ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция № 08 — Алгоритмы синхронизации

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2023.03.28

Оглавление

Алгоритмы синхронизации

Внешние проблемы кооперации процессов связаны с организацией их взаимодействия.

Даже, если надежная связь процессов друг с другом организована, и они умеют обмениваться информацией, для организации правильного решения задачи взаимодействия этого недостаточно.

Чередование, состояние гонок и взаимоисключения[1]

Активность (сущ.) — *последовательное* выполнение некоторых действий, направленных на достижение определенной цели.

Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных.

Активности можно разбить на некоторые **неделимые** или **атомарные** операции.

Например, активность «приготовление бутерброда» можно разбить на следующие атомарные операции:

1) отрезать ломтик хлеба;
2) отрезать ломтик колбасы;
3) намазать ломтик хлеба маслом;
4) положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь некоторые внутренние невидимые действия (взять батон хлеба в одну руку, взять нож в другую руку, собственно произвести отрезание).

Их называют **неделимыми** потому, что считают их одним целым, выполняемыми за один раз, без прерывания действий.

---

1 Interleaving, Race condition, Mutual exclusion

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 **P Q** мы получаем следующую последовательность атомарных действий:

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

При исполнении этих активностей псевдопараллельно, в режиме разделения времени активности могут расслоиться на неделимые операции с различным их 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
......
d e f a b c
```

Атомарные операции в составе активностей могут чередоваться всевозможными способами с сохранением своего порядка расположения внутри своих активностей.

Поскольку псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, то результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения.

Пример

Пусть у нас есть две активности P и Q, состоящие из двух атомарных операций каждая:

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

В результате их псевдопараллельного выполнения, если переменные **x** и **y** являются общими, для активностей, возможны шесть вариантов чередования и четыре разных результата для пары **(x, y): (3, 4)**, **(2, 1)**, **(2, 3)** и **(3, 2)**.

Определение

Говорят, 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.

Детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени.

Для недетерминированного набора такое исполнение нежелательно.

Для определения, является ли набор активностей детерминированным или нет, существуют **достаточные условия Бернстайна**.

Достаточные условия Бернстайна

Вводятся 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 активности (программы) **IN(P)** представляет собой объединение наборов *входных* переменных для всех ее неделимых действий.

Аналогично, набор выходных переменных активности **OUT(P)** представляет собой объединение наборов *выходных* переменных для всех ее неделимых действий.

В итоге имеем **IN(P)**, **OUT(P)**, **IN(Q)**, **OUT(Q)**. Например, для программы

```
P: x = u + v
   y = x * w
IN(P) = {u, v, x, w}, OUT(P) = {x, y}.
```

Следует заметить, что переменная **x** присутствует как в **IN(P)**, так и в **OUT(P)**.

Итак, условия Бернстайна

Если для двух данных активностей **P** и **Q**:

1) пересечение **OUT(P)** и **OUT(Q)** пусто, (P и Q записывают разные данные)

2) пересечение **OUT(P)** с **IN(Q)** пусто, (Q не читает того, что записывает P)

3) пересечение **IN(P)** и **OUT(Q)** пусто, (P не читает того, что записывает Q)

тогда выполнение **P** и **Q** детерминировано.

Если эти условия не соблюдены, возможно, что параллельное выполнение **P** и **Q** детерминировано, но возможно, что и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

Условия Бернстайна гарантируют детерминизм, но слишком жесткие – они требуют практически невзаимодействующих процессов.

А нам необходимо, чтобы детерминированный набор образовывали активности, ***совместно использующие*** информацию и обменивающиеся ей.

Для этого необходимо ограничить число возможных чередований атомарных операций, исключив некоторые из них с помощью механизмов синхронизации выполнения процессов.

Тем самым это обеспечит упорядоченный доступ процессов к совместно используемым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет состояние гонки[2] или состояние состязания.

В приведенном ранее примере

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

имеет место состязание процессов (race condition) за вычисление значений переменных **x** и **y**.

2 race condition

Взаимное исключение (mutual exclusion)

Если не важна очередность доступа к совместно используемым данным, задачу упорядоченного к ним доступа (устранение *race condition*) можно решить, если обеспечить каждому процессу ***эксклюзивное*** право доступа к этим данным.

Каждый процесс, обращающийся к совместно используемым ресурсам, исключает для всех других процессов возможность общения с этими ресурсами одновременно с ним, если это может привести к недетерминированному поведению набора процессов.

Такой прием называется **взаимоисключением** (**mutual exclusion**).

Если же для получения правильных результатов очередность доступа к совместно используемым ресурсам важна, то одними взаимоисключениями уже не обойтись.

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 состояния гонок по отношению к некоторому ресурсу.

Чтобы исключить эффект гонок, 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 выполняется как атомарная операция***.

Пример

Существует пример, в котором *псевдопараллельные взаимодействующие процессы* представлены действиями различных студентов (три американских студента и пустой холодильник).

Критический участок представлен набором действий от «обнаружил, что пива нет» до «вернулся из магазина».

В результате отсутствия взаимоисключения из ситуации «нет пива» возникает ситуация «слишком много пива».

Если бы этот критический участок выполнялся как атомарная операция — «достает из холодильника ящик пива», то проблема образования излишков была бы снята.

Процесс добывания пива атомарной операцией можно было бы сделать следующим образом:

- перед началом этого процесса закрыть дверь изнутри на засов и уйти за пивом через окно;
- по окончании процесса вернуться в комнату через окно и отодвинуть засов.

В этом случае пока один студент добывает пиво, все остальные находятся в состоянии ожидания под дверью.

Т.о. для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем критическом участке, другие процессы не могли войти в свои критические участки.

Критический участок должен сопровождаться **прологом** (**entry section**) — «закрыть дверь изнутри на засов», и **эпилогом** (**exit section**) — «отодвинуть засов».

Пролог и эпилог не имеют отношения к основной активности одиночного процесса.

Во время выполнения пролога процесс должен, в частности, ***получить разрешение на вход*** в критический участок, а во время выполнения эпилога — ***сообщить другим процессам, что он покинул критическую секцию***.

Пролог (Entry Section) – это часть процесса, которая обеспечивает вход конкретного процесса из многих других процессов в критическую секцию.

Критическая секция (Critical Section) – это часть, в которой только одному процессу разрешено выполняться и изменять общий ресурс. Эта часть процесса гарантирует, что только никакой другой процесс не сможет получить доступ к общему ресурсу (к общим данным, например).

Эпилог (Exit Section) – это часть процесса, которая позволяет другому процессу, ожидающему в *Прологе*, войти в критическую секцию.

Регулярная часть кода (Remainder Section[3]) – так иногда называются все остальные части кода, кроме вышеперечисленных.

---

3 остальная часть

Программные алгоритмы организации взаимодействия процессов

Требования, предъявляемые к алгоритмам

Организация взаимоисключения для критических участков позволяет избежать возникновения состояния гонок, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов.

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

1) Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения (на самом деле сегодня такую машину найти трудно). 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 количестве процессоров, на которых они исполняются.

3) Если процесс P_i исполняется в своей критической секции, то не существует никаких других процессов, которые исполняются в своих соответствующих критических секциях.

Это условие получило название условия взаимоисключения (**mutual exclusion**).

4) Процессы, которые находятся вне своих критических секций и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические секции.

Если нет процессов в критических секциях, и имеются процессы, желающие войти в них (в прологе), то только те процессы, которые не исполняются в регулярной секции (**regular section**), должны принимать решение о том, какой процесс войдет в свою критическую секцию (только те, что в прологе). Такое решение не должно приниматься бесконечно долго.

Это условие получило название условия прогресса (**progress**).

Регулярная часть кода
Пролог
Критическая секция
Эпилог
Регулярная часть кода

5) Не должно возникать бесконечного ожидания для входа процесса в свою критическую секцию. 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 секции лишь ограниченное число раз.

Это условие получило название условия ограниченного ожидания (**bound waiting**).

Описание соответствующего алгоритма означает описание способа организации пролога и эпилога для критической секции.

Запрет прерываний

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

```
regular_section
interrupts_disable – запретить все прерывания
   critical_section
interrupts_enable – разрешить все прерывания
regular_section
```

Поскольку, выход процесса из состояния исполнения без его завершения осуществляется по прерыванию, то внутри критической секции никто не может вмешаться в его работу.

Однако такое решение чревато далеко идущими последствиями, поскольку разрешает процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе.

Допустим, что в результате ошибки или злого умысла пользователь запретил прерывания в системе и зациклил или завершил свой процесс.

Тем не менее, запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например, при обновлении содержимого PCB — блока управления процессами.

CLI/STI – запрет/разрешение прерываний.

Переменная-замок (lock)

Определим переменную **lock**, доступную всем процессам, и положим ее начальное значение равным 0.

Процесс может войти в критическую секцию только тогда, когда значение этой переменной равно 0, одновременно со входом изменяя ее значение на 1 — закрывая замок.

При выходе из критической секции процесс сбрасывает ее значение в 0 — замок открывается.

```
shared int lock = 0; // свободно/занято

regular section
while(lock);
    lock = 1; // не атомарное действие
    critical section
    lock = 0;
regular section
```

К сожалению, такое решение, не удовлетворяет условию взаимоисключения, так как действие **while(lock); lock = 1;** не является атомарным.

Строгое чередование

Попробуем решить задачу сначала для двух процессов — **P0** и **P1**

Как и в предыдущем используется общая для них обоих переменная с начальным значением 0.

Пусть имя ей **turn**.

Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него — если **turn == 0**, в КС входит **P0**, если **turn == 1**, в КС входит **P1**.

```
shared int turn = 0; // чья очередь
...
regular section before
while(turn != i);    // ждем, пока второй не выйдет
   critical section
turn = 1-i;
regular section after
```

В этом случае взаимоисключение гарантируется и процессы входят в критическую секцию строго по очереди: **P0**, **P1**, **P0**, **P1**, ...

Но наш алгоритм не удовлетворяет условию прогресса — процессы, которые находятся вне своих критических участков и не собираются входить в них, не должны препятствовать другим процессам входить в их собственные критические участки.

Например, если **turn == 0** и процесс **P1** готов войти в критический участок, он не может сделать этого, если процесс **P0** находится в регулярной секции **before**.

Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени.

Пусть два наши процесса имеют совместно используемый массив флагов готовности входа процессов в критический участок

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

Когда k-й процесс готов войти в КС, он присваивает своему элементу массива **ready[i]** значение равное **1**. После выхода из КС он сбрасывает это значение в **0**.

Процесс не входит в критическую секцию, если другой процесс уже готов ко входу в критическую секцию или находится в ней.

```
...
ready[i] = 1;
while(ready[1-i]);
   critical_section
ready[i] = 0;
...
```

Алгоритм обеспечивает взаимоисключение и 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**).

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;

...
ready[i] = 1;                          (1)
turn     = 1-i;                        (2)
while(ready[1-i] && turn == 1-i);
   critical section
ready[i] = 0;
...
```

При исполнении пролога критической секции процесс заявляет о своей готовности выполнить критический участок (1) и одновременно предлагает другому процессу приступить к его выполнению (2).

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 последует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Все пять требований данным алгоритмом удовлетворяются.

Алгоритм регистратуры (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов.

Соответствующий алгоритм для нескольких взаимодействующих процессов получил название алгоритм регистратуры (булочной).

Каждый вновь прибывающий клиент (процесс) получает талончик на обслуживание с номером.

Клиент с наименьшим номером на талончике обслуживается следующим.

Из-за неатомарности операции вычисления следующего номера не гарантируется, 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).

Совместно используемые структуры данных для алгоритма — два массива

```
shared enum {false, true} choosing[n]; // получает талончик
shared int number[n];                  // номер талончика
```

Изначально элементы этих массивов инициируются значениями **`false`** и **`0`** соответственно.

Введем следующие обозначения:

1) $(a.b) < (c.d)$, если $a < c$ или если $a = c$ и $b < d$

2) $\sup(a_0, a_1, \ldots, a_n)$ – это число k такое, что $k \geq a_i$ для всех $i = 0, \ldots, n$

Структура алгоритма для процесса P[k]

```
shared enum {false, true} choosing[n]; // получает талончик
shared int number[n];                  // номер талончика


...
do {
     choosing[i] = true;
     number[i] = sup(number[0], number[1], …, number[n – 1]) + 1; (автоинкремент)
     choosing[i] = false;

     for (j = 0; j < n; j++) {                 // ждем, пока не подойдет очередь
         while (choosing[j]);                  // ждем, пока j получает талончик
         while ((number[j] != 0) &&            // ждем если j в очереди и
                (number[j],j)<(number[i],i)); // очередь i (наша) после j
     }
     critical section
     number[i] = 0;
     remainder section
} while (1);
```

Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность.

Мы уже обращались к аппаратной поддержке для решения задачи реализации взаимоисключений, когда говорили об использовании механизма запрета/разрешения прерываний (**`cli`**/**`sti`**).

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции.

Концепции обеих команд могут быть использованы для реализации взаимоисключений.

Команда Test-and-Set (проверить и установить в 1)

О выполнении команды **Test-and-Set**, осуществляющей проверку значения логической переменной с одновременной установкой ее значения в **1**, можно думать, как о выполнении функции

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

С использованием этой атомарной команды мы можем модифицировать алгоритм для переменной-замка, так чтобы он обеспечивал взаимоисключения

```
shared int lock = 0;

...
while(Test_and_set(&lock));
    critical section
lock = 0;
...
```

К сожалению, полученный алгоритм не удовлетворяет условию ограниченного ожидания. Процесс может ждать неограниченно долго, пока переменная-замок освободится.

Команда Swap (Обменять значения)

Выполнение команды **Swap**, обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией

```
void Swap(int *a, int *b) {

   int tmp = *a;
   *a      = *b;
   *b      =  tmp;
}
```

Применяя атомарную команду **Swap**, мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную **key** локальную для каждого процесса:

```
shared int lock = 0;
int key;

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

XCHG — (Exchange) обмен операндов

```
XCHG    r1, r2
XCHG    r/m, r
```

Описание

Обменивает содержимое целевого (первого) и исходного (второго) операндов.

Операндами могут быть два регистра общего назначения или регистр и ячейка памяти.

Если имеется ссылка на операнд памяти, на время операции обмена автоматически реализуется протокол блокировки процессора независимо от наличия или отсутствия префикса **LOCK** или значения **IOPL**.

Инструкция заменяет три инструкции **MOV** и не требует временного сохранения содержимого операнда, пока загружается другое.

Эта инструкция полезна для реализации семафоров или аналогичных информационных объектов для синхронизации процессов.

Инструкция **XCHG** также может использоваться вместо инструкции **BSWAP** для 16-битных операндов.

Флагов не изменяет.

CMPXCHG — (compare and exchange) сравнить и обменять

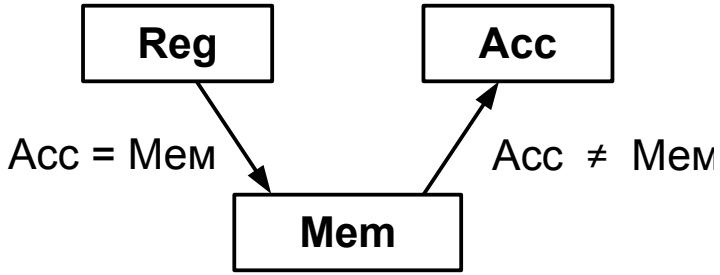
```
CMPXCHG     r/m, r
```

Сравнивает значение в регистре **AL**, **AX**, **EAX** или **RAX** (регистр аккумулятора **Acc**) с первым операндом (операндом назначения). Если два значения равны, второй операнд (исходный операнд) загружается в операнд назначения. В противном случае операнд назначения загружается в регистр **AL**, **AX**, **EAX** или **RAX**. Регистр **RAX** доступен только в 64-битном режиме.

Эта инструкция может использоваться с префиксом **LOCK**, чтобы позволить ее атомарное выполнение.

Инструкция используется для синхронизации операций в системах, использующих несколько процессоров, а также для работы с семафорами.

Инструкция **CMPXCHG** проверяет, свободен ли семафор. Если семафор свободен, он помечается как выделенный; в противном случае инструкция получает идентификатор текущего владельца.



Все это делается за одну непрерывную операцию. В однопроцессорной системе инструкция CMPXCHG избавляет от необходимости переключаться на уровень защиты 0 (для отключения прерываний) перед выполнением нескольких инструкций для проверки и модификации семафора.

Флаги ZF устанавливается, если **Mem == Acc**; в противном случае он очищается.

CF, **PF**, **AF**, **SF** и **OF** устанавливаются по результатам операции сравнения.

Механизмы синхронизации

Рассмотренные выше алгоритмы хотя и являются корректными, но:

- достаточно громоздки и не обладают элегантностью;
- процедура ожидания входа в критический участок включает в себя достаточно длительное вращение процесса в пустом цикле, пожирая драгоценное время процессора.

Существуют и другие серьезные недостатки у алгоритмов, построенных средствами обычных языков программирования.

Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них H — высокоприоритетный, а другой L — низкоприоритетный.

Пусть планировщик устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он готов к исполнению, и занимает процессор на все время своего выполнения (если не появится процесс с еще большим приоритетом).

Тогда в случае, когда процесс L находится в своей критической секции, а процесс H, получив процессор, подошел ко входу в критическую область, мы получаем тупиковую ситуацию — процесс H не может войти в критическую область, находясь в цикле, а процесс L не получает управления, чтобы покинуть критический участок.

Для того чтобы устранить возникновение подобных проблем были разработаны различные механизмы синхронизации более высокого уровня:

- семафоры;
- мониторы;
- сообщения.

Семафоры

Семафор (semaphore) — объект, ограничивающий количество процессов/потоков, которые могут войти в участок кода критической секции. Определение введено **Эдсгером Дейкстрой**.

Семафоры используются для синхронизации и защиты передачи данных через совместно используемую память, а также для синхронизации работы процессов и потоков.

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции:

P (от датского слова proberen — проверять);
V (от verhogen — увеличивать).

Сегодня эти операции называют **down** и **up**.

Операция **P(S)** или **down(s)** выясняет, отличается ли значение семафора **S** от **0**.

Если отличается, она уменьшает это значение на **1** и выполнение продолжается (входит в критическую секцию).

Если значение == **0**, процесс приостанавливается, не завершая операцию **down** (блокируется).

Все операции — проверка значения семафора, его изменение, и, возможно, приостановка процесса осуществляются как единое и неделимое атомарное действие. Тем самым гарантируется, что с началом семафорной операции никакой другой процесс не может получить доступ к семафору до тех пор, пока операция не будет завершена (**`S >= 0`**) или заблокирована (**`S == 0`**).

Операция **V(S)** или **up(S)** увеличивает значение переменной семафора, на **1**.

Если с этим семафором связаны один или более приостановленных процессов, способных завершить ранее начатые операции **down**, система выбирает один из них и позволяет ему завершить его операцию **down**.

Таким образом, после применения операции **up** в отношении семафора, с которым были связаны приостановленные процессы, значение семафора так и останется нулевым, но количество приостановленных процессов уменьшится на **1**.

Операция увеличения значения семафора на 1 и активизации одного из процессов также является неделимой – ни один из процессов не может быть заблокирован при выполнении операции **up**.

Подобные переменные-семафоры могут быть с успехом применены для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях применяются через использование системных вызовов.

Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций **down** и **up**, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов.

Если при выполнении операции **P** заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

Решение проблемы producer-consumer с помощью семафоров

Одной из типовых задач, требующих организации взаимодействия процессов, является задача producer-consumer (производитель-потребитель).

Пусть два процесса обмениваются информацией через буфер ограниченного размера.

Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда.

Грубо говоря, на этом уровне деятельность потребителя и производителя можно описать следующим образом.

Producer

```
while(1) {
   produce_item;
   put_item;
}
```

Consumer

```
while(1) {
   get_item;
   consume_item;
}
```

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации.

Если буфер пуст, то потребитель должен дожидаться, пока в буфере не появится сообщение.

Реализация с помощью семафоров

Возьмем три семафора — **items**, **free_space** и **mutex**.

items — потребитель будет ждать, пока в буфере не появится то, что ему нужно.

free_space — производитель будет ждать, пока в буфере не появится место.

mutex — для организации взаимоисключения на критических участках, которыми являются действия **put_item** и **get_item**, поскольку эти операции не могут пересекаться.

Решение задачи организации взаимоисключения на критическом участке и синхронизации скорости работы процессов выглядит так:

```
Semaphore mutex      = 1;
Semaphore free_space = N; // емкость буфера;
Semaphore items      = 0; // пусто

Producer                        Consumer
while(1) {                      while(1) {
    produce_item;                   down(items);
    down(free_space);               down(mutex);
    down(mutex);                    get_item;
    put_item;                       up(mutex);
    up(mutex);                      up(free_space);
    up(items);                      consume_item;
}                                }
```

Мьютексы

Мьютекс (mutex, mutual exclusion — «взаимное исключение») — аналог одноместного семафора, служащий для синхронизации одновременно выполняющихся потоков.

Мьютекс отличается от семафора тем, что только владеющий им поток может его освободить.

Мьютексы — это один из вариантов семафорных механизмов для организации взаимного исключения. Они реализованы во многих ОС, их основное назначение — организация взаимного исключения для потоков из одного и того же или из разных процессов.

Мьютексы могут находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт, соответственно, или свободен/захвачен).

Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта **mutex**, он переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом.

В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом. Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток засыпает до тех пор, пока мьютекс не будет освобождён.

Цель использования мьютексов — защита данных от повреждения в результате асинхронных изменений (состояние гонки), однако мьютексы могут порождать другие проблемы — такие, как взаимная блокировка (клинч).

Немного об отличии одноместных (бинарных, двоичных) семафоров и мьютексов

Строго говоря, **мьютекс – это механизм блокировки**, используемый для синхронизации доступа к ресурсу. Только одна задача (поток или процесс на основе абстракции ОС) может получить мьютекс. Это означает, что с мьютексом будет связано владение, и только владелец может снять блокировку (мьютекс). Мьютексы всегда используют следующую последовательность:

```
SemTake (захватить)
    Критическая секция
SemGive (отдать)
```

Семафор – это сигнальный механизм (вид сигнала «Вы можете продолжить»). Например, задача откладывается в ожидании чего-либо (например, срабатывания датчика, места в очереди, …). Когда датчик срабатывает (появляется место в очереди), семафор уведомляет задачу о срабатывании (появлении места в очереди). Задача продолжает выполнение, предпринимает соответствующие действия по срабатыванию датчика (появлению места в очереди), затем возвращается в состояние ожидания.

Таким образом, **мьютекс** можно рассматривать как ***токен***, передаваемый от задачи к задачам, а семафор – как сигнал светофора (он сигнализирует кому-то, что он может продолжить).

На теоретическом уровне семантически они не отличаются. Можно реализовать мьютекс с использованием семафоров и наоборот. На практике же реализации разные, и они предлагают немного разные услуги.

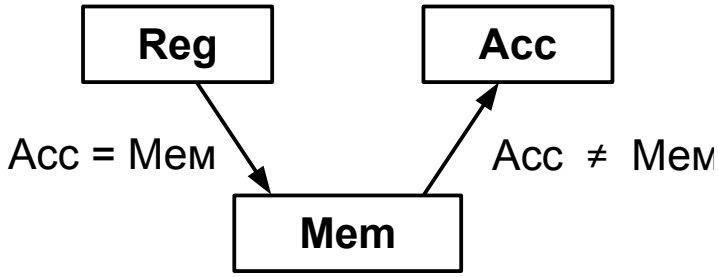
Практическое отличие (с точки зрения окружающих их системных служб) состоит в том, что реализация мьютекса нацелена на то, чтобы быть более легким механизмом синхронизации.

Мьютексы называются защелками, а семафоры — ожиданиями.

На самом низком уровне и семафор и мьютекс оба используют какой-либо атомарный механизм **test&set**.

Этот механизм считывает текущее значение ячейки памяти, вычисляет своего рода условное выражение и записывает значение в этой ячейке в одной атомарной инструкции, которая не может быть прервана. Это означает, что можно «приобрести» мьютекс или проверить, кто его был ли он у кого-нибудь до этого.

CMPXCHG — (compare and exchange) сравнить и обменять



Типичная реализация

Типичная реализация мьютекса состоит в том, что процесс или поток, выполняет инструкцию **test-and-set** и узнает, захватил ли этот мьютекс кто-нибудь еще.

Ключевым моментом здесь является отсутствие взаимодействия с планировщиком, поэтому неизвестно (это и не нужно), кто установил блокировку. Затем мы либо отказываемся от нашего кванта времени и пытаемся выполнить его снова, когда задача будет перепланирована, либо выполняем спин-блокировку.

Спин-блокировка

Устанавливаем счетчик обратного отсчета, например, в 500 и выполняем в цикле:

1. Выполняем инструкцию **`test-and-set`**.

2. Проверяем, очищен ли мьютекс, если да, мы завладели им в предыдущей инструкции и можем выйти из цикла.

3. Когда мы дойдем до нуля, отказываемся от нашего кванта времени.

Пример[4]

Мьютекс

Ключ от туалета.

Только один человек может иметь ключ и занимать туалет одновременно.

Когда заканчивает, он дает (освобождает) ключ следующему человеку в очереди.

Официально: «Мьютексы обычно используются для сериализации доступа к разделу реентерабельного кода, который не может выполняться одновременно более чем одним потоком. Объект мьютекса позволяет только одному потоку войти в контролируемую секцию, вынуждая другие потоки, которые пытаются получить доступ к этот раздел, чтобы дождаться, пока первый поток не выйдет из этой секции"»

Семафор

Это некоторое количество одинаковых бесплатных ключей от туалета.

Например, у нас есть четыре туалета с одинаковыми замками и ключами. Счетчик семафоров – счетчик ключей – вначале установлен на 4 (все четыре туалета свободны), затем значение счетчика уменьшается по мере того, как входят люди. Если все туалеты заполнены, т.е. свободных ключей не осталось, счетчик семафоров равен 0. Теперь, когда один человек выходит из туалета, семафор увеличивается до 1 (один свободный ключ) и передается следующему человеку в очереди.

Официально: «Семафор ограничивает количество одновременных пользователей общего ресурса до некоторого максимального числа. Потоки могут запрашивать доступ к ресурсу (уменьшая семафор) и могут сигнализировать, что они закончили использование ресурса (увеличивая семафор).»

---

4 Библиотека разработчика Symbian

Сообщения

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи — **send** и **receive**.

В случае прямой адресации их обозначают обычно так:

`send(P, message)` — послать сообщение **`message`** процессу P;

`receive(Q, message)` — получить сообщение **`message`** от процесса Q.

В случае непрямой адресации их обозначают обычно так:

`send(A, message)` — послать сообщение **`message`** в почтовый ящик A;

`receive(A, message)` — получить сообщение **`message`** из почтового ящика A.

Примитивы **send** и **receive** уже имеют скрытый механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер.

Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной.

Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

Мониторы

Решение задачи «производитель-потребитель» с помощью семафоров выглядит достаточно элегантно, однако программирование с их использованием требует повышенной осторожности и внимания, чем, отчасти, напоминает программирование на языке ассемблера.

Допустим, что в рассмотренном примере мы случайно поменяли местами операции **down**, сначала выполняя ее для семафора **mutex**, а уже затем для семафоров **free_space** и **items**.

В этом случае потребитель, войдя в свой критический участок (**mutex** сброшен), обнаруживает, что буфер пуст. Он блокируется на **items** и начинает ждать непустого буфера.

Производитель при этом не может войти в критический участок для заполнения буфера, так как тот заблокирован потребителем. Результат – тупик.

В сложных программах произвести анализ правильности использования семафоров с карандашом в руках становится очень непростым занятием. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от interleaving'а атомарных операций, и ошибки могут быть трудно воспроизводимы.

Для того чтобы облегчить труд программистов, в 1974 году Хором (Hoare) был предложен механизм еще более высокого уровня, чем семафоры, получивший название мониторов.

Итак, мониторы

Монитор — высокоуровневый механизм взаимодействия и синхронизации процессов, обеспечивающий доступ к общим ресурсам (аппаратура или набор переменных).

Компилятор или интерпретатор прозрачно для программиста вставляет код блокировки-разблокировки в оформленные соответствующим образом процедуры, избавляя программиста от явного обращения к примитивам синхронизации.

Монитор — это специальный объектный тип данных, состоящий из:

- переменных, связанных с общим ресурсом;
- набора процедур, взаимодействующих с этим ресурсом (функции-методы);
- мьютекса;
- инварианта, определяющего условия, позволяющие избежать состояние гонки.

Процедура монитора захватывает мьютекс перед началом работы и держит его или до выхода из процедуры, или до момента ожидания условия.

Если каждая процедура в составе монитора гарантирует, что перед освобождением мьютекса инвариант истинен, то никакая задача не может получить ресурс в состоянии, ведущем к гонке.

Значения переменных снаружи монитора могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, функции-методы могут использовать только данные, находящиеся внутри монитора и свои параметры.

На абстрактном уровне можно описать структуру монитора следующим образом:

```
monitor monitor_name {
    variable declarations; // переменные состояния монитора
    void m1(...) {...}
    void m2(...) {...}
    ...
    void mn(...) {...}
    {конструктор -- блок инициализации внутренних переменных;}
}
```

Функции **m1(), m2(), ..., mn()** представляют собой функции-методы монитора, а блок инициализации внутренних переменных содержит операции, которые выполняются только один раз – при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

В C++ может быть реализован как класс, который инициализируется через список инициализации конструктора.

Взаимоисключения

Важной особенностью мониторов является то, что в любой момент времени только один процесс внутри данного монитора может быть активен, т. е. находиться в состоянии «готовность» или «исполнение».

Поскольку мониторы представляют собой особые конструкции языка программирования, то компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующие взаимоисключение.

Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность появления ошибок становится меньше.

Однако одних только взаимоисключений не достаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов — необходимы средства организации очередности процессов, подобно семафорам **free_space** и **items** в примере «Производитель — Потребитель».

Для этого в мониторах было введено понятие **условных переменных** (condition variables), над которыми можно совершать две операции **wait** и **signal**, до некоторой степени похожие на операции P/down и V/up над семафорами.

Мониторы и задача производитель-потребитель

```
monitor Producer_Consumer {
  condition free_space, items;
  int count;

  void put() {
    if(count == N) free_space.wait;
    put_item;
    count += 1;
    if(count == 1) items.signal;
  }
  void get() {
    if (count == 0) items.wait;
    get_item();
    count -= 1;
    if(count == N-1) free_space.signal;
  }
  {count = 0;}
}
```

Producer:

```
while(1) {
  produce_item;
  ProducerConsumer.put();
}
```

Consumer:

```
while(1) {
  ProducerConsumer.get();
  consume_item;
}
```

Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию **wait** над какой-либо условной переменной. При этом процесс, выполнивший операцию **wait**, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода данного монитора выполняет операцию **signal** над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным.

Если несколько процессов дожидались операции **signal** для этой переменной, то активным становится только один из них.

Что нам нужно предпринять для того, чтобы не нарушились условия взаимоисключения — т.е. у нас не оказалось двух одновременно активных внутри монитора процессов, разбудившего и пробужденного?

Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор.

Несколько позже Хансен (Hansen) предложил другой механизм — разбудивший процесс покидает монитор немедленно после исполнения операции **signal**.