Лекция 38
Нити (threads), ч. 2

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

Принудительное завершение нитей

- Как правило, нити следует завершать только «добровольно»

```
int pthread_cancel(pthread_t thread);
```

- Выполняет запрос на завершение нити
  - По умолчанию нить может быть завершена только в т. н. cancellation points (обычно — системные вызовы)
  - Может быть включен «асинхронный режим», по которому нить будет завершена немедленно

```
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

Проблемы принудительного завершения

- Принудительное завершение нити может оставить синхронизационные объекты в этой нити в «закрытом» состоянии
- Может ввести весь процесс в состояние тупика
- Причина: в целях оптимизации производительности средства синхронизации нитей не регистрируются в ядре ОС

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

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)

```
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:
- 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);
```

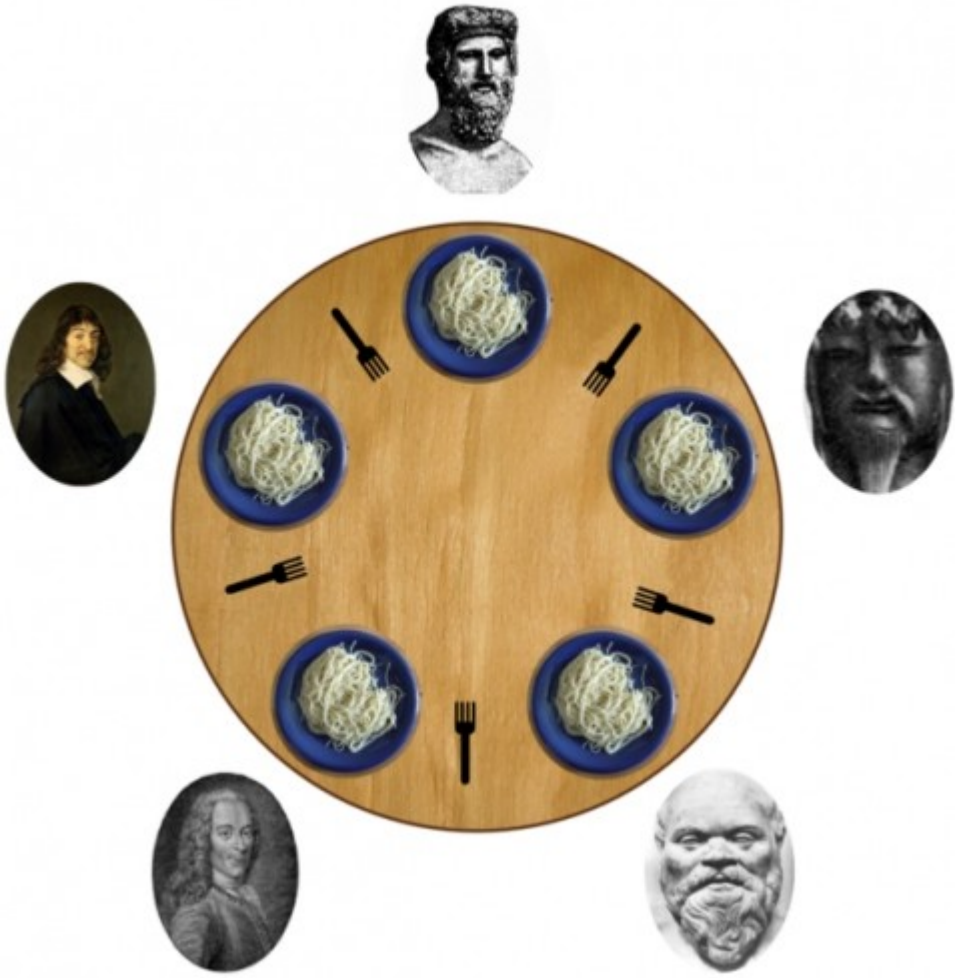
Монитор

- 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;
  }
}
```

Обедающие философы



Наивное решение

```
void philosopher ( int i ) {
  while (TRUE) {
    think () ;
    take_fork ( i ) ;
    take_fork ( ( i + 1 ) % N ) ;
    eat () ;
    put_fork ( i ) ;
    put_fork ( ( i + 1 )  % N ) ;
  }
  return;
}
```

Deadlock

- Возможна ситуация, когда все философы одновременно захотят есть и возьмут левую от себя вилку — никто не сможет начать есть
-
- «Обедающие философы» показывает важность корректного порядка занятия ресурсов при входе в критическую секцию

Избежание блокировки

- При N = 2 задача сводится к:

- Процесс 1:

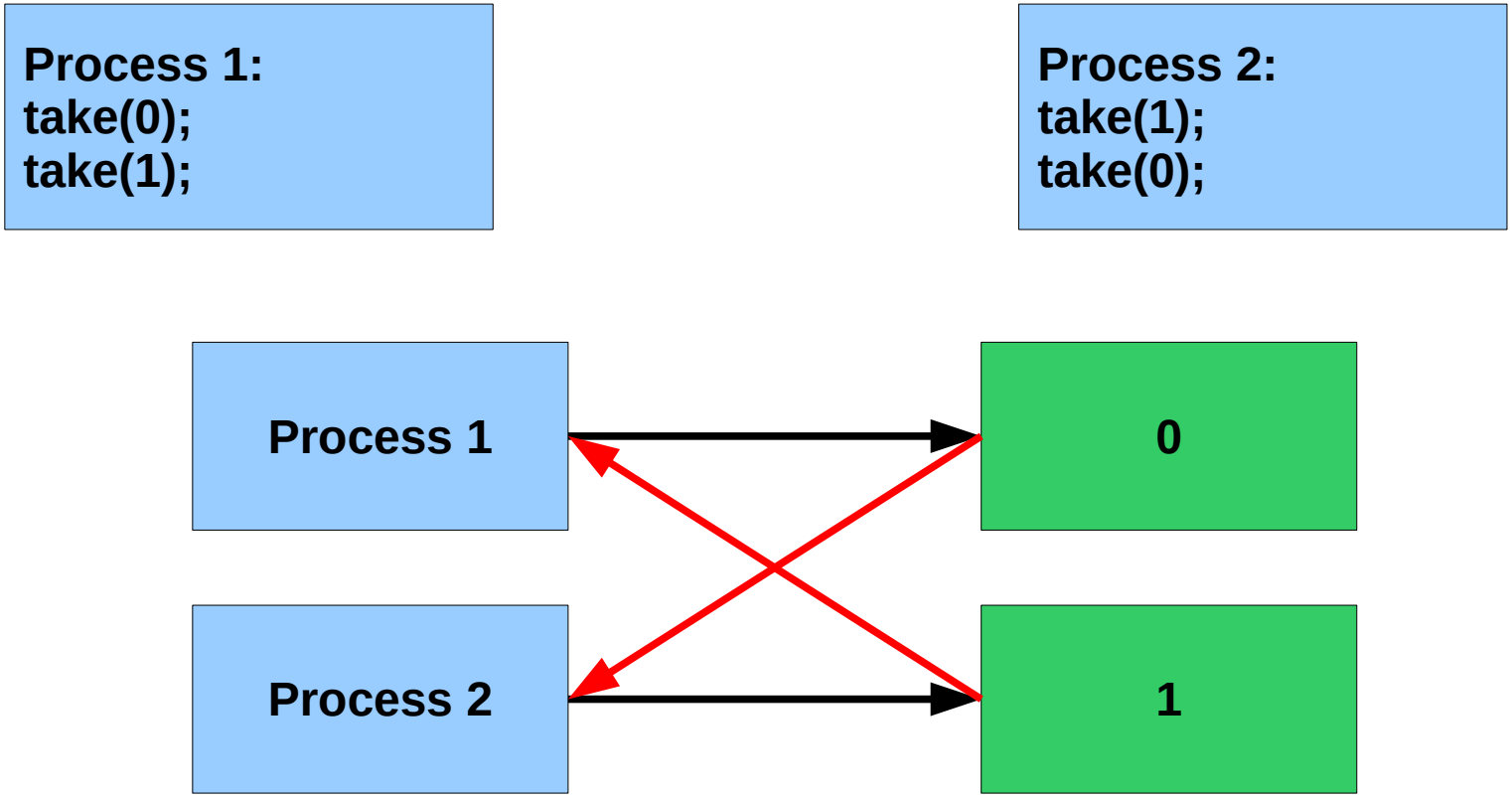
```
take_fork(0);
take_fork(1);
```

- Процесс 2:

```
take_fork(1);
take_fork(0);
```

- Изменение порядка взятия вилок в процессе 2 решает проблему!

Обнаружение тупиков



- Захваченный ресурс — дуга от процесса к ресурсу
- Ожидаемый ресурс — дуга от ресурса к процессу
- Если в графе есть цикл, система попала в состояние тупика

Избежание блокировки

- Каждый процесс может сначала взять вилку с меньшим номером, потом взять вилку с большим номером
- Недостатки:
  - Операция по взятию вилок неатомарна
  - Могут появляться цепочки философов, которые взяли вилку с меньшим номером, но ждут вилку с большим номером — такая цепочка разрушится только когда философ с максимальным номером закончит есть
  - Требуется отношение порядка на множестве вилок

Избежание блокировки

- Проверяем состояние соседей философа под мьютексом
- Если философ не может начать есть, он засыпает на условной переменной
- Когда состояние изменится, его разбудят
- Недостатки:
  - Мьютекс на весь стол, только один философ может проверить состояние
  - Немасштабируемо

Condition variables

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

Использование condvar

```
pthread_cond_t c; // для ожидания
pthread_mutex_t m; // для блокировки
volatile int f; // для передачи значения

// отсылка уведомления
pthread_mutex_lock(&m);
f = 1;
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);

// ожидание уведомления
pthread_mutex_lock(&m);
while (f == 0) pthread_cond_wait(&c, &m);
f = 0;
pthread_mutex_unlock(&m);
```

Классические задачи синхронизации

- Барьер (barrier)
- Обедающие философы (dining philosophers)
- Читатели и писатели (readers-writers)
- Производители-потребители (producers-consumers)
- Спящий парикмахер (sleeping barber)

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);
```

Читатели и писатели

- Дана некоторая разделяемая область память
- К этой структуре данных может обращаться произвольное количество «читателей» и произвольное количество «писателей»
- Несколько читателей могут получить доступ одновременно, писатели в этот момент не допускаются
- Только один писатель может получить доступ, другие писатели и читатели должны ждать

Решение 1

- Первое решение: читатель может войти в критическую секцию, если нет писателей
- Это решение несправедливо, так как отдает предпочтение читателям
- Плотный поток запросов от читателей может привести к тому, что писатель никогда не получит доступа к критической секции: ситуация «голодания» (starvation)

Решение 2

- Отдадим предпочтение писателям, то есть читатель не входит в критическую секцию, если есть хотя бы один ожидающий писатель
- Данное решение отдает приоритет писателям, и тоже несправедливо
- Возможно «голодание» (starvation) читателей

Решение 3

- Третье решение: не отдавать никому приоритета, просто использовать мьютекс
- Не используется возможность одновременного чтения

Решение 4

- Формируем очередь запросов
- Несколько идущих подряд в очереди запросов на чтение могут выполняться параллельно
- Запросы на запись выполняются в эксклюзивном режиме

Производители-потребители (producer-consumer problem)

- Дан буфер фиксированного размера (N), в котором размещается очередь.
- Производители добавляют элементы в конец очереди, если буфер заполнился, производители засыпают
- Потребители забирают элементы из начала очереди, если буфер пуст, потребители засыпают

Спящий парикмахер (sleeping barber)

- В парикмахерской имеется одно кресло для стрижки и N кресел для ожидающих посетителей
- Если нет посетителей, парикмахер спит
- Если приходит посетитель и кресло для стрижки свободно, посетитель садится в него и парикмахер начинает его стричь
- В противном случае посетитель садится в кресло для ожидающих
- Если все кресла заняты, посетитель уходит