SPINLOCKS, Part I

by Mike Rieker

Introduction

A method is needed in a multiprocessor system to keep the cpu's from interfering with each other. For example, let's say you have a list of threads that are ready to execute instructions, so they are waiting for a cpu. There needs to be a way to make sure that only one of the cpu's will start processing that thread.

Now you may say you're not writing a multiprocessor os, so you don't need these. You're right, you don't. But you need something similar, so with little extra effort, you can make your os multiprocessor from the start. We'll delay this discussion for the end.

Getting along without them

So let's take our example started above. We have a list of threads that are ready to be executed, and we want to be sure that only one cpu will start executing the thread. Here's the idea:

lock all other cpu's out of the list of threads waiting for a cpu unlink the first one that's waiting, if any unlock the list if we didn't find a thread, repeat the whole thing else, jump to the thread

Now how do we do that 'lock all other cpu's...'?

If we just:

```
static int threads_waiting_to_run_lock = 0;
static Thread *threads_waiting_to_run = NULL;
static Thread **threads_waiting_to_run_end = &threads_waiting_to_run;

Thread *find_a_thread_to_run (void)

{
  Thread *thread_to_start;

  do {
    while (threads_waiting_to_run_lock) {}         // repeat while some other cpu is using the queue
    threads_waiting_to_run_lock = 1;               // tell other cpu's the queue is being used

    thread_to_start = threads_waiting_to_run;
    if (thread_to_start != NULL) {
      threads_waiting_to_run = thread_to_start -> next_thread;
      if (threads_waiting_to_run == NULL) threads_waiting_to_run_end = &threads_waiting_to_run;
    }

    lock = 0;                                      // tell other cpu's the queue is no longer busy
  } while (thread_to_start == NULL);               // repeat if I didn't get anything to do

  return (thread_to_start);
}
```

That won't work because if two cpu's simultaneously hit the while statement, they will both see lock as being zero, then they both will set it to one. And they both will dequeue the same top thread, which is no good.

No simple C statements will solve this problem.

What makes it work

So, for Intel processors, there is the 'lock' instruction prefix. This says, that for the following instruction, this instruction shall be the only one that accesses that memory location. So we can make a routine like this:

```
    int test_and_set (int new_value, int *lock_pointer);

    .globl test_and_set
  test_and_set:
    movl 4(%esp),%eax  # get new_value into %eax
    movl 8(%esp),%edx  # get lock_pointer into %edx
    lock               # next instruction is locked
    xchgl %eax,(%edx)  # swap %eax with what is stored in (%edx)
                       # ... and don't let any other cpu touch that
```

```
                        # ... memory location while you're swapping
    ret                 # return the old value that's in %eax
```

Now we can correctly make our routine:

```
Thread *find_a_thread_to_run (void)

{
  Thread *thread_to_start;

  do {
    while (test_and_set (1, &threads_waiting_to_run_lock)) {} // repeat while some other cpu is using the queue
                                                              // ... then tell other cpu's the queue is being used

    thread_to_start = threads_waiting_to_run;
    if (thread_to_start != NULL) {
      threads_waiting_to_run = thread_to_start -> next_thread;
      if (threads_waiting_to_run == NULL) threads_waiting_to_run_end = &threads_waiting_to_run;
    }

    threads_waiting_to_run_lock = 0;                          // tell other cpu's the queue is no longer busy
  } while (thread_to_start == NULL);                          // repeat if I didn't get anything to do

  return (thread_to_start);
}
```

So the 'while' loop can be said to be 'spinning' on that test_and_set call while some other cpu is using the lock.

Dealing with interrupts

Now for the complications...

You may ask yourself, isn't this thing going to sit there forever in that do..while loop if there is nothing in the queue? What puts things in the queue?

So some device gives an hardware interrupt while we're in the loop. This interrupt routine checks the disk or keyboard status, does it's thing, and realizes it is time to wake up a thread that is waiting for the I/O. Here is where it puts the thread in the 'threads_waiting_to_run' queue.

Now it can't just:

```
void wake_thread (Thread *thread_to_wake)

{
  thread_to_wake -> next_thread = NULL;
  *threads_waiting_to_run_end = thread_to_wake;
  threads_waiting_to_run_end = &(thread_to_wake -> next_thread);
}
```

... because the interrupt may have happened between the second and third lines of:

```
    thread_to_start = threads_waiting_to_run;
    if (thread_to_start != NULL) {
      threads_waiting_to_run = thread_to_start -> next_thread;
      if (threads_waiting_to_run == NULL) threads_waiting_to_run_end = &threads_waiting_to_run;
    }
```

So you in essence have the following happening:

```
    thread_to_start = threads_waiting_to_run;
    if (thread_to_start != NULL) {

      --> interrupt
        thread_to_wake -> next_thread = NULL;
        *threads_waiting_to_run_end = thread_to_wake;
        threads_waiting_to_run_end = &(thread_to_wake -> next_thread);
      <-- return from interrupt

      threads_waiting_to_run = thread_to_start -> next_thread;
      if (threads_waiting_to_run == NULL) threads_waiting_to_run_end = &threads_waiting_to_run;
    }
```

... and we end up possibly losing our 'thread_to_wake'.

To solve this, we just inhibit hardware interrupt delivery as well as lock our spinlock:

```
Thread *find_a_thread_to_run (void)

{
  Thread *thread_to_start;

  do {
    inhib_hw_int_delivery ();                                 // make sure interrupt routine doesn't interfere
    while (test_and_set (1, &threads_waiting_to_run_lock)) {} // repeat while some other cpu is using the queue
                                                              // ... then tell other cpu's the queue is being used

    thread_to_start = threads_waiting_to_run;
    if (thread_to_start != NULL) {
      threads_waiting_to_run = thread_to_start -> next_thread;
      if (threads_waiting_to_run == NULL) threads_waiting_to_run_end = &threads_waiting_to_run;
    }

    threads_waiting_to_run_lock = 0;                          // tell other cpu's the queue is no longer busy
    enable_hw_int_delivery ();                                // allow interrupts to happen now
  } while (thread_to_start == NULL);                          // repeat if I didn't get anything to do

  return (thread_to_start);
}
```

Interrupts on a multiprocessor

Now we're almost done. If we are on a multiprocessor, one of the cpu's might be executing the do..while that is waiting for something to work on, and another cpu might be doing the interrupt processing. So we end up with our bad situation again, where we have the four statements executing simultaneously. Easy to fix. We just put our spinlock around the statements in our interrupt routine as well:

```
void wake_thread (Thread *thread_to_wake)

{
  int hwi;

  thread_to_wake -> next_thread = NULL;

  hwi = inhib_hw_int_delivery ();                           // make sure interrupt delivery is inhibited
  while (test_and_set (1, &threads_waiting_to_run_lock)) {} // repeat while some other cpu is using the queue
                                                            // ... then tell other cpu's the queue is being used
  *threads_waiting_to_run_end = thread_to_wake;
  threads_waiting_to_run_end = &(thread_to_wake -> next_thread);
  if (hwi) enable_hw_int_delivery ();                       // restore interrupt delivery state
}
```

Rules to live by

It seems very complicated from all that. Well, if you follow a couple simple rules, it isn't!

1. When modifying a variable that can possibly be modified by more than one cpu at a time, then you need to protect it with a spinlock. You must *always* have this spinlock when accessing that variable.
2. If the variable can also be modified by an interrupt routine, you must disable (at least that) interrupt while the spinlock is being held.

Uniprocessors

So now, you uniprocessor people wonder why all this is necessary. For a uniprocessor, you have:

```
Thread *find_a_thread_to_run (void)

{
  do {
    inhib_hw_int_delivery ();                                 // make sure interrupt routine doesn't interfere

    thread_to_start = threads_waiting_to_run;
    if (thread_to_start != NULL) threads_waiting_to_run = thread_to_start -> next_thread;


    enable_hw_int_delivery ();                                // allow interrupts to happen now
  } while (thread_to_start == NULL);                          // repeat if I didn't get anything to do
  return (thread_to_start);
}


void wake_thread (Thread *thread_to_wake)

{
  int hwi;

  thread_to_wake -> next_thread = NULL;
  hwi = inhib_hw_int_delivery ();                           // make sure interrupt delivery is inhibited
  *threads_waiting_to_run_end = thread_to_wake;
  threads_waiting_to_run_end = &(thread_to_wake -> next_thread);
  if (hwi) enable_hw_int_delivery ();                       // restore interrupt delivery state
}
```

So you can see that it takes all of 2 extra lines per routine to make it multiprocessor ready!

Part 2 talks about some more practical stuff.

Mike's home page can be found at http://www.o3one.org/