Лекция 19
Синхронизация нитей

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;
}
```

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

retrieve(30);

deposit(10);

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

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

Результат?

Разделяемые ресурсы

volatile int amount = 100;

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

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

retrieve(30);

deposit(10);

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

```
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)

```
volatile int s = 1;
volatile 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();
}
```

Atomic (C++ 11)

- Можно использовать с любым типом, но не с любым типом он будет реализован на уровне инструкций процессора

```
std::atomic<int> p{1};
```

- Volatile подразумевается
- Предоставляются разные операции, например

```
int cur = p.exchange(0, std::memory_order_acquire);
p.store(1);
```

Atomic lock/unlock

```
std::atomic<int> lock_var{1};
void lock(std::atomic<int> &p)
{
    while (1) {
        int cur=p.exchange(0,std::memory_order_acquire);
        if (cur) break;
        sched_yield();
    }
}
void unlock(std::atomic<int> &p)
{
    p.store(1);
}
```

Семафор

- Семафор — это переменная s (целого типа), над которой можно выполнять две операции:
- down(s,v) — если значение s >= v, то s = s - v; в противном случае процесс блокируется — помещается в список процессов, ожидающих освобождения данного семафора
- 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;
  }
}
```

Ожидание наступления события

- Часто требуется, чтобы одна нить ждала наступление некоторого условия
- Например, главная нить может дожидаться завершения расчетов созданных нитей чтобы объединить результаты расчетов нитей
- Вариант решения: мьютекс + активное ожидание — не подходит
- Вариант решения: использовать канал — требует использования операций ввода-вывода

Условные переменные

- Механизм для рассылки уведомлений
- Одна или несколько нитей ждут наступления события (заблокированы)
- При наступлении события нить посылает уведомление ожидающим нитям, пробуждая одну из них или все
- Для блокировки доступа к условной переменной используется мьютекс

Условные переменные pthread

```
int pthread_cond_init(pthread_cond_t *c,
                      const pthread_contattr_t *a);
int pthread_cond_destroy(pthread_cond_t *c);
int pthread_cond_wait(pthread_cond_t *c,
                      pthread_mutex_t *m);
int pthread_cond_broadcast(pthread_cond_t *c);
int pthread_cond_signal(pthread_cond_t *c);
```

- Перед использованием условная переменная должна быть проинициализирована

Отправка нотификаций

- Если в момент выполнения pthread_cond_signal или pthread_cond_broadcast целевая нить не находится в ожидании в pthread_cond_wait, НОТИФИКАЦИЯ ПОТЕРЯЕТСЯ!
- Поэтому нужна переменная-флаг (обычно bool или int), которая устнавливается в 1
- Для блокировки доступа к ней нужен мьютекс

Пример: ожидание всех рабочих нитей (барьер)

- Пусть есть N рабочих нитей и есть главная нить, которая ожидает прохождения контрольной точки

```
// условная переменная
pthread_mutex_t wait_mutex;
pthread_cond_t wait_cond;
int wait_count;
// рабочие нити
pthread_mutex_lock(&wait_mutex);
if (++wait_count == N)
  pthread_cond_signal(&wait_cond);
pthread_mutex_unlock(&wait_mutex);
```

Пример: ожидание всех рабочих нитей

- Пусть есть N рабочих нитей и есть главная нить, которая ожидает прохождения контрольной точки

```
// условная переменная
pthread_mutex_t wait_mutex;
pthread_cond_t wait_cond;
int wait_count;
// главная нить
pthread_mutex_lock(&wait_mutex);
while (wait_count != N)
  pthread_cond_wait(&wait_cond, &wait_mutex);
pthread_mutex_unlock(&wait_mutex);
```