Operating System Principles: Mutual Exclusion and Asynchronous Completion

CS 111

Operating Systems

Peter Reiher

Outline

- Mutual Exclusion
- Asynchronous Completions

Mutual Exclusion

- Critical sections can cause trouble when more than one thread executes them at a time
  - Each thread doing part of the critical section before any of them do all of it
- Preventable if we ensure that only one thread can execute a critical section at a time
- We need to achieve *mutual exclusion* of the critical section

Critical Sections in Applications

- Most common for multithreaded applications
  - Which frequently share data structures
- Can also happen with processes
  - Which share operating system resources
  - Like files
- Avoidable if you don't share resources of any kind
  - But that's not always feasible

Recognizing Critical Sections

- Generally involves updates to object state
  - May be updates to a single object
  - May be related updates to multiple objects
- Generally involves multi-step operations
  - Object state inconsistent until operation finishes
  - Pre-emption compromises object or operation
- Correct operation requires mutual exclusion
  - Only one thread at a time has access to object(s)
  - Client 1 completes before client 2 starts

Critical Sections and Atomicity

- Using mutual exclusion allows us to achieve *atomicity* of a critical section
- Atomicity has two aspects:

1. Before or After atomicity
   - A enters critical section before B starts
   - B enters critical section after A completes
   - There is no overlap
2. All or None atomicity
   - An update that starts will complete
   - An uncompleted update has no effect

- Correctness generally requires both

Options for Protecting Critical Sections

- Turn off interrupts
  - We covered that in the last class
  - Prevents concurrency
- Avoid shared data whenever possible
- Protect critical sections using hardware mutual exclusion
  - In particular, atomic CPU instructions
- Software locking

Avoiding Shared Data

- A good design choice when feasible
- Don’t share things you don’t need to share
- But not always an option
- Even if possible, may lead to inefficient resource use
- Sharing read only data also avoids problems
  - If no writes, the order of reads doesn’t matter
  - But a single write can blow everything out of the water

Atomic Instructions

- CPU instructions are uninterruptable
- What can they do?
  – Read/modify/write operations
  – Can be applied to 1-8 contiguous bytes
  – Simple: increment/decrement, and/or/xor
  – Complex: test-and-set, exchange, compare-and-swap
- Either do entire critical section in one atomic instruction
- Or use atomic instructions to implement locks
  – Use the lock operations to protect critical sections

Atomic Instructions – Test and Set

A C description of a machine language instruction

```
bool TS( char *p) {
   bool rc;
   rc = *p;                  /* note the current value              */
   *p = TRUE;           /* set the value to be TRUE           */
   return rc;                 /* return the value before we set it   */
}

if !TS(flag) {
        /* We have control of the critical section! */
}
```

Atomic Instructions – Compare and Swap

Again, a C description of machine instruction

```
bool compare_and_swap( int *p, int old, int new ) {
  if (*p == old) {    /* see if value has been changed     */
      *p = new;       /* if not, set it to new value       */
      return( TRUE);  /* tell caller he succeeded          */
  } else              /* value has been changed       */
    return( FALSE);   /* tell caller he failed             */
}

if (compare_and_swap(flag,UNUSED,IN_USE) {
      /* I got the critical section! */
} else {
      /* I didn’t get it.  */
}
```

Preventing Concurrency Via Atomic Instructions

- CPU instructions are hardware-atomic
  - So if you can squeeze a critical section into one instruction, no concurrency problems
- What can you do in one instruction?
  - Simple operations like read/write
  - Some slightly more complex operations
  - With careful design, some data structures can be implemented this way
- Limitations
  - Unusable for complex critical sections
  - Unusable as a waiting mechanism

Locking

- Protect critical sections with a data structure
  – Use atomic instructions to implement that structure
- Locks
  – The party holding a lock can access the critical section
  – Parties not holding the lock cannot access it
- A party needing to use the critical section tries to acquire the lock
  – If it succeeds, it goes ahead
  – If not . . .?
- When finished with critical section, release the lock
  – Which someone else can then acquire

Using Locks

- Remember this example?

thread #1	**thread #2**
counter = counter + 1;	counter = counter + 1;

What looks like one instruction in C gets compiled to:

```
mov counter, %eax
add $0x1, %eax
mov %eax, counter
```

Three instructions . . .

- How can we solve this with locks?

Using Locks For Mutual Exclusion

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
…
if (pthread_mutex_lock(&lock) == 0) {
   counter = counter + 1;
   pthread_mutex_unlock(&lock);
}
```

Now the three assembly instructions are mutually exclusive

What Happens When You Don't Get the Lock?

- You could just give up
  - But then you'll never execute your critical section
- You could try to get it again
- But it still might not be available
- So you could try to get it again . . .

Spin Waiting

- The computer science equivalent
- Check if the event occurred
- If not, check again
- And again
- And again
- . . .

Spin Locks: Pluses and Minuses

- Good points
  - Properly enforces access to critical sections
    - Assuming properly implemented locks
  - Simple to program
- Dangers
  - Wasteful
    - Spinning uses processor cycles
  - Likely to delay freeing of desired resource
    - Spinning uses processor cycles
  - Bug may lead to infinite spin-waits

How Do We Build Locks?

- The very operation of locking and unlocking a lock is itself a critical section
  - If we don't protect it, two threads might acquire the same lock
- Sounds like a chicken-and-egg problem
- But we can solve it with hardware assistance
- Individual CPU instructions are atomic
  - So if we can implement a lock with one instruction . . .

Single Instruction Locks

- Sounds tricky
- The core operation of acquiring a lock (when it's free) requires:
  1. Check that no one else has it
  2. Change something so others know we have it
- Sounds like we need to do two things in one instruction
- No problem – hardware designers have provided for that

Building Locks From Single Instructions

- Requires a complex atomic instruction
  - Test and set
  - Compare and swap
- Instruction must atomically:
  - Determine if someone already has the lock
  - Grant it if no one has it
  - Return something that lets the caller know what happened
- Caller must honor the lock . . .

Using Atomic Instructions to Implement a Lock

- Assuming C implementation of test and set

```
bool getlock( lock *lockp) {
  if (TS(lockp) == 0 )
     return( TRUE);
  else
     return( FALSE);
}
void freelock( lock *lockp ) {
  *lockp = 0;
}
```

The Asynchronous Completion Problem

- Parallel activities move at different speeds
- One activity may need to wait for another to complete
- The *asynchronous completion problem* is how to perform such waits without killing performance
- Examples of asynchronous completions
  - Waiting for an I/O operation to complete
  - Waiting for a response to a network request
  - Delaying execution for a fixed period of real time

How Can We Wait?

- Spin locking/busy waiting
- Yield and spin …
- Either spin option may still require mutual exclusion
- Completion events

Spin Waiting For Asynchronous Completions

- Wastes CPU, memory, bus bandwidth
  - Each path through the loop costs instructions
- May actually delay the desired event
  - One of your cores is busy spinning
  - Maybe it could be doing the work required to complete the event instead
  - But it's spinning . . .

Spinning Sometimes Makes Sense

1. When awaited operation proceeds in parallel
   – A hardware device accepts a command
   – Another CPU releases a briefly held spin-lock
2. When awaited operation is guaranteed to be soon
   – Spinning is less expensive than sleep/wakeup
3. When spinning does not delay awaited operation
   – Burning CPU delays running another process
   – Burning memory bandwidth slows I/O
4. When contention is expected to be rare
   – Multiple waiters greatly increase the cost

A Classic "spin-wait"

```
/* set a specified register in the ZZ controller to a specified value   */

zzSetReg( struct zzcontrol *dp, short reg, long value ) {
    while( (dp->zz_status & ZZ_CMD_READY) == 0)
          ;
    dp->zz_value = value;
    dp->zz_reg = reg;
    dp->zz_cmd = ZZ_SET_REG;
}

/* program the ZZ for a specified DMA read or write operation            */

zzStartIO( struct zzcontrol *dp, struct ioreq *bp ) {

    zzSetReg(dp, ZZ_R_ADDR, bp->buffer_start);

    zzSetReg(dp, ZZ_R_LEN, bp->buffer_length);

    zzSetReg(dp, ZZ_R_CMD, bp->write ? ZZ_C_WRITE : ZZ_C_READ );

    zzSetReg(dp, ZZ_R_CTRL, ZZ_INTR + ZZ_GO);

}
```

No guarantee that hardware is ready when this routine returns.

Yield and Spin

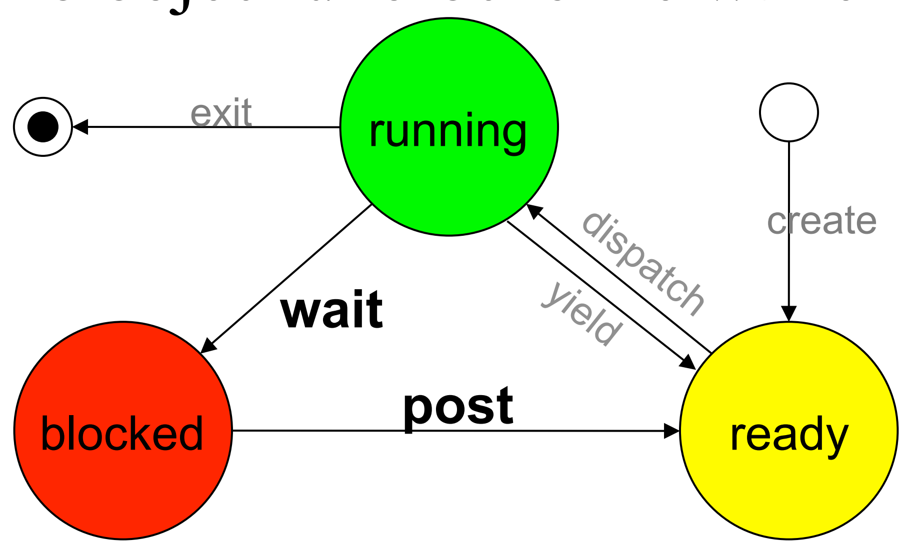
- Check if your event occurred
- Maybe check a few more times
- But then yield
- Sooner or later you get rescheduled
- And then you check again
- Repeat checking and yielding until your event is ready

Problems With Yield and Spin

- Extra context switches
  - Which are expensive
- Still wastes cycles if you spin each time you’re scheduled
- You might not get scheduled to check until long after event occurs
- Works very poorly with multiple waiters

Another Approach: Condition Variables

- Create a synchronization object associated with a resource or request
  - Requester blocks awaiting event on that object
  - Upon completion, the event is "posted"
  - Posting event to object unblocks the waiter



Condition Variables and the OS

- Generally the OS provides condition variables
  - Or library code that implements threads does
- It blocks a process or thread when condition variable is used
  - Moving it out of the ready queue
- It observes when the desired event occurs
- It then unblocks the blocked process or thread
  - Putting it back in the ready queue
  - Possibly preempting the running process

Waiting Lists

- Likely to have threads waiting on several different things
- Pointless to wake up everyone on every event
  - Each should wake up when <u>his</u> event happens
- Suggests all events need a waiting list
  - When posting an event, look up who to awaken
    - Wake up everyone on the list?
    - One-at-a-time in FIFO order?
    - One-at-a-time in priority order (possible starvation)?
  - Choice depends on event and application

Who To Wake Up?

- Who wakes up when a condition variable is signaled?
  - pthread_cond_wait … at least one blocked thread
  - pthread_cond_broadcast … all blocked threads
- The broadcast approach may be wasteful
  - If the event can only be consumed once
  - Potentially unbounded waiting times
- A waiting queue would solve these problems
  - Each post wakes up the first client on the queue

Evaluating Waiting List Options

- Effectiveness/Correctness
  - Should be very good
- Progress
  - There is a trade-off involving *cutting* in line
- Fairness
  - Should be very good
- Performance
  - Should be very efficient
  - Depends on frequency of spurious wakeups

Locking and Waiting Lists

- Spinning for a lock is usually a bad thing
  - Locks should probably have waiting lists
- A waiting list is a (shared) data structure
  - Implementation will likely have critical sections
  - Which may need to be protected by a lock
- This seems to be a circular dependency
  - Locks have waiting lists
  - Which must be protected by locks
  - What if we must wait for the waiting list lock?

A Possible Problem

- The sleep/wakeup race condition

Consider this sleep code:

```
void sleep( eventp *e ) {
  while(e->posted == FALSE) {
      add_to_queue( &e->queue,
      myproc );
      myproc->runstate |= BLOCKED;
      yield();
  }
}
```

And this wakeup code:

```
void wakeup( eventp *e) {
      struct proce *p;

      e->posted = TRUE;
      p = get_from_queue(&e->
queue);
      if (p) {
         p->runstate &= ~BLOCKED;
         resched();
      }  /* if !p, nobody's
waiting */
}
```

What's the problem with this?

A Sleep/Wakeup Race

- Let's say thread B is using a resource and thread A needs to get it
- So thread A will call `sleep()`
- Meanwhile, thread B finishes using the resource
  - So thread B will call `wakeup()`
- No other threads are waiting for the resource

The Race At Work

Thread A

```
void sleep( eventp *e ) {
 while(e->posted == FALSE) {
```

CONTEXT SWITCH!

Nope, nobody's in the queue!

CONTEXT SWITCH!

```
   add_to_queue( &e->queue, myproc );
   myproc->runsate |= BLOCKED;
   yield();
  }
 }
```

Thread B

Yep, somebody's locked it!

```
void wakeup( eventp *e) {
 struct proce *p;
 e->posted = TRUE;
 p = get_from_queue(&e-> queue);
 if (p) {
  }  /* if !p, nobody's waiting */
 }
```

The effect?

Thread A is sleeping   But there's no one to wake him up

Solving the Problem

- There is clearly a critical section in `sleep()`
  - Starting before we test the posted flag
  - Ending after we put ourselves on the notify list
- During this section, we need to prevent
  - Wakeups of the event
  - Other people waiting on the event
- This is a mutual-exclusion problem
  - Fortunately, we already know how to solve those

Progress vs. Fairness

- Consider …
  - P1: lock(), park()
  - P2: unlock(), unpark()
  - P3: lock() (*before P2's unpark()*)
- Progress says:
  - It is available, so P3 gets it
  - Spurious wakeup of P1
- Fairness says:
  - FIFO, P3 gets in line
  - And a convoy forms

```
void lock(lock_t *m) {
    while(true) {
        while (TestAndSet(&m->guard, 1) == 1);
        if (!m->locked) {
            m->locked = 1;
            m->guard = 0;
            return;
        }
        queue_add(m->q, me);
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    m->locked = 0;
    if (!queue_empty(m->q))
        unpark(queue_remove(m->q);
    m->guard = 0;
}
```