CS 3204
Operating Systems

Lecture 14

Godmar Back

Announcements

- Project 2 due Tuesday Oct 17, 11:59pm
- Midterm Oct 12
  - Posted Sample Midterm
- Reading assignment:
  - Read Chapter 5 & 19.

Schedule

- Multiprogramming Basics (today)
- Sep 28 Thursday: Scheduling part 1
- Oct 3 Tuesday: (out of town) Guest lecture on real-time scheduling
- Oct 5 Thursday + Oct 10 Tuesday:
  - Wrap-up Scheduling, Monitors, & Deadlock
- Oct 10 Tuesday:
  - Deadlock
- Oct 12: (out of town) Midterm

Scheduling

Basic Scheduling: Summary

- FCFS: simple
  - unfair to short jobs & poor I/O performance (convoy effect)
- RR: helps short jobs
  - loses when jobs are equal length
- SPN: optimal average waiting time
  - which, if ignoring blocking time, leads to optimal average completion time
  - unfair to long jobs
  - requires knowing (or guessing) the future
- MLFQS: approximates SPN without knowing execution time
  - Can still be unfair to long jobs

Proportional Share Scheduling

- Aka “Fair-Share” Scheduling
- None of algorithms discussed so far give direct way of assigning CPU shares
  - E.g., give 30% of CPU to process A, 70% to process B
- Proportional Share algorithms assign “tickets” or “shares” to processes
  - Process get to use resource in proportion of their shares to total number of shares
- Lottery Scheduling, Stride Scheduling [Waldspurger 1995]

Lottery Scheduling

- Idea: number tickets between 1…N
  - every process gets p_i tickets according to importance
  - process 1 gets tickets [1… p_1-1]
  - process 2 gets tickets [p_1… p_2-1] and so on.
- Scheduling decision:
  - Hold a lottery and draw ticket, holder gets to run for next timeslice
- Nondeterministic algorithm
- Q.: what's the complexity of this algorithm?
- Q.: what if a process is blocked?
- Q.: how to implement priority donation?

Scheduling Summary

- OS must schedule all resources in a system
  - CPU, Disk, Network, etc.
- CPU Scheduling affects indirectly scheduling of other devices
- Goals:
  - Minimizing latency
  - Maximizing throughput
  - Provide fairness
- In Practice: some theory, lots of tweaking

Concurrency & Synchronization

Continued

Recap: Synchronization

- Covered:
  - Critical Section Problem
  - Implementation uniprocessor vs multiprocessor
  - Using locks to express mutual exclusion constraint
- Now:
  - Higher-level synchronization constructs that can express precedence constraints
  - Have already discussed semaphores

Monitors

- A monitor combines a set of shared variables & operations to access them
  - Think of an enhanced C++ class with no public fields
- A monitor provides implicit synchronization (only one thread can access private variables simultaneously)
  - Single lock is used to ensure all code associated with monitor is within critical section
- A monitor provides a general signaling facility
  - Wait/Signal pattern (similar to, but different from semaphores)
  - May declare & maintain multiple signaling queues

Monitors (cont'd)

- Classic monitors are embedded in programming language
  - Invented by Hoare & Brinch-Hansen 1972/73
  - First used in Mesa/Cedar System @ Xerox PARC 1978
  - Limited version available in Java/C#
- (Classic) Monitors are safer than semaphores
  - can't forget to lock data – compiler checks this
- In contemporary C, monitors are a *synchronization pattern* that is achieved using locks & condition variables
  - Must understand monitor abstraction to use it

Infinite Buffer w/ Monitor

```
monitor buffer {
    /* implied: struct lock mlock;*/
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}
```

```
buffer::produce(item i)
{  /* try { lock_acquire(&mlock); */
   buffer[head++] = i;
   /* } finally {lock_release(&mlock);} */
}

buffer::consume()
{  /* try { lock_acquire(&mlock); */
   return buffer[tail++];
   /* } finally {lock_release(&mlock);} */
}
```

- Monitors provide implicit protection for their internal variables
  - Still need to add the signaling part

Condition Variables

- Variables used by a monitor for signaling a condition
  - a general (programmer-defined) condition, not just integer increment as with semaphores
- Monitor can have more than one condition variable
- Three operations:
  - Wait(): leave monitor, wait for condition to be signaled, reenter monitor
  - Signal(): signal one thread waiting on condition
  - Broadcast(): signal all threads waiting on condition

Bounded Buffer w/ Monitor

```
monitor buffer {
    condition items_avail;
    condition slots_avail;
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}
```

```
buffer::produce(item i)
{
  while ((tail+1–head)%CAPACITY==0)
     slots_avail.wait();
  buffer[head++] = i;
  items_avail.signal();
}
buffer::consume()
{
   while (head == tail)
     items_avail.wait();
   item i = buffer[tail++];
   slots_avail.signal();
   return i;
}
```


Bounded Buffer w/ Monitor

```
monitor buffer {
    condition items_avail;
    condition slots_avail;
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}
```

```
buffer::produce(item i)
{
  while ((tail+1–head)%CAPACITY==0)
     slots_avail.wait();
  buffer[head++] = i;
  items_avail.signal();
}
buffer::consume()
{
   while (head == tail)
     items_avail.wait();
   item i = buffer[tail++];
   slots_avail.signal();
   return i;
}
```

Q1.: How is lost update problem avoided?

Q2.: Why *while()* and not *if()*?

lock_release(&mlock);
block_on(items_avail);
lock_acquire(&mlock);

Implementing Condition Variables

- State is just a queue of waiters:
  - Wait(): adds current thread to (end of queue) & block
  - Signal(): pick one thread from queue & unblock it
    - Hoare-style Monitors: gives lock directly to waiter
    - Mesa-style monitors (C, Pintos, Java): signaler keeps lock – waiter gets READY, but can't enter until signaler gives up lock
  - Broadcast(): unblock all threads
- Compare to semaphores:
  - Condition variable signals are lost if nobody's on the queue (semaphore's V() are remembered)
  - Condition variable wait() always blocks (semaphore's P() may or may not block)

Monitors in C

- POSIX Threads & Pintos
- No compiler support, must do it manually
  - must declare locks & condition vars
  - must call lock_acquire/lock_release when entering&leaving the monitor
  - must use cond_wait/cond_signal to wait for/signal condition
- Note: cond_wait(&c, &m) takes monitor lock as parameter
  - necessary so monitor can be left & reentered without losing signals
- Pintos cond_signal() takes lock as well
  - only as debugging help/assertion to check lock is held when signaling
  - pthread_cond_signal() does not

Mesa vs Hoare Style

- Mesa-style:
  - Cond_signal leaves signaling thread in monitor
  - so must always use “while()” when checking loop condition
  - POSIX Threads & Pintos are Mesa-style (and so are C# & Java)
- Alternative is “Hoare”-style where cond_signal leads to exit from monitor and immediate reentry of waiter
  - Not commonly used

Monitors in Java

- synchronized *block* means
  - enter monitor
  - *execute block*
  - leave monitor
- wait()/notify() use condition variable associated with receiver
  - Every object in Java can function as a condition var

```
class buffer {
  private char buffer[];
  private int head, tail;
  public synchronized produce(item i) {
    while (buffer_full())
        this.wait();
    buffer[head++] = i;
    this.notify();
  }
  public synchronized item consume() {
    while (buffer_empty())
        this.wait();
    buffer[tail++] = i;
    this.notify();
  }
}
```


Per Brinch Hansen’s Criticism

- See *Java’s Insecure Parallelism* [Brinch Hansen 1999]
- Says Java abused concept of monitors because Java does not *require* all accesses to shared variables to be within monitors
- Why did designers of Java not follow his lead?
  - Performance: compiler can’t easily decide if object is local or not - conservatively, would have to make all public methods synchronized – pay at least cost of atomic instruction on entering every time

Readers/Writer w/ Monitor

```
struct lock mlock; // protects rdrs & wrtrs
int readers = 0, writers = 0;
struct condvar canread, canwrite;
void read_lock_acquire() {
  lock_acquire(&mlock);
  while (writers > 0)
    cond_wait(&canread, &mlock);
  readers++;
  lock_release(&mlock);
}
void read_lock_release() {
  lock_acquire(&mlock);
  if (--readers == 0)
    cond_signal(&canwrite, &mlock);
  lock_release(&mlock);
}
```

```
void write_lock_acquire() {
  lock_acquire(&mlock);
  while (readers > 0 || writers > 0)
    cond_wait(&canwrite, &mlock);
  writers++;
  lock_release(&mlock);
}

void write_lock_release() {
  lock_acquire(&mlock);
  writers--;
  ASSERT(writers == 0);
  cond_signal(&canread, &mlock);
  cond_signal(&canwrite, &mlock);
  lock_release(&mlock);
}
```

Q.: does this implementation prevent starvation?

Summary

- Semaphores & Monitors are both higher-level constructs
- Monitors in C is just a programming pattern that involves mutex+condition variables
- When should you use which?