SPINLOCKS, Part III

by Mike Rieker

Increment/Decrement

In some cases, there are simple and effective alternatives to spinlocks.

Consider the case:

```
void dec_thread_refcount (Thread *thread)

{
  int new_refcount;

  acquire_spinlock (&(thread -> spinlock));  // lock access to thread -> refcount
  new_refcount = -- (thread -> refcount);    // decrement it, save new value
  release_spinlock (&(thread -> spinlock));  // unlock
  if (new_refcount == 0) kfree (thread);     // if no one is using it anymore, free it
}
```

Seems like a lot of work acquiring and releasing the spinlock just to decrement the refcount. Well, remember the 'lock' prefix instruction we used to make the test_and_set thing? We can be a little fancier about it and make a 'dec_and_testz' routine:

```
  int locked_dec_and_testz (int *refcount);  // decrement *refcount, return whether or not it went

    .globl locked_dec_and_testz
  locked_dec_and_testz:
    xorl %eax,%eax    # clear all of %eax
    movl 4(%esp),%edx # get pointer to refcount
    lock              # keep other cpu's out of next instruction
    decl (%edx)       # decrement refcount, keeping other cpu's out of refcount
    setz %al          # set %eax=1 if result is zero, set %eax=0 otherwise
    ret               # return %eax as the result
```

So now we can write:

```
void dec_thread_refcount (Thread *thread)

{
  if (locked_dec_and_testz (&(thread -> refcount)) {
    kfree (thread);
  }
}
```

Now this has a little gotcha. 'refcount' must now be thought of as being a variable not protected by a spinlock, but by 'lock instruction prefix' access only! So any modifications to refcount must be done with the lock prefix. So we can no longer:

```
  acquire_spinlock (&(thread -> spinlock));
  (thread -> refcount) ++;
  release_spinlock (&(thread -> spinlock));
```

... because the locked_dec_and_testz routine might be in progress on another cpu, and our 'spinlock' won't do anything to stop it!

So we have to supply a 'locked_inc' routine:

```
  void locked_inc (int *refcount);

    .globl locked_inc
  locked_inc:
```

```
  movl 4(%esp),%edx
  lock
  incl (%edx)
  ret
```

But we still come out ahead, because there is no possibility of doing any spinning in any of these routines. (The actual spinning is done at the CPU bus level, which is very very fast).

Atomic arithmetic

Now we can generally apply increment/decrement to any single arithmetic operation on a single variable. It is possible to write routines to do add, subtract, or, and, xor, etc, and have the routine return the previous value.

For example, this routine or's in 'new_bits' into *value, and returns the previous contents of value:

```
int atomic_or_rtnold (int new_bits, int *value);

  .globl atomic_or_rtnold
atomic_or_rtnold:
  movl 8(%esp),%edx    # point to value
  movl (%edx),%eax     # sample the current contents of value
atomic_or_loop:
  movl 4(%esp),%ecx    # get the new_bits
  orl  %eax,%ecx       # or them together
  lock                 # bus lock the next instruction
  cmpxchgl %ecx,(%edx) # if 'value' hasn't changed, store our new value there
  jne atomic_or_loop   # repeat if 'value' has changed
  ret                  # return with old (sampled) contents of value
```

Now you notice this does have a little 'spin' in it, it has a loop back. But the loop is so short, that the chances of conflicting with other modifications of the *value is very slim, so it is highly unlikely that it will ever loop.

If we don't want the old value, we can do this:

```
void atomic_or (int new_bits, int *value);

  .globl atomic_or
atomic_or:
  movl 4(%esp),%eax    # get the new_bits
  movl 8(%esp),%edx    # point to value
  lock                 # bus lock the next instruction
  orl  %eax,(%edx)     # update value
  ret
```

... and not have any loop in there at all.

Atomic linked stack

You can get 'nasty' and implement a LIFO linked stack with atomic operations.

Suppose we have:

```
typedef struct Block Block;
struct Block { Block *next_block;
                ...
             };

static Block *free_blocks = NULL;

void free_a_block (Block *block_to_free)

{
  Block *old_top_free_block;

  do {
```

```
    old_top_free_block = free_blocks;
    block_to_free = old_top_free_block;
  } while (!atomic_setif_ptr (&free_blocks, block_to_free, old_top_free_block));
}

atomic_setif_ptr says:

  if (free_blocks != old_top_free_block) return (0);
  else {
    free_blocks = block_to_free;
    return (1);
  }

  .globl atomic_setif_ptr
atomic_setif_ptr:
  movl  4(%esp),%edx    # get pointer to free_blocks
  movl  8(%esp),%ecx    # get block_to_free
  movl 12(%esp),%eax    # get old_free_top_block
  lock                  # bus lock the next instruction
  cmpxchgl %ecx,(%edx)  # if free_blocks == old_free_top_block, then free_blocks = block_to_free
  jne atomic_setif_failed
  movl $1,%eax          # success, return 1
  ret
atomic_setif_failed:
  xorl %eax,%eax        # failure, return 0
  ret
```

Now we can use that same routine to write the alloc routine:

```
Block *alloc_a_block (void)

{
  Block *sample_block;

  do {
    sample_block = free_blocks;
    if (sample_block == NULL) break;
  } while (!atomic_setif_ptr (&free_blocks, sample_block -> next_block, sample_block));
  return (sample_block);
}
```

But again, the gotcha is that 'free_blocks' can only be modified with atomic operations. So if you must also scan the list, you will have to use a spinlock.

Summary

Simple spinlocked sequences can be replaced with atomic operations and can result in performance improvements. Now you have more fun stuff to play with!

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