Модуль 8 . Семафоры

Конкуренция и ситуация гонок

Параллелизм и блокировка

Методы синхронизации необходимы тогда, когда существует параллелизм. Параллелизм – это ситуация, когда одновременно выполняется два или более процесса, которые могут потенциально взаимодействовать друг с другом (например, использовать один и тот же набор ресурсов).

Параллелизм может встречаться на однопроцессорных рабочих станциях, где несколько потоков используют один и тот же процессор, вытесняя друг друга и создавая ситуацию гонки. Вытеснением называется прозрачное совместное использование процессора путем временной приостановки одного потока для обеспечения возможности выполнения другого. Ситуация гонки - это ситуация, когда два или более потока управляют одними и теми же данными и результат зависит от порядка выполнения. Параллелизм также существует на многопроцессорных машинах, где одновременно выполняемые потоки разных процессоров используют одни и те же данные. Обратите внимание, что в случае многопроцессорных систем имеет место истинный параллелизм, поскольку потоки выполняются одновременно. В случае однопроцессорной системы параллелизм создаётся механизмом вытеснения. В обоих режимах параллелизма имеются свои сложности.

Ядро Linux поддерживает параллелизм в обоих вариантах. Ядро само по себе динамично, и ситуации гонки могут возникать в различных случаях. Ядро Linux также поддерживает многопроцессорный режим, известный как симметричная многопроцессорность (SMP).

Для решения проблемы ситуаций гонки было разработана концепция критической секции. Критическая секция - это часть кода, защищенная от одновременного доступа. Эта часть кода может управлять данными и службами общего пользования (например, периферийным оборудованием). Критические секции работают по принципу взаимного исключения (когда поток находится в критической секции, вход любых других потоков не допускается).

Однако с критическими секциями связана проблема тупиковой взаимной блокировки. Рассмотрим две отдельные критические секции, каждая из которых защищает свой ресурс. Каждый ресурс имеет свою блокировку; назовем их A и B. Рассмотрим два потока, которым необходим доступ к нашим ресурсам. Поток X захватывает блокировку A, а поток Y - блокировку B. Пока эти блокировки удерживаются, каждый из потоков пытается захватить блокировку, удерживаемую другим потоком (поток X пытается захватить блокировку B, а поток Y – блокировку A). Теперь потоки находятся в тупиковой ситуации, поскольку каждый из них блокирует нужный другому ресурс. Простое решение состоит в том, чтобы всегда захватывать блокировки в одном и том же порядке, что позволяет завершить работу потока. Другое решение состоит в обнаружении таких ситуаций. В таблице 1

определены наиболее важные обсуждаемые здесь понятия из области параллелизма.

Таблица 1. Важные понятия параллелизма

Понятие	Описание
Условие гонки	Ситуация, в которой одновременные манипуляции нескольких потоков с ресурсом приводят к неоднозначным результатам.
Критическая секция	Сегмент кода, координирующий доступ к общим ресурсам.
Взаимное исключение	Свойство программного обеспечения, которое гарантирует эксклюзивный доступ к общим ресурсам.
Взаимная блокировка	Особое состояние, создаваемое двумя или более процессами и двумя или более блокировками, которые не дают процессам нормально работать.

Методы синхронизации Linux

Теперь, когда мы немного ознакомились с теорией и поняли, какую проблему предстоит решать, давайте рассмотрим различные способы реализации параллельного исполнения и взаимных исключений в Linux. Раньше обработка взаимных исключений выполнялась путем отключения прерываний, но такая блокировка неэффективна (хотя её следы еще можно найти в ядре). Кроме того, этот метод не очень хорошо масштабируется и не гарантирует взаимное исключение на других процессорах.

В приведенном ниже обзоре механизмов блокировки мы сначала рассмотрим атомарные операции, которые обеспечивают защиту простых переменных (счетчиков и битовых масок). После этого

рассматриваются простые взаимные блокировки (спинлоки) и взаимные блокировки чтения и записи в качестве эффективного механизма активного ожидания блокировок для архитектур SMP. И, наконец, мы рассмотрим взаимные исключения (мьютексы) ядра, которые построены на атомарном API.

Типы блокировок

До появления и широкого распространения SMP, когда физической реализации параллелизма ещё не существовало, блокировки использовались классическим способом, защищая критические области программы от одновременного исполнения несколькими процессами. Такие механизмы работают за счёт вытеснения запрашивающих процессов в блокированное состояние до времени освобождения запрошенных ресурсов. Подобные блокировки мы будем называть пассивными блокировками, и в этом случае процессор прекращает выполнение текущего процесса в точке блокирования и переключается на выполнение другого процесса (возможно idle).

Принципиально другой вид блокировок —активные блокировки — появился вместе с SMP системами, когда процессор, ожидая освобождения недоступного ресурса, не переводится в блокированное состояние, а выполняет пустые циклы. В этом случае, процессор не освобождается для выполнения другого ожидающего процесса в системе, а продолжает активное выполнение ("пустых" циклов) в контексте текущей ветви исполнения.

Эти два типа блокировок (каждый из которых включает

несколько подвидов) принципиально отличаются по ключевым параметрам:

возможностью использования : пассивно заблокировать (переключить контекст) можно только фрагмент кода, имеющий свой собственный контекст (запись задачи), куда позже можно вернуться (активировать процесс), а в обработчиках прерываний или тасклетах это условие не соблюдается;

эффективностью: активные блокировки не всегда проигрывают пассивным в производительности, так как переключение контекста в системе это трудоёмкий процесс, поэтому для ожидания короткого интервала времени активные блокировки могут оказаться даже эффективнее, чем пассивные.

Семафоры (мьютексы). Реализация.

Семафоры ядра определены в файле <linux/semaphore.h>. Так как задачи, конфликтующие при захвате блокировки, переводятся в состояние ожидания и в этом состоянии ожидают освобождения блокировки, то семафоры хорошо подходят для блокировок, которые могут удерживаться в течение длительного времени. С другой стороны, семафоры не очень пригодны для блокировок, которые удерживаются в течение короткого интервала, так как накладные затраты на перевод процессов в состояние ожидания могут превысить время, в течение которого удерживается блокировка. Кроме того, существует и явное ограничение на использование семафоров в ядре, так как их невозможно

использовать в коде, который не должен переходить в блокированное состояние, например, при обработке верхней половины прерываний.

Спин-блокировки позволяют удерживать блокировку только одной задачи в любой момент времени, но для семафора количество задач (count), которые разрешено одновременно удерживать с его помощью (владеть семафором), может быть задано при декларации переменной семафора в соответствующем поле структуры:

```
struct semaphore {
  spinlock_t lock;
  unsigned int count;
  struct list_head wait_list;
};
```

Если значение count больше 1, то семафор называется счетным семафором и допускает количество потоков, которые одновременно удерживают блокировку, не больше, чем значение счетчика использования (count). Встречается ситуация, когда разрешенное количество потоков, которые одновременно могут удерживать семафор, равно 1 (как и для спин-блокировок), и такие семафоры называются бинарными или взаимоисключающими блокировками (mutex, мютекс, потому что он гарантирует взаимоисключающий доступ — mutual exclusion). Бинарные семафоры (мьютексы) чаще всего используются для обеспечения взаимоисключающего доступа к фрагментам кода, называемым критической секцией.

Независимо от того, определено ли поле владельца, захватившего мютекс (так как это делается по разному в различных POSIX-совместимых ОС), принципиальными особенностями мютекса, в

отличии от счётного семафора будет то, что:

1. у захваченного мютекса всегда будет **единственный** владелец, захвативший его;
2. освободить блокированные на мютексе потоки (освободить мютекс) может только один владеющий мютексом поток.

В случае счётного семафора освободить потоки, блокированные на семафоре, может любой из потоков, владеющий семафором.

Статическое определение и инициализация семафоров выполняется макросом:

```
static DECLARE_SEMAPHORE_GENERIC( name, count );
```

Для создания взаимоисключающей блокировки (mutex) есть более короткий синтаксис:

```
static DECLARE_MUTEX( name );
```

— где в обоих случаях name— это имя переменной типа семафор.

Но чаще всего семафоры создаются динамически, как составная часть более крупных структур данных. В таком случае для инициализации счётного семафора используется функция:

```
void sema_init( struct semaphore *sem, int val );
```

А для инициализации бинарных семафоров (мютексов) используются макросы:

```
init_MUTEX( struct semaphore *sem );
init_MUTEX_LOCKED( struct semaphore *sem );
```

В ОС Linux для захвата семафора (мютекса) используется операция down(), уменьшающая его счетчик на единицу. Если значение счетчика больше или равно нулю, то блокировка захвачена успешно и

задача может входить в критический участок. Если значение счетчика (после декремента) меньше нуля, то задание помещается в очередь ожидания и процессор переходит к выполнению других задач. Метод up() используется для того, чтобы освободить семафор (после завершения выполнения критического участка), его выполнение увеличивает счётчик семафора на единицу, при этом один из имеющихся заблокированных потоков может захватить блокировку (принципиальным является то, что невозможно повлиять на то, какой конкретно поток из числа заблокированных будет выбран). Ниже перечислены другие операции над семафорами.

- void down(struct semaphore *sem) — переводит задачу в блокированное состояние ожидания с флагом TASK_UNINTERRUPTIBLE. В большинстве случаев это нежелательно, так как процесс, который ожидает освобождения семафора, не будет отвечать на сигналы.
- int down_interruptible(struct semaphore *sem)— выполняет попытку захватить семафор. Если эта попытка неудачна, то задача переводится в блокированное состояние с флагом TASK_INTERRUPTIBLE (в структуре задачи). Такое состояние процесса означает, что задание может быть возвращено к выполнению с помощью сигнала, а такая возможность обычно очень ценна. Если сигнал приходит в то время, когда задача блокирована на семафоре, то задача возвращается к выполнению, а функция down_interruptible() возвращает значение —EINTR.

- int down_trylock(struct semaphore *sem)— используется для неблокирующего захвата семафора. Если семафор уже захвачен, то функция немедленно возвращает ненулевое значение. В случае успешного захвата семафора возвращается нулевое значение и захватывается блокировка.
- int down_timeout(struct semaphore *sem, long jiffies)— используется для попытки захвата семафора на протяжении интервала времени jiffies системных тиков.

Взаимоблокировки (Спин-блокировки)

Блокирующая попытка входа в критическую секцию при использовании семафоров означает потенциальный перевод задачи в блокированное состояние и переключение контекста, что является дорогостоящей операцией. Спин-блокировки (spinlock_t) используются для синхронизации в случаях, когда:

- контекст выполнения не позволяет переходить в блокированное состояние (контекст прерывания);
- или требуется кратковременная блокировка без переключение контекста.

Они представляют собой активное ожидание освобождения в пустом цикле. Если необходимость синхронизации связана только с наличием в системе нескольких процессоров, то для небольших критических секций следует использовать спин - блокировку , основанную на простом ожидании в цикле. Спин-блокировка может

быть только бинарной. Определения, относящиеся к spinlock_t, распределены по нескольким заголовочным файлам:

```
$ ls spinlock*
spinlock_api_smp.h  spinlock_api_up.h  spinlock.h  spinlock_types.h
spinlock_types_up.h  spinlock_up.h

typedef struct {
  raw_spinlock_t raw_lock;
...
} spinlock_t;
```

Для инициализации spinlock_t и родственного типа rwlock_t, о котором будет подробно рассказано ниже, раньше (и в литературе) использовались макросы:

```
spinlock_t lock = SPIN_LOCK_UNLOCKED;
rwlock_t lock = RW_LOCK_UNLOCKED;
```

Но сейчас мы можем читать в комментариях:

```
// SPIN_LOCK_UNLOCKED and RW_LOCK_UNLOCKED defeat lockdep
state tracking and
// are hence deprecated.
```

То есть, эти макроопределения объявлены не поддерживаемыми и могут быть исключены в любой последующей версии. Поэтому для определения и инициализации используются новые макросы (эквивалентные по смыслу записанным выше) вида:

```
DEFINE_SPINLOCK( lock );
DEFINE_RWLOCK( lock );
```

Эти макросы представляют собой статические определения отдельных (автономных) переменных типа spinlock_t. И так же, как и для других примитивов, возможна динамическая инициализация ранее объявленной переменной (чаще всего эта переменная является полем в составе более сложной структуры):

```
void spin_lock_init( spinlock_t *sl );
```

Основной интерфейс spinlock_t содержит пару вызовов для

захвата и освобождения блокировки:

```
spin_lock ( spinlock_t *sl );
spin_unlock( spinlock_t *sl );
```

Если при компиляции ядра не было активировано SMP (использование многопроцессорности) и не сконфигурировано вытеснение кода в ядре (обязательно выполнение обоих условий), то spinlock_t вообще не компилируются (и на их месте останутся пустые места) за счёт препроцессорных директив условной трансляции.

Примечание: В отличие от реализаций в некоторых других операционных системах, спин-блокировки в операционной системе Linux не рекурсивны. Это означает, что показанный ниже код автоматически приведёт к ситуации deadlock (процессор будет бесконечно выполнять этот фрагмент и произойдёт деградация системы, так как число процессоров, доступных в системе, уменьшится):

```
DEFINE_SPINLOCK( lock );
spin_lock( &lock );
spin_lock( &lock );
```

Рекурсивный захват спин-блокировки может неявно произойти в обработчике прерываний, поэтому перед захватом такой блокировки нужно запретить прерывания на локальном процессоре. Это общий случай, поэтому для него предоставляется специальный интерфейс:

```
DEFINE_SPINLOCK( lock );
unsigned long flags;
spin_lock_irqsave( &lock, flags );
/* критический участок ... */
spin_unlock_irqre_store( &lock, flags );
```

Для спин-блокировки определены ещё такие вызовы, как:

- int spin_try_lock(spinlock_t *sl)— попытка захвата без блокирования, если блокировка уже захвачена, функция

возвратит ненулевое значение;

- int spin_is_locked(spinlock_t *sl)— возвращает ненулевое значение, если блокировка в данный момент захвачена.

Блокировки чтения-записи

Особым, но часто встречающимся, случаем синхронизации являются сценарий "чтения-записи". "Читатели" только считывают состояние некоторого ресурса, и поэтому могут) иметь к нему совместный параллельный доступ. "Писатели" изменяют состояние ресурса, и поэтому писатель должен иметь к ресурсу монопольный доступ, причем чтение ресурса для всех читателей в этот момент времени так же должно быть заблокировано. Для реализации блокировок чтения-записи в ядре Linux существуют отдельные версии семафоров и спин-блокировок. Мьютексы реального времени не имеют реализации, подходящей для использования в данном сценарии.

Примечание: Обратим внимание на то, что в точности той же функциональности можно достичь, используя классические примитивы синхронизации (мьютекс или спин-блокировку), просто захватывая критический участок независимо от типа предстоящей операции. Блокировки чтения-записи введены из соображений эффективности реализации для массового сценария такого использования.

Для семафоров вместо структуры struct semaphore вводится структура struct rw_semaphore, а набор интерфейсных функций для захвата/освобождения (простые down()/up()) расширяется до:

- down_read(&rwsem)— попытка захватить семафор для чтения;
- up_read(&rwsem) — освобождение семафора для чтения;
- down_write(&rwsem)— попытка захватить семафор для записи;
- up_write(&rwsem)— освобождение семафора для записи;

Семантика этих операций следующая:

- если семафор ещё не захвачен, то любой захват (down_read() или down_write()) будет успешным (без блокирования);
- если семафор захвачен уже для чтения, то последующие попытки захвата семафора для чтения (down_read()) будут завершаться успешно (без блокирования), но запрос на захват такого семафора для записи (down_write()) закончится блокированием;
- если семафор захвачен уже для записи, то любая последующая попытка захвата семафора (down_read() или down_write()) закончится блокированием;

Статически определенный семафор чтения-записи создаётся макросом:

```
static DECLARE_RWSEM( name );
```

Семафоры чтения-записи, которые создаются динамически, должны быть инициализированы с помощью функции:

```
void init_rwsem( struct rw_semaphore *sem );
```

Примечание: Из описания инициализации видно, что семафоры чтения-записи являются исключительно бинарными (не счётными), то есть (в терминологии Linux) фактически не семафорами, а мютексами.

Ниже представлен пример того, как семафоры чтения-записи могут быть использованы при работе (обновлении и считывании)

циклических списков Linux (о которых мы говорили ранее):

```
struct data {
  int value;
  struct list_head list;
};
static struct list_head list;
static struct rw_semaphore rw_sem;
int add_value( int value ) {
  struct data *item;
  item = kmalloc( sizeof(*item), GFP_ATOMIC );
  if ( !item ) goto out;
  item->value = value;
  down_write( &rw_sem );            /* захватить для записи */
  list_add( &(item->list), &list );
  up_write( &rw_sem );              /* освободить по записи */
  return 0;
out:
  return -ENOMEM;
}
int is_value( int value ) {
  int result = 0;
  struct data *item;
  struct list_head *iter;
  down_read( &rw_sem );             /* захватить для чтения */
  list_for_each( iter, &list ) {
    item = list_entry( iter, struct data, list );
    if( item->value == value ) {
      result = 1; goto out;
    }
  }
out:
  up_read( &rw_sem );               /* освободить по чтению */
  return result;
}
void init_list( void ) {
  init_rwsem( &rw_sem );
  INIT_LIST_HEAD( &list );
}
```

Точно так же, как это сделано для семафоров, вводится и блокировка чтения-записи для спин-блокировки:

```
typedef struct {
  raw_rwlock_t raw_lock;
...
} rwlock_t;
```

С набором операций:

```
read_lock( rwlock_t *rwlock );
```

```
read_unlock( rwlock_t *rwlock );
write_lock( rwlock_t *rwlock );
write_unlock ( rwlock_t *rwlock );
```

Примечание: Если при компиляции ядра не было установлено SMP и не сконфигурировано вытеснение кода в ядре, то spinlock_t вообще не скомпилируются (на их месте останутся пустые места), а, значит, и соответствующие им rwlock_t.

Также, блокировку, захваченную для чтения, уже нельзя далее повысить до блокировки, захваченной для записи.:

```
read_lock( &rwlock );
write_lock( &rwlock );
```

Такая последовательностью операторов гарантирует возникновение deadlock, так как при захвате блокировки на запись будет выполняться периодическая проверка, пока все потоки, которые захватили блокировку для чтения, ее не освободят, это касается и текущего потока, который не сможет сделать этого никогда...

Но несколько потоков чтения могут безопасно захватывать одну и ту же блокировку чтения-записи, поэтому один поток также может безопасно рекурсивно захватывать одну и ту же блокировку для чтения несколько раз, например в обработчике прерываний без запрета прерываний.