09

СИНХРОНИЗАЦИЯ И ВЗАИМОИСКЛЮЧЕНИЯ

Курс лекций

«Системное программное обеспечение»
«System Software»
«Операционные системы»

для студентов специальностей АСОИ и ИИ

Павел Кочурко
доцент кафедры ИИТ, к.т.н.

БрГТУ

Активности и операции

- Активность – неатомарна, набор операций
- Операция – атомарна, неделима

P: a b c
Q: d e f

Псевдопараллельное выполнение PQ:

a b c d e f
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

Детерменированность

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

- Возможные результаты:
- (x, y): (3, 4), (2, 1), (2, 3) и (3, 2)

Набор активностей **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**

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.

+: детерминированность выполнения

-: слишком жёсткие условия ☐ фактически невзаимодействующие активности

Взаимоисключение

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет **race condition** (**состояние гонки** , состояние состязания)

Устранение race condition = эксклюзивное право доступа к данным для каждого процесса

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).**

Критическая секция

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 (**критическому ресурсу**), необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом.

Общая структура процесса

```
while (some condition) {
    entry section       пролог критической секции
    critical section    критическая секция
    exit section        эпилог критической секции
    remainder section   остальная часть программы
}
```

Требования к алгоритмам реализации взаимоисключения

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс P_i исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях - **Условие взаимоисключения**
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 - **Условие прогресса**
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 - **Условие ограниченного ожидания**

БрГТУ

Запрет прерываний

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

Сбой в критической секции приведет к неработоспособности системы (с запрещёнными прерываниями)

Применяется как пролог и эпилог к критическим секциям внутри самой операционной системы

Переменная-замок

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

Нарушено условие взаимоисключения: неатомарный пролог приводит к попаданию в критическую секцию двух процессов

Строгое чередование

```
shared int turn = 0;
while (some condition) {
   while(turn != i);
      critical section
   turn = 1-i;
      remainder section
}
```

Взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P_0, P_1, P_0, P_1, P_0, ...

Не удовлетворяет **условию прогресса**. Например, если значение turn равно 1, и процесс P_0 готов войти в критический участок, он не может сделать этого, даже если процесс P_1 находится в 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
}
```

Нарушает условие прогресса: при одновременном выполнении неатомарного пролога с чередованием оба процесса бесконечно ждут друг друга на входе в критическую секцию.

Алгоритм Петерсона

Предложено Деккером, усовершенствовано Петерсоном

```
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
}
```

Удовлетворяет всем условиям

Алгоритм булочной

(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

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

Команда Test-and-Set

```
int Test_and_Set (int *target){
   int tmp = *target;
   *target = 1;
   return tmp;
}

shared int lock = 0;

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

Команда Swap

```
void Swap (int *a, int *b){
   int tmp = *a;
   *a = *b;
   *b = tmp;
}

shared int lock = 0;
int key;

while (some condition) {
   key = 1;
   do Swap(&lock,&key);
   while (key);
      critical section
   lock = 0;
      remainder section
}
```

ВОПРОСЫ?

http://iit.bstu.by/ss