CS 3204 Operating Systems

Lecture 10

Godmar Back

Announcements

- Project 1 Feb 20 (Tuesday) 11:59pm
  - Additional office hours: Xiaomo Th 4-6pm, Jai Fr 10am-12pm, Mo 12pm-2pm
- Project 0 graded, has it been handed back already?
  - If not, will be later today
  - Read feedback before submitting project 1

Concurrency & Synchronization

continued

How many locks should I use?

- Could use one lock for all shared variables
  - Disadvantage: if a thread holding the lock blocks, no other thread can access *any* shared variable, even unrelated ones
  - Sometimes used when retrofitting non-threaded code into threaded framework
  - Examples:
    - "BKL" Big Kernel Lock in Linux
    - fslock in Pintos Project 2
- Ideally, want fine-grained locking
  - One lock only protects one (or a small set of) variables – how to pick that set?

Multiple locks, correct (2)

```
static struct li
static struct li

static struct lo
static struct lo
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_release(&freelock);
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&usedlock);
    return b->data;
}
```

```
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_release(&usedlock);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
}
```

Correct, but not necessarily better!
On uniprocessor:
No throughput from fine-grained locking, since no blocking inside critical sections – but pay twice the price compared to one-lock solution
On multiprocessor:
Gain from being able to manipulate free & used lists in parallel, but increased risk of contended locks

---

Conclusion

- Choosing which lock should protect which shared variable(s) is not easy – must weigh:
  - Whether all variables are always accessed together (use one lock if so)
  - Whether code inside critical section can block (if not, no throughput gain from fine-grained locking on uniprocessor)
  - Whether there is a consistency requirement if multiple variables are accessed in related sequence (must hold single lock if so)
    - See "Subtle race condition in Java" below
  - Cost of multiple calls to lock/unlock (increasing parallelism advantages may be offset by those costs)

Rules for Easy Locking

- Every shared variable must be protected by a lock
  - One lock may protect more than one variable, but not too many
  - Acquire lock before touching (reading or writing) variable
  - Release when done, on all paths
- If manipulating multiple variables, acquire locks protecting each
  - Acquire locks always in same order (doesn't matter which order, but must be same)
  - Release in opposite order
  - Don't mix acquires & release (two-phase locking)

Locks in Java/C#

```
synchronized void method() {
    code;
    synchronized (obj) {
        more code;
    }
    even more code;
}
```

is transformed to →

```
void method() {
  try {
    lock(this);
    code;
    try {
      lock(obj);
      more code;
    } finally { unlock(obj); }
    even more code;
  } finally { unlock(this); }
}
```

- Every object can function as lock – no need to declare & initialize them!
- `synchronized` (locked in C#) brackets code in lock/unlock pairs – either entire method or block {}
- `finally` clause ensures unlock() is always called

Subtle Race Condition

```
public synchronized StringBuffer append(StringBuffer sb) {
  int len = sb.length();          // note: StringBuffer.length() is synchronized
  int newcount = count + len;
  if (newcount > value.length)     Not holding lock on 'sb' – other
    expandCapacity(newcount);      Thread may change its length
  sb.getChars(0, len, value, count); // StringBuffer.getChars() is synchronized
  count = newcount;
  return this;
}
```

- Race condition even though individual accesses to "sb" are synchronized (protected by a lock)
  - But "len" may no longer be equal to "sb.length" in call to getChars()
- This means simply slapping lock()/unlock() around every access to a shared variable does not thread-safe code make
- Found by Flanagan/Freund

Concurrency & Synchronization

Higher-level constructs

Infinite Buffer Problem

```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  lock_release(buffer);
}
```

```
consumer()
{
  lock_acquire(buffer);
  while (buffer is empty) {
    lock_release(buffer);
    thread_yield();
    lock_acquire(buffer);
  }
  item = buffer[tail++];
  lock_release(buffer);
  return item
}
```

- Trying to implement infinite buffer problem with locks alone leads to a very inefficient solution (busy waiting!)
- Locks cannot express precedence constraint: A must happen before B.

Infinite Buffer Problem, Take 2

```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
    for c in consumers {
      thread_unblock(c);
    }
  lock_release(buffer);
}
```

```
consumer()
{
  lock_acquire(buffer);
  while (buffer is empty) {
    lock_release(buffer);
    consumers.add(current);
    thread_block(current);
    lock_acquire(buffer);
  }
```

Context switch here would cause *Lost Wakeup* problem: producer will put item in buffer, but won't unblock consumer thread (since consumer thread isn't in consumers yet)

Infinite Buffer Problem, Take 3

```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
     for c in consumers {
       thread_unblock(c);
     }
  lock_release(buffer);
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      consumers.add(current);
      lock_release(buffer);
      thread_block(current);
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
}
```

- What if consumers.add is done before lock is released?

Infinite Buffer Problem, Take 4

```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
     for c in consumers {
       thread_unblock(c);
     }
  lock_release(buffer);
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      consumers.add(current);
      lock_release(buffer);
      thread_block(current);
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
}
```

- This is correct, but complicated and very easy to get wrong
  - Want abstraction that does not require direct block/unblock call

Low-level vs. High-level Synchronization

- Low-level synchronization primitives:
  - Disabling preemption, (Blocking) Locks, Spinlocks
  - implement mutual exclusion
- Implementing precedence constraints directly via thread_unblock/thread_block is problematic because
  - It's complicated (see last slides)
  - It may violate encapsulation from a software engineering perspective
  - You may not have that access at all (unprivileged code!)
- We need well-understood higher-level constructs
  - Semaphores
  - Monitors

Semaphores

Source: inter.scoutnet.org

- Invented by Edsger Dijkstra in 1965s
- Counter S, initialized to some value, with two operations:
  - P(S) or "down" or "wait" – if counter greater than zero, decrement. Else wait until greater than zero, then decrement
  - V(S) or "up" or "signal" – increment counter, wake up any threads stuck in P.
- Semaphores don't go negative:
  - #V + InitialValue - #P >= 0
- Note: direct access to counter value after initialization is not allowed
- Counting vs Binary Semaphores
  - Binary: counter can only be 0 or 1
- Simple to implement, yet powerful
  - Can be used for many synchronization problems

Infinite Buffer w/ Semaphores (1)

```
semaphore items_avail(0);

producer()
{
  lock_acquire(buffer);
  buffer[head++] = item;
  lock_release(buffer);
  sema_up(items_avail);
}
```

```
consumer()
{
   sema_down(items_avail);
   lock_acquire(buffer);
   item = buffer[tail++];
   lock_release(buffer);
   return item;
}
```

- Semaphore "remembers" items put into queue (no updates are lost)

Infinite Buffer w/ Semaphores (2)

```
semaphore items_avail(0);
semaphore buffer_access(1);

producer()
{
  sema_down(buffer_access);
  buffer[head++] = item;
  sema_up(buffer_access);
  sema_up(items_avail);
}
```

```
consumer()
{
   sema_down(items_avail);
   sema_down(buffer_access);
   item = buffer[tail++];
   sema_up(buffer_access);
   return item;
}
```

- Can use semaphore instead of lock to protect buffer access

Bounded Buffer w/ Semaphores

```
semaphore items_avail(0);
semaphore buffer_access(1);
semaphore slots_avail(CAPACITY);
producer()
{
  sema_down(slots_avail);
  sema_down(buffer_access);
  buffer[head++] = item;
  sema_up(buffer_access);
  sema_up(items_avail);
}
```

```
consumer()
{
   sema_down(items_avail);
   sema_down(buffer_access);
   item = buffer[tail++];
   sema_up(buffer_access);
   sema_up(slots_avail);
   return item;
}
```

- Semaphores allow for scheduling of resources

Rendezvous

- A needs to be sure B has advanced to point L, B needs to be sure A has advanced to L

```
semaphore A_madeit(0);

A_rendezvous_with_B()
{
  sema_up(A_madeit);
  sema_down(B_madeit);
}
```

```
semaphore B_madeit(0);

B_rendezvous_with_A()
{
  sema_up(B_madeit);
  sema_down(A_madeit);
}
```


Waiting for an activity to finish

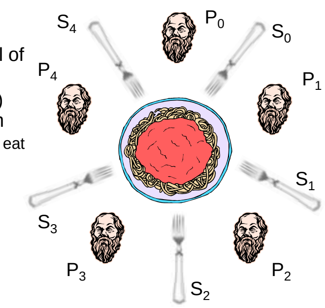
```
semaphore done_with_task(0);
thread_create(
      do_task,
      (void*)&done_with_task);

sema_down(done_with_task);
// safely access task's results
```

```
void
do_task(void *arg)
{
   semaphore *s = arg;
   /* do the task */
   sema_up(*s);
}
```

- Works no matter which thread is scheduled first after thread_create (parent or child)
- Elegant solution that avoids the need to share a "have done task" flag between parent & child
- Two applications of this technique in Pintos Project 2
  - signal successful process startup ("exec") to parent
  - signal process completion ("exit") to parent

Dining Philosophers (Dijkstra)

- A classic
- 5 Philosophers, 1 bowl of spaghetti
- Philosophers (threads) think & eat ad infinitum
  - Need left & right fork to eat (!?)
- Want solution that prevents starvation & does not delay hungry philosophers unnecessarily



Dining Philosophers (1)

```
semaphore fork[0..4](1);
philosopher(int i)                    // i is 0..4
{
  while (true) {
     /* think … finally */
     sema_down(fork[i]);              // get left fork
     sema_down(fork[(i+1)%5]);        // get right fork
      /* eat */
      sema_up(fork[i]);               // put down left fork
      sema_up(fork[(i+1)%5]);         // put down right fork
  }
}
```

- What is the problem with this solution?
- Deadlock if all pick up left fork

Dining Philosophers (2)

```
semaphore fork[0..4](1);
semaphore at_table(4);   // allow at most 4 to fight for forks
philosopher(int i)                    // i is 0..4
{
  while (true) {
     /* think … finally */
     sema_down(at_table);             // sit down at table
     sema_down(fork[i]);              // get left fork
     sema_down(fork[(i+1)%5]);        // get right fork
      /* eat … finally */
      sema_up(fork[i]);               // put down left fork
      sema_up(fork[(i+1)%5]);         // put down right fork
      sema_up(at_table);              // get up
  }
}
```