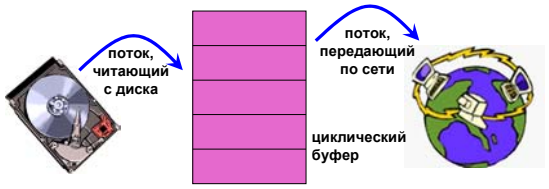
Лекция 7: Синхронизация

Алексей Линёв
Александр Мощук
Кирилл Погорельский

somes slides are adapted from the OS course at the University of Washington

Синхронизация

- В многопоточных программах потоки взаимодействуют друг с другом
  - они *разделяют* ресурсы, используют одни и те же структуры данных
    - например, потоки web-сервера, совместно используют его файловый кэш
  - также они должны *согласовывать* свое выполнение
    - пример: поток, выполняющий чтение данных с диска, может передавать блоки данных потоку, выполняющему передачу по сети, через циклический буфер



Синхронизация

- Для поддержания корректности, мы должны контролировать подобные операции
  - при этом нужно учитывать, что относительные скорости развития потоков неизвестны и различны
    - порядок выполнения потоков и размер выделенных им интервалов времени ЦП зависит от работы планировщика, и не зависит от разработчика прикладной программы
- Мы управляем взаимодействием, используя *синхронизацию*
  - она позволяет упорядочивать исполнение потоков
- Замечание: синхронизация может использоваться также для потоков, исполняющихся в разных процессах
  - в будущем мы более не будем использовать термин "процесс", говоря об активных исполняющихся объектах
- Также, она может использоваться для потоков, выполняющихся на разных машинах в распределенной системе

Разделяемые ресурсы

- Мы сосредоточимся на координации доступа к разделяемым ресурсам (shared resources)
  - основная проблема:
    - два параллельных потока имеют доступ к разделяемой переменной
    - если переменная читается/изменяется/записывается обоими потоками, доступом к ней обязательно надо управлять
    - в противном случае, могут возникнуть неожиданные эффекты
- В течение нескольких следующих лекций мы рассмотрим:
  - механизмы управления доступом к разделяемым ресурсам
    - низкоуровневые механизмы, такие как блокировки
    - высокоуровневые механизмы, такие как мьютексы, семафоры, мониторы, условные переменные
  - шаблоны (patterns) организации координации доступа к разделяемым ресурсам
    - задача о кольцевом буфере (bounded buffer), поставщик-потребитель,…

Классический пример

- Предположим, нам нужно реализовать функцию списания денег с банковского счета

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

- Теперь предположим, что Вы и Ваша супруга используете один банковский счет с текущим балансом $100.00
  - что произойдет, если вы одновременно на двух разных банкоматах снимете со счета по $10.00?

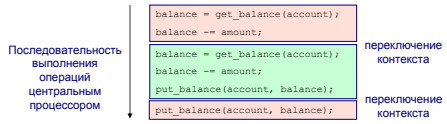
- Представим реализацию описанной процедуры посредством создания отдельного потока для каждого запроса на списание со счета
  - оба запроса выполняются на одном и том же банковском сервере

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Чередующееся исполнение

- Проблема в том, что при использовании вытесняющего планирования выполнение потоков может чередоваться:



- Каков будет баланс после окончания такого выполнения?
  - для кого это хорошо – для вас или для банка?
- Как вы думаете, как часто может реализовываться такая "неудачная" последовательность?

Другие варианты чередования выполнения

- Какие сценарии перекрытия дают правильный результат? Какие – неправильный?

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

А в случае следующего кода?

```
int xfer(from, to, amt) {
  int bal = withdraw(from, amt);
  deposit( to, amt );
  return bal;
}
```

```
int xfer(from, to, amt) {
  int bal = withdraw(from, amt);
  deposit( to, amt );
  return bal;
}
```

А такого?

```
i++;
```

```
i++;
```

Суть проблемы

- Проблема в том, что два параллельных потока (или процесса) выполняют доступ к *разделяемому ресурсу* (в нашем примере – к счету), не используя *синхронизацию*
  - возникают "*гонки*" (race conditions)
    - ситуация, когда результат выполнения операции становится недетерминированным и зависящим от относительной скорости выполнения потоков
- Нам нужны механизмы контроля доступа к разделяемым ресурсам в условиях параллельного выполнения потоков
  - при их наличии мы сможем вернуться к обсуждению выполнения программ
    - фактически, мы представим *новые подходы для поддержки детерминированности*

Какие ресурсы разделяют потоки?

- Локальные переменные *не* разделяются
  - они располагаются в стеке, а каждый поток имеет свой собственный стек
  - никогда не передавайте/разделяйте/сохраняйте указатель на локальную переменную одного потока в стеке другого потока!
- Глобальные переменные – разделяются
  - они расположены в регионе данных, к которому имеют доступ все потоки
- Динамически выделенные объекты – также разделяются
  - они расположены в куче и доступны, если вы имеете их имена
    - в C доступ к объектам можно получить по указателю
      - например, void *x = (void *) 0xDEADBEEF
    - в Java строгая типизация запрещает подобные фокусы
      - вы должны использовать явно определенную ссылку

Взаимное исключение

- Для синхронизации доступа к разделяемым ресурсам мы будем использовать *взаимное исключение (mutual exclusion)*
- Использование взаимного исключения позволяет достаточно просто изменить структуру и поведение программ
  - проще изменения – меньше ошибок
- Код, использующий взаимное исключение для синхронизации своего выполнения, называется *критической секцией (critical section)*
  - в каждый момент времени только один поток может выполнять код из критической секции (находиться в критической секции)
  - все остальные потоки должны блокироваться на входе в критическую секцию
  - когда один поток покидает критическую секцию, один из ожидающих потоков может в нее войти

Требования к реализации критических секций

- Реализация критических секций должна удовлетворять следующим требованиям:
  - **взаимное исключение**
    - в критической секции должно находится не более одного потока
  - **progress**
    - если поток T выполняется вне критической секции, он не может помешать потоку S войти в критическую секцию
  - **граничное ожидание** (bounded waiting) - остутствие голодания
    - если поток T ожидает возможности войти в критическую секцию, в конце концов он в нее войдет
      - предполагается, что потоки со временем покидают критические секции
    - имеет ли в данном случае смысл термин "справедливость"?
  - **производительность**
    - накладные расходы на вход и выход из критической секции должны быть малы по сравнению с работой, выполняемой в критической секции

Механизмы реализации критических секций

- Использование признаков блокировки (locks)
  - очень прост, обладает минимальной семантикой и является основой для построения других механизмов
- Семафоры
  - основной механизм, прост в понимании – труден в использовании
- Мониторы
  - высокоуровневый механизм со скрытой функциональностью, требует поддержки со стороны языка программирования
  - прост в использовании; пример – “`synchronized()`” в Java
- Сообщения
  - простая модели передачи данных и синхронизации, основанная на атомарной передаче данных по некоторому каналу
  - широко применяется в распределенных системах

Признаки блокировки (Locks)

- Признак блокировки – это объект (в памяти), предоставляющий две следующие операции
  - `acquire()`: поток вызывает ее перед входом в критическую секцию
  - `release()`: поток вызывает ее после выхода из критической секции
- Потоки используют вызовы `acquire()` и `release()` в паре
  - между вызовами `acquire()` и `release()` поток захватывает признак блокировки (или захватывает блокировку)
  - `acquire()` не возвращает управление до тех пор, пока вызвавший поток не сможет захватить признак блокировки
    - как правило, в каждый момент времени только один поток может владеть признаком блокировки
  - вопрос: что произойдет, если вызовы не используются в паре?
- Две основных особенности признаков блокировки
  - активное ожидание (spinlocks)
  - реализуют взаимное исключение ("мьютекс")

Использование признаков блокировки

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}
```

критическая секция

```
acquire(lock)
balance = get_balance(account);
balance -= amount;
```

```
acquire(lock)
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
```

- Что происходит, когда зеленый поток пытается захватить признак блокировки?
- Почему "return" расположен за пределами критической секции?
  - это нормально?

Реализация признаков блокировки…

- Как реализовать признаки блокировки? Может, так:

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while (lock->held);
   lock->held = 1;
}
void release(lock) {
  lock->held = 0;
}
```

вызвавший поток использует "активное ожидание", то есть крутится в цикле, ожидая освобождения признака блокировки
⇒ это называется spinlock

- Почему такой вариант не будет работать?
  - где в этом случае возникают гонки?

Реализация признаков блокировки

- Проблема в том, что реализация признаков блокировки сама содержит критическую секцию!
  - операции acquire/release должны быть *атомарными*
    - атомарный – выполнение операции не может быть прервано
  - это должен быть код, который либо выполняется целиком, либо не выполняется совсем
- Необходима поддержка со стороны аппаратного обеспечения
  - атомарные инструкции
    - проверка-и-установка (test-and-set), сравнение-и-обмен (compare-and-swap), …
  - возможность запрещать/разрешать прерывания
    - для предотвращения смены контекста

Активное ожидание с использованием Test-and-Set (Проверка-И-Установка)

- ЦП предоставляет test-and-set в виде атомарной инструкции:

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

- Запомните: это одна инструкция ЦП…

Активное ожидание с использованием Test-and-Set (Проверка-И-Установка)

- Изменим наш цикл активного ожидания следующим образом:

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while(test_and_set(&lock->held));
}
void release(lock) {
  lock->held = 0;
}
```

- Реализованы ли
  - взаимное исключение?
  - progress?
  - граничное ожидание?
  - производительность?

Вспомним использование …

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}
```

критическая секция

```
acquire(lock)
balance = get_balance(account);
balance -= amount;
```

```
acquire(lock)
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
```

- Каким образом поток, вызвавший “acquire()” (то есть, выполняющий в цикле операцию проверка-и-установка) освобождает ЦП?
  - вызывает yield() (после каждой проверки)
  - вытесняется планировщиком

Недостатки активного ожидания

- Активное ожидание работает, но оно ужасно затратно!
  - если какой-то поток находится в состоянии активного ожидания, поток, захвативший признак блокировки, не может продолжать свое исполнение
  - и никакой другой поток тоже не может!
- Активное ожидание лучше использовать только как базу для построения высокоуровневых механизмов
  - почему это возможно?
- *В каких случаях вышеперечисленные пункты не должны вас смущать?*

Другой подход: запрещение прерываний

```
struct lock {
}
void acquire(lock) {
   cli();   // disable interrupts
}
void release(lock) {
   sti();   // reenable interrupts
}
```

Недостатки использования запрета прерываний

- Доступно только в режиме ядра
  - Пользовательским программам нельзя позволять запрещать прерывания!
- Не подходит для использования на многопроцессорных системах
  - Каждый процессор имеет собственный механизм прерываний
- При запрещении прерываний на длительное время могут возникнуть сложности с функционированием у устройств, не получавших должного внимания
- Так же как и активное ожидание, запрещение прерываний лучше использовать только для создания высокоуровневых механизмов синхронизации

Выводы

- Синхронизацию выполнения можно реализовать, используя признаки блокировки, семафоры, мониторы, сообщения,…
- Признаки блокировки (locks) – механизм самого низкого уровня
  - очень прост с точки зрения семантики, использование сопряжено с возможностью возникновения ошибок
  - реализуется через активное ожидание (грубо) или запрет прерываний (тоже грубо, а также доступно только в режиме ядра)
- В следующей лекции
  - семафоры – немного более высокий уровень абстракции
    - с менее грубой реализацией
  - мониторы – еще более высокий уровень
    - позволяет уменьшать количество ошибок на уровне языка программирования