Объекты синхронизации

Операционные системы 2011/12

Татьяна Романова

8 октября 2011 г.

План на сегодня

- Возможности для реализации блокировок
- Атомарные операции процессора
- Спин-блокировки
- Семафоры
- Проблема производителя и потребителя
- Мониторы и условные переменные (добавление в очередь)
- Проблема читателей и писателей

Материалы

- CS 162, лекции 7, 8 (видео лекции на YouTube).
- Э. Таненбаум, Современные операционные системы, глава 2 (межпроцессное взаимодействие).
- Ю. Вахалия, UNIX изнутри, глава 7.

Повторение: необходимость блокировок

Объекты, требующие синхронизации:

- Очередь запросов при реализации веб-сервера
- Счет в банке при операциях с банкоматами
- Холодильник, в который кладут молоко

Решение задачи про молоко (активное ожидание)

Рабочее решение задачи про молоко:

Поток A

```
leave note A
while (note B) {
    do nothing;
}
if (no milk) {
    buy milk
}
remove note A
```

Поток B

```
leave note B
if (no note A) {
    if (no milk) {
        buy milk
    }
}
remove note B
```

Минусы: запутанно, немасштабируемо, несимметрично, A не дает B работать.

Еще один вариант

Поток A

```
trylockA() {
    leave note A
    while (note B) {
        do nothing;
    }
    if (no note C) {
        leave note C
        remove note A
        return true
    }
    remove note A
    return false
}
```

Поток B

```
trylockB() {
    leave note B
    if (no note A) {
        if (no note C) {
            leave note C
            remove note B
            return true
        }
    }
    remove note B
    return false
}
```

- Заменили длинное ожидание коротким
- Использовать так: `if (trylockA()) {if (no milk) buy milk}`

Ожидаемый интерфейс блокировки

- Операции захвата и освобождения блокировки атомарны.
- Только один поток может, захватив блокировку, попасть в критическую секцию.
- Интерфейс:
  - mutex.lock() заставляет процесс ждать, если блокировка захвачена кем-то другим
  - mutex.unlock() снимает блокировку, будит ожидающие потоки
- Решение задачи про молоко выглядит просто:

```
milk_mutex.lock()
if (no milk)
    buy milk
milk_mutex.unlock()
```

Как реализовать блокировку?

Реализация блокировок

Отключение прерываний

- Наивный подход:
  lock() { disable interrupts; }
  unlock() { enable interrupts; }
- lock и unlock недоступны в пространстве пользователя (иначе пользователь может «повесить» систему).
- При нахождении потока в критической области могут теряться прерывания.
- Нельзя реализовать на многопроцессорной машине
  - отключать на всех процессорах дорого
  - и к тому же невозможно, т. к. один может продолжать выполнять текущий код в момент отключения прерываний.

Отключение на короткое время

Блокировка только для изменения переменной var.
Изначально var = FREE.

```
lock() {
    disable interrupts
    if (var == BUSY) {
        add to wait queue
        sleep()
        // enable interrupts?
    }
    enable interrupts
}
```

```
unlock() {
    disable interrupts;
    if (!wqueue.empty())
        wakeup(wqueue.front())
    else
        var = FREE
    enable interrupts
}
```

Проблема: lock() засыпает с отключенной обработкой прерываний.

Отключение на короткое время (продолжение)

Прерывания можно разрешить внутри реализации функции sleep():

Поток S	Поток T
disable interrupts	
sleep	
switch	
	switch return
	sleep return
	enable interrupts
	...
	disable interrupts
	sleep
	switch
switch return	
sleep return	
enable interrupts	

Атомарные операции процессора

Проблемы с отключением прерываний:

- Не подходит для реализации потоков в пространстве пользователя
- Не работает с многопроцессорными машинами

SMP (symmetric multiprocessing) — архитектура многопроцессорных компьютеров, в которой два или больше одинаковых процессоров подключаются к общей памяти.

Атомарные операции:

- можно считать значение из памяти и записать туда новое, не прерываясь
- работают и с SMP-системами (при записи приходится блокировать шину данных).
- позволяют реализовать неблокирующие структуры данных

Примеры атомарных операций

- Проверить и установить

```
test&set(address) {
    res = Memory[address];
    Memory[address] = 1;
    return res;
}
```

  Если венулся 0 — мы захватили блокировку, 1 — нужно ждать.

- Сравнение с обменом

```
compare&swap(address, reg1, reg2) {
    if (Memory[address] == reg1) {
        M[address] = reg2;
        return success;
    }
    else
        return failure;
}
```

Базовые объекты синхронизации

Спин-блокировка

Реализация через test&set:

```
lock(spin_t *s) {
    while(test&set(s));
}
unlock(spin_t s) {
    s = 0;
}
```

Более правильный подход:

```
lock(spin_t *s) {
    while(test&set(s))
        while (*s != 0);
}
unlock(spin_t s) {
    s = 0;
}
```

- Ждем пока блокировка освободится и атомарно захватываем ее.
- В первом случае шина блокируется постоянно, во втором — только когда похоже, что значение s поменялось.
- В обоих случаях используется активное ожидание.

Спин-блокировка: хорошо или плохо?

Достоинства:

- Простота: нет отключения прерываний и очередей сна.
- Работает на многопроцессорных системах
- Не использует системные вызовы

Недостатки:

- Бессмысленна на однопроцессорной машине
- Не подходит для продолжительных блокировок
- Может привести к инверсии приоритетов

What really happened on Mars?

Mars Pathfinder

- После начала сбора данных — частые перезагрузки.
- ОС VxWorks, вытесняющее планирование.
- Потоки на каждую задачу, общая шина данных, мьютекс.
- Рабочие потоки:
  1. задача управления шиной (выс. приор.),
  2. задача обмена данными (сред.)
  3. сбор данных (низк.).
- Инверсия приоритетов: 3 блокирует мьютекс, 1 ждет на мьютексе, при прерывании переключаемся на 2 $\Longrightarrow$ 1 не выполнятеся в срок, система перезагружается

Решение: повторение ситуации на Земле, просмотр логов, включение наследования приоритетов.

Возможная реализация мьютекса

Вместо долгого активного ожидания используем короткое и очередь сна:

```
int guard = 0;
int val = FREE;
```

```
lock() {
    while(test&set(&guard));
    if (val == BUSY) {
        add to wait queue
        sleep() & guard = 0;
    }
    else {
        value = BUSY;
        guard = 0;
    }
}
```

```
unlock() {
    while(test&set(&guard));
    if (not wqueue.empty()) {
        add to ready queue
        wqueue.front()
    }
    else
        val = FREE;
    guard = 0;
}
```

Семафоры

Что это:

- Обобщение мьютексов
- Тип, инициализирующийся целым числом и поддерживающий 2 атомарные операции:
  - P() (sem_wait) — уменьшает значение на 1, блокируется, если получается отрицательное число.
  - V() (sem_post) — увеличивает значение на 1, если в очереди сна есть потоки, будит один из них.

Примеры использования:

1. Мьютекс: семафор с начальным значением 1.
   lock() {sem_wait()}, unlock() {sem_post()}
2. Ожидание событий: семафор с начальным значением 0.
   thread_join() { sem_wait(); }
   thread_exit() { sem_post(); }

Проблема производителя и потребителя

- Два процесса используют буфер ограниченного размера.
- Производитель пишет в буфер, потребитель читает из буфера
- Если буфер пуст, потребитель ждет, если полон — ждет производитель.
- Кладя элемент в пустой буфер, производитель будит потребителя (удаляя из полного, потребитель будит производителя)

Решение в лоб

```
int N = 100, count = 0;
void producer() {
    while(1) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}
void consumer() {
    while(1) {
        if (count == 0) sleep();
        item = remove_item(item);
        count = count - 1;
        if (count == N-1) wakeup(producer);
        process_item(item);
    }
}
```

- Работают в двух потоках
- Общий доступ к очереди и к count
- Состояние состязания

Решение с семафорами

```
int N = 100;
sem_t mutex = 1, empty = N, full = 0;
void producer() {
    while(1) {
        item = produce_item();
        empty.down();
        mutex.down();
        insert_item(item);
        mutex.up();
        full.up();
    }
}

void consumer() {
    while(1) {
        full.down();
        mutex.down();
        item = del_item(item);
        mutex.up();
        empty.up();
        consume_item();
    }
}
```

Решение с семафорами

```
int N = 100;
sem_t mutex = 1, empty = N, full = 0;
void producer() {
    while(1) {
        item = produce_item();
        empty.down();
        mutex.down();
        insert_item(item);
        mutex.up();
        full.up();
    }
}
```

```
void consumer() {
    while(1) {
        full.down();
        mutex.down();
        item = del_item(item);
        mutex.up();
        empty.up();
        consume_item();
    }
}
```

Вопросы:

- Зачем так много объектов синхронизации?
- Важен ли порядок операций down? А up?
- Работает ли для 2 производителей и потребителей?

Проблемы предыдущего решения

- Запутанность. Трудно доказать корректность.
- Неправильный порядок вызова приводит к взаимоблокировке.
- Семафоры используются для взаимного исключения и для сигналов.

Лучше использовать два типа объектов:

- Мьютексы для взаимного исключения
- Условные переменные для планирования, кому когда спать и выполняться

Сложные объекты синхронизации

Условные переменные

- Объект с тремя операциями:
  - wait(&lock) — ждет сингала внутри критической области
  - signal() — перемещает процесс, заблокированный на данной условной переменной, из очереди сна в очередь выполнения
  - broadcast() — перемещает все заблокированные процессы из очереди сна в очередь выполнения.
- В отличие от семафоров: может ждать внутри критической сексии (блокировка освобождается перед уходом в сон и снова захватывается после)
- Просто использовать, если не думать о реализации.
- Posix-реализация теряет сингнал, если его никто не ждет. В реализации на семафорах такого поведения не добьешься.

Мониторы

- Подход (парадигма), позволяющий гарантировать, что поток, вызывающий текущую процедуру не будет прерван кем-то еще.
- Встроены в некоторых языка (synchronized в Java), в других могут быть реализованы через использование мьютексов и условных переменных.
- Существуют два подхода к планированию после того, как поток, делающий signal покинет монитор:
  - Мониторы Хоара: после потока, вызывающего signal, сразу будет выполняться разбуженный поток. Сложная реализация, не используются на практике.
  - Мониторы Хансена: при вызове signal спящий поток становится готов к выполнению, но кто начнет выполняться на самом деле — решает планировщик. Используются на практике.

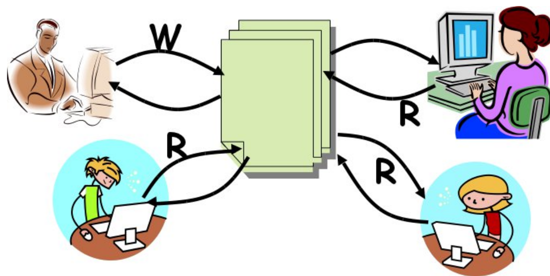
Пример использования монитора: очередь

```
mutex_t m;
queue q;
cond_t c;
add_to_queue(item) {
    m.lock();
    q.enqueue(item);
    c.signal(); // Будим ждущего
    m.unlock();
}
del_from_queue() {
    m.lock();
    while (q.empty())
        c.wait(&m); //Если пуста, заснуть
    item = q.dequeue();
    m.unlock();
    return item;
}
```

Реализация условных переменных

1. На семафорах:
   - в wait нужно отпустить блокировку, заблокироваться на семафоре, затем снова захватить блокировку.
   - в signal разблокировать семафор.
2. В ядре:
   - wait:
     - добавить поток в очередь сна (операция с очередью обернута спин-блокировкой)
     - отпустить блокировку
     - switch()
     - захватить блокировку
   - signal:
     - удалить поток из очереди сна
     - добавить его в очередь на выполнение

Проблема читателей и писателей



- Существуют 2 класса пользователей:
  - Читатели читают и никогда не меняют базу
  - Писатели читают и меняют базу
- Блокировать всю базу при доступе каждого невыгодно.

Заключение

Итоги лекции

- Синхронизацию можно реализовать без использования атомарных операций (но сложно или неуневерсально)
- В большинстве современных процессоров доступны test&set, compare&swap (или аналоги)
- Спин-блокировки используют активное ожидание, но не делают системных вызовов
- Существует проблема инверсии приоритета
- Кроме простых взаимоблокировок и спин-блокировок, существуют сложные объекты синхронизации: семафоры, условные переменные, блокировки чтения/записи.
- Мониторы являются принципом использования блокировок, встроенным в некоторых языках.