

Лекция 29

Синхронизация нитей

Thread-Local Storage (TLS)

- По умолчанию все глобальные переменные являются общими для всех нитей
- Каждая нить имеет свой стек, но другие нити имеют доступ в стек нити (например, если был передан адрес)
- Ключевое слово `_Thread_local` (C11) или `thread_local` (C++) для обозначения TLS, например:

```
_Thread_local volatile int count = 5;
```

TLS

- При создании нити выделяется память под TLS
- Начальные значения переменных берутся те, которые были заданы при инициализации
- Каждая нить работает со своей копией TLS
- Все TLS находятся в общем адресном пространстве
- В реализации Linux используется сегментный регистр `gs`

Параллельные процессы

- Параллельные процессы (нити) — процессы (нити), выполнение которых хотя бы частично перекрывается по времени
- Независимые процессы (нити) — используют независимые ресурсы
- Взаимодействующие процессы (нити) — используют ресурсы совместно, выполнение одного может оказать влияние на результат другого
- **Результат выполнения не должен зависеть от порядка переключения между процессами**

Разделяемые ресурсы

```
int amount = 100;
```

```
void retrieve(int m)
{
    amount -= m;
}
```

```
retrieve(30);
```

```
movl    amount, %eax
subl    $30, %eax
movl    %eax, amount
```

```
void deposit(int m)
{
    amount += m;
}
```

```
deposit(10);
```

```
movl    amount, %eax
addl    $10, %eax
movl    %eax, amount
```

Результат?

Разделяемые ресурсы

```
int amount = 100;
```

```
void retrieve(int m)
{
    amount -= m;
}
```

```
retrieve(30);
```

```
movl    amount, %eax
subl    $30, %eax
movl    %eax, amount
```

```
void deposit(int m)
{
    amount += m;
}
```

```
deposit(10);
```

```
movl    amount, %eax
addl    $10, %eax
movl    %eax, amount
```

Правильный: 80
Неправильный: 70
Неправильный: 110

Гонки (race condition)

- Результат работы зависит от порядка переключения выполнения между параллельными процессами
- Очень сложно обнаруживаемые ошибки
- Могут проявляться очень редко при редкой комбинации условий

Критическая секция

- Взаимное исключение — способ работы с разделяемым ресурсом, при котором во время работы процесса (нити) с разделяемым ресурсом другие процессы (нити) не имеют доступ к разделяемому ресурсу
- Критическая секция — фрагмент кода процесса, который выполняется в режиме взаимного исключения

Требования к механизмам взаимного исключения

- Корректность: только один процесс может находиться в критической секции в каждый момент
- Не должно быть никаких предположений о количестве процессоров или скорости работы процессов
- Процесс вне критической секции не должен быть причиной блокировки других процессов
- Справедливость: не должна возникать ситуация, когда некоторый процесс никогда не получит доступа в критическую секцию
- Масштабируемость: процесс в состоянии ожидания не должен расходовать процессорного времени

Пример (наивный)

```
int s = 1;  
int amount = 100;
```

```
void lock()  
{  
    while (s == 0) ;  
    s = 0;  
}  
void unlock()  
{  
    s = 1;  
}
```

**Не обеспечивается корректность!
Используется активное ожидание!**

**Требуется: атомарность операции
проверки значения и установки его в 0,
изменение состояния ожидающего процесса,
оповещение ожидающих процессов**

```
void retrieve(int m)  
{  
    lock();  
    amount -= m;  
    unlock();  
}
```

```
void deposit(int m)  
{  
    lock();  
    amount += m;  
    unlock();  
}
```

Семафор

- Семафор — это переменная s (целого типа), над которой можно выполнять две операции:
- $\text{down}(s, v)$ — если значение $s \geq v$, то $s = s - v$; в противном случае процесс блокируется — помещается в список процессов, ожидающих освобождения данного семафора
- $\text{up}(s, v)$ — $s = s + v$, разблокировать все процессы в списке ожидания
- Операции down и up атомарны

Семафоры

- Если максимальное значение семафора $= 1$, то есть даны операции $up(s)$, $down(s)$ — это **бинарный** семафор
- Если максимальное значение > 1 , это — **считающий** семафор
- Поднимать семафор может любой процесс/нить, не обязательно тот, кто его опускал
- Семафоры могут использоваться и для взаимного исключения, и для посылки нотификаций (один процесс/нить может разбудить другой процесс/нить)

Мьютексы

- Мьютекс (mutex — mutual exclusion) — это специальный вид семафора
- Мьютекс может находиться в состоянии 0 (закрыт) и в состоянии 1 (открыт)
- У закрытого мьютекса есть процесс-владелец, только владелец может открыть мьютекс.

Мьютексы pthread

```
int pthread_mutex_init(pthread_mutex_t *mutex, |
                        const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Мьютекс должен быть предварительно проинициализирован
- Исходное состояние — открыт
- Различаются рекурсивные и нерекурсивные мьютексы

Рекурсивные мьютексы

- У обычных мьютексов две операции lock одной нитью подряд приведут к дедлоку
- У рекурсивных мьютексов повторные lock той же самой нитью увеличивают счетчик вложений, unlock уменьшают счетчик вложений

Рекурсивные мьютексы

```
pthread_mutex_t mutex;  
pthread_mutexattr_t attr;
```

```
pthread_mutexattr_init(&attr);  
pthread_mutexattr_settype(&attr,  
    PTHREAD_MUTEX_RECURSIVE);  
pthread_mutex_init(&mutex, &attr);  
pthread_mutexattr_destroy(&attr);
```


Монитор

- Монитор — это совокупность некоторых переменных и методов, т. е. класс
- В каждый момент времени может выполняться не более одной процедуры, манипулирующей с этими переменными
- Поддержка мониторов находится на уровне языка программирования (Ada, Java, C#)
- Обычная реализация монитора — с помощью рекурсивных мьютексов

Пример монитора (Java)

```
class Account
{
    private double amount;
    public synchronized void update(double m)
    {
        amount += m;
    }
    public synchronized double get()
    {
        return amount;
    }
}
```