Алгоритмы синхронизации

Активность "приготовление бутерброда" можно разбить на следующие *атомарные операции*:

* Отрезать ломтик хлеба.
* Отрезать ломтик колбасы.
* Намазать ломтик хлеба маслом.
* Положить ломтик колбасы на подготовленный ломтик хлеба.

Пусть имеется две *активности*

* P: a b cQ: 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

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

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+1Что 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*** .

Теперь сформулируем *условия Бернстайна*.

* Если для двух данных *активностей* P и Q:
* пересечение W(P) и W(Q) пусто,
* пересечение W(P) с R(Q) пусто,
* пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

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.

Таблица 5.1.			
Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Обнаруживает, что хлеба нет		
17-09	Уходит в магазин		
17-11		Приходит в комнату	
17-13		Обнаруживает, что хлеба нет	
17-15		Уходит в магазин	
17-17			Приходит в комнату
17-19			Обнаруживает, что хлеба нет
17-21			Уходит в магазин
17-23	Приходит в магазин		
17-25	Покупает 2 батона на всех		
17-27	Уходит из магазина		
17-29		Приходит в магазин	
17-31		Покупает 2 батона на всех	
17-33		Уходит из магазина	
17-35			Приходит в магазин
17-37			Покупает 2 батона на всех
17-39			Уходит из магазина
17-41	Возвращается в комнату		
17-43			
17-45			
17-47		Возвращается в комнату	
17-49			
17-51			
17-53			Возвращается в комнату

Таблица 5.2.			
Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Достает два батона хлеба		
17-43		Приходит в комнату	
17-47			Приходит в комнату

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*.

Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих *критические участки*, если они могут проходить их в произвольном порядке.

* 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*.
* Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
* Если процесс P_i 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)*.
* 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)*** .
* 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

Запрет прерываний

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

```
while (some condition)
{
запретить все прерывания
critical section
разрешить все прерывания
 remainder section
}
```

Переменная-замок

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

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

Строгое чередование

```
shared int turn = 0;

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

Флаги готовности

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

Алгоритм Петерсона

```
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
}
```

Алгоритм булочной (Bakery algorithm)

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

shared int number[n];

(a,b) < (c,d), если a < c или если a == c и b < d
max(a_0, a_1, ..., a_n) – это число k такое, что k >= a_i для всех i = 0, ...,n

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 }