CS 3204
Operating Systems

Lecture 10

Godmar Back

Announcements

- Project 1 due on Sep 29, 11:59pm
  - Additional GTA office hours
    - Peter: Fri 9-11am
- Midterm Oct 2
  - Will cover content up until including Tuesday's lecture
- Project 2 Help Session TBA
- Reading:
  - Read carefully 1.5, 3.1-3.3, 6.1-6.4

Project 1 Suggested Timeline

- By now, you have:
  - Have read relevant project documentation, set up CVS, built and run your first kernel, designed your data structures for alarm clock
- And should be finishing:
  - Alarm clock by Sep 16
- Pass all basic priority tests by Sep 18
- Priority Inheritance & Advanced Scheduler will take the most time to implement & debug, start them in parallel
  - Should have design for priority inheritance figured out by Sep 23
  - Develop & test fixed-point layer independently by Sep 23
- Due date Sep 29

Concurrency & Synchronization

Monitors

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 languages
  - Invented by Hoare & Brinch-Hansen 1972/73
  - First used in Mesa/Cedar System @ Xerox PARC 1978
  - Adapted 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
  - The actual condition is typically some boolean predicate of monitor variables, e.g. "buffer.size > 0"
- 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();
}
```

Q1.: How is lost update problem avoided?

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

```
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;
}
```

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


Recall: Infinite Buffer Problem, Take 5

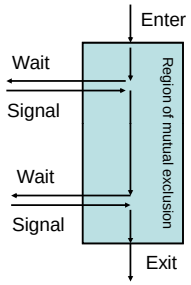
```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
    thread_unblock(
       consumers.pop()
    );
  lock_release(buffer);
}
```

- Aside: this solution to the infinite buffer problem essentially reinvented monitors!

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


Implementing Condition Variables

- A condition variable's state is just a queue of waiters:
  - Wait(): adds current thread to (end of queue) & block
  - Signal(): pick one thread from queue & unblock it
  - Broadcast(): unblock all threads



Mesa vs Hoare Style

- Mesa-style monitors signaler keeps lock
  - cond_signal keeps lock, so it leaves signaling thread in monitor
  - waiter is made READY, but can't enter until signaler gives up lock
  - There is no guarantee whether signaled thread will enter monitor next (or some other thread) - so must always use "while()" when checking condition – cannot assume that condition set by signaling thread will still hold when monitor is reentered
  - POSIX Threads & Pintos are Mesa-style (and so are C# & Java)
- Alternative is Hoare-style (after C.A.R. Hoare)
  - cond_signal leads to signaling thread's exit from monitor and immediate reentry of waiter (e.g., monitor lock is passed from signaler to signalee)
  - not commonly used

Condition Variables vs. Semaphores

- Condition Variables
  - Signals are lost if nobody's on the queue (e.g., nothing happens)
  - Wait() always blocks
- Semaphores
  - Signals (calls to V() or sema_up()) are remembered even if nobody's current waiting
  - Wait (e.g., P() or sema_down()) 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

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

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_broadcast(&canread, &mlock);
  cond_signal(&canwrite, &mlock);
  lock_release(&mlock);
}
```

Q.: does this implementation prevent starvation?

Summary

- Semaphores & Monitors are both higher-level constructs
- Monitors can be included in a language (Mesa, Java)
  - in C, however, they are just a programming pattern that involves a structured way of using mutex+condition variables
- When should you use which?

Semaphores vs. Monitors

- Semaphores & Monitors are both higher-level constructs
- Use semaphores where updates must be remembered – where # of signals must match # of waits
- Otherwise, use monitors.
- Prefer semaphore if they are applicable

High vs Low Level Synchronization

- The bounded buffer problem (and many others) can be solved with higher-level synchronization primitives
  - semaphores and monitors
- In Pintos kernel, one could also use thread_block/unblock directly
  - this is not always efficiently possible in other concurrent environments
- Q.: when should you use low-level synchronization (a la thread_block/thread_unblock) and when should you prefer higher-level synchronization?
- A.: Except for the simplest scenarios, higher-level synchronization abstractions are always preferable
  - They're well understood; make it possible to reason about code.

Nonblocking Synchronization

Nonblocking Synchronization

- Alternative to locks: instead of serializing access, detect when bad interleaving occurred, retry if so

```
void increment_counter(int *counter) {
  do {
    int oldvalue = *counter;
    int newvalue = oldvalue + 1;
    [ BEGIN ATOMIC COMPARE-AND-SWAP INSTRUCTION ]
    if (*counter == oldvalue) { *counter = newvalue; success = true; }
    else { success = false; }
    [ END CAS ]
  } while (!success);
}
```


Nonblocking Synchronization (2)

- Also referred to a "optimistic concurrency control"
- x86 supports this model via cmpxchg instruction
- Advantages:
  - Less overhead for uncontended locks (can be faster, and need no storage for lock queue)
  - Synchronizes with IRQ handler automatically
  - Can be easier to clean up when killing a thread
  - No priority inversion or deadlock
- Disadvantages
  - Can require lots of retries, leading to wasted CPU cycles
  - Requires careful memory/ownership management – must ensure that memory is not reclaimed while a thread may hold reference to it (this can lead to blocking, indirectly, when exhausting memory – real implementations need to worry about this!)

Aside: Nonblocking Properties

- Different NBS algorithms can be analyzed with respect to their progress guarantees
- Lock-freedom:
  - One thread will eventually make progress
- Wait-freedom guarantee: (strongest)
  - All threads will eventually make progress
- Obstruction-freedom: (weakest)
  - Thread will make progress if it is unobstructed

Recent Developments (1)

- As multi- and manycore architectures become abundant, need for better programming models becomes stronger
  - See "The Landscape of Parallel Computing Research: A View From Berkeley"
- Distinguish programming models along 5 categories (each explicit or implicit):
  - Task identification
  - Task mapping
  - Data distribution
  - Communication Mapping
  - Synchronization

Transactional Memory

- Software (STM) or hardware-based
- Idea:
  - Break computations into pieces called transactions
    - Transaction must have the same "atomicity" semantics as locks
    - NB: not as in a database transaction (no persistence on stable storage!)
  - Don't use locks to prevent bad interleavings, and occur cost of serialization, rather: focus on the results of the computation: "Could they have been obtained in some serial order (i.e., if locks had been used?)" – if so, allow them. Otherwise, undo computations
- Many approaches possible – goal is to relieve the programmer from having to use locks explicitly (and avoid their pitfalls such as forgetting them, potential for deadlock, and potential for low CPU utilization)
  - Challenge is to implement this efficiently and in a manner that integrates with existing languages
  - See [Larus 2007] for a survey of approaches

Closing Thoughts on Concurrency and Synchronization

- Have covered most frequently used concepts and models today:
  - Locks, semaphores, monitors/condition variables
- Have looked at them from both users' and implementers' perspective
  - And considered both correctness and performance perspective
- Will use these concepts in projects 2, 3, and 4 (Pintos is a fully preemptive kernel!)
- Yet overall, have barely scratched the surface
  - Many exciting developments may lie ahead in coming years