Синхронизация

Синхронизация

Как мы помним ОС служит для управления ресурсами. Если есть набор ресурсов, то есть и множество некоторых процессов или потоков, которым нужны эти ресурсы. Отсюда вытекает проблема – как этому набору процессов сопоставить набор ресурсов, ведь ресурсы могут быть нужны в определенное время или одновременно. Нужно управлять доступом к ресурсам.

Нужны примитивы и синхронизация.

Как организуется управление, как организуется синхронизация доступа процессов к определенным ресурсам?

Критические секции (Critical section)

Последовательность инструкций, **одновременное выполнение** которой может привести к неправильным результатам называется **критической секцией**.

Это основной примитив, применяемый в синхронизации. Важно в понятии «одновременное выполнение» например в разных потоках одного процесса, либо в разных процессах.

Атомарная операция – выполняется без прерывания, либо все, либо ничего

Критические секции (Critical section)

Для гарантии правильных результатов необходимо гарантировать взаимное исключение (mutual exclusion) между двумя критическими секциями достаточно для корректного исполнения.

Один из способов гарантировать взаимное исключение – использовать блокировки

LOCK – по англ. замок, защелка

Пример Критической секции

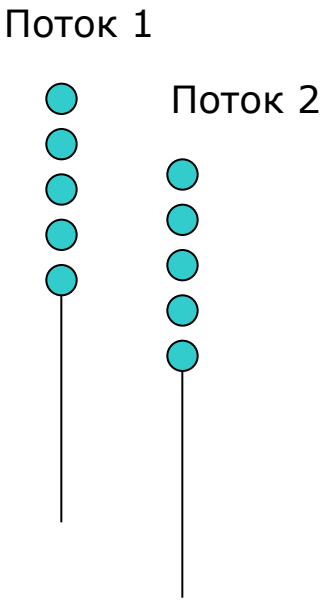


Рис 1

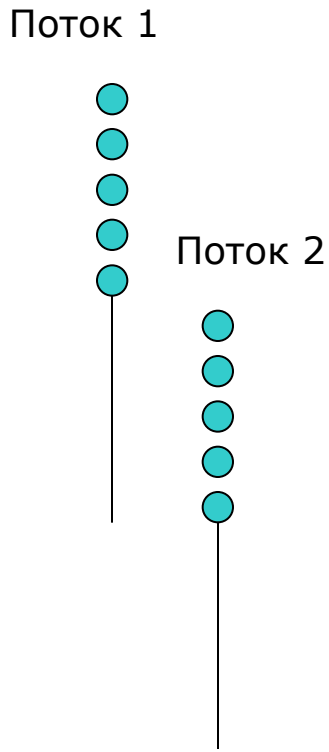


Рис 2

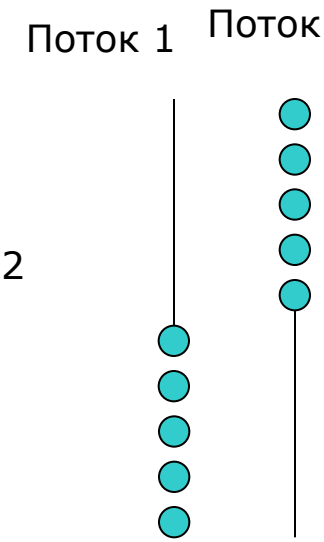


Рис 3

есть 2 потока, выполняются одновременно. Кружочками отмечены их критические секции.

Когда они выполняются, как показано на рис.1, может получится некорректный результат. Если потоки выполняются как на рис, 2и3 результат будет правильный. Задача синхронизации из рис.1 сделать 2 или 3

Условия создания критических секций. Когда получается некорректный результат.

1) Код должен выполняться одновременно (либо это одновременное выполнение на двух физических процессорах или двух выч.ядрах одного процессора, либо это одновременное выполнение в результате многозадачности)
2) Выполнение последовательности действий: прочитать-изменить – записать
3) Критические секции имеют одну или более общих переменных. Общие переменные могут быть:

   - глобальные и выделены из кучи (доступ к ним будет из двух и более выполняемых критических секций)

   -любые др. нелокальные переменные, например в стэке.

Классический пример - Функция снятия денег со счета фирмы в банке.

В банке у компании есть счет, на нем лежат деньги. С него банковская программа переводит деньги на персональные счета сотрудников. Банк написал хитрое ПО, при котором данная операция происходит не последовательно, а параллельно (запускается сразу много потоков и операция осуществляется).

Код функции (язык СИ)

```
// функция, которая принимает на вход 2 параметра
Void WithdrawMoney(account, amount) {
                   номер счета   сумма денег
 //далее выполняется 3 действия
1 int balance = GetBalance(account); //чтение –
    считываем баланс денег на счете фирмы
2 balance - = amount; // изменение – вычитаем из
    баланса сумму, которую надо перевести работнику
3 SetBalance(account, balance); //запись, –
    устанавливаем значение счета фирмы в новое
    значение
4 GiveCashTonser(); // выдача наличных, зачисление на
    счет работника
```

Проблема – есть неопределенность

- Если на счете 100000 и надо сразу двум работникам списать по 10000, то фирма может пострадать.
- Надо написать программу для двух потоков, которая будет выполняться одновременно в двух местах.
- Операции разбиты, поэтому может произойти следующее: в двух местах одновременно считали один и тот же баланс, потом его уменьшили в 2 раза, записали и в итоге вышло непонятно что – грязное чтение, запись, некорректный результат.
- Чтобы данной проблемы не было нужно, чтобы в один момент времени операции 1+2+3 должны выполняться только одним потоком. Операция 4 – зачисление на счет работника может производится одновременно.
- Нужно гарантировать, что 3 операции пройдут в нужной последовательности. Используя вытесняющую многозадачность это гарантировать невозможно – в любой момент времени по таймеру наш код может прерваться и вытеснен другим, произойдет смена контекст, а потом по таймеру возврат обратно.

Требования к критическим секциям

- Как организовать критические секции так, чтобы они правильно работали? Требования, которым должны удовлетворять правильная организация критических секций

3 основные вещи:

1) **Взаимное исключение**

Максимум один поток может быть в критической секции (т.е. один поток может выполнять КС в один момент времени)

Другой поток вне КС не может вытеснить тот поток, который находится в КС, не может его прервать, вытеснить, переключить контекст.

2) **Требование ограниченного ожидания**

Если поток ожидает входа в КС, то рано или поздно это произойдет, поток не может ждать входа вечно

3) **Производительность**

Накладные расходы на вход и выход из КС малы по сравнению с временем работы выполняемой внутри КС

Механизмы реализации критических секций

Исторически было предложено целый ряд решений:

- **Запрет прерываний** – самый простой способ, но в нем много недостатков
- **Алгоритм Петерсона**
- **Спинлоки** (spinlock) – базовый примитив, который часто используется для построения более сложных блокировок
- **Семафоры**- (non spinning locks) просты для понимания, но сложны в исполнении
- **Мониторы** – примитив синхронизации, высокоуровневая блокировка, требующая поддержки со стороны языка программирования, легко использовать
- **Сообщения** – важное понятие в распределенных системах. Простая модель взаимодействия и синхронизации на основе передаваемых сообщений через канал. Используется в основном в распределенных системах. Взаимный доступ к ресурсу в них осуществляется с использованием модели сообщений.

Есть и другие примитивы синхронизации.

Запрет прерываний

Суть - **Временно отключаются прерывания**.

(-) Недостатки

- В режиме пользователя пользовательская программа не может просто так отключать прерывания, эта опция доступна только ядру ОС, иначе в ОС был бы хаос
- На многопроцессорной системе это работать вообще не будет, ведь у каждого ЦП есть свои прерывания и их надо как то объединять
- Длительный запрет прерываний может привести к неработоспособности устройств.

Данный подход используется только **в ядрах ОС**

Алгоритм Петерсона

Не требует аппаратной поддержки, но он не универсальный, имеет ограничения:

- Работает для ограниченного, определенного заранее количества процессов для 2
- Не требует аппаратной поддержки атомарных(непрерывных) операций
- Если мы возьмем алгоритм перечисления з/пл, то по А.П. он будет выглядеть так:

Алгоритм Петерсона

Int turn; // Чья очередь?
Int interested[2];// массив «интерес о входе», исходно все значения равны false
1 **Void EnterRegion(int process);** //функция входа в КС, process (процесс) – число 0 или 1
(int other = 1-process; //номер второго процесса
Interested[process]=TRUE; //обозначение интереса о входе (чья очередь?)
Turn=process; //установка флага – переменная выбирает один из 2-х процессов, т.е. процесс ставит заинтересованность (в Turn- тот процесс, кто будет ждать)
While(turn==process && interested[other]; // активное ожидание с условием – проверяет – пока интерес у одного не пропал, второй стоит и ждет
2 **Void LeaveRegion(int process);** // process, который выходит. Функция, которая вызывается при выходе из КС
)
(
Interested[process]=FALSE; сообщаем о выходе из КС
)

Пояснения к алгоритму Петерсона

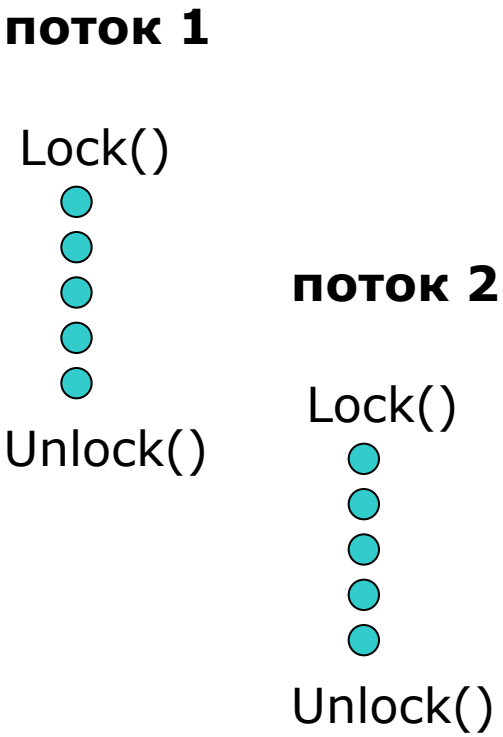
- Используя алгоритм Петерсона в алгоритме перечисления з/пл перед тремя операциями необходимо вызвать функцию №1АП, далее выполнить три неделимых атомарных действия и покинуть эту КС функцией 2АП
- Реализация достаточно простая, вся ее прелесть в том, что она работает без какой –либо аппаратной поддержки
- Есть 2 переменные, один массив – интерес о входе, второй процесс, тот , который хочет занять эту КС.
- Есть активное ожидание с условием.
- Переменная Turn – это тот, кто будет ждать. Если мы ждем и другой процесс еще есть, то мы продолжаем ждать.
- Если мы ждем и другого процесса уже нет, то мы пользуемся условием **turn==process && interested[other].** Оно выполняется если у другого процесса есть интерес
- **Process –** аргумент функции, это не глобальная переменная, поэтому ее чтение «грязным» не будет.

Блокировщики /защелки (locks) - замок

- Защелка – это объект с двумя операциями:
  - Lock() –войти в критическую секцию – операция входа в КС(операция «взятия» защелки).
  - Unlock() – выйти из КС

  Lock() - Эта операция блокирует дальнейшее выполнение потока до тех пор, пока не будет выполнен вход в КС. Можно сказать, что поток держит эту защелку, а потом освобождает ее.

Блокировщики /защелки (locks) - замок



Первый поток входит в КС первым. Второй попытался тоже войти в эту же КС, но у второго потока начинается ожидание. Управление не вернется во второй поток, пока в первом не произойдет выход(т.е.освобождение защелки и выход из КС)

Пример с банком и защелкой

// функция, которая принимает на вход 2 параметра
Void WithdrawMoney(account, amount) {
Lock(global_lock); // закрыли защелку
//далее выполняется 3 действия
1 int balance = GetBalance(account); //чтение – считываем баланс денег на счете фирмы
2 balance - = amount; // изменение – вычитаем из баланса сумму, которую надо перевести работнику
3 SetBalance(account, balance); //запись, – устанавливаем значение счета фирмы в новое значение
UnLock(global_lock); // освободили защелку
4 GiveCashTonser(); // выдача наличных, зачисление на счет работника

В итоге 3 действия выполняются атомарно (без прерывания)

Спинлок (защелка)

Часто применяемый примитив синхронизации, так как работает быстро и простой для понимания.

Реализуется аппаратно. Существует 2 способа реализации:

- Производители процессоров ввели удобную инструкцию, которая называется Test_and_set
- Временное отключение прерываний на период выполнения спинлока

Спинлок (защелка) Аппаратный Test_and_set

В каждом ЦП достаточно давно введена и является ключевой следующая атомарная операция

```
Int TestAndSet(int*val) {
Int oldValue=*val; //считываем старое значение
*val=1; //устанавливаем новое значение
Return oldValue;//возвращаем старое значение вызывающего
}
```

Для ЦП эти 3 инструкции воспринимаются как одна и не делятся.

Код Спинлока (защелка) – в ней есть некий цикл активного ожидания

```
Typedef struct spinlock_s {
int acguired=0;
} spinlock_t
Void Lock(spilock_t *s {
While (TestAndTest(&s->acguired)); //процессорная инструкция
    применяется как функция над значением
}
Void UnLock(spinlock_t *s) {
S->acguired=0; //эквайер
}
```

Если защелка априори никем не “взята”, то она проходит сразу и мы ее берем, и выполняем аппаратно 3 неразрывных действия, если она кем то взята, то мы ждем выполнениях 3 действий и пока эквайер не станет =0 и будет произведен выход из КС.

Т.О. два потока не войдут в КС одновременно, без взаимного исключения

Проблемы Спинлока

Неэффективность: Если поток заблокирован спинлоком, то он будет находится в цикле до окончания кванта времени ЦП

Спинлоки используются как примитивы для создания средств высокоуровневой синхронизации.

Мы рассмотрели базовые низкоуровневые подходы синхронизации. На основе них строится более высокоуровневая синхронизация, имеющая более совершенный механизм.

Высокоуровневая синхронизация

- Семафоры
- Мониторы
- Тупиковые ситуации

Семафоры

Примитив синхронизации, который в теории ОС проходит красной нитью. Это более высокий уровень абстракции, чем защелки.

Его изобретатель Dijkstra (Нидерланды, 1968г)

Семафор – это переменная. Которая может управлять с помощью двух операций P и V

- P(sem) – ожидать, пока переменная семафора sem не станет больше 0, затем уменьшить ее на 1 и продолжить
- V(sem) - увеличить sem на 1
- P() – ожидать Probeer –нидерл. пробовать
- V() – увеличить Verhoog – нидерл. увеличить

Важно помнить, когда будем работать с семафором, то перед Verhoog нужно выполнить Probeer, и наоборот.

Два типа Семафоров

- Можно сделать 2 варианта семафоров:
- **Бинарный(мьютексный) семафор**
  - Переменная семафора sem либо 0 либо 1
  - Гарантирует взаимоисключающий доступ к ресурсу
  - Только один поток/процесс может захватить ресурс
  - Логически эквивалентен защелке с блокировкой, а не спинлоку.
- **Считающий семафор**
  - Переменная семафора sem от 0 до N, где N-число единиц потоков, которые могут использовать ресурсы
  - Одновременно до N потоков могут захватывать этот ресурс

Семафоры

- У каждого семафора есть своя, ассоциированная с ним, очередь потоков, ожидающих на нем
- Когда поток вызывает P(sem), то
  - Если семафор доступен(т.е.>0), то уменьшим sem на 1 и вернуть управление потоку
  - Если семафор недоступен(=0), то поместить поток в соответствующую очередь для ожидания и дать управление другому потоку
- Когда поток сделал все свои дела в КС, он вызывает V(sem)
  - Если есть потоки в очереди, ожидающие семафора, разблокировать один из них
  - Если это тот же поток, который вызвал V просто передать ему управление
  - Если нет ожидающих, то просто увеличить sem на 1

Проблемы Семафоров

Они могут быть использованы для решения традиционных задач синхронизации. Но легко совершить ошибки:

- По сути своей семафоры это глобальные переменные с общим доступом
- Нет связи между этими переменными и данными, доступом к которым он управляет
- Нет контроля за их использованием, можно два раза вызвать P из одного и того же потока или забыть вызвать V… освобождение

Появляются ошибки.

Следующим этапом в развитии эволюции синхронизационных примитивов было создание мониторов

Монитор – поддержка синхронизации внутри языка программирования

Монитор – это конструкция яз. программирования, которая поддерживает контролируемый доступ к общим данным. Код синхронизации добавляется компилятором. Решает ключевые проблемы семафоров.

Монитор – это класс, в котором каждый метод автоматически выполняет Lock() при входе в метод и Unlock() при выходе из него.

Поэтому он объединяет:

- Общие структуры данных
- Процедуры, работающие над этими данными (методы объекта)
- Синхронизацию между потоками, вызывающими эти процедуры.

(Т.е. данные с защитой доступа этих данных)

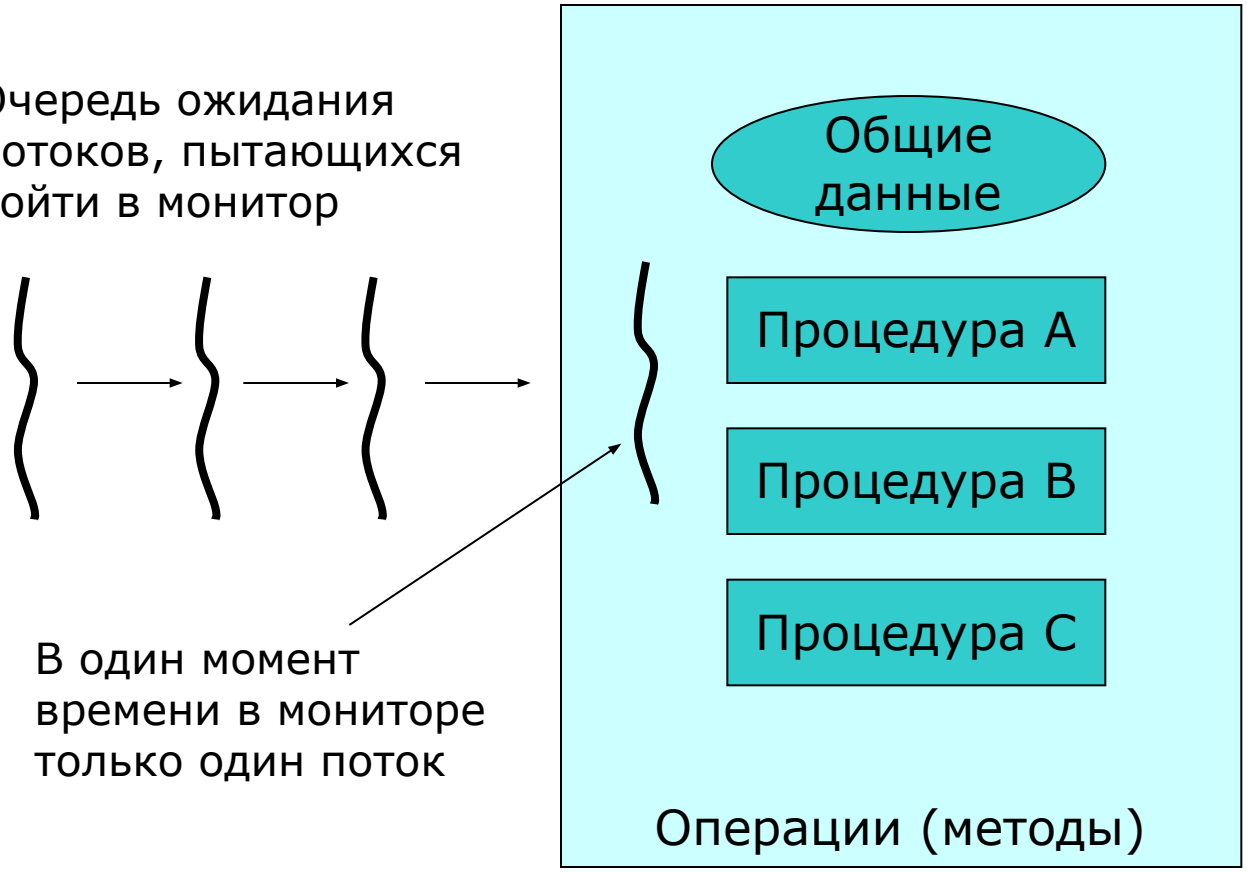
Доступ к данным может быть только из монитора, используя его процедуры

Нет связи между этими переменными и данными, доступом к которым он управляют

- Защита доступа
- Точное понимание, какие данные защищены монитором

Монитор

Монитор представлен как какой-то класс, в нем есть набор процедур с общими данными. Потоки, которые поступают в него. В один момент времени только один поток может работать с этими общими данными. Все это реализуется на уровне языка программирования.



Этот прямоугольник не обязательно один процесс