8.1 СЕМАФОРИ

Розглянуті раніше алгоритми синхронізації хоча і є коректними, але досить громіздкі. Більше того, процедура очікування входу в критичну ділянку, як для алгоритму Петерсона, так і при використанні апаратних інструкцій, припускає досить тривале обертання процесу в порожньому циклі, тобто марну витрату дорогоцінного часу процесора, перевіряючи можливість увійти до критичної області.

Існують і інші серйозні недоліки алгоритмів, побудованих засобами звичайних мов програмування. Припустимо, що в обчислювальній системі знаходяться два взаємодіючі процеси: один з них – H з високим пріоритетом, інший – L з низьким пріоритетом. Нехай планувальник влаштований так, що процес з високим пріоритетом витісняє низькопріоритетний процес всякий раз, коли той готовий до виконання, і займає процесор на весь відведений йому час (якщо не з'явиться процес з ще більшим пріоритетом). Тоді в разі, якщо процес L знаходиться у своїй критичній секції, а процес H, отримавши процесор, підійшов до входу в критичну область, ми отримуємо тупикову ситуацію. Процес H не може ввійти до критичної області, знаходячись в циклі, а процес L не отримує управління, щоб покинути критичну ділянку.

Для того щоб не допустити виникнення подібних проблем, були розроблені різні механізми синхронізації вищого рівня. Опису ряду з них – семафорів, моніторів і повідомлень – і присвячений цей розділ.

Першою великою роботою, присвяченою питанням паралельних обчислень, стала монографія Дейкстри, яка була видана в 1965 році [14]. У ній розглядалася розробка ОС як побудова певної кількості співпрацюючих послідовних процесів і створення ефективних та надійних механізмів підтримки цієї співпраці. Ці механізми легко застосовуються і призначеними для користувача процесами, якщо процесор і ОС роблять їх загальнодоступними.

Дейкстра запропонував узагальнений засіб синхронізації процесів, ввівши два нових примітиви. В абстрактній формі ці примітиви, позначені Дейкстрою як P і V (це перші букви голландських слів перевірка (Proberen) і збільшення (Verhogen)),

оперують над цілими не від'ємними змінними S, що називаються **семафорами** (semaphore).

Класичне визначення цих операцій виглядає таким чином:

```
P(S): доки S = 0 процес блокується;
S = S - 1;
V(S): S = S + 1;
```

Позначимо ці примітиви відповідно як Wait і Signal (або Down(s) і Up(s), як це було ще в нотації Algol-68).

Для передачі сигналу через семафор S процес застосовує примітив Signal(s), а для отримання сигналу – примітив Wait(s). В останньому випадку процес призупиняється до тих пір, поки не буде прийнятий відповідний сигнал.

Над семафором визначені три операції, що наводяться нижче.

1. Семафор може ініціалізуватися тільки додатнім числом.

2. Операція Wait(s) зменшує значення семафора на 1. Якщо це значення стає негативним, то процес блокується, тобто чекає, поки це зменшення стане можливим. Успішна перевірка, зменшення і перехід процесу в стан очікування виконуються як єдина і неподільна елементарна (атомарна) дія.

3. Операція Signal(s) збільшує значення семафора на 1 однією неподільною дією (вибірка, інкремент і запам'ятовування). Якщо це значення від'ємне, то заблокований операцією Wait процес деблокується. Тобто, один з процесів вибирається системою і йому дозволяється завершити свою операцію Wait(s). Операція збільшення значення семафора і активізації процесу також неподільна. Є різні організації семафорів щодо операції Signal(s), застосованої до семафора, пов'язаного з декількома очікуючими процесами, значення семафора так і залишається рівним 0, але число очікуючих процесів зменшується на одиницю.

Таким чином, якщо значення семафора більше нуля, операція Wait(s) зменшує його і просто повертає управління. Якщо значення дорівнює нулю, процедура Wait(s) не повертає управління процесу, а процес переводиться в стан очікування.

При виконанні операції Signal(s), якщо з цим семафором пов'язані один або декілька процесів, які не можуть завершити ранішу операцію Wait(s), один з них

вибирається системою і йому дозволяється завершити свою операцію Wait(s). Таким чином, після операції Signal(s), що застосовується до семафора, пов'язаного з декількома очікуючими процесами, значення семафора так і залишається рівним нулю, але число очікуючих процесів зменшується на одиницю. Жоден процес не може бути блокований під час виконання операції Signal(s).

В окремому випадку, коли семафор S може набувати тільки значень 0 і 1, він перетворюється на блокуючу змінну. Операція Wait містить в собі потенційну можливість переходу процесу, який її виконує, в стан очікування, тоді як операція Signal може при деяких обставинах активізувати інший процес, призупинений операцією Wait. Така спрощена версія семафора, називається м’ютексом (mutex, скорочення від mutual exclusion). У загальному випадку м’ютексом називають примітив синхронізації, який не допускає виконання деякого фрагмента коду більше як одним процесом. М’ютекс не здатний обчислювати, він може лише управляти взаємним виключенням доступу до спільно використовуваних ресурсів або коду.

Розглянемо використання семафорів на класичному прикладі взаємодії двох процесів «письменник-читач», що виконуються в режимі мультипрограмування, один з яких пише дані в буферний пул, а інший прочитує їх з буферного пулу.

Нехай буферний пул складається з N буферів, кожен з яких може містити один запис. Процес «письменник» повинен призупинятися, коли всі буфери виявляються зайнятими, і активізуватися при звільненні хоч би одного буфера. Навпаки, процес «читач» призупиняється, коли усі буфери порожні, і активізується при появі хоч би одного запису.

Введемо два семафори: e – число порожніх буферів і f – число заповнених буферів, причому в початковому стані e =N, а f =0. Припустимо, що запис у буфер і зчитування з буфера є критичними секціями. Введемо також двійковий семафор b, використовуваний для забезпечення взаємного виключення. Тоді робота потоків (процесів) із загальним буферним пулом може бути представлена блок-схемою (рис. 8.1).

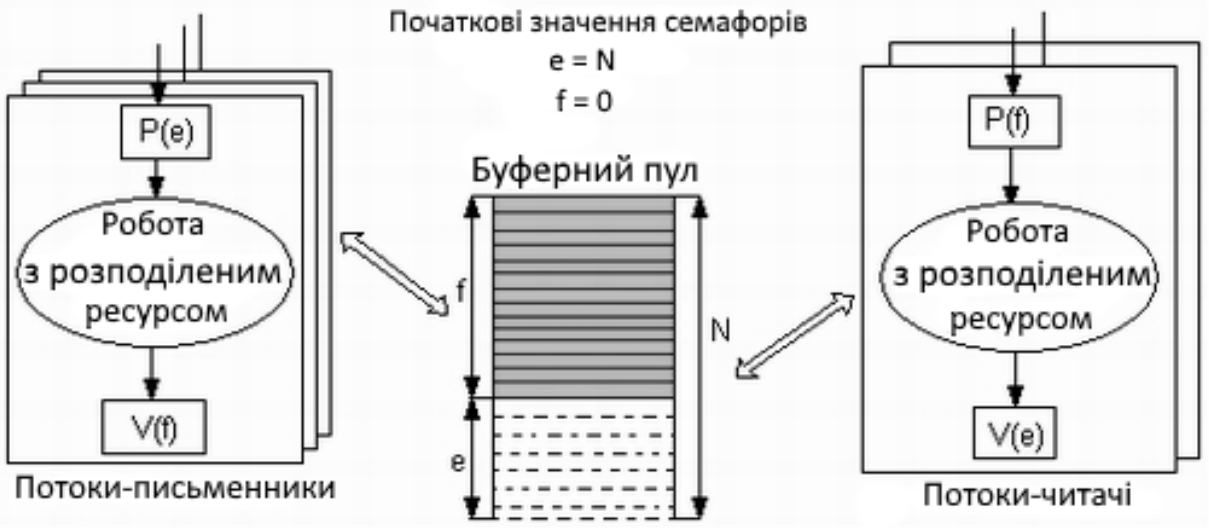

Рисунок 8.1 – Використання семафорів для синхронізації потоків

Фрагмент програми на С-подібній мові виглядає таким чином:

Лістинг 8.1 – Визначення семафорних примітивів

```
// Глобальні змінні
#define N 256
int e = N, f = 0, b = 1; void Writer () {
while(1){
PrepareNextRecord(); /* підготовка нового запису */ Wait(e); /* Зменшити число вільних буферів, якщо вони є */
/* інакше - чекати, поки вони звільняться */ Wait(b); /* Вхід в критичну секцію */ AddToBuffer(); /* Додати новий запис у буфер / Signal(b); /* Вихід з критичної секції */
Signal(f); /* Збільшити число зайнятих буферів */
}
}
void Reader () { while(1){
Wait(f); /* Зменшити число зайнятих буферів, якщо вони есть*/
/* інакше чекати, поки вони з'являться */ Wait(b); /* Вхід в критичну секцію */ GetFromBuffer(); /* Узяти запис з буфера */ Signal(b); * Вихід з критичної секції */
Signal(e); /* Збільшити число вільних буферів */ ProcessRecord(); /* Обробити запис */
}
}
```

Семафор може використовуватися і в якості блокуючої змінної. У розглянутому вище прикладі, для того щоб виключити колізії при роботі з областю пам'яті, що розділяється, запис у буфер і прочитування з буфера є критичними секціями. Взаємне виключення забезпечується за допомогою двійкового семафора b. Обидва потоки після перевірки доступності буферів повинні виконати перевірку доступності критичної секції. У лістингу 8.2 наведено формальніше визначення примітивів семафорів. Передбачається, що примітиви атомарні, тобто вони не можуть бути перервані, і кожна з підпрограм розглядається як єдиний крок.

Лістинг 8.2 – Визначення семафорних примітивів

```
Struct Semaphore { Int count;
QueueTypy queue;
}
void Wait(semaphore S) { s.count--;
if (s.count < 0) {
/* Помістити процес s.queue, s.count++ або =0 */
/* Заблокувати процес */
}
}
void Signal(semsphore S)
{
s.count++;
if (s.count <= 0) {
/* Видалити процес P з s.queue */
/* Помістити процес P в список активних */
}
}
```

Для зберігання процесів використовується черга. При цьому виникає питання про порядок витягання з черги. Найкоректніший спосіб – використання принципу «першим увійшов – першим вийшов» (First In First Out – FIFO). При цьому з черги звільняється процес, який був заблокований довше за інших. Семафор, що використовує цей метод, називається сильним семафором (strong semaphore).

Семафор, порядок витягання процесів з черги якого не визначений, називається слабким семафором (weak semaphore).

8.1.1 Взаємні виключення

Розглянемо алгоритм взаємовиключень з використанням сильного семафора S (лістинг 8.3), в якому гарантується неможливість голодування, але слабкий семафор такої гарантії не дає. Далі вважатимемо, що працюють сильні семафори, оскільки саме вони використовуються в ОС.

Нехай у нас є N процесів, що ідентифікуються масивом P(i). У кожному з процесів перед входом в критичний розділ виконується виклик Wait(S). Якщо значення S стає від'ємним, процес призупиняється. Якщо ж значення S дорівнює 1, воно зменшується до нуля і процес негайно входить в критичний розділ. Оскільки S більше не є додатнім, жоден інший процес не може ввійти до критичного розділу.

Лістинг 8.3 – Взаємні виключення з використанням семафорів

```
/* Програма взаємного виключення */ const int n = /* Кількість процесів */; semaphore s = 1;
void P(int i)
{
while(true) {
wait(s);
/* Критичний розділ */ signal(s);
/* Інший код */
}
}
void main() {
perbegin(P(1),P(2),…,P(n));
}
```

Семафор ініціалізувався значенням 1. Отже, перший процес, що виконує інструкцію Wait, зможе негайно потрапити в критичний розділ, встановлюючи значення семафора, рівним 0. Будь-який інший процес при спробі увійти до критичного розділу виявить, що він зайнятий. Відповідно, станеться блокування процесу, а значення семафора буде зменшено до -1. Намагатися увійти до критичного

розділу може будь-яка кількість процесів; кожна спроба неуспіху зменшує значення семафора.

Після того, як процес, що увійшов до критичного розділу першим, покидає його, S збільшується, і один із заблокованих процесів (якщо такий є) видаляється з пов'язаної з семафором черги і активується. Таким чином, як тільки планувальник ОС надасть йому можливість виконання, процес тут же зможе увійти до критичного розділу.

8.1.2 Задача «Виробник-Споживач» («Письменник-Читач»)

А тепер додамо до нашої задачі нове сховище – розглядатимемо кінцевий буфер b(n) як циклічне сховище, при роботі з яким значення покажчиків повинні виражатися за модулем розміру буфера. При цьому виконуються умови, що наведені нижче.

Блокування:

1. Виробник: вставка в повний буфер.
2. Споживач: видалення з порожнього буфера.

Деблокування:

1. Виробник: вставка елемента в буфер.
2. Споживач: видалення елемента з буфера.

Функції виробника і споживача при цьому можуть бути записані таким чином (in і out ініціалізувані значенням 0):

```
Виробник: (AddToBuffer())
  While (true) {
  /* Виробництво елементу v /;
while((in+1) % n == out); /*чекати*/
     b[in] = v;
     in % n;
  }
```

```
Споживач: (GetFromBuffer())
     While (true) {
     while(in == out); /* чекати */
          w = b[out];
          out = (out+1) % n;
          /* споживання елементу w*/
     }
```

У лістингу 8.4 наведеній розв'язок цієї задачі з використанням **узагальнених семафорів** (семафорами з лічильниками). Для відстежування порожнього місця в буфері в програму доданий семафор e.

Лістинг 8.4 – Розв'язок задачі Виробник-Споживач з обмеженим циклічним буфером

```
const int SizeBuffer = /* Розмір буфера */;
semafore s = 1;
semafore n = 0; /* число зайнятих буферів */
semafore e = SizeBuffer; /* число порожніх буферів */
void Writer () {
    while(1){
    PrepareNextRecord(); /* підготовка нового запису */
    Wait(e); /* Зменшити число вільних буферів, якщо вони є */
    /* інакше чекати, поки вони звільняться */
    Wait(s);   /* Вхід в критичну секцію */
    AddToBuffer(); /* Додати новий запис у буфер */
    Signal(s); /* Вихід з критичної секції */
    Signal(n); /* Збільшити число зайнятих буферів */
     }
}
void Reader () {
    while(1){
    Wait(n); /* Зменшити число зайнятих буферів, якщо вони є */
    /* інакше чекати, поки вони з'являться */
    Wait(s); /* Вхід в критичну секцію */
    GetFromBuffer(); /* Узяти запис з буфера */
    Signal(s); /* Вихід з критичної секції */
    Signal(e); /* Збільшити число вільних буферів */
    ProcessRecord(); /* Обробити запис */
    }
}
void main(){
    parbegin (Writer, Reader);
}
```

Припустимо тепер, що при переписуванні цієї програми сталася помилка: програміст переставив операції Signal(s) і Signal(n) в процедурі Writer (). Це призведе до того, що операція Signal(n) виконуватиметься в критичному розділі виробника без переривання споживача. Чи вплине це на виконання програми? Ні, споживач у будь-

якому випадку повинен чекати установки обох семафорів перед продовженням роботи.

Тепер припустимо, що переставлені операції Wait(n) і Wait(s) в процедурі Reader (). Це призведе до фатальних наслідків. Якщо споживач увійде до критичного розділу, коли буфер порожній (n = 0), то виробник не зможе додати дані в буфер і система виявиться в стані взаємного блокування.

З усього вищесказаного ясно, що використовувати семафори треба дуже обережно, і програміст при написанні програм з паралельно працюючими процесами повинен мати певну культуру програмування.

У разі потоків у призначеному для користувача просторі немає проблеми доступу потоків, наприклад до семафора, оскільки в усіх потоків загальний адресний простір. Проте, у більшості попередніх моделей, зокрема в алгоритмі Петтерсона і семафорах, передбачалося, що декілька процесів мають доступ до спільно використовуваної ділянки пам'яті. Якщо адресні простори процесів несумісні, то як вони можуть спільно використати змінну turn в алгоритмі Петтерсона, або семафори, або загальний буфер? На це питання існує дві відповіді.

По-перше, деякі із спільно використовуваних структур даних, скажімо, семафори, можуть зберігатися в ядрі з доступом тільки через системні запити. Цей підхід вирішує проблему.

По-друге, більшість сучасних ОС (Windows, Unix) представляють можливість спільного використання процесами деякої частини адресного простору. У цьому випадку можливе розділення буфера і інших структур даних. У крайньому випадку можна використати файл.

8.1.3 Java семафори

Семафори представляють ще один засіб синхронізації для доступу до ресурсу. У Java семафори представлені класом **Semaphore** з пакету *java.util.concurrent.*

Для управління доступом до ресурсу семафор використовує лічильник, що представляє кількість дозволів. Якщо значення лічильника більше нуля, то потік дістає доступ до ресурсу, при цьому лічильник зменшується на одиницю. Після закінчення роботи з ресурсом потік звільняє семафор, і лічильник збільшується на

одиницю. Якщо ж лічильник дорівнює нулю, то потік блокується і чекає, поки не отримає дозвіл від семафора.

Встановити кількість дозволів для доступу до ресурсу можна за допомогою конструкторів класу Semaphore:

```
1 Semaphore(int permits)
2 Semaphore(int permits, boolean fair)
```

Параметр `permits` вказує на кількість допустимих дозволів для доступу до ресурсу. Параметр `fair` у другому конструкторі дозволяє встановити черговість отримання доступу. Якщо він рівний true, то дозволи надаватимуться очікуючим потокам в тому порядку, в якому вони просили доступ. Якщо ж він рівний false, то дозволи надаватимуться в невизначеному порядку.

Для отримання дозволу в семафора потрібно викликати метод **acquire(),** який має дві форми:

```
1 void acquire() throws InterruptedException
2 void acquire(int permits) throws InterruptedBxception
```

Для отримання одного дозволу застосовується перший варіант, а для отримання декількох дозволів – другий варіант.

Після виклику цього методу доки потік не отримає дозвіл, він блокується. Після закінчення роботи з ресурсом отриманий раніше дозвіл потрібно звільнити за допомогою методу release():

```
1 void release()
2 void release(int permits)
```

Перший варіант методу звільняє один дозвіл, а другий варіант – кількість дозволів, вказаних в permits. Використаємо семафор Java в простому прикладі (лістинг 8.5).

Лістинг 8.5 – Приклад використання семафорів у Java.

```
1  import java.util.concurrent.Semaphore;
2
3  public class ThreadsApp {
4
5    public static void main(String[] args) {
6
7    Semaphore sem = new Semaphore(1); // 1 дозвіл
8    CommonResource res = new CommonResource();
9    new Thread(new CountThread(res, sem, "CountThread 1")).start();
10   new Thread(new CountThread(res, sem, "CountThread 2")).start();
11   new Thread(new CountThread(res, sem, "CountThread 3")).start();
12   }
13 }
14 class CommonResource{
15
16    int x=0;
17 }
18
19 class CountThread implements Runnable{
20
21   CommonResource res;
22   Semaphore sem;
23   String name;
24   CountThread(CommonResource res, Semaphore sem, String name){
25        this.res=res;
26        this.sem=sem;
27        this.name=name;
28        }
29
30   public void run(){
31
32        try{
33        System.out.println(name + " чекає дозвіл");
34        sem.acquire();
35        res.x=1;
```

```
36        for (int i = 1; i < 5; i++){
37        System.out.println(this.name + ": " + res.x);
38        res.x++;
39        Thread.sleep(100);
40        }
41    }
42    catch(InterruptedException e)
43      {System.out.println(e.getMessage());}
44    System.out.println(name + " звільняє дозвіл");
45    sem.release();
46    }
47  }
```

У даному прикладі є загальний ресурс CommonResource з полем x, яке змінюється кожним потоком. Потоки представлені класом CountThread, який отримує семафор і виконує деякі дії над ресурсом. В основному класі програми (main) ці потоки запускаються.

У результаті ми отримаємо такі повідомлення на екрані:

CountThread 1 чекає дозвіл

CountThread 2 чекає дозвіл

CountThread 3 чекає дозвіл

CountThread 1: 1

CountThread 1: 2

CountThread 1: 3

CountThread 1: 4

CountThread 1 звільняє дозвіл

CountThread 3: 1

CountThread 3: 2

CountThread 3: 3

CountThread 3: 4

CountThread 3 звільняє дозвіл

CountThread 2: 1

CountThread 2: 2

CountThread 2: 3

CountThread 2: 4

CountThread 2 звільняє дозвіл

Семафори чудово підходять для розв'язання задач, де потрібно обмежувати доступ. Наприклад, класична задача про філософів, що обідають, яка використовується в інформатиці для ілюстрації проблем синхронізації при розробці паралельних алгоритмів. Задача була сформульована в 1965 році Едсгером Дейкстрою як екзаменаційна вправа для студентів. Як приклад був узятий конкуруючий доступ до стрічкового накопичувача. Незабаром задача була сформульована Річардом Хоаром в тому вигляді, в якому вона відома сьогодні. Суть її полягає в наступному.

П'ять безмовних філософів сидять навколо круглого столу, перед кожним філософом стоїть тарілка спагеті. Вилки лежать на столі між кожною парою найближчих філософів. Кожен філософ може або їсти, або роздумувати (сидячи за столом або ходити). Їда не обмежена кількістю спагеті – мається на увазі нескінченний запас. Проте, філософ може їсти тільки тоді, коли тримає дві вилки – узяту справа і ліворуч.

Кожен філософ може узяти найближчу вилку (якщо вона доступна), або покласти, якщо він вже тримає її. Узяття кожної вилки і повернення її на стіл є роздільними діями, які повинні виконуватися одне за іншим.

Суть проблеми полягає в тому, щоб розробити модель поведінки (паралельний алгоритм), при якій жоден з філософів не голодуватиме, тобто вічно буде чергувати їжу і роздуми.

Задача сформульована так, щоб ілюструвати проблему уникнення взаємного блокування – стану системи, при якому прогрес неможливий. Відносно просте рішення задачі досягається шляхом додавання офіціанта біля столу. Філософи повинні чекати дозволу офіціанта перед тим, як узяти вилку. Оскільки офіціант знає скільки вилок використовується в даний момент, то він може приймати рішення відносно розподілу вилок і тим самим запобігти взаємного блокування філософів. Якщо чотири вилки з п'яти вже використовуються, то наступний філософ, що

запросив вилку, змушений буде чекати дозволу офіціанта, який не буде отримано, поки вилка не буде звільнена. Передбачається, що філософ завжди намагається спочатку узяти ліву вилку, а потім – праву (чи навпаки), що спрощує логіку. Офіціант працює, як семафор.

Лістинг 8.6 – Задача про обідаючих філософів.

```
1    import java.util.concurrent.Semaphore;
2
3    public class ThreadsApp {
4
5        public static void main(String[] args) {
6
7         Semaphore sem = new Semaphore(2);
8         for(int i=1;i<6;i++)
9              new Philosopher(sem,i).start();
10        }
11   }
12   // клас філософа
13   class Philosopher extends Thread
14   {
15        Semaphore sem; // семафор, що обмежує число філософів
16        // кількість їди
17        int num = 0;
18        // умовний номер філософа
19        int id;
20        // в якості параметрів конструктора передаємо
21        // ідентифікатор філософа і семафор
22        Philosopher (Semaphore sem, int id)
23             {
24             this.sem=sem;
25             this.id=id;
26             }
27
28        public void run()
29        {
```

```
30          try
31            {
32             while // доки кількість їди не досягне 3
33             {
34             //Просимо у семафора дозвіл на виконання
35            sem.acquire();
36            System.out.println("Філософ " +id+" сідає за стіл");
37            // філософ їсть
38             sleep(500);
39            num++;
40
41            System.out.println("Філософ " +id+" виходить із-за
столу");
42            sem.release();
43
44            // філософ гуляє
45            sleep(500);
46            }
47        }
48        catch(InterruptedException e)
49        {
50        System.out.println("у філософа " +id+ " проблеми");
51        }
52       }
53   }
```

У результаті тільки два філософи зможуть одночасно знаходитися за столом, а інші чекатимуть:

Філософ 1 сідає за стіл

Філософ 3 сідає за стіл

Філософ 3 виходить із-за столу

Філософ 1 виходить із-за столу

Філософ 2 сідає за стіл

Філософ 4 сідає за стіл

Філософ 2 виходить із-за столу

Філософ 4 виходить із-за столу

Філософ 5 сідає за стіл

Філософ 1 сідає за стіл

Філософ 1 виходить із-за столу

Філософ 5 виходить із-за столу

Філософ 3 сідає за стіл

Філософ 2 сідає за стіл

Філософ 3 виходить із-за столу

Філософ 4 сідає за стіл

Філософ 2 виходить із-за столу

Філософ 5 сідає за стіл

Філософ 4 виходить із-за столу

Філософ 5 виходить із-за столу

Філософ 1 сідає за стіл

Філософ 3 сідає за стіл

Філософ 1 виходить із-за столу

Філософ 2 сідає за стіл

Філософ 3 виходить із-за столу

Філософ 5 сідає за стіл

Філософ 2 виходить із-за столу

Філософ 4 сідає за стіл

Філософ 5 виходить із-за столу

Філософ 4 виходить із-за столу