних в моніторі тільки шляхом виклику процедур цього монітора. Взаємне виключення забезпечується неявно тим, що процедури в одному моніторі не можуть виконуватися паралельно. Умовна синхронізація в моніторах забезпечується явно за допомогою *умовних змінних* (*condition variable*). Вони аналогічні семафорам, але мають істотні відмінності у визначенні та, отже, в використанні для сигналізації.

Паралельна програма, яка використовує монітори для взаємодії і синхронізації, містить два типи модулів: активні процеси і пасивні монітори. За умови, що всі колективні змінні знаходяться всередині моніторів, два процеси взаємодіють, викликаючи процедури одного і того ж монітора. Отримана модульність має два важливих переваги. Перше — процес, що викликає процедуру монітора, може не знати про конкретну реалізації процедури; роль грають лише видимі результати виклику процедури. Друге — програміст монітора може не піклуватися про те, де і як використовуються процедури монітора, і вільно змінювати його реалізацію. Ці переваги дозволяють розробляти процеси і монітори відносно незалежно, що полегшує створення та розуміння паралельної програми.

10.1. Синтаксис та семантика

Монітор використовується, щоб згрупувати зображення і реалізацію спільного ресурсу (класу). Він складається з інтерфейсу та тіла. Інтерфейс визначає надані ресурсом операції (методи). Тіло містить змінні, що описують стан ресурсу, і процедури, що реалізують операції інтерфейсу.

Надалі будемо вважати, що монітор є статичним об'єктом, а його тіло і інтерфейс описані так:

```
monitor mname {
    оголошення постійних змінних
    оператори ініціалізації
    процедури
}
```

Процедури реалізують видимі операції. Постійні змінні поділяються всіма процедурами тіла монітора. Вони називаються *постійними*, оскільки існують і зберігають своє значення, поки існує монітор. У процедурах, як зазвичай, можна використовувати локальні змінні, копії яких створюються для кожного виклику функції.

Монітор як представник абстрактних типів даних має три властивості. По перше, поза монітором видно тільки імена процедур. Щоб змінити стан ресурсу, процес повинен викликати одну з процедур монітора. Виклик процедури монітора має такий вигляд.

```
mname.opname(arguments)
```

Тут `mname` — ім'я монітора, `opname` — ім'я однієї з його операцій (процедур), що викликається з аргументами `arguments`.

По-друге, оператори всередині монітора (в оголошеннях і процедурах) *не можуть звертатися до змінних, оголошених поза монітором*.

По-третє, постійні змінні ініціалізуються до виклику його процедур.

10.1.1. Взаємне виключення

Синхронізацію найпростіше зрозуміти і запрограмувати, якщо взаємне виключення і умовна синхронізація виконуються різними способами. Найкраще, якщо взаємне виключення відбувається неявно, чим автоматично усувається взаємний вплив. Крім того, програми легше читати, оскільки в них немає явних протоколів входу в критичні секції і виходу з них.

На відміну від взаємного виключення, умовну синхронізацію потрібно програмувати явно, оскільки різні програми вимагають різних умов синхронізації. Хоча часто простіше синхронізувати за допомогою логічних умов, як в операторах `await`, низькорівневі механізми можна реалізувати набагато ефективніше.

Відповідно до цих зауваженнями взаємне виключення в моніторах *забезпечується неявно*, а умовна синхронізація програмується за допомогою так званих *умовних змінних*.

Зовнішній процес викликає процедуру монітора. Поки певний процес виконує оператори процедури, вона *активна*. У будь-який момент часу може бути активним тільки один екземпляр тільки однієї процедури монітора, тобто одночасно не можуть бути активними ні два виклики різних процедур, ні два виклики однієї і тієї ж процедури.

Процедури моніторів за визначенням виконуються із взаємним виключенням. Воно забезпечується реалізацією мови, бібліотекою або операційною системою, але не програмістом, який використовує монітори. На практиці взаємне виключення в мовах і бібліотеках реалізується за допомогою блокування та семафорів.

10.1.2. Умовні змінні

Умовна змінна використовується для припинення роботи процесу, безпечне виконання якого неможливо до переходу монітора в стан, що задовольняє деяку логічну умову. Умовні змінні також застосовуються для запуску призупинених процесів, коли певна логічна умова стає істинною. Умовна змінна оголошується наступним чином:

```
cond cv;
```

Таким чином, `cond` — це новий тип даних. Умовні змінні можна оголошувати і використовувати тільки в межах моніторів. Значенням умовної змінної `cv` є черга призупинених процесів (черга затримки). Спочатку вона порожня. Програміст не може безпосередньо звертатися до значення змінної `cv`. Замість цього він отримує непрямий доступ до черги за допомогою декількох спеціальних операцій, описаних нижче.

Процес може запросити стан умовної змінної за допомогою виклику

```
empty(cv);
```

Якщо черга змінної `cv` порожня, ця функція повертає значення `true`, інакше — `false`.

Процес блокується на умовній змінній `cv` за допомогою виклику

```
wait(cv);
```

Операція `wait` змушує працюючий процес затриматися в кінці черги змінної `cv`. Щоб інший процес міг врешті-решт увійти в монітор для запуску призупиненого процесу, виконання операції `wait` відбирає у процесу, що викликав її, винятковий доступ до монітора.

Процеси, заблоковані на умовних змінних, запускаються операторами `signal`. При виконанні виклику

```
signal(cv);
```

перевіряється черга затримки змінної cv. Якщо вона порожня, ніякі дії не відбуваються. Однак, якщо призупинені процеси є, оператор `signal` запускає процес *у початку черги*. Таким чином, операції `wait` та `signal` забезпечують порядок сигналізації FIFO: процеси припиняються в порядку викликів операції `wait`, а запускаються в порядку викликів операції `signal`.

10.1.3. Дисципліни сигналізації

Виконуючи оператор `signal`, процес працює в моніторі і, отже, може управляти блокуванням, неявно пов'язаним із монітором. В результаті виникає дилема. Якщо операція `signal` запускає інший процес, то виходить, що могли б виконуватися два процеси: той, що викликав операцію `signal` та запущений нею. Але наступним може виконуватися тільки один з них, оскільки лише один процес може мати винятковий доступ до монітора. Таким чином, можливі два варіанти:

- *сигналізувати і продовжити*: сигналізатор продовжує роботу, а процес, що отримав сигнал, виконується пізніше;
- *сигналізувати і очікувати*: сигналізатор чекає деякий час, а процес, який отримав сигнал, виконується відразу.

Дисципліна (порядок) "сигналізувати і продовжити" *не перериває обслуговування*. Процес, що виконує операцію `signal`, зберігає винятковий доступ до монітора, а процес, що запускається, почне роботу трохи пізніше, коли отримає винятковий доступ до монітора. По суті, операція `signal` просто вказує процесу, який запускається, на можливість виконання, після чого він повертається в чергу процесів, що очікують.

Порядок "сигналізувати і очікувати" має властивість *переривання обслуговування*. Процес, що виконує операцію `signal`, передає управління блокуванням монітора про-

цесу, який запускається, тобто переривається робота процесу-сигналізатора. В цьому випадку сигналізатор переходить в чергу процесів, які очікують на звільнення монітора.

Діаграма станів на рис. 10.1 ілюструє роботу синхронізації в моніторах. Викликаючи процедуру монітора, процес поміщається у вхідну чергу, якщо в моніторі виконується ще один процес; в іншому випадку процес, який викликав операцію, негайно починає виконання в моніторі. Коли монітор звільняється (після повернення з процедури або виконання операції `wait`), один процес з вхідної черги може перейти до роботи в моніторі. Виконуючи операцію `wait(cv)`, процес переходить від роботи в моніторі в чергу, пов'язану з умовною змінною. Якщо процес виконує операцію `signal(cv)`, то при порядку "Сигналізувати і продовжити" (Signal and Continue — SC) процес з початку черги умовної змінної переходить до вхідної. При порядку "сигналізувати і очікувати" (Signal and Wait — SW) процес, що виконується в моніторі, переходить до вхідної черги, а процес з початку черги умовної змінної переходить до виконання в моніторі.

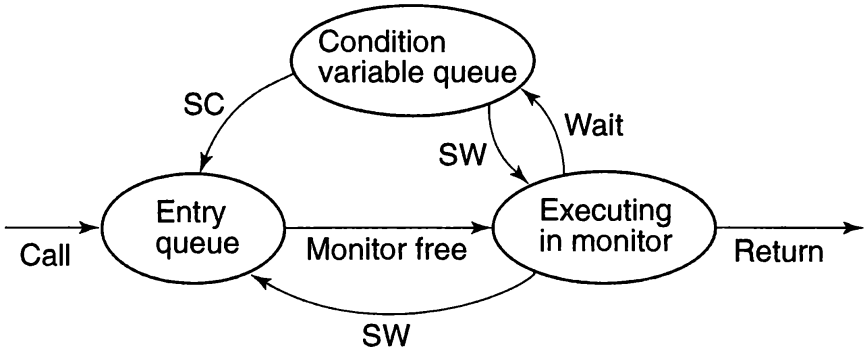

Рис. 10.1. Діаграма станів

Розглянемо приклад, який демонструє можливість реалізації семафор за допомогою монітора.

```
# Реалізація семафора за допомогою монітора
monitor Semaphore {
    int s = 0;
    cond pos; # отримує сигнал, коли s > 0
    procedure Psem() {
        while (s == 0) wait(pos);
```

```
        s = s-1;
    }
    procedure Vsem() {
        s = s+1;
        signal(pos);
    }
}
```

Викликаючи операцію `Psem`, процес призупиняється, поки значення змінної `s` не стане позитивним, потім зменшує його на 1. Затримка програмується за допомогою циклу `while`, який призводить процес до очікування на умовній змінній `pos`, якщо `s` дорівнює 0. Операція `Vsem` збільшує `s` на 1 і виробляє сигнал для змінної `pos`. Якщо є призупинені процеси, запускається "найстаріший" з них.

Програма коректно працює як при порядку "сигналізувати і очікувати" (SW), так і при "сигналізувати і продовжити" (SC). Порядки роботи відрізняються тільки послідовністю виконання процесів. При порядку SW процес, що запускається, виконується відразу та зменшує значення змінної `s`. При порядку SC процес, що запускається, виконується через деякий час *після* процесу, який виробив сигнал. Тому він має ще раз перевірити значення семафора щоб переконатися, що воно ще додатне. Це треба зробити для того, що можливо інший процес із вхідної черги викликав `Psem` і вже зменшив `s`. (Тому, на відміну від порядку SW, для SC ми не обов'язково отримуємо FIFO-семафор). Вищенаведений монітор можна змінити так, щоб він коректно працював при обох порядках запуску процесів (SC і SW), не використав цикл `while` та реалізовував семафор з порядком обслуговування FIFO. Як вже зазначалося проблема полягає у тому, що збільшене значення `s` може прочитати не той процес, який був запущений за допомогою `signal`. Вихід — виклик `VSem` у випадку наявності призупинених процесів не має збільшувати значення `s`. Такий метод називається *передачею умови*:

```
# Семафор FIFO з передачею умови
monitor FIFOsemaphore {
    int s = 0;
    cond pos; # Отримує сигнал, коли s > 0
    procedure Psem() {
        if (s == 0)
            wait(pos);
        else
            s = s-1;
    }
```

```
        procedure Vsem() {
            if (empty(pos))
                s = s + 1;
            else
                signal(pos);
        }
    }
```

Метод передачі умови можна застосовувати всюди, де в процедурах з викликами `wait` та `signal` є дії, що доповнюють одна одну. У програмі такими є збільшення змінної `s` в процедурі `Psem` та її зменшення в `Vsem`.

З попередньої програми видно, що умовні змінні аналогічні операціям `P` і `V` семафора. Операція `wait` призупиняє процес, а операція `signal` запускає його. Однак є дві істотні відмінності. Перша — операція `wait` завжди призупиняє процес до подальшого виконання операції `signal`, тоді як операція `P` викликає зупинку процесу, тільки якщо поточне значення семафора дорівнює нулю. Друге — операція `signal` не виробляє ніяких дій, якщо немає процесів, призупинених на умовній змінній, тоді як `V` або запускає призупинений процес, або збільшує значення семафора, тобто факт здійснення операції `signal` не запам'ятовується. Через ці відмінності умовна синхронізація з моніторами програмується не так, як з семафорами.

Надалі для моніторів *будемо використовують лише порядок SC*, який був прийнятий в ОС Unix, мові Java та бібліотеці Pthreads.

10.2. Методи синхронізації

10.2.1. Кільцеві буфери: базова умовна синхронізація

Повернемося до задачі про кільцевий буфер (див. підрозділ 7.2). Процес-виробник і процес-споживач взаємодіють за допомогою спільного буфера, який містить `n` комірок.

Нижче наведений монітор, який реалізує кільцевої буфер. Для черги повідомлень знову використані масив `buf` і дві цілочислові змінні `front` та `rear`. У змінній `count` зберігається кількість повідомлень в буфері. Операції з буфером `deposit` та `fetch` стають процедурами монітора. Умовна синхронізація реалізована за допомогою двох умовних змінних. Обидва оператора `wait` знаходяться в циклах.

```
# Реалізація кільцевого буфера за допомогою монітора
monitor Bounded_Buffer {
    typeT buf[n];    # масив деякого типу T
```

```
        int front = 0,   # індекс першої заповненої комірки
        rear = 0,        # індекс першої порожньої комірки
        count = 0;       # Число заповнених комірок
        # rear == (front + count)%n
        cond not_full;   # отримує сигнал, коли count < n
        cond not_empty;  # отримує сигнал, коли count > 0
        procedure deposit(typeT data) {
            while (count == n) wait(not_full);
            buf[rear] = data;
            rear = (rear+1)%n;
            count++;
            signal(not_empty);
        }
        procedure fetch(typeT& result) {
            while (count == 0) wait(not_empty);
            result = buf[front];
            front = (front+1)% n;
            count--;
            signal(not_full);
        }
    }
```

Виконуючи операцію `signal`, процес просто повідомляє, що тепер деяка умова істинна. Оскільки процес-сигналізатор і, можливо, інші процеси можуть виконуватися в моніторі до відновлення процесу, запущеного операцією `signal`, в момент початку його роботи умова запуску може вже не виконуватися. Наприклад, процес-виробник був припинений в очікуванні вільної комірки у буфері, потім процес-споживач зчитав повідомлення і запустив призупинений процес. Однак до того, як цьому виробникові прийшла черга виконуватися, інший процес-виробник міг уже увійти в процедуру `deposit` і зайняти вільну позицію. Аналогічна ситуація може виникнути із споживачами. Таким чином, умову припинення необхідно перевіряти ще раз. Оператори `signal` в процедурах `deposit` і `fetch` виконуються безумовно, оскільки в момент їх виконання умова, про яке вони сигналізують, є істиною. Оператори `wait` знаходяться в циклах, тому оператори `signal` можуть виконуватися в будь-який момент часу, оскільки вони просто підказують припиненим процесам. Однак програма виконується більш ефективно, коли `signal` виконується, тільки якщо відомо напевно (або хоча б з великою ймовірністю), що деякий призупинений процес може бути продовжений.

10.2.2. Читачі та письменники: сигнал сповіщення

Задача про читачів і письменників була наведена у підрозділі 9.4. Хоча база даних — загальний ресурс, її не можна уявити монітором, оскільки тоді читачі не зможуть працювати з нею паралельно (весь код усередині монітора виконується зі взаємним виключенням). Монітор використовується для упорядкування доступу до бази даних.

У задачі про читачів і письменників монітор дає дозвіл на доступ до бази даних. Для цього необхідно, щоб процеси інформували монітор про своє бажання отримати доступ і про завершення роботи з базою даних. Є два типи процесів і по два види дій на процес, тому отримуємо чотири процедури монітора:

`request_read`, `release_read`, `request_write`, `release_write`.

Для синхронізації доступу до бази даних необхідно вести облік числа процесів-читачів та письменників. Як і раніше, нехай значення змінної `nr` — це число читачів, а `nw` — письменників. У початковому стані `nr` і `nw` рівні 0. Їх значення збільшуються при виклику процедур запиту і зменшуються при виклику процедур звільнення.

Нижче наведено монітор, що відповідає цій специфікації.

```
# Розв'язок задачі про читачів та письменників
monitor RW_Controller {
    int nr = 0, nw = 0;
    # (nr == 0 || nw == 0) && nw <= 1
    cond oktoread;  # Отримує сигнал, коли nw == 0
    cond oktowrite; # Отримує сигнал, коли nr == 0 і nw == 0

    procedure request_read() {
        while (nw > 0) wait(oktoread);
        nr = nr + 1;
    }

    procedure release_read() {
        nr = nr - 1;
        if (nr == 0) signal(oktowrite);
        # запустити процес-письменник
    }

    procedure request_write() {
        while (nr > 0 || nw > 0) wait(oktowrite);
        nw = nw + 1;
    }

    procedure release_write() {
        nw = nw - 1;
        signal(oktowrite);     # запустити процес-письменник
        signal_all(oktoread); # запустити всі процеси-читачі
```

```
    }
}
```

Програма не встановлює порядок чергування процесів-читачів і процесів-письменників. Замість цього дана програма запускає *всі* призупинені процеси і дозволяє стратегії планування процесів визначити, який з них першим отримає доступ до бази даних. Якщо це процес-письменник, то припиняться *всі* запущені процеси-читачі. Якщо ж першим отримає доступ процес-читач, то призупиниться відновлений процес-письменник.

10.2.3. Розподіл ресурсів за схемою "найкоротше завдання": пріоритетне очікування

Умовна змінна за замовчуванням є FIFO-чергою, тому, виконуючи оператор `wait`, процес потрапляє в кінець черги очікування. Оператор пріоритетного очікування `wait(cv, rank)` має призупинені процеси в порядку зростання рангу. Він використовується для реалізації стратегій планування, відмінних від FIFO. Знову розглянемо задачу розподілу ресурсів за схемою "найкоротше завдання", наведену в 7.5.

Для розподілу ресурсів за схемою "найкоротше завдання" потрібні дві операції: `request` та `release`. У наступній програмі наведено монітор, який реалізує розподіл ресурсів згідно стратегії НЗ. Використовується логічна змінна `free` для індикації того, що ресурс вільний, і умовна змінна `turn` для припинення процесів.

Процедури використовують метод передачі умови. Пріоритетний оператор `wait` застосовується для сортування призупинених процесів за часом, протягом якого вони будуть використовувати ресурс. Функція `empty` використовується для перевірки, чи є призупинені процеси. Коли ресурс звільняється, при наявності призупинених процесів запускається той, якому потрібно найменше часу, інакше ресурс позначається як вільний. Якщо процес отримує сигнал, то відмітки про звільнення ресурсу не робиться, щоб інший процес не отримав до нього доступ першим.

```
# Розподіл ресурсів за стратегією НЗ з використанням моніторів
monitor Shortest_Job_Next {
    bool free = true;
    cond turn; # Отримує сигнал, коли ресурс доступний
    procedure request(int time) {
        if (free)
            free = false;
        else
            wait(turn, time);
```

```
    }
    procedure release() {
        if (empty(turn))
            free = true;
        else
            signal(turn);
    }
}
```

10.2.4. Інтервальний таймер: покриваючі умови

Звернемося до нової задачі — розробки таймера, який дозволяє процесу перейти в стан сну на кілька одиниць часу. Така можливість часто забезпечується операційними системами, щоб дозволити користувачам, наприклад, періодично виконувати службові команди. Розробимо два розв’язки, які ілюструють два корисних методи. У першому використані так звані покриваючі умови; у другому (для створення компактного і ефективного механізму затримки) — пріоритетний оператор `wait`.

Монітор, який реалізує інтервальний таймер, є прикладом контролера ресурсів. Ресурсом є логічні годинник. Можливі дві операції з годинником: `delay(interval)`, яка призупиняє процес на відрізок часу тривалістю `interval` "тіків" таймера, і `tick`, яка збільшує значення логічних годин.

Прикладні процеси викликають операцію `delay(interval)` з невід'ємним значенням `interval`. Операцію `tick` викликає процес, який періодично запускається апаратним таймером. Цей процес зазвичай має великий пріоритет виконання, щоб значення логічних годин залишалося точним.

Для подання значення логічних годин використовуємо цілочислову змінну `tod` (Time of day — час дня). Спочатку її значення дорівнює нулю. Процес, що викликає операцію `delay`, спочатку повинен обчислити бажаний час запуску. Це робиться за допомогою коду:

```
wake_time = tod + interval;
```

Тут змінна `wake_time` локальна по відношенню до тіла функції `delay`; отже, кожний процес, що викликає `delay`, обчислює власне значення часу запуску. Далі процес повинен очікувати, поки не буде достатня кількість разів викликана процедура `tick`. Для цього використовується цикл while з умовою закінчення `wake_time >= tod`.

Необхідну синхронізацію просто реалізувати, використовуючи одну умовну змінну і метод так званої *покриваючої умови*. Логічний вираз, пов’язаний з умовною змінною, "покриває" умови запуску всіх очікуючих процесів. Коли будь-яка з покриваючих умов виконується, запускаються всі процеси, що очікують. Кожен такий процес перевіряє ще раз свою умову і поновлюється або знову чекає.

У моніторі `Timer` можна використовувати одну умовну змінну `check`, пов'язану з покриваючою умовою "значення `tod` збільшено". Процеси очікують на зміну `check` в тілі функції `delay`. При кожному виклику процедури `tick` запускаються всі процеси, що очікують. У процедурі `tick` для запуску всіх припинених процесів використаний оператор `signal_all`.

```
# Інтервальний таймер з покриває умовою
monitor Timer {
    int tod = 0;
    cond check; # Отримує сигнал, коли tod збільшено
    procedure delay(int interval) {
        int wake_time;
        wake_time = tod + interval;
        while (wake_time > tod)
            wait(check);
    }
    procedure tick() {
        tod = tod + 1;
        signal_all(check);
    }
}
```

Розв’язок, наведений вище, не є ефективним для даної задачі. Застосування покриваючих умов підходить тільки для ситуацій, коли витрати на помилкові сигнали (запускається процес, який визначає, що його умова помилкова, і відразу повертається в стан очікування) менше, ніж витрати на обслуговування умов всіх очікуючих процесів і запуск тільки того процесу, для якого умова виконується. В даній ситуації ймовірно, що процеси затримуються на тривалий час і, отже, будуть без потреби багаторазово запускатися.

Використовуючи пріоритетний оператор `wait`, можна перетворити програму в більш просту і ефективну. Для цього використовуємо пріоритетний `wait` всюди, де є статична впорядкованість умов для різних очікуючих процесів. У даній ситуації процеси можна впорядкувати за часом їх запуску. Викликана процедура `tick` використовує

функцію minrank, щоб визначити, чи настав час запустити перший процес, який був загальмований на змінній check. Якщо так, цей процес отримує сигнал.

```
# Інтервальний таймер з пріоритетним очікуванням
monitor Timer {
    int tod = 0;
    cond check; # отримує сигнал, коли minrank(check) <= tod
    procedure delay (int interval){
        int wake_time;
        wake_time = tod + interval;
        if (wake_time > tod)
            wait(check, wake_time);
    }
    procedure tick(){
        tod = tod + 1;
        while (!empty (check) && minrank(check) <= tod)
            signal(check);
    }
}
```

10.2.5. Сплячий перукар: рандеву

В якості останнього базового прикладу розглянемо ще одну класичну задачу синхронізації: задачу про сплячого перукаря. Вона має практичні застосування, наприклад у плануванні роботи головки дискового накопичувача. Ця задача ілюструє важливість зв'язків типу клієнт-сервер, які часто існують між процесами. Для неї необхідний особливий тип синхронізації, так зване рандеву.

Задача про сплячого перукаря. У тихому містечку є перукарня з двома дверима і декількома кріслами. Відвідувачі входять через одні двері і виходять через іншу. Салон малий, і ходити по ньому може тільки перукар і один відвідувач. Перукар все життя обслуговує відвідувачів. Коли в салоні нікого немає, він спить у своєму кріслі. Коли відвідувач приходить і бачить сплячого перукаря, він будить його, сідає в крісло і спить, поки той зайнятий стрижкою. Якщо перукар зайнятий, коли приходить відвідувач, той сідає в одне з вільних крісел і засинає. Після стрижки перукар відкриває відвідувачеві вихідні двері і закриває її за ним. Якщо є очікуючі відвідувачі, перукар будить одного з них і чекає, поки той сяде в крісло перукаря. Якщо нікого немає, він знову йде спати до приходу наступного відвідувача.

Відвідувачі і перукар є процесами, що взаємодіють в моніторі — перукарні. Відвідувачі — це клієнти, які запитують сервіс (стрижку) у перукаря. Перукар — це сервер, постійно забезпечує сервіс.

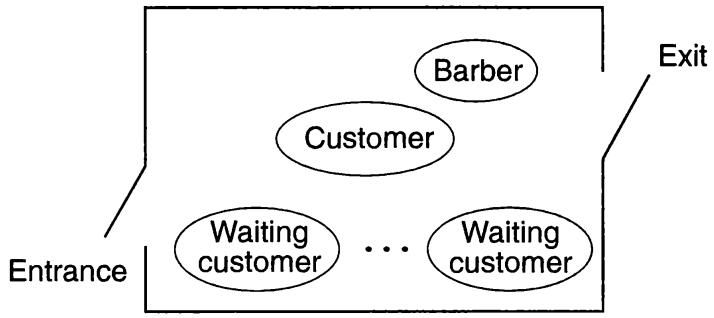

Рис. 10.2. Ілюстрація до задачі про сплячого перукаря

Для реалізації описаних взаємодій перукарню можна промоделювати монітором з трьома процедурами: `get_haircut` (постригтися), `get_next_customer` (покликати наступного) і `finished_cut` (закінчити стрижку). Відвідувачі викликають процедуру `get_haircut`; вихід з неї відбувається після того, як перукар закінчить стрижку даного відвідувача. Перукар циклічно викликає процедуру `get_next_customer`, запрошуючи клієнта в своє крісло, стриже його і випускає з перукарні за допомогою виклику процедури `finished_cut`. Постійні змінні служать для зберігання стану процесів і крісел, в яких процеси сплять. Дії перукаря і відвідувачів необхідно синхронізувати в моніторі. По перше, перукаря і відвідувачеві необхідна зустріч — *рандеву*, тобто перукар повинен дочекатися приходу відвідувача, а відвідувач — звільнення перукаря. Рандеву аналогічне бар'єру для двох процесів, оскільки для продовження роботи до нього повинні прийти обидві сторони. Однак рандеву відрізняється від двопроцесного бар’єру тим, що перукар може зустрітися з будь-яким з відвідувачів.

По-друге, відвідувачеві необхідно чекати, поки перукар закінчить його стригти, що визначається відкриттям вихідних дверей для відвідувача. Нарешті, перед тим, як закрити вихідні двері, перукар повинен почекати, поки піде відвідувач. Таким чином, перукар і відвідувач проходять через послідовність етапів, що починаються з рандеву.

Найпростіший спосіб визначити подібні етапи синхронізації — використовувати зростаючі лічильники для запам'ятовування числа процесів, які досягли кожного

етапу. У відвідувачів є два важливих етапи: перебування в кріслі перукаря і вихід з перукарні. Для цих етапів будемо використовувати лічильники `cinchair` і `cleave`. Перукар циклічно проходить через три етапи: звільнення від роботи, стрижка і завершення стрижки. Використовуємо для них лічильники `bavail`, `bbusy` і `bdone`. Всі лічильники в початковому стані мають значення нуль. Оскільки процеси проходять свої етапи послідовно, для лічильників виконується наступний інваріант:

```
cinchair >= cleave && bavail >= bbusy >= bdone
```

Щоб забезпечити рандеву відвідувача і перукаря перед початком стрижки, відвідувач не може сідати в крісло перукаря частіше, ніж перукар звільняється від роботи. Крім того, перукар не може починати стрижку частіше, ніж відвідувачі сідають в його крісло. Отже, виконується умова:

```
cinchair <= bavail && bbusy <= cinchair
```

Нарешті, відвідувачі не можуть виходити частіше, ніж перукар завершує стрижку:

```
cleave <= bdone
```

Зростаючі лічильники застосовуються для запам'ятовування етапів, через які проходять процеси, проте їх значення можуть зростати необмежено. Якщо синхронізація залежить тільки від різниці значень лічильників, зростання можна уникнути, змінивши змінні. У задачі є три ключові різниці, для яких виділимо три нові змінні `barber`, `chair` і `open`.

```
barber == bavail - cinchair
chair == cinchair - bbusy
open == bdone - cleave
```

Вони ініціалізуються 0, а під час роботи програми можуть приймати значення 0 або 1. Значення `barber` дорівнює 1, якщо перукар очікує відвідувача і сидить в своєму кріслі. Змінна `chair` має значення 1, якщо відвідувач вже сів в крісло, але перукар ще не зайнятий, а змінна `open` приймає значення 1, коли вихідні двері вже відкриті, але відвідувач ще не вийшов.

```
# Монітор для завдання про сплячому перукаря
monitor Barber_Shop {
    int barber = 0, chair = 0, open = 0;
```

```
    cond barber_available; # Отримує сигнал, коли barber > 0
    cond chair_occupied;    # Отримує сигнал, коли chair > 0
    cond door_open;         # Отримує сигнал, коли open > 0
    cond customer_left;     # Отримує сигнал, коли open == 0

    procedure get_haircut() {
        while (barber == 0)
            wait(barber_available);
        barber = barber - 1;
        chair = chair + 1; signal(chair_occupied);
        while (open == 0) wait(door_open);
        open = open - 1; signal(customer_left);
    }

    procedure get_next_customer() {
        barber = barber + 1;
        signal(barber_available);
        while (chair == 0)
            wait(chair_occupied);
        chair = chair - 1;
    }

    procedure finished_cut() {
        open = open + 1;
        signal(door_open);
        while (open > 0)
            wait(customer_left);
    }
}
```

У наведеному моніторі ми вперше бачимо процедуру `get_haircut`, що містить два оператора `wait`. Справа в тому, що відвідувач проходить через два етапи: спочатку він чекає, поки не звільниться перукар, потім — поки не закінчиться стрижка.

11. БАГАТОПОТОКОВЕ ПРОГРАМУВАННЯ ЗАСОБАМИ .NET FRAMEWORK

11.1. Базові можливості

11.1.1. М'ютекс

М'ютекс (mutex) — це взаємно виключний синхронізуючий об'єкт. Це означає, потоки можуть його отримати тільки по черзі. М'ютекс призначений для тих випадків, коли загальний ресурс може бути одночасно використаний тільки у одному потоці.

М'ютекс реалізований у класі `System.Threading.Mutex`. Він має кілька конструкторів. Нижче наведені два найбільш уживані:

```
public Mutex()
public Mutex(bool initiallyOwned)
```

Перший створює м'ютекс, яким ніхто не володіє. При виклику другого із параметром `true` м'ютексом заволодіває потік, у якому викликається конструктор.

Для того, щоб отримати м'ютекс, в коді програми треба викликати метод `WaitOne()` для цього м'ютекса. Метод `WaitOne()` успадковується класом `Mutex` від класу `Thread.WaitHandle`. Нижче наведена його найпростіша форма:

```
public bool WaitOne()
```

Метод `WaitOne()` очікує до тих пір, поки не буде отриманий м'ютекс, для якого він був викликаний. Тобто, цей метод блокує виконання викликаючого потоку, поки не стане доступним вказаний м'ютекс. Він завжди повертає логічне значення `true`. М'ютекс звільняється викликом методу `ReleaseMutex()`:

```
public void ReleaseMutex()
```

Це дає змогу іншим потокам отримати даний м'ютекс. Якщо потік вже отримав і ще не звільнив м'ютекс і повторно викликає метод `WaitOne()`, то він не заблоковується.

Схема використання м'ютекса:

```
Mutex myMutex = new Mutex();
// ........................
myMutex.WaitOne(); // чекати отримання м'ютекса
// Отримати доступ до ресурса
myMutex.ReleaseMutex(); // звільнити м'ютекс
```

Приклад. Розв'язування задачі про виробників та споживачів за допомогою м'ютекса.

11.1.2. Семафор

Семафор надає одночасний доступ до спільного ресурсу не одному, а кільком потокам. Семафор керує доступом до загального ресурсу, використовуючи лічильник. Якщо значення лічильника більше за нуль, то доступ до ресурсу дозволений. Якщо значення рівне нулю, то доступ заборонений.

Семафор реалізований класом `System.Threading.Semaphore`. Найпростіша форма конструктора класу:

```
public Semaphore(int initialCount, int maximumCount)
```

де `initialCount` — це початкове значення лічильника семафора, тобто кількість початкових дозволів; `maximumCount` — максимальне можливе значення лічильника, тобто максимальна кількість дозволів, які може дати семафор.

Семафор використовується таким чином, як і м'ютекс. Для отримання доступу до ресурсу в коді викликається метод `WaitOne()` для об'єкта семафора, який очікує до тих пір, поки не буде отриманий семафор, для якого він викликається. Він блокує виконання викликаючого потоку до тих пір, поки семафор не надасть дозвіл на доступ до ресурсу. Семафор звільняється викликом методу `Release()`. Є дві форми цього методу:

```
public int Release()
public int Release(int releaseCount)
```

В першій формі `Release()` звільняє один дозвіл, а в другій — `releaseCount` дозволів. В обох формах метод повертає кількість дозволів, які існували до звільнення.

Якщо потік вже отримав доступ до семафора і ще не звільнив свій доступ, а потім повторно викликає метод `WaitOne()`, то на відміну від м'ютекса він може і не отримати доступ (якщо всі дозволи вже вичерпані).

Задача. Написати клас «заправка». Реалізувати метод для заправки автомобіля з урахуванням числа колонок та запасу палива на заправці. Передбачити синхронізацію.

11.1.3. Бар'єр

Клас бар'єр дає змогу кільком задачам паралельно працювати із алгоритмом, використовуючи кілька фаз. Кожний учасник виконується, поки його код не досягне точки бар'єра. Бар'єр означає закінчення одного етапу роботи. Коли учасник досягає бар'єра,

його виконання блокується, поки усі учасники не досягнуть цього самого бар'єра. Коли всі учасники досягнуть бар'єра, при необхідності можна почати наступний етап.

Конструктори:

```
Barrier(Int32)
Barrier(Int32, Action<Barrier>)
```

Числовий параметр — кількість учасників, які мають зібратися біля бар'єра. Додатковий параметр — делегат, який буде викликатися після кожної фази.

У роботі з бар'єром використовується метод

```
SignalAndWait()
```

Він повідомляє, що учасник досяг бар'єра та очікує досягнення бар'єра іншими учасниками. Властивість `CurrentPhaseNumber` вказує номер поточної фази бар'єра.

Приклад. Кожний з потоків випадковим чином перемішує слова у відповідному йому масиві. Обчислити кількість етапів, потрібних для того, щоб конкатенація слів у масивах дала потрібне речення.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace BarrierSimple
{
    class Program
    {
        static string[] words1 = new string[] { "brown", "jumped", "the", "fox"};
        static string[] words2 = new string[] { "dog", "lazy","the","over"};
        static string solution = "the brown fox jumped over the lazy dog.";
        static bool success = false;
        static Barrier barrier = new Barrier(2, (b) =>
        {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < words1.Length; i++)
            {
                sb.Append(words1[i]);
                sb.Append(" ");
            }
            for (int i = 0; i < words2.Length; i++)
            {
                sb.Append(words2[i]);

                if(i < words2.Length - 1)
                    sb.Append(" ");
```

```
            }
            sb.Append(".");
            Console.CursorLeft = 0;
            Console.Write("Current phase: {0}", barrier.CurrentPhaseNumber);
            if (String.CompareOrdinal(solution, sb.ToString()) == 0)
            {
                success = true;
                Console.WriteLine("\r\nThe solution was found in {0}
                                attempts", barrier.CurrentPhaseNumber);
            }
        });

        static void Main(string[] args)
        {

            Thread t1 = new Thread(() => Solve(words1));
            Thread t2 = new Thread(() => Solve(words2));
            t1.Start();
            t2.Start();

            // Keep the console window open.
            Console.ReadLine();
        }

        // We require that both wordArrays be solved in the same phase.
        static void Solve(string[] wordArray)
        {
            while(success == false)
            {
                Random random = new Random();
                for (int i = wordArray.Length - 1; i > 0; i--)
                {
                    int swapIndex = random.Next(i + 1);
                    string temp = wordArray[i];
                    wordArray[i] = wordArray[swapIndex];
                    wordArray[swapIndex] = temp;
                }
                // We need to stop here to examine results
                // of all thread activity. This is done in the post-phase
                // delegate that is defined in the Barrier constructor.
                barrier.SignalAndWait();
            }
        }
    }
}
```

11.1.4. Монітор

У *статичному* класі `Monitor`, який знаходиться в просторі імен `System.Threading` визначено ряд методів для керування синхронізацією. Наприклад, для отримання блокування об'єкта викликається метод `Enter()`, а для зняття блокування — метод `Exit()`. Нижче наведені загальні форми цих методів:

```
public static void Enter(object obj)
```

```
public static void Exit(object obj)
```

де obj позначає синхронізований об'єкт. Якщо ж об'єкт недоступний, то після виклику методу Enter() викликаючий потік очікує до тих пір, поки об'єкт не стане доступним (на практиці методи Enter() та Exit() застосовуються рідко, оскільки оператор lock автоматично забезпечує еквівалентні засоби синхронізації потоків).

В класі Monitor описані методи TryEnter(), які можуть мати наступні форм:

```
public static bool TryEnter(object obj)
public static bool TryEnter(object obj, int milliseconds)
```

Ці методи повертають значення true, якщо викликаючий потік отримує блокування для об'єкта obj (відразу, або на протязі часу, вказаного за допомогою параметру timeout), а інакше вони повертають false. За допомогою методу TryEnter() можна реалізувати альтернативний варіант синхронізації потоків, якщо необхідний об'єкт тимчасово недоступний. Розглянемо приклад:

```
var lockObj = new Object();
int timeout = 500;

if (Monitor.TryEnter(lockObj, timeout)) {
   try {
      // The critical section.
   }
   finally {
      // Ensure that the lock is released.
      Monitor.Exit(lockObj);
   }
}
else {
   // The lock was not acquired.
}
```

В класі Monitor визначені методи Wait(), Pulse() та PulseAll(), які дають змогу перевести / вивести потоки із стану очікування та забезпечують комунікацію між потоками. Ці методи можуть викликатися тільки з заблокованого фрагмента блоку (після входу у монітор). Вони застосовуються наступним чином. Коли виконання потоку тимчасово заблоковано, він викликає метод Wait(). В результаті потік переходить в стан очікування, а *блокування з відповідного об'єкта знімається*, що дає можливість використовувати цей об'єкт в іншому потоці. Надалі очікуючи потік акти-

вується, коли інший потік увійде в аналогічний стан блокування, і викликає метод `Pulse()` або `PulseAll()`. При виклику методу `Pulse()` поновлюється виконання першого потоку, що очікує своєї черги на отримання блокування. А виклик методу `PulseAll()` сигналізує про зняття блокування всім очікуючим потокам. Нижче наведено дві найбільш вживані форми методу `Wait()`:

```
public static bool Wait(object obj)
public static bool Wait(object obj, int milliseconds)
```

У першій формі очікування триває аж до повідомлення про звільнення об'єкта, а в другій — як до повідомлення про звільнення об'єкта, так і до закінчення періоду часу, на який вказує кількість міллісекунд. В обох формах `obj` позначає об'єкт, звільнення якого очікується.

Нижче наведені загальні форми методів `Pulse()` і `PulseAll()`:

```
public static void Pulse(object obj)
public static void PulseAll(object obj)
```

Якщо методи `Wait()`, `Pulse()` і `PulseAll()` викликаються з коду, що знаходиться за межами синхронізованого коду, то генерується `SynchronizationLockException`.

Слід зазначити, що засіб для блокування вбудовано в мову C#. Завдяки цьому всі об'єкти можуть бути синхронізовані. Синхронізація організовується за допомогою ключового слова `lock`. Нижче наведена загальна форма блокування:

```
lock(lockObj)
{
    // оператори, що синхронізуються
}
```

де `lockObj` позначає посилання на синхронізований об'єкт (якщо потрібно синхронізувати тільки один оператор, то фігурні дужки не потрібні). Оператор `lock` є замінником пари `Monitor.Enter(lockObj)` та `Monitor.Exit(lockObj)`. Блокується такий об'єкт, який є синхронізованим ресурсом. У деяких випадках їм надається екземпляр самого ресурсу або ж довільний екземпляр об'єкта, який використовується для синхронізації. Треба мати на увазі, що об'єкт, що блокується не має бути загальнодоступним, бо у протилежному випадку він може бути заблокований з іншого, неконтрольованого в програмі фрагмента коду і надалі взагалі не розблокується. У мину-

лому для блокування об'єктів дуже часто застосовувалася конструкція `lock(this)`. Але вона придатна тільки в тому випадку, якщо `this` є посиланням на закритий об'єкт. У зв'язку з можливими програмними і концептуальними помилками, до яких може привести конструкція `lock(this)`, застосовувати її більше не рекомендується. Замість неї краще створити закритий об'єкт, щоб потім заблокувати його.

У наведеній нижче програмі синхронізація демонструється на прикладі управління доступом до методу `SumIt()`, який обчислює суму елементів цілочисельного масиву.

```
using System;
using System.Threading;
class SumArray {
    int sum;
    object lockOn = new object(); // об’єкт для блокування
    public int SumIt(int[] nums) {
        lock(lockOn) { // заблокувати увесь метод
            sum = 0;
            for(int i = 0; i < nums.Length; i++) {
                sum += nums[i];
                Console.WriteLine("Сума для потока " +
                  Thread.CurrentThread.Name + " = " + sum);
                Thread.Sleep(10); // дозволити перемикання
            }
            return sum;
        }
    }
}
class MyThread {
    public Thread Thrd;
    int[] a;
    int answer;
    // Один об’єкт типу SumArray для всіх екземплярів
    static SumArray sa = new SumArray();
    // Створити новий потік.
    public MyThread(string name, int[] nums) {
        a = nums;
        Thrd = new Thread(this.Run);
        Thrd.Name = name;
        Thrd.Start();
    }
    // Почати виконання нового потоку
    void Run() {
        Console.WriteLine (Thrd.Name + " стартував");
        answer = sa.SumIt(a);
```

```
            Console.WriteLine("Сумма для потоку {0} рівна
            {1}", Thrd.Name, answer);
            Console.WriteLine(Thrd.Name + " завершений");
        }
    }
    class Sync {
        static void Main() {
            int[] a = {1, 2, 3, 4, 5};
            MyThread mtl = new MyThread("Нащадок #1", a);
            MyThread mt2 = new MyThread("Нащадок #2", a);
            mtl.Thrd.Join();
            mt2.Thrd.Join();
        }
    }
```

Незважаючи на всю простоту і ефективність блокування коду методу, як показано в наведеному вище прикладі, такий засіб синхронізації виявляється придатним далеко не завжди. Припустимо, що потрібно синхронізувати доступ до метода класу, який був створений кимось іншим і сам не синхронізований. Вихідний код класу недоступний. Як же тоді синхронізувати об'єкт такого класу? На щастя, ця проблема розв’язується досить просто: доступ до об'єкта може бути заблокований з зовнішнього коду по відношенню до даного об'єкту, для чого достатньо вказати цей об'єкт в операторі `lock`. Нижче наведений відповідний рядок коду, в якому здійснюється подібне блокування :

```
lock(sa) answer = sa.SumIt(a);
```

Об'єкт `sa` має бути закритим для успішного блокування заблокований.

Приклад. Задача про перетин мостів.

11.1.5. Застосування подій

Для синхронізації в C # передбачений ще один тип об'єкта: подія. Існують два різновиди подій: встановлюються в початковий стан вручну і автоматично. Вони підтримуються в класах `ManualResetEvent` та `AutoResetEvent` відповідно. Ці класи є похідними від класу `EventWaitHandle`, що знаходиться на верхньому рівні ієрархії класів, і застосовуються в тих випадках, коли один потік чекає появи деякої події в іншому потоці. Як тільки така подія з'являється, другий потік повідомляє про нього перший потік, дозволяючи тим самим відновити його виконання. Нижче наведені конструктори класів:

```
    public ManualResetEvent(bool initialState)
    public AutoResetEvent(bool initialState)
```

Якщо в обох формах параметр `initialState` має значення `true`, то про подію спочатку повідомляється. А якщо він має логічне значення `false`, то про подію спочатку не буде повідомлено.

Використання подій дуже просте. Так, для події типу `ManualResetEvent` порядок застосування наступний. Потік, що очікує деяку подію, викликає метод `WaitOne()` для об'єкта, що пов'язаний із цією подією. Якщо об'єкт події знаходиться в *сигнальному стані*, то відбувається негайне повернення з методу `WaitOne()`. В іншому випадку виконання викликаючого потоку призупиняється до тих пір, поки не буде отримано повідомлення про подію. Як тільки подія відбудеться в іншому потоці, цей потік *встановить* об'єкт події в сигнальний стан, викликавши метод `Set()`. Метод `Set()` треба розглядати як такий, що повідомляє про те, що подія відбулася. Після встановлення об'єкта події в сигнальний стан відбувається негайне повернення з методу `WaitOne()`, і перший потік продовжує своє виконання. А в результаті виклику методу `Reset()` об'єкт події повертається в несигнальний стан.

Подія типу `AutoResetEvent` відрізняється від події типу `ManualResetEvent` лише способом переведення у початковий стан. Якщо для події типу `ManualResetEvent` об'єкт події залишається в сигнальному стані доти, поки не буде викликаний метод `Reset()`, то для події типу `AutoResetEvent` об'єкт автоматично переходить в несигнальний стан, як тільки потік, очікує цю подію, отримає повідомлення про нього і відновить своє виконання. Тому, якщо застосовується подія типу `AutoResetEvent`, то викликати метод `Reset()` необов'язково.

У наведеному нижче прикладі програми демонструється застосування подій.

```
// Потік повідомляє, що подія передана його конструктору
class MyThread {
    public Thread Thrd;
    ManualResetEvent mre;
    public MyThread(string name, ManualResetEvent evt) {
        Thrd = new Thread(this.Run);
        Thrd.Name = name;
        mre = evt;
        Thrd.Start();
    }
```

```
    // Точка входу в потік
    void Run(){
        Console.WriteLine("Всередині потоку" + Thrd.Name);
        for(int i = 0; i < 5; i ++) {
            Console.WriteLine(Thrd.Name);
            Thread.Sleep(500);
        }
        Console.WriteLine(Thrd.Name + "завершено!");
        mre.Set(); // Повідомити про подію
    }
}
class ManualEventDemo {
    static void Main () {
        ManualResetEvent evtObj =
                                new ManualResetEvent(false);
        MyThread mtl = new MyThread("Потік події 1",evtObj);
        Console.WriteLine("Основний потік очікує подію.");
        // Очікувати повідомлення про подію.
        evtObj.WaitOne();
        Console.WriteLine("Основний потік отримав" +
            "Повідомлення про подію від першого потоку.");
        // Установити об'єкт події в початковий стан.
        evtObj.Reset();
        mtl = new MyThread("Потік події 2", evtObj);
        // Очікувати повідомлення про подію.
        evtObj.WaitOne();
        Console.WriteLine("Основний потік отримав" +
            "Повідомлення про подію від другого потоку.");
    }
}
```

Подія передається безпосередньо конструктору класу `MyThread`. Коли завершується метод `Run()` класу `MyThread`, він викликає для об'єкта події метод `Set()`, встановлює цей об'єкт в сигнальний стан. Якби не об'єкт події, то всі потоки виконувалися б одночасно, а результати їх виконання виявилися б заплутаними. Якщо об’єкт `AutoResetEvent` використовувався би замість об'єкта типу `ManualResetEvent`, то викликати метод `Reset()` в методі `Main()` не довелося б.

11.1.6. Клас Interlocked

Клас `Interlocked` є альтернативою до інших засобів синхронізації у випадках, коли потрібно тільки змінити значення змінної, спільної для кількох потоків. Методи, доступні в класі `Interlocked`, гарантують, що їх дія буде виконуватися як єдина, неперервна операція. Це означає, що ніякої синхронізації в даному випадку взагалі не

вимагається. У класі `Interlocked` надаються статичні методи для додавання двох цілих значень, інкременту та декременту цілих чисел, порівняння і встановлення значень об'єкта, обміну об'єктами і отримання 64-розрядного значення. Всі ці операції виконуються без переривання.

У наведеному нижче прикладі демонструється застосування методів `Increment()` та `Decrement()`:

```
public static int Increment(ref int location)
public static int Decrement(ref int location)
```

де `location` — це відповідна змінна.

```
// Спільний ресурс
class SharedRes {
   public static int Count = 0;
}
//В цьому потоці змінна SharedRes.Count інкрементуєтся
class IncThread {
   public Thread Thrd;
   public IncThread(string name) {
      Thrd = new Thread(this.Run);
      Thrd.Name = name;
      Thrd.Start();
   }
   // Точка входу у потік
   void Run() {
      for(int i = 0; i < 5; i++) {
         Interlocked.Increment(ref SharedRes.Count);
         Console.WriteLine(Thrd.Name + " Count = {0}",
         SharedRes.Count);
      }
   }
}
// В цьому потоці змінна SharedRes.Count
декрементується
class DecThread {
   // аналогічно до попереднього класу
}
// Точка входу в поток,
   void Run () {
      for(int i = 0; i < 5; i++) {
      Interlocked.Decrement(ref SharedRes.Count);
      Console.WriteLine(Thrd.Name + " Count = {0}",
      SharedRes.Count);
      }
   }
```

```
}
class InterlockedDemo {
    static void Main() {
        IncThread mtl = new IncThread("Потік 1");
        DecThread mt2 = new DecThread("Потік 2");
        mtl.Thrd.Join();
        mt2.Thrd.Join();
    }
}
```

11.1.7. Запуск окремого процесу

При програмуванні на C# багатозадачність найчастіше організовується на основі використання потоків. Але там, де це доречно, можна організувати і багатозадачність на основі процесів. У цьому випадку замість запуску іншого потоку в одній і тій же програмі одна програма починає виконання іншої. При програмуванні на C# це робиться за допомогою класу `Process`, визначеного в просторі `System.Diagnostics`.

Найпростіший спосіб запустити інший процес — скористатися методом `Start()` класу `Process`. Нижче наведена одна з найпростіших форм цього методу:

```
public static Process Start(string fileName)
```

де `fileName` позначає конкретне ім'я файлу, який повинен виконуватися або ж пов'язаний з виконуваним файлом.

Коли створений процес завершується, слід викликати метод `Close()`, щоб звільнити пам'ять, виділену для цього процесу:

```
public void Close()
```

Процес може бути перерваний двома способами. Якщо процес є GUI-додатком Windows, то для переривання такого процесу викликається метод

```
public bool CloseMainWindow()
```

Цей метод посилає процесу повідомлення, що пропонує йому зупинитися. Він повертає логічне значення `true`, якщо повідомлення отримано, і логічне значення `false`, якщо програма не має графічного інтерфейсу або головного вікна. Слід, однак, мати на увазі, що метод `CloseMainWindow()` служить тільки для запиту зупинки процесу. Якщо додаток проігнорує такий запит, то він не буде перерваний.

Для безумовного переривання процесу слід викликати метод `Kill()`:

```
public void Kill()
```

Методом `Kill()` слід користуватися акуратно, так як він призводить до неконтрольованого переривання процесу. Будь-які незбережені дані, пов'язані з процесом, переривається, будуть, швидше за все, втрачені.

Для того щоб організувати очікування завершення процесу, можна скористатися методом `WaitForExit()`. Нижче наведено дві його форми:

```
public void WaitForExit()
public bool WaitForExit(int milliseconds)
```

У першій формі очікування триває до тих пір, поки процес не завершиться, а в другій формі — тільки протягом зазначеної кількості мілісекунд. В другому випадку `WaitForExit()` повертає `true`, якщо процес завершився, і `false`, якщо він все ще виконується.

У наведеному прикладі демонструється створення, очікування і закриття процесу.

```
using System;
using System.Diagnostics;
class StartProcess {
    static void Main() {
        Process newProc = Process.Start( "wordpad.exe");
        Console.WriteLine("Новий процес запущений");
        newProc.WaitForExit();
        newProc.Close(); // звільнити виділені ресурси
        Console.WriteLine("Новий процес завершено");
    }
}
```

При виконанні цієї програми запускається стандартний додаток WordPad, і на екрані з'являється повідомлення. Потім програма очікує закриття WordPad. Після закінчення роботи WordPad на екрані з'являється заключне повідомлення.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 608 с.
2. Антонов А. С. Параллельное программирование с использованием технологии OpenMP. — М. Изд-во МГУ, 2009. — 77 с.
3. Оленев Н. Основы параллельного программирования в системе MPI. — М.: Вычислительный центр им. А. А. Дородицына РАН, 2005 — 81 с.
4. Сердюк Ю. П. Введение в параллельное программирование на языке MC#. Переславль-Залесский: Институт программных систем РАН, 2007. — 51с.
5. Боресков А. В., Харламов А. А. Основы работы с технологией CUDA. М.: ДМК Пресс, 2010. — 232 с.
6. Воеводин В. В. Математические основы параллельных вычислений. – М.: МГУ, 1991. – 345 с.
7. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ, 2-е издание. – М.: "Вильямс", 2005. — 1296 с.
8. Шилдт Г. Java. Полное руководство. — М.: ООО "И.Д. Вильямс", 2012. — 1104 с.
9. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0, 5-е изд. — М.: "Вильямс", 2011. — 1392 с.
10. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. — М.: Издательский дом "Вильямс", 2003. — 512 с.

Інформаційні ресурси

11. OpenMP Architecture Review Board (**http://www.openmp.org/**)
12. http://www.gridforum.org
13. http://www.mpiforum.org
14. http://parallel.ru