# Лабораторная работа №6

# Синхронизация процессов

# 1. Теоретические сведения

## 1.1 Синхронизация процессов

Синхронизация процессов имеет важное значение, при использовании различными процессами одних и тех же ресурсов. Введем некоторые понятия. Под **активностями** мы будем понимать последовательное выполнение ряда операций, направленных на достижение определенной цели. Неделимые операции могут иметь внутренние невидимые действия. Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.
Пусть имеется две активности

```
P:   a b c
Q:   d e f
```

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

```
PQ: a b c d e f
```

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередования:

```
a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
......
d e f a b c
```

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

```
P:  x=2          Q:  x=3
    y=x-1            y=x+1
```

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор активностей (например, программ) **детерминирован**, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает **одинаковые** выходные данные. В противном случае он **недетерминирован**. Выше приведен пример недетерминированного набора программ.
Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

```
P:   x=u+v
     y=x*w
```

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

1. пересечение W(P) и W(Q) пусто;
2. пересечение W(P) с R(Q) пусто;
3. пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением (mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

## 1.2 Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (critical section) программы. Критическая секция – это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам,

участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция.
В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

```
while (some condition) {
   entry section
      critical section
   exit section
      remainder section
}
```

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию.

**Требования, предъявляемые к алгоритмам:**

1. задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test) являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutual exclusion).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting).

# 2. Программные алгоритмы организации взаимодействия процессов.

## 2.1 Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе "Критическая секция").

```
shared int lock = 0;
/* shared означает, что */
/* переменная является разделяемой */

while (some condition) {
   while(lock); lock = 1;
      critical section
   lock = 0;
      remainder section
}
```

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

## 2.2 Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

```
shared int turn = 0;

while (some condition) {
   while(turn != i);
      critical section
   turn = 1-i;
      remainder section
}
```

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в remainder section.

## 2.3 Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

```
shared int ready[2] = {0, 0};
```

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

```
while (some condition) {
   ready[i] = 1;
   while(ready[1-i]);
      critical section
   ready[i] = 0;
      remainder section
}
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock).

## 2.4 Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
   ready[i] = 1;
   turn =1-i;
   while(ready[1-i] && turn == 1-i);
      critical section
   ready[i] = 0;
      remainder section
}
```

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно

из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

## 2.5 Алгоритм булочной (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

```
shared enum {false, true} choosing[n];
shared int number[n];
```

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

```
(a,b) < (c,d), если a < c
или если a == c и b < d
max(a0, a1, ...., an) - это число k такое, что
k >= ai для всех i = 0, ...,n
```

Структура процесса Pi для алгоритма булочной приведена ниже

```
while (some condition) {
   choosing[i] = true;
   number[i] = max(number[0], ...,
                   number[n-1]) + 1;
   choosing[i] = false;
   for(j = 0; j < n; j++){
      while(choosing[j]);
      while(number[j] != 0 && (number[j],j) <
         (number[i],i));
   }
      critical section
   number[i] = 0;
      remainder section
}
```

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

# 2. Индивидуальные задания

## 2.1 Алгоритм взаимодействия двух процессов «**Переменная – замок**»

Выполнить алгоритм синхронизации двух процессов (P0, P1) «*переменная – замок*», использующих общие ресурсы, для данных приведенных в таблице 2.1. Алгоритм планирования процессов ***Round Robin (RR)***, величина кванта времени **3**. Результаты оформить в виде таблицы иллюстрирующей работу процессов.

Таблица 2.1 Варианты заданий

| № | Время возникновения входа в критическую секцию для **P0** | Время возникновения входа в критическую секцию для **P1** | Время выполнения критической секции **P0** | Время выполнения критической секции **P1** |
|---|---|---|---|---|
| 1 | 1-4-11-15-19-28 | 8-13-18-22-27-30 | 2-1-1-1-2-1 | 1-1-1-1-1-4 |
| 2 | 2-5-12-19-22-27 | 8-15-19-22-28-30 | 1-1-1-1-1-2 | 1-1-2-1-1-3 |
| 3 | 1-4-10-16-21-29 | 8-12-15-18-22-26 | 2-1-1-1-1-1 | 1-1-1-1-2-1 |
| 4 | 2-5-13-18-20-23 | 8-13-15-19-22-29 | 1-1-1-1-1-1 | 1-1-2-1-1-2 |
| 5 | 1-5-10-14-19-25 | 7-10-15-20-26-30 | 2-1-1-2-2-1 | 1-2-1-1-2-1 |
| 6 | 2-4-16-22-29-36 | 8-13-19-23-27-32 | 1-1-1-2-1-3 | 1-2-2-1-1-1 |
| 7 | 1-5-19-23-26-34 | 9-13-17-23-27-33 | 2-1-2-1-1-2 | 1-1-2-1-1-1 |
| 8 | 2-5-16-22-29-35 | 9-15-22-25-29-35 | 2-1-1-1-1-1 | 1-1-1-2-2-1 |
| 9 | 1-4-14-18-25-32 | 8-14-18-22-27-35 | 1-1-1-1-1-2 | 1-2-1-1-2-1 |
| 10 | 1-4-16-21-26-32 | 9-14-18-25-30-35 | 2-1-1-1-1-2 | 1-1-1-1-1-3 |
| 11 | 2-5-12-16-20-24 | 10-13-20-23-27-32 | 1-1-2-1-1-2 | 1-1-1-2-1-1 |
| 12 | 1-5-12-18-25-32 | 10-16-19-22-26-30 | 1-1-1-1-2-2 | 1-1-1-2-1-4 |
| 13 | 2-5-12-17-23-32 | 10-14-18-25-30-33 | 1-1-2-1-2-3 | 1-1-1-1-2-1 |
| 14 | 1-5-9-15-22-27-33 | 10-16-19-25-28-30 | 1-1-1-1-1-2 | 1-1-1-1-2-1 |
| 15 | 1-5-11-18-27-34 | 10-13-19-24-27-34 | 1-1-1-1-2-1 | 1-1-1-1-1-1 |

**Пример**

Выполнить алгоритм синхронизации двух процессов (P0, P1) «*переменная – замок*»:

| Время возникновения входа в критическую секцию для **P0** | Время возникновения входа в критическую секцию для **P1** | Время выполнения критической секции **P0** | Время выполнения критической секции **P1** |
|---|---|---|---|
| 1-6-15 | 3-9-15 | 4-3-1 | 2-4-2 |

КС – выполнение критической секции, ГК – готовность критической секции к выполнению, И – выполнение не критической секции, Г – готовность некритической секции к выполнению.

| T | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P0 | КС | КС | КС | КС | Г | ГК | ГК | КС | КС | КС | Г | Г | Г | Г | КС | И | И | Г | Г |
| P1 | Г | Г | Г | ГК | КС | КС | И | Г | ГК | ГК | КС | КС | КС | КС | ГК | ГК | ГК | КС | КС |

## 2.2 Алгоритм взаимодействия двух процессов «**Строгое – чередование**»

Выполнить алгоритм синхронизации двух процессов (P0, P1) «*строгое – чередование*», использующих общие ресурсы, для данных приведенных в таблице 2.1. Алгоритм планирования процессов ***Round Robin (RR)***, величина кванта времени **3**. Результаты оформить в виде таблицы иллюстрирующей работу процессов.

## 2.3 Алгоритм взаимодействия трех процессов

Выполнить алгоритмы синхронизации процессов (P0, P1) «*переменная – замок*» и «*строгое – чередование*», использующих общие ресурсы, при наличии третьего процесса (P2), не использующего ресурсы процессов P0, P1. Данные процессов (P0, P1) «приведенных в таблице 2.1, процесс P2 появляется каждый 6 квант времени, длительность процесса равна **3** квантам. Алгоритм планирования процессов ***Round Robin (RR)***, величина кванта времени **3**. Если процесс P2 выполниться не успел, новый его экземпляр в очередь не ставится. Процесс P2 не может прервать выполнение критической секции. Результаты оформить в виде таблиц иллюстрирующих работу процессов.

## 2.4 Алгоритм взаимодействия нескольких процессов

Выполнить алгоритм синхронизации четырех процессов (P0, P1, P2, P3) «*алгоритм булочной*», использующих общие ресурсы. Процессы выбираются из таблицы 2.1, согласно таблице 2.2. При каждой постановке в очередь критической секции, вычисляется номер присваиваемый процессу.
Алгоритм планирования процессов ***Round Robin (RR)***, величина кванта времени **3.**
Результаты оформить в виде таблицы иллюстрирующей работу процессов, в таблице указывать номер

Таблица 2.2 Варианты заданий

| № | Процессы P0, P1 | Процессы P2, P3 |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 2 | 3 |
| 3 | 4 | 5 |
| 4 | 6 | 7 |
| 5 | 8 | 9 |
| 6 | 10 | 11 |
| 7 | 12 | 13 |
| 8 | 14 | 15 |
| 9 | 1 | 8 |
| 10 | 2 | 9 |
| 11 | 3 | 10 |
| 12 | 4 | 11 |
| 13 | 5 | 12 |
| 14 | 6 | 13 |
| 15 | 7 | 14 |

## Вопросы

1. Охарактеризовать проблему синхронизации потоков.
2. Понятия активностей, детерминированных и недетерминирован активностей.
3. Условия детерминированности активностей Бернстайна.
4. Понятие критической секции.
5. Требования, предъявляемые к алгоритмам взаимодействия процессов.
6. Алгоритм взаимодействия процессов – «Запрет прерываний».
7. Алгоритм взаимодействия процессов – «Пременная замок».
8. Алгоритм взаимодействия процессов – «Строгое чередование».
9. Алгоритм взаимодействия процессов – «Флаги готовности».
10. Алгоритм взаимодействия процессов – Петерсона.
11. Алгоритм «Булочной» взаимодействия процессов.