Semaphores, Mutexes, and Spinlocks

Basic Concurrency Features of the Linux Kernel

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Defining “Concurrency”

Linux can do more than one thing at once:

- User programs
- System calls
- Device drivers
- Interrupts
- Kernel threads
- ...

Defining “Concurrency”

User programs and system calls:

- Simultaneous reading and writing
- Multiple programs using the same device

Device drivers:

- Simultaneous reading and writing
- Waiting for a device
- Blocking I/O
- Interrupt handling

Defining “Concurrency”

Modern Linux kernels are *preemptible*:

- System calls will interrupt each other
- Interrupt handlers run as “tasks”

Older (pre-2.6) kernels:

- The BKL, a.k.a. “The Big Kernel Lock”
- Only one system call at a time
- Poor performance, but simpler code

Races

Race condition:

- Flow of execution is timing- or event-related
- Usually related to poor control of shared resources
- Net result is an unstable, unpredictable system

Code for concurrency, or else!

- You will lose data
- You will hang the system

Can you spot the race?

```
int head, tail;
char buf[BUF_SIZE];

interrupt_handler()
{
  int old_head = head;

  if (++head >= BUF_SIZE) head = 0;
  if (head == tail)
     if (++tail >= BUF_SIZE) tail = 0;

  buf[old_head] = *RXBUF;
}
```

Can you spot the race?

```
int read()
{
  int c;

  if (head != tail) {
    c = buf[tail];



    if (++tail >= BUF_SIZE) tail = 0;
  }

  return c;
}
```

Can you spot the race?

```
int read()
{
  int c;

  if (head != tail) {
    c = buf[tail];

    /* hint: and then a byte comes in ... */

    if (++tail >= BUF_SIZE) tail = 0;
  }

  return c;
}
```

Races

Data sharing across contexts:

- The most common source of concurrency problems

So:

- Avoid shared resources whenever possible
- When sharing, do it conservatively
- When sharing, use concurrency-related facilities
- (Some shared resources are unavoidable)

Races

Concurrency-related facilities:

- Semaphores and mutexes
- Completions
- Spinlocks
- Lock-free algorithms

“Haven’t I seen this before?”

“Waitaminit. Isn’t this just like with an RTOS?”

- Yes!

“Haven’t I seen this before?”

“Waitaminit. Isn’t this just like with an RTOS?”

- Yes!

<meg_ryan>

- Yes!
- Yes!
- YES!

</meg_ryan>

Going to Sleep

Yielding the processor to another context:

- Waiting for I/O
- Waiting for a signal from another context
- Waiting for memory to become available
- Waiting for a request for services

Going to Sleep

Contexts that can’t sleep:

- Interrupt handlers
- Tasklets
- Critical sections (spinlocks)

Going to Sleep

Some kernel services will sleep:

- Memory allocations
- Delays
- Copying data to/from userspace

Concurrency facilities:

- Facilitate sleeps when they're needed
- Prevent sleeps when they need to be avoided

Critical Sections

A *critical section* is code that must be atomic:

- Updates to shared data (careful!)
- Reconfiguration of hardware resources

Not all critical sections are alike!

- Pick the concurrency facility that fits best

Semaphores

Semaphore:

- An integer, plus
- `up()` and `down()` functions

```
#include <linux/semaphore.h>
```

Semaphores

`down()`

- Sleeps until the integer is nonzero
- Decrements the integer and continues
- “Down” is not the same as “lock”!

```
void down(struct semaphore *sem)
```

Semaphores

`up()`

- Increments the integer

```
void up(struct semaphore *sem)
```

Semaphore API

Initialize a `struct semaphore`:

- MUST initialize before `up()` or `down()`

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

Semaphore API

```
void down(struct semaphore *sem)
```

- Waits (indefinitely) if needed
- Decrements the semaphore

```
void up(struct semaphore *sem)
```

- Increments the semaphore

Interruptible waits

Waiting that can be interrupted by a signal:

- Send the process a `signal(2)` to interrupt
- Lets users kill a process that's hung in a wait

Use an interruptible wait whenever possible:

- Non-interruptible waits are truly noninterruptible!

Interruptible waits

`int down_interruptible(struct semaphore *sem)`

- Waits (indefinitely) if needed
- Can be interrupted by a `signal(2)`
- Returns nonzero if interrupted

`int down_trylock(struct semaphore *sem)`

- Never waits
- Returns 0 if waiting is necessary
- If successful, equivalent to `down()`

“How do I send a `signal(2)`?”

To send a `signal(2)` to a process:

```
$ ps -C my_process_name
 PID   TTY     TIME CMD
5964 pts/2 00:00:00 my_process_name
$ kill 5964
```

Mutex API

Mutex:

- Blocking, mutual exclusion locks

```
#include <linux/mutex.h>
```

Mutex API

```
void mutex_lock(struct mutex *lock)
```

- Locks a mutex
- Sleeps if locked, potentially indefinitely

Mutex API

```
int mutex_lock_interruptible(struct mutex *lock)
```

- Returns 0 on successful lock
- Sleeps if mutex is already locked
- Returns `-EINTR` if a signal arrives

Mutex API

```
int mutex_trylock(struct mutex *lock)
```

- Returns 1 if the mutex was acquired
- Returns 0 if the mutex was already locked
- Never sleeps

Mutex API

```
void mutex_unlock(struct mutex *lock)
```

- Unlocks a mutex
- Only the owner can unlock the mutex
- Interrupt handlers cannot unlock mutexes!

Reader-Writer Semaphores

Semaphores are a worst-case facility:

- All callers will wait, if necessary

Reader-writer patterns can be optimized:

- Only one writer
- As many readers as necessary
- Readers wait while writer is busy

Reader-Writer Semaphores

Linux reader-writer semaphores (`rwsem`):

- Optimized for reader-writer patterns
- Infrequently used, but powerful when needed
- See also `drivers/leds/leds-bd2802.c`

```
#include <linux/rwsem.h>
```

Reader-Writer Semaphores

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

- Initializes a reader-writer semaphore

```
void down_read(struct rw_semaphore *sem)
```

- Obtains a read-only lock
- Concurrent with other readers, if any
- Uninterruptible wait

Reader-Writer Semaphores

```
int down_read_trylock(struct rw_semaphore *sem)
```

- Obtains a read-only lock
- Will not wait on contention
- Returns 0 if waiting is necessary

```
void up_read(struct rw_semaphore *sem)
```

- Frees a read-only lock

Reader-Writer Semaphores

`void down_write(struct rw_semaphore *sem)`

- Obtains read/write access
- Readers denied access until `up_write()` or `downgrade_write()`

`int down_write_trylock(struct rw_sempahore *sem)`

- Will not wait on contention
- Returns 0 if waiting is necessary

Reader-Writer Semaphores

```
void up_write(struct rw_semaphore *sem)
```

- Frees a read/write lock

```
void downgrade_write(struct rw_semaphore *sem)
```

- Converts a read/write lock to a read-only

Notes on Semaphores

Semaphores explicitly sleep:

- So you can't call `down()` just anywhere!

You can't sleep in:

- Interrupt handlers
- Tasklets
- Critical sections
- Any time you are holding a spinlock!

Notes on Semaphores

Reader-writer semaphores don’t enforce anything:

- Nothing physically prevents writing after `down_read()`
- (But it would be unwise to do so!)

simple_semaphore.c

```
struct simple_semaphore_data {
  struct file_operations fops;
  struct cdev cdev;
  struct semaphore sem;
};

static char stuff[32];
static int in, out;
#define NSTUFF (sizeof(stuff) / sizeof(*stuff))
```

simple_semaphore.c

```c
static ssize_t
simple_semaphore_write (struct file *file, const char __user *buf,
                        size_t count, loff_t *offp)
{
  int orig_count = count;

  while(count--) {
    int new_in = in + 1;
    if (new_in >= NSTUFF) new_in = 0;

    if (new_in != out) {
        copy_from_user(&stuff[in], buf++, 1);
        up(&simple_semaphore_data.sem);
        in = new_in;
      }
    else break;
  }

  return orig_count;
}
```

simple_semaphore.c

```
static ssize_t
simple_semaphore_read (struct file *file, char __user *buf,
                       size_t count, loff_t *offp)
{
  if (down_interruptible(&simple_semaphore_data.sem))
    return -EINTR;

  copy_to_user(buf, &stuff[out], 1);
  if (++out >= NSTUFF) out = 0;

  return 1;
}
```

Spinlocks

A variation on mutexes:

- During contention, waiters “spin” instead of sleep
- Suitable for interrupt handlers, among others

```
#include <linux/spinlock.h>
```

corgi_ssp.c

```
unsigned long corgi_ssp_ads7846_putget(ulong data)
{
        unsigned long flag;
        u32 ret = 0;

        spin_lock_irqsave(&corgi_ssp_lock, flag);
        if (ssp_machinfo->cs_ads7846 >= 0)
                GPCR(ssp_machinfo->cs_ads7846) = GPIO_bit(ssp_machinfo->cs_ads7846);

        ssp_write_word(&corgi_ssp_dev,data);
        ssp_read_word(&corgi_ssp_dev, &ret);

        if (ssp_machinfo->cs_ads7846 >= 0)
                GPSR(ssp_machinfo->cs_ads7846) = GPIO_bit(ssp_machinfo->cs_ads7846);
        spin_unlock_irqrestore(&corgi_ssp_lock, flag);

        return ret;
}
```

Spinlocks

Common uses:

- Critical code sections
- Resource protection e.g. data structures

Spinlocks

```
void spin_lock_init(spinlock_t *lock)
```

- Initializes a spinlock

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

- Define-plus-initialize

Spinlocks

`void spin_lock(spinlock_t *lock)`

- Acquire the lock
- Spin while waiting, if necessary
- By definition, spinlocks are uninterruptible
- Don't sleep while holding a lock!

`int spin_trylock(spinlock_t *lock)`

- Will not wait on contention
- Returns 0 if waiting is necessary

Spinlocks

```
void spin_unlock(spinlock_t *lock)
```

- Releases the lock

Spinlocks

An ugly scenario:

- Device driver takes spinlock
- Device issues interrupt request
- Interrupt handler tries to take spinlock
- *kaboom!*

Spinlocks

```
void spin_lock_irqsave(spinlock_t *lock,
                       unsigned long flags)
```

- Takes spinlock
- Disables interrupts
- Saves interrupt state in `flags`

Spinlocks

```
void spin_unlock_irqrestore(spinlock_t *lock,
                            unsigned long flags)
```

- Releases spinlock
- Restores interrupt state per `flags`

mach-pxa/ssp.c

```
int ssp_read_word(struct ssp_dev *dev, u32 *data)
{
        int timeout = TIMEOUT;

        while (!(SSSR_P(dev->port) & SSSR_RNE)) {
                if (!--timeout)
                        return -ETIMEDOUT;
                cpu_relax();
        }

        *data = SSDR_P(dev->port);
        return 0;
}
```

Spinlocks

```
void read_lock(rwlock_t *lock)
```

- Locks a read-only spinlock

```
void read_lock_irqsave(rwlock_t *...)
```

- Read-only spinlock, disables interrupts

Spinlocks

```
void write_lock(rwlock_t *lock)
```

- Locks a read-write spinlock

```
void write_lock_irqrestore(rwlock_t *...)
```

- Read-write spinlock, restores interrupts

“How is the resource being shared?”

With an interrupt handler:

- Must use a spinlock or rwlock

Between peer contexts:

- Semaphore, if counting is needed
- Mutex, if only one resource

Lock-Free Algorithms

Avoid the need for locking whenever possible:

- Recast the problem
- Use a lock-free algorithm
- (Circular buffers can often be lock-free)

Advantages of lock-free algorithms:

- Avoids risks of locking
- Often better performance

Can you spot the race?

```
int head, tail;
char buf[BUF_SIZE];

interrupt_handler()
{
  int old_head = head;

  if (++head >= BUF_SIZE) head = 0;
  if (head == tail)
     if (++tail >= BUF_SIZE) tail = 0;

  buf[old_head] = *RXBUF;
}
```

Can you spot the race?

```
int read()
{
  int c;

  if (head != tail) {
    c = buf[tail];

    /* hint: and then a byte comes in ... */

    if (++tail >= BUF_SIZE) tail = 0;
  }

  return c;
}
```

Can you spot the race?

If `read()` is interrupted, `tail` moves!

- The reader gets the newest data, not oldest, or
- The `tail` index gets corrupted, and data is lost
- ... but only if interrupted in the critical section!

```
if (head != tail) {
  c = buf[tail];
  if (++tail >= BUF_SIZE) tail = 0;
}
```

Can you spot the race?

Instead of:

```
/* we’re full; discard oldest data to make room */
if (head == tail)
   if (++tail >= BUF_SIZE) tail = 0;
```

Do this:

```
/* we’re full; discard incoming data */
if (head == tail) return;
```

Can you spot the race?

This fixes the race:

- The `tail` object isn't shared
- No contention == no race
- No contention == no need for a lock!

```
/* we're full; discard incoming data */
if (head == tail) return;
```

Semaphores, Mutexes, and Spinlocks

Basic Concurrency Features of the Linux Kernel

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer