Concurrency and Synchronisation

Details

Donato Capitella

Department of Computer Science
University of Birmingham

Linux Kernel Programming, 2011

Outline

Primitives offered by Linux

- Linux implements a wide range of synchronization primitives
  - Atomic operations
  - Spinlocks
  - Semaphores and Mutexes
  - Reader/Writer Locks
  - The big kernel lock
- Note: Atomic operations
  - not exactly syhncoronisation primitives
  - but can be used as a basis for implementing all of the other primitives

Primitives offered by Linux

- Linux implements a wide range of synchronization primitives
  - Atomic operations
  - Spinlocks
  - Semaphores and Mutexes
  - Reader/Writer Locks
  - The big kernel lock
- Note: Atomic operations
  - not exactly syhncoronisation primitives
  - but can be used as a basis for implementing all of the other primitives

Atomic operations (I)

- Basic mechanism
  - guarantee that simple operations on integers are executed atomically on every supported architecture
- Linux offers an abstract data type `atomic_t`
  - defined in `arch/x86/include/asm/atomic.h` (for x86)
  - initialized with `ATOMIC_INIT()` macro
  - accessed with a set of operations `atomic_*()`

Warning

Variables of type `atomic_t` MUST be accessed only through these functions.

Atomic operations (I)

- Basic mechanism
  - guarantee that simple operations on integers are executed atomically on every supported architecture
- Linux offers an abstract data type `atomic_t`
  - defined in `arch/x86/include/asm/atomic.h` (for x86)
  - initialized with `ATOMIC_INIT()` macro
  - accessed with a set of operations `atomic_*()`

Warning

Variables of type atomic_t MUST be accessed only through these functions.

Atomic operations (II)

Operation	Effect
`atomic_read(atomic_t *v)`	Reads the value of the atomic variable.
`atomic_set(atomic_t *v, int i)`	Sets v to i.
`atomic_add(int i, atomic_t *v)`	Adds i to v.
`atomic_add_return(int i, atomic_t *v)`	Adds i to v and returns the result.
`atomic_sub(int i, atomic_t *v)`	Subtracts i from v.
`atomic_sub_return(int i, atomic_t *v)`	Subtracts i from v and returns the result.
`atomic_inc(atomic_t *v)`	Adds 1 to v.
`atomic_inc_and_test(atomic_t *v)`	Adds 1 to v. Returns true if the result is 0, otherwise false.
`atomic_dec(atomic_t *v)`	Subtracts 1 from v.
`atomic_dec_and_test(atomic_t *v)`	Subtracts 1 from v. Returns true if the result is 0, otherwise false.

*Table adapted from W. Mauerer, *Professional Linux Kernel Architecture*, Wiley

counter_atomic.c

- To enforce mutual exclusion:
  - change counter type to atomic_t
  - initialise counter using ATOMIC_INIT macro
  - use atomic_read() to read the variable
  - use atomic_add() to increment the variable

```
/* Shared counter variable */
atomic_t counter = ATOMIC_INIT(0);
...
static int counter_read (...)  {
  ..
  /* Read variable */
  return sprintf(buf, "%li",
atomic_read(&counter));
}
...
static int counter_write(...)  {
  ..
  /* Increment variable */
  atomic_add(digit, &counter);
  ..
}

..
```

Outline

Spinlocks (I)

- **Spinlocks** are mutexes with *active wait*
  - if the mutex is not available, the thread remains spinning at the entry point of the critical section (thus *active wait*)
- Why active wait?
  - (*Performance*) if the critical section is very small and quick, the cost of putting a process to sleep and then awaking it can be higher then the cost of spinning
  - (*Need*) there are some situations where the kernel cannot sleep (e.g. interrupts)

Spinlocks (I)

- **Spinlocks** are mutexes with *active wait*
  - if the mutex is not available, the thread remains spinning at the entry point of the critical section (thus *active wait*)
- Why active wait?
  - (*Performance*) if the critical section is very small and quick, the cost of putting a process to sleep and then awaking it can be higher then the cost of spinning
  - (*Need*) there are some situations where the kernel cannot sleep (e.g. interrupts)

Spinlocks (II)

- A spinlock is represented by the `spinlock_t` data structure
  - defined in `linux/spinlock.h`
  - initialised with `SPIN_LOCK_UNLOCKED` macro
- A spinlock in essentially manipulated with two functions:
  - `spin_lock(spinlock_t *lock)`
  - `spin_unlock(spinlock_t *lock)`
- Example >> `counter_spinlock.c`

Warning

- A critical region protected by a spinlock MUST NOT go to sleep in ANY circumstances.
- Be careful: your code might not go to sleep directly, but some of the functions it calls may(e.g. kmalloc)

Spinlocks (II)

- A spinlock is represented by the `spinlock_t` data structure
  - defined in `linux/spinlock.h`
  - initialised with `SPIN_LOCK_UNLOCKED` macro
- A spinlock in essentially manipulated with two functions:
  - `spin_lock(spinlock_t *lock)`
  - `spin_unlock(spinlock_t *lock)`
- Example >> `counter_spinlock.c`

Warning

- A critical region protected by a spinlock MUST NOT go to sleep in ANY circumstances.
- Be careful: your code might not go to sleep directly, but some of the functions it calls may(e.g. kmalloc)

Spinlocks (II)

- A spinlock is represented by the `spinlock_t` data structure
  - defined in `linux/spinlock.h`
  - initialised with `SPIN_LOCK_UNLOCKED` macro
- A spinlock in essentially manipulated with two functions:
  - `spin_lock(spinlock_t *lock)`
  - `spin_unlock(spinlock_t *lock)`
- Example >> `counter_spinlock.c`

Warning

- A critical region protected by a spinlock MUST NOT go to sleep in ANY circumstances.
- Be careful: your code might not go to sleep directly, but some of the functions it calls may(e.g. kmalloc)

Spinlocks (II)

- A spinlock is represented by the `spinlock_t` data structure
  - defined in `linux/spinlock.h`
  - initialised with `SPIN_LOCK_UNLOCKED` macro
- A spinlock in essentially manipulated with two functions:
  - `spin_lock(spinlock_t *lock)`
  - `spin_unlock(spinlock_t *lock)`
- Example >> `counter_spinlock.c`

Warning

• A critical region protected by a spinlock MUST NOT go to sleep in ANY circumstances.
• Be careful: your code might not go to sleep directly, but some of the functions it calls may(e.g. kmalloc)

Spinlocks (III)

- If the lock is busy, `spin_lock()` is a blocking call
  - sometimes we don't want this behaviour
- Thus, the kernel offers a non-blocking equivalent
  - `int spin_trylock(spinlock_t *lock)`
- Semantics
  - if `spin_trylock` acquires the lock successfully, it returns non-zero value
  - otherwise (it cannot acquire the lock) it returns 0

Warning

- If 0 is returned, the thread MUST NOT enter the critical section because another thread is already there.

Spinlocks (III)

- If the lock is busy, `spin_lock()` is a blocking call
  - sometimes we don't want this behaviour
- Thus, the kernel offers a non-blocking equivalent
  - `int spin_trylock(spinlock_t *lock)`
- Semantics
  - if `spin_trylock` acquires the lock successfully, it returns non-zero value
  - otherwise (it cannot acquire the lock) it returns 0

Warning

- If 0 is returned, the thread MUST NOT enter the critical section because another thread is already there.

Spinlocks (III)

- If the lock is busy, `spin_lock()` is a blocking call
  - sometimes we don't want this behaviour
- Thus, the kernel offers a non-blocking equivalent
  - `int spin_trylock(spinlock_t *lock)`
- Semantics
  - if `spin_trylock` acquires the lock successfully, it returns non-zero value
  - otherwise (it cannot acquire the lock) it returns 0

Warning

- If 0 is returned, the thread MUST NOT enter the critical section because another thread is already there.

Spinlocks (III)

- If the lock is busy, `spin_lock()` is a blocking call
  - sometimes we don't want this behaviour
- Thus, the kernel offers a non-blocking equivalent
  - `int spin_trylock(spinlock_t *lock)`
- Semantics
  - if `spin_trylock` acquires the lock successfully, it returns non-zero value
  - otherwise (it cannot acquire the lock) it returns 0

Warning

• If 0 is returned, the thread MUST NOT enter the critical section because another thread is already there.

Outline

Semaphores

- Linux offers Dijkstra's sempahores as well
  - `struct semaphore`
  - defined in `include/linux/semaphore.h`
- Semaphore structure
  - `lock`: used to make the operations on the semaphore atomic
  - `count`: threads allowed to enter the critical region (other releases used `atomic_t`)
  - `wait_list`: processes sleeping on the sempahore

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

Semaphores

- Linux offers Dijkstra's sempahores as well
  - `struct semaphore`
  - defined in `include/linux/semaphore.h`

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

- Semaphore structure
  - `lock`: used to make the operations on the semaphore atomic
  - `count`: threads allowed to enter the critical region (other releases used `atomic_t`)
  - `wait_list`: processes sleeping on the sempahore

Semaphores in Use

- Initialization
  - macros: `DEFINE_SEMAPHORE(name)`[1], `__SEMAPHORE_INITIALIZER(name, n)`
  - function: `sema_init(struct semaphore *sem, int val)`
- Functions
  - `down(struct semaphore *sem)`
  - `up(struct semaphore *sem)`

```
DEFINE_SEMAPHORE(sem)
...
down(&sem);
/*Critical section*/
up(&sem);

...
```

Look at the source

- Use >> `counter_sem.c`
- Very straight-forward implementation >> `kernel/semaphore.c`

[1] kernel > 2.6.26

Semaphores in Use

- Initialization
  - macros: `DEFINE_SEMAPHORE(name)`[1], `__SEMAPHORE_INITIALIZER(name, n)`
  - function: `sema_init(struct semaphore *sem, int val)`

- Functions
  - `down(struct semaphore *sem)`
  - `up(struct semaphore *sem)`

```
DEFINE_SEMAPHORE(sem)
...
down(&sem);
/*Critical section*/
up(&sem);

...
```

Look at the source

- Use >> `counter_sem.c`
- Very straight-forward implementation >> `kernel/semaphore.c`

[1] kernel > 2.6.26

Semaphores in Use

- Initialization
  - macros: `DEFINE_SEMAPHORE(name)`[1], `__SEMAPHORE_INITIALIZER(name, n)`
  - function: `sema_init(struct semaphore *sem, int val)`

- Functions
  - `down(struct semaphore *sem)`
  - `up(struct semaphore *sem)`

```
DEFINE_SEMAPHORE(sem)
...
down(&sem);
/*Critical section*/
up(&sem);

...
```

Look at the source

- Use >> `counter_sem.c`
- Very straight-forward implementation >> `kernel/semaphore.c`

[1] kernel > 2.6.26

Outline

Special-purpose Mutexes (I)

- As we have seen mutexes are a particular case of sempahores
  - initialized to 1
  - used to enforce mutual exclusion in critical sections
- However, the generality of semaphores causes unneeded **overhead**
- Thus, Linux offers a special-purpose implementation of mutexes
  - datatype `struct mutex`
  - defined in `include/linux/mutex.h`

Special-purpose Mutexes (I)

- As we have seen mutexes are a particular case of sempahores
  - initialized to 1
  - used to enforce mutual exclusion in critical sections
- However, the generality of semaphores causes unneeded **overhead**
- Thus, Linux offers a special-purpose implementation of mutexes
  - datatype `struct mutex`
  - defined in `include/linux/mutex.h`

Special-purpose Mutexes (I)

- As we have seen mutexes are a particular case of sempahores
  - initialized to 1
  - used to enforce mutual exclusion in critical sections
- However, the generality of semaphores causes unneeded **overhead**
- Thus, Linux offers a special-purpose implementation of mutexes
  - datatype `struct mutex`
  - defined in `include/linux/mutex.h`

Special-purpose Mutexes (II)

- Initialisation
  - static: `DECLARE_MUTEX` macro
  - dynamic: `mutex_init()`
- Operations
  - `mutex_lock(struct mutex *lock)`
  - `mutext_unlock(struct mutex *lock)`
  - `mutext_trylock(struct mutex *lock)`

```
#include <linux/mutex.h>
DEFINE_MUTEX(mymutex);

...
/* Acquire the mutex */
mutex_lock(&mymutex);
/* Critical Section */
mutex_unlock(&mymutex);

...
```

Limitations

- A mutex can only be released by the same thread that acquired it.
- The thread may not exit without first unlocking the mutex
- Recursive locking is not allowed

Special-purpose Mutexes (II)

- Initialisation
  - static: `DECLARE_MUTEX` macro
  - dynamic: `mutex_init()`
- Operations
  - `mutex_lock(struct mutex *lock)`
  - `mutext_unlock(struct mutex *lock)`
  - `mutext_trylock(struct mutex *lock)`

```
#include <linux/mutex.h>
DEFINE_MUTEX(mymutex);

...
/* Acquire the mutex */
mutex_lock(&mymutex);
/* Critical Section */
mutex_unlock(&mymutex);

...
```

Limitations

- A mutex can only be released by the same thread that acquired it.
- The thread may not exit without first unlocking the mutex
- Recursive locking is not allowed

Special-purpose Mutexes (II)

- Initialisation
  - static: `DECLARE_MUTEX` macro
  - dynamic: `mutex_init()`
- Operations
  - `mutex_lock(struct mutex *lock)`
  - `mutext_unlock(struct mutex *lock)`
  - `mutext_trylock(struct mutex *lock)`

```
#include <linux/mutex.h>
DEFINE_MUTEX(mymutex);

...
/* Acquire the mutex */
mutex_lock(&mymutex);
/* Critical Section */
mutex_unlock(&mymutex);

...
```

Limitations

- A mutex can only be released by the same thread that acquired it.
- The thread may not exit without first unlocking the mutex
- Recursive locking is not allowed

Outline

Consumer/Producer (I)

- Let us consider the classical Consumer/Producer problem
- Two groups of threads manipulate a **shared buffer**
  - **producers**: produce items and add them to the buffer
  - **consumers**: extract items from the buffer and accomplish some task with them
- The buffer is shared, so we must guarantee **mutual execlusion** when manipulating it
- However, this is not enough
  - if the buffer is empty, consumers must wait for producers to put something in it

Consumer/Producer (I)

- Let us consider the classical Consumer/Producer problem
- Two groups of threads manipulate a **shared buffer**
  - **producers**: produce items and add them to the buffer
  - **consumers**: extract items from the buffer and accomplish some task with them
- The buffer is shared, so we must guarantee **mutual execlusion** when manipulating it
- However, this is not enough
  - if the buffer is empty, consumers must wait for producers to put something in it

Consumer/Producer (I)

- Let us consider the classical Consumer/Producer problem
- Two groups of threads manipulate a **shared buffer**
  - **producers**: produce items and add them to the buffer
  - **consumers**: extract items from the buffer and accomplish some task with them
- The buffer is shared, so we must guarantee **mutual execlusion** when manipulating it
- However, this is not enough
  - if the buffer is empty, consumers must wait for producers to put something in it

Consumer/Producer (I)

- Let us consider the classical Consumer/Producer problem
- Two groups of threads manipulate a **shared buffer**
  - **producers**: produce items and add them to the buffer
  - **consumers**: extract items from the buffer and accomplish some task with them
- The buffer is shared, so we must guarantee **mutual execlusion** when manipulating it
- However, this is not enough
  - if the buffer is empty, consumers must wait for producers to put something in it

Consumer/Producer (II)

- Solution:
  - use a mutex to enforce mutual exclusion on accesses to the buffer
  - use a sempahore (say 'available') to stop the consumers when the buffer is empty
- The semaphore
  - is initialised to 0 (or to the number of items already in the buffer)
  - before extracting an item, consumers have to **wait** on the sempahore issuing down()
  - after adding an item, a producer **signals** that a new item is available by calling up() on the semaphore

Consumer/Producer (II)

- Solution:
  - use a mutex to enforce mutual exclusion on accesses to the buffer
  - use a sempahore (say 'available') to stop the consumers when the buffer is empty
- The semaphore
  - is initialised to 0 (or to the number of items already in the buffer)
  - before extracting an item, consumers have to **wait** on the sempahore issuing down()
  - after adding an item, a producer **signals** that a new item is available by calling up() on the semaphore

cons_prod.c (I)

```
/* List */
struct item { list_head list; int integer};
LIST_HEAD(integers);
/* Synchronisation structures */
DEFINE_MUTEX(mutex);
struct semaphore available = __SEMAPHORE_INITIALIZER(available, 0);
...
static int cons_read (...)  {
  int ret;
  /* This is the consumer's code */
  down(&available);
  mutex_lock(&mutex);
  ret = get_head(integers);
  mutex_unlock(&mutex);
  return sprintf(buf, "%d", ret);
}
```

cons_prod.c (II)

```
static int prod_write(...)  {
  ..
  /* This is the producer code */
  struct item *new_item = build_new_item(digit);
  mutex_lock(&mutex);
  list_add_tail(&(new_item->list), &integers);
  mutex_unlock(&mutex);
  up(&available);
  ..
}
```

Look at the source

- Play with the code >> `prod_cons.c`

Outline

Reader/Writer Locks (I)

- Classical mutual exclusion constraint
  - only one thread can access a resource at a time
- Sometime we want to relax this constraint for performance reasons
  - a group of threads may be allowed to execute concurrently the same operation
  - but we do not allow concurrent execution of any other operation
- *Reader/Writer* classical problem
  - multiple threads are allowed to read concurrently from a data structure
  - while write access is restricted to a single thread at a time

Reader/Writer Locks (I)

- Classical mutual exclusion constraint
  - only one thread can access a resource at a time
- Sometime we want to relax this constraint for performance reasons
  - a group of threads may be allowed to execute concurrently the same operation
  - but we do not allow concurrent execution of any other operation
- *Reader/Writer* classical problem
  - multiple threads are allowed to read concurrently from a data structure
  - while write access is restricted to a single thread at a time

Reader/Writer Locks (I)

- Classical mutual exclusion constraint
  - only one thread can access a resource at a time
- Sometime we want to relax this constraint for performance reasons
  - a group of threads may be allowed to execute concurrently the same operation
  - but we do not allow concurrent execution of any other operation
- *Reader/Writer* classical problem
  - multiple threads are allowed to read concurrently from a data structure
  - while write access is restricted to a single thread at a time

Reader/Writer Locks (II)

- Linux provides additional implementations of semaphores and spinlocks for the reader/writer problem
- Reader/Writer locks - `rwlock_t` (`include/linux/rwlock_types.h`)
  - `read_lock()` and `read_unlock()`
  - `write_lock()` and `write_unlock()`
- Reader/Writer semaphores - `struct rw_semaphore` (`include/linux/rwsem.h`)
  - `down_read()` and `up_read()`
  - `down_write()` and `up_write()`

Look at the source

- counter LKM using reader/writer sempahores >> `counter_rwsem.c`

Reader/Writer Locks (II)

- Linux provides additional implementations of semaphores and spinlocks for the reader/writer problem
- Reader/Writer locks - `rwlock_t` (`include/linux/rwlock_types.h`)
  - `read_lock()` and `read_unlock()`
  - `write_lock()` and `write_unlock()`
- Reader/Writer semaphores - `struct rw_semaphore` (`include/linux/rwsem.h`)
  - `down_read()` and `up_read()`
  - `down_write()` and `up_write()`

Look at the source

- counter LKM using reader/writer sempahores >> `counter_rwsem.c`

Reader/Writer Locks (II)

- Linux provides additional implementations of semaphores and spinlocks for the reader/writer problem
- Reader/Writer locks - `rwlock_t` (`include/linux/rwlock_types.h`)
  - `read_lock()` and `read_unlock()`
  - `write_lock()` and `write_unlock()`
- Reader/Writer semaphores - `struct rw_semaphore` (`include/linux/rwsem.h`)
  - `down_read()` and `up_read()`
  - `down_write()` and `up_write()`

Look at the source

- counter LKM using reader/writer sempahores >> `counter_rwsem.c`

Reader/Writer Locks (II)

- Linux provides additional implementations of semaphores and spinlocks for the reader/writer problem
- Reader/Writer locks - `rwlock_t` (`include/linux/rwlock_types.h`)
  - `read_lock()` and `read_unlock()`
  - `write_lock()` and `write_unlock()`
- Reader/Writer semaphores - `struct rw_semaphore` (`include/linux/rwsem.h`)
  - `down_read()` and `up_read()`
  - `down_write()` and `up_write()`

Look at the source

- counter LKM using reader/writer sempahores >> `counter_rwsem.c`

Outline

The Big Kernel Lock

- Big Kernel Lock (BKL)
  - introduced in the early days of multiprocessor support
  - to prevent more than one processor from running in parallel in kernel mode
- Still used in some parts of the kernel, but DEPRECATED
  - new code should never use it
  - prefer the finer-grained options described before
- Operations
  - `lock_kernel()`
  - `unlock_kernel()`

Summary

- Various synchronisation mechanisms in the kernel.
- How to some some common synchronization problems
  - Mutual exclusion
  - Produer/Consumer
  - Reader/Writer
- Next $\rightarrow$ Advanced concepts
  - Kernel threads and completion structures
  - Deferred work