Lecture Topics: 11/8

- Why synchronization?
  - Atomic operations
  - Interrupts
  - Critical sections
- How to do synchronization
  - Semaphores
  - Monitors

A Quick Motivating Example

```
void Queue::InsertAtHead(Item *item) {
    item->next = head;
    head = item;
}
```

- Suppose two threads, red and blue, share this code and a Queue q.
- The two threads both operate on q: each calls `q->InsertAtHead()`.
- Their execution is *interleaved* by interrupts.

Queue Example

- Now suppose that an interrupt occurs at an "inconvenient" time, so that the actual execution order is like so:

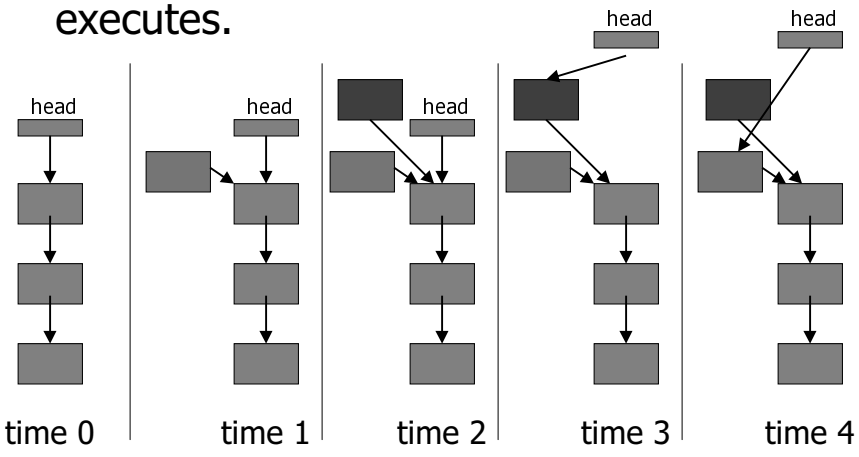
```
1  item->next = head;
2  item->next = head;
3  head = item;
4  head = item;
```

interrupt; switch red to blue

interrupt; switch blue to red

Disaster Strikes

- Let's watch what happens as this code executes.



Unsynchronized Data Access

- This type of problem is called a *race condition*
  - The outcome of the execution varies according to when interrupts occur
  - Which thread gets its way depends on the outcome of a "race"
- The solution is to use *synchronization*: have threads communicate with each other to keep them from tripping each other up

Characterizing the Problem

- The issue in the queue example was that the queue is temporarily *inconsistent*
- If an interrupt occurs during the time when the queue is inconsistent, then bad things can happen
- Is there any kind of data or data structure not susceptible to inconsistency?

Atomic Operations

- Interrupts always occur between instructions
  - If an interrupt occurs while an instruction is partly through the pipeline, the partially computed instruction is *flushed*
- Therefore, any operation that can be executed using only one instruction is safe from interrupts
  - Such operations are called *atomic*

Read/Modify/Write Not Atomic

- All of the instructions we learned in part one of the course (add, sub, lw, sw, etc.) are atomic
- But remember that MIPS and many other architectures are *load-store*
- No memory-memory operation is atomic!

Counter Example

```
int counter=0;
main() {
  thread* t1 = thread_fork(count);
  thread* t2 = thread_fork(count);
}

void count() {
  int i;
  for(i=0; i<1000; i++)
    counter++;
}
```

Two Kinds of Code

- Sometimes threads are operating on shared data
  - This is what we're worried about
  - Only one thread can operate on the data at a time
- But sometimes they're off "doing their own thing"
  - When threads are *not* accessing shared data, they should be allowed freedom

Critical Sections

- We need to identify the code that manipulates the shared data
- That code is called the *critical section*
- Only one thread can be in the critical section at the same time
- Use special *entry* and *exit* procedures to get in and out of the critical section
- With this in mind, let's fix the counter example

Critical Section Solutions

- Three essential qualities of a CS solution:
  - Mutual Exclusion
    - Only one thread in the critical section at a time
  - Progress
    - If no one is in the CS, and A and B want to get in, then only A and B can participate in the decision for who's next, and the decision must happen in a timely fashion
  - Bounded Waiting
    - If A wants to get in, B should not get arbitrarily many turns before A gets a turn

Implementing Synchronization

- There are lots of ways to do this:
  - Turn off interrupts
  - Bakery algorithm
  - Spinlock
  - Semaphores
  - Conditional Critical Regions
  - Condition Variables
  - Monitors

Turning off Interrupts

- The problem is that interrupts may occur while a data structure is inconsistent
- Solution:
  - Entry procedure = turn off interrupts
  - Exit procedure = turn on interrupts
- What happens to interrupts that occur during the critical section?
  - maybe deferred, maybe dropped

Problems with Disabling Interrupts

- The system clock depends on interrupts, and it may drift
- Doesn't work well on a multiprocessor, since time to disable and enable on all processors is high
- Should user code be allowed to do this, even via a system call?
- What about nested critical sections?

Spinlocks

- Suppose you have an instruction `tsl`, test and set lock
  - Takes an address as an argument
  - Checks the value stored at that address
    - 0 means critical section is empty
    - 1 means critical section is full
    - if value is 0, set it to 1 and return 0
    - if value is 1, leave it at 1 and return 1
- Both the *test* and the *set* happen together, atomically

Entry and Exit Using Spinlocks

- Now we can write the entry procedure:

```
while(test_and_set(&lock));
```

- And the exit procedure:

```
lock = 0;
```

```
getlock:  tsl  $t0, lock
          bne  $t0, $zero, getlock
          # critical section here
          sw   $zero, lock
```

What's Wrong with Spinlocks?

- What if you don't have a special test and set instruction?
  - Sometimes there's something equivalent
    - compare-and-swap is common
  - You can disable interrupts, just long enough to do the test and set, rather than for the entire critical section
- Think about processor efficiency
  - Can (sort of) fix this problem with yield()

Semaphores

- Define a *semaphore* to guard the critical section
- The semaphore has two operations:
  - P() is the entry procedure
  - V() is the exit procedure
- Who came up with those names?
  - They're from the Dutch for *probieren* (to try) and *verhogen* (to increment)
  - Thanks, Dijkstra

How Semaphores Work

- The semaphore contains a number, s, which starts at 1
- When s is 1, the critical section is empty; when s is 0, the CS is full

```
sem::P() {
  while(s <= 0);
  s--;
}
```

```
sem::V() {
  s++;
}
```

- Test and decrement in P() are atomic; increment in V() is atomic

Semaphores vs. Spinlocks

- This version of semaphore is only a thin layer above a spinlock
  - P() is pretty much "test and unset"
- We still have the *busy waiting* problem
- Let's try again
  - This time, give each semaphore a *wait queue*
  - Threads trying to get into the critical section can wait on the queue

Semaphores version 2

```
Semaphore::P() {
  s--;
  if(s < 0) {
    waitQ->Enqueue(currentThread);
    currentThread->stop();
  }
}

Semaphore::V() {
  s++;
  if(s <= 0) {
    nextThread = waitQ->Dequeue();
    start(nextThread);
  }
}
```

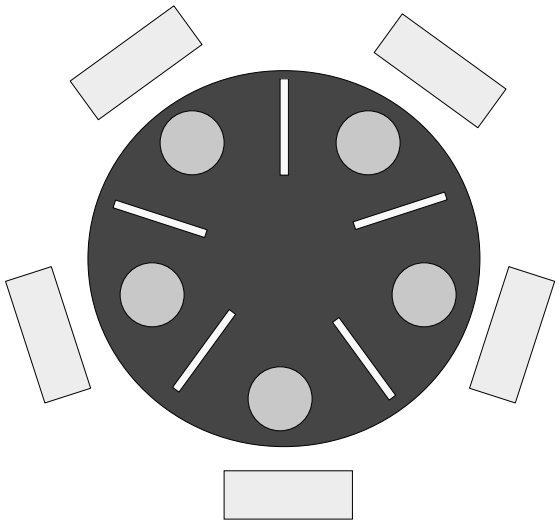
Building up to Semaphores

- Many of the operations in the P() and V() implementations still need to occur atomically with each other
  - We can disable interrupts or use spinlocks to acheive that
  - It's OK to use these primitives in very limited doses
  - The key here is that they'll definitely never be held for long

The Dining Philosophers

- Five philosophers sit together around a circular table
- Philosophers do two things: they think and they eat.
- Between each pair of philosophers lies a single chopstick.
  - You need a pair of chopsticks to eat
  - If either of your neighbors is eating, you'll have to wait

For the Visual Learners...



Implementing the Philosophers

- One solution:
  - Make eating the critical section, like so:

```
philosopher(int i) {
  while(1) {
    sem->P();
    eat();
    sem->V();
    think();
}
```

- What's wrong here?

Trying Again

- This time, make one semaphore per chopstick

```
semaphore sem[5];
void philosopher(int i) {
  while(1) {
    sem[i]->P();
    sem[((i+1)%5)]->P();
    eat();
    sem[i]->V();
    sem[((i+1)%5)]->V();
    think();
  }
}
```

Deadlock

- Suppose each philosopher picks up the left chopstick
- Each philosopher waits for the right chopstick
- No one ever backs down, no one ever makes progress
- This is called *deadlock*; more on this later

Starvation

- Suppose we fix the deadlock problem by giving Philosopher 0 first crack at the chopsticks, then Philosopher 1, etc.
- Philosopher 2 must give up the chopstick if philosopher 1 wants to eat
  - This is actually tricky with semaphores
- Now our solution is *starvation*-prone: there is no guarantee that Philosopher 4 ever gets to eat

Who Cares About Philosophers?

- This problem has important systems implications. For example:
  - Suppose the chopsticks are really queues
  - Suppose the philosophers are really threads trying to move objects between queues
  - A philosopher must lock both the source and destination queue (and no other queues)
- We'll see a good solution later

Readers and Writers

- You have a database full of records
- Many threads may read the same record at the same time
- If any thread is writing the record, then no other thread may read or write
  - when a reader enters, it must block if there is a writer inside
  - when a writer enters, it must block if there is anyone inside

Implementing Readers and Writers

```
reader() {
  lockSem->P();
  readers++;
  if(readers == 1)
    writeSem->P();
  lockSem->V();
  read();
  lockSem->P();
  readers--;
  if(readers == 0)
    writeSem->V();
  lockSem->V();
}
```

```
writer() {
  writeSem->P();
  write();
  writeSem->V();
}
```

In this version, readers are never kept waiting

Writers may starve

Semaphore Conclusions

- Semaphores are nice because they're a little higher level than, e.g., spinlocks
  - no busy waiting
- Semaphores are used often in practice
- However, they're tricky to get right
  - Deadlock and starvation prone solutions are common even for experienced coders
- No built-in support for checking correctness; programmer's responsibility

Monitors

- Implemented within the programming language
- Think of a monitor as a special kind of C++ class
  - Contains code and data like a regular class
  - All of the data is private, so you have to use the monitor code to access it
  - Only one thread is allowed to run code in the monitor at the same time. If someone else is already in, block until they leave.

Entry Procedures

- Some procedures are special *entry procedures*
  - threads can enter the monitor by calling one of these procedures
  - think of these as public
- Other procedures are for internal use only
  - only threads already inside the monitor may call them

Conditions

- What we have so far not quite enough
- In addition, there's a new variable type called a *condition*
- Two operations on a condition:
  - *Wait*, which suspends the calling thread
  - *Signal*, which resumes zero or one threads waiting for the condition
    - If no one is waiting, the signal is lost forever
    - Different from V(), which always had an effect

Dining Philosopher Monitor

```
Monitor Philosopher {
  int state[5];
  condition self[5];

  void entry Pickup(me) {
    state[me] = HUNGRY;
    test(me);
    if(state[me] != EATING)
      wait(self[me]);
  }

  void entry Putdown(me) {
    state[me] = THINKING;
    test(left);
    test(right);
  }
```

```
void test(me) {
  if((state[left] != EATING)
  && (state[me] == HUNGRY) &&
  (state[right] != EATING)) {
    state[me] = EATING;
    signal(self[me]);
}

  Philosopher() {
    state[me] = THINKING;
  }
};
```

Monitor Conclusions

- Still not totally straightforward
  - Synchronization is just hard
- Java uses something akin to monitors
  - You don't necessarily protect the whole class
  - Sometimes you just mark a critical section, as with semaphores
- Monitors not widely used because popular languages don't have them