Announcements

• Homework 2 is due Monday.
• PeerWise question & reviews due tomorrow.
• Project 1:
  – Milestone Friday 7/10
  – Deadline Monday 7/13
KQS Cards

• Use real code snippets
  – Execute examples in class.
• Make slides available before class
  – Annotate slides?
• Talk about the project.
• Talk about past influential OS’s and new OS technologies.
PeerWise

• A new thread gets its own __________:
  – heap
  – data segment
  – text segment
  – address space
  – stack space
Review: Threads

Address Space

Thread 1

Thread 2

Stack (Thread 1)

Stack (Thread 2)

Stack (Thread 3)

Heap

Data Segment

Text Segment

0x00....... (Starting Address)

0xFFF..... (Ending Address)

SP1

SP2

SP3

PC1

PC2

PC3

Thread 3
Review

• Threads
  – Separate execution states within the same process.
  – Preemptive vs. Non-preemptive scheduling

• Intro to Synchronization
  – Mutual exclusion & Critical sections
  – Locks
Goals for Today

• What’s a Semaphore?
  – Binary vs. Counting Semaphores

• What’s a Monitor?

• What’s a Condition Variable?

• When are these mechanisms more efficient than using a Lock?

• Learn enough to do Project 1
Higher-Level Synchronization

• Drawbacks of locks when critical sections are long
  – Spin locks – inefficient
  – Disabling interrupts – can miss or delay important events

• Instead, we want synchronization mechanisms that
  – Block waiters
  – Leave interrupts enabled inside the critical section

• Look at two common high-level mechanisms
  – Semaphores: binary (mutex) and counting
  – Monitors: mutexes and condition variables
Semaphores

• Semaphores are another data structure that provides mutual exclusion to critical sections
  – Block waiters, interrupts enabled within critical section
  – Described by Dijkstra in THE system in 1968
  – Two states: opened or closed

• Semaphores support two operations:
  – wait(semaphore): block until semaphore is open, decrement
    • Also P(), after the Dutch word for test, or down()
  – signal(semaphore): increment, allow another thread to enter
    • Also V() after the Dutch word for increment, or up()

• Semaphores can also be used as atomic counters
  – More on that in a bit
Blocking in Semaphores

• Associated with each semaphore is a queue of waiting processes

• When wait(semaphore) is called by a thread:
  – If semaphore is open, thread continues
  – If semaphore is closed, thread blocks on queue

• Then signal(semaphore) opens the semaphore:
  – If a thread is waiting on the queue, the thread is unblocked
  – If no threads are waiting on the queue, the signal is remembered for the next thread
    • In other words, signal() has “history” (different from condition vars later)
    • This “history” is a counter
### Semaphores Behavior

- **wait(semaphore):**
  - block until semaphore is open
  - decrement
- **signal(semaphore):**
  - Increment
  - allow another thread to enter

- These operations happen atomically.

- **What should we initialize “S” to?**

```c
int S;

wait(S) {
    while ( S <= 0 );
    S--
}

signal(S) {
    S++;
}
```
Semaphore Types

• Semaphores come in two types
  – Determined by semaphore “value” (aka “count”)

• **Mutex** semaphore (count = 1)
  – Represents single access to a resource
  – Guarantees mutual exclusion to a critical section

• **Counting** semaphore (count = N)
  – Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  – N threads can pass the semaphore
Semaphores Illustrated

• Example of counting semaphore for resource control in a railway [joseph, ucb]
  – Semaphore has initial value of 2 (1 for each line)
Using Semaphores

- Use is similar to our locks, but semantics are different

```c
struct Semaphore {
    int value; // start as open
    Queue q;
} S;

withdraw(account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance - amount;
    signal(S);
    printf("balance %f", balance);
    return balance;
}
```

```
wait(S);
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
signal(S);
printf("balance %f", balance)
```
Using Semaphores (2)

```c
struct Semaphore {
    int value;
    Queue q;
} S;

withdraw(account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    signal(S);
    printf("balance %f", balance)
    return balance;
}
```

- It is undefined which thread runs after a `signal()`
Semaphores in Nachos

```java
public Semaphore(int initialValue) {
    value = initialValue;
}

public void P() {
    boolean intStatus = Machine.interrupt().disable();
    if (value == 0) {
        waitQueue.waitForAccess(KThread.currentThread());
        KThread.sleep();
    } else {
        value--;
    }
    Machine.interrupt().restore(intStatus);
}

public void V() {
    boolean intStatus = Machine.interrupt().disable();
    KThread thread = waitQueue.nextThread();
    if (thread != null) {
        thread.ready();
    } else {
        value++;
    }
    Machine.interrupt().restore(intStatus);
}
```

- In nachos/threads/Semaphore.java
- Note that some kernel operations require interrupts to be disabled
  - Can you identify the critical section?
  - How can the thread sleep with interrupts disabled?
Using Semaphores

• We’ve looked at a simple example for using synchronization
  – Mutual exclusion while accessing a bank account
• Now we’re going to use semaphores to look at more interesting examples
  – Readers/Writers Problem
  – Bounded Buffers Problem
Readers/Writers Problem

• Motivation: Consider a shared database
  – Two classes of users:
    • Readers – never modify database
    • Writers – read and modify database
  – Is using a single lock on the whole database sufficient?
    • Like to have many readers at the same time
    • Only one writer at a time

![Diagram showing a database with two writers and multiple readers accessing it]
Readers/Writers Problem (2)

• Readers/Writers Problem:
  – An object is shared among several threads
  – Some threads only read the object, others only write it
  – We can allow multiple readers
  – But only one writer
  – A lock is inefficient

• How can we use semaphores to control access to the object to implement this protocol?

• Use three variables
  – int readcount – number of threads reading object
  – Semaphore mutex – control access to readcount
  – Semaphore w_or_r – exclusive writing or reading
Readers/Writers Problem (3)

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex = 1;

// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock our readers
    Write;
    signal(w_or_r); // up for grabs
}

reader {
    wait(mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/writers
    signal(mutex); // unlock readcount
    Read;
    wait(mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex); // unlock readcount
}
Readers/Writers Problem (4)

• If there is a writer
  – First reader blocks on w_or_r
  – All other readers block on mutex

• Once a writer exits, all readers can fall through
  – Which reader gets to go first?

• The last reader to exit signals a waiting writer
  – If no writer, then readers can continue

• If readers and writers are waiting on w_or_r, and a writer exits, who goes first?

• Why doesn’t a writer need to use a mutex?
Bounded Buffer

• Problem: There is a set of resource buffers shared by producer and consumer threads
• **Producer** inserts resources into the buffer set
  – Output, disk blocks, memory pages, processes, etc.
• **Consumer** removes resources from the buffer set
  – Whatever is generated by the producer
• Producer and consumer execute at different rates
  – No serialization of one behind the other
  – Tasks are independent (easier to think about)
  – The buffer set allows each to run without explicit handoff
Bounder Buffer (2)

• Use three semaphores:
  – **Mutex** – mutual exclusion to shared set of buffers
    • Binary semaphore
  – **Empty** – count of empty buffers
    • Counting semaphore
  – **Full** – count of full buffers
    • Counting semaphore
Bounded Buffer (3)

Semaphore mutex = 1;    // mutual exclusion to shared set of buffers
Semaphore empty = N;    // count of empty buffers (all empty to start)
Semaphore full = 0;      // count of full buffers (none full to start)

**producer** {
    while (1) {
        Produce new resource;
        // wait for empty buffer
        wait(empty);
        // lock buffer list
        wait(mutex);
        Add resource to an empty buffer;
        // unlock buffer list
        signal(mutex);
        // note a full buffer
        signal(full);
    }
}

**consumer** {
    while (1) {
        // wait for a full buffer
        wait(full);
        // lock buffer list
        wait(mutex);
        Remove resource from a full buffer;
        // unlock buffer list
        signal(mutex);
        // note an empty buffer
        signal(empty);
        Consume resource;
    }
}

24
Bounded Buffer (4)

- Why need the mutex at all?
- Where are the critical sections?
- What happens if operations on mutex and full/empty are switched around?
  - The pattern of signal/wait on full/empty is a common construct called an interlock
- **Reader/Writer** and **Bounded Buffer** are classic examples of synchronization problems
Break & KQS Cards

• K: what do you want me to **Keep** doing?
• Q: what do you want me to **Quit** doing?
• S: what do you want me to **Start** doing?

• **Is** there a specific topic that we’ve gone over that is still unclear to you?
Semaphore Summary

• Semaphores can be used to solve any of the traditional synchronization problems

• However, they have some drawbacks
  – They are essentially shared global variables
    • Can potentially be accessed anywhere in program
  – No connection between the semaphore and the data being controlled by the semaphore
  – Used both for critical sections (mutual exclusion) and coordination (scheduling)
    • Note that I had to use comments in the code to distinguish
  – No control or guarantee of proper usage

• Sometimes hard to use and prone to bugs
  – Another approach: Use programming language support
Monitors

• A monitor is a programming language construct that controls access to shared data
  – Synchronization code added by compiler, enforced at runtime
  – Why is this an advantage?

• A monitor is a module that encapsulates
  – Shared data structures
  – Procedures that operate on the shared data structures
  – Synchronization between concurrent threads that invoke the procedures

• A monitor protects its data from unstructured access
• It guarantees that threads accessing its data through its procedures interact only in legitimate ways
Monitor Semantics

• A monitor guarantees mutual exclusion
  – Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
  – If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    • So the monitor has to have a wait queue...
  – If a thread within a monitor blocks, another one can enter

• What are the implications in terms of parallelism in a monitor?
Hey, that was easy

But what if a thread wants to wait inside the monitor?
  – How do we solve the Bounded Buffer Problem with Monitors?
Monitors & Bounded Buffer

Monitor bounded_buffer{
    produce() {
        Produce new resource;
        Add resource to an empty buffer;
    }

    consume() {
        Remove resource from a full buffer;
        Consume resource;
    }
}

• Does this work?
  – Why not?
  – What is missing?
Condition Variables

- **Condition variables** provide a mechanism to wait for events (a “rendezvous point”)
  - Resource available, no more writers, etc.

- **Condition variables** support three operations:
  - **Wait**: release monitor lock, wait for C/V to be signaled
    - So CVs have wait queues too
  - **Signal**: wakeup one waiting thread
  - **Broadcast**: wakeup all waiting threads
Monitor Bounded Buffer

Monitor **bounded_buffer** {
    Resource buffer[N];
    // Variables for indexing buffer
    Condition not_full, not_empty;

    void put_resource(Resource R) {
        while (buffer array is full) {
            wait(not_full);
            Add R to buffer array;
            signal(not_empty);
        }
    }
}

Resource get_resource() {
    while (buffer array is empty) {
        wait(not_empty);
        *Get resource R from buffer array;*
        signal(not_full);
        return R;
    }
} // end monitor
Monitor bounded_buffer {
    Condition not_full;
    …other variables…
    Condition not_empty;

    void put_resource() {
        …wait(not_full)…
        …
        …signal(not_empty)…
    }

    Resource get_resource() {
        …
    }
}
Semaphore Bounded Buffer

Semaphore \texttt{mutex} = 1; \quad \text{\textit{\small // mutual exclusion to shared set of buffers}}
Semaphore \texttt{empty} = N; \quad \text{\textit{\small // count of empty buffers (all empty to start)}}
Semaphore \texttt{full} = 0; \quad \text{\textit{\small // count of full buffers (none full to start)}}

\begin{align*}
\textbf{producer} \{ \\
\quad \textbf{while} (1) \{ \\
\quad\quad \textit{Produce new resource;} \\
\quad\quad \textit{\textit{\small // wait for empty buffer}} \\
\quad\quad \text{wait(\texttt{empty});} \\
\quad\quad \textit{\textit{\small // lock buffer list}} \\
\quad\quad \text{wait(\texttt{mutex});} \\
\quad\quad \textit{\textit{\small Add resource to an empty buffer;}} \\
\quad\quad \textit{\textit{\small // unlock buffer list}} \\
\quad\quad \text{signal(\texttt{mutex});} \\
\quad\quad \textit{\textit{\small // note a full buffer}} \\
\quad\quad \text{signal(\texttt{full});} \\
\quad \} \\
\}
\end{align*}

\begin{align*}
\textbf{consumer} \{ \\
\quad \textbf{while} (1) \{ \\
\quad\quad \textit{\textit{\small // wait for a full buffer}} \\
\quad\quad \text{wait(\texttt{full});} \\
\quad\quad \textit{\textit{\small // lock buffer list}} \\
\quad\quad \text{wait(\texttt{mutex});} \\
\quad\quad \textit{\textit{\small Remove resource from a full buffer;}} \\
\quad\quad \textit{\textit{\small // unlock buffer list}} \\
\quad\quad \text{signal(\texttt{mutex});} \\
\quad\quad \textit{\textit{\small // note an empty buffer}} \\
\quad\quad \text{signal(\texttt{empty});} \\
\quad\quad \textit{\textit{\small Consume resource;}} \\
\quad \} \\
\}
\end{align*}
Monitor vs Semaphore BB

producer {
    while (1) {
        Produce new resource;
        // wait for empty buffer
        wait(empty);
        // lock buffer list
        wait(mutex);
        Add resource to an empty buffer;
        // unlock buffer list
        signal(mutex);
        // note a full buffer
        signal(full);
    }
}

Semaphore Producer

Monitor bounded_buffer {
    Resource buffer[N];
    // Variables for indexing buffer
    Condition not_full, not_empty;

    void put_resource(Resource R) {
        while (buffer array is full)
            wait(not_full);
        Add R to buffer array;
        signal(not_empty);
    }
}

Monitor Producer
Condition Variables and Semaphores

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics (such is life, worse yet to come)
  - However, they each can be used to implement each other
    - See nachos/threads/Condition.java

- Access to the monitor is controlled by a lock
  - `wait()` blocks the calling thread, and gives up the lock
    - To call `wait`, the thread has to be in the monitor (has the lock)
    - `Semaphore::wait` just blocks the thread on the queue
  - `signal()` causes a waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - `Semaphore::signal` increases the semaphore count, allowing future entry even if no thread is waiting
    - Condition variables have no history
Signal Semantics

• There are two flavors of monitors that differ in the scheduling semantics of signal()
  – Hoare monitors (original)
    • signal() immediately switched from the caller to a waiting thread
    • The condition that the waiter was anticipating is guaranteed to hold when waiter executes
    • signaler must restore monitor invariants before signaling
  – Mesa monitors (Mesa, Java)
    • signal() places a waiter on the ready queue, but signaler continues inside monitor
    • condition is not necessarily true when waiter runs again
      – Returning from wait() is only a hint that something changed
      – Must recheck conditional case
Hoare vs Mesa Monitors

• Hoare
  
  if (empty)
    wait(condition)

• Mesa
  
  while(empty)
    wait(condition)

• Tradeoffs
  
  – Mesa monitors easier to use, more efficient
    • Fewer context switches, easy to support broadcast
  
  – Hoare monitors leave less to chance
    • Easier to reason about the program
Condition Variables and Locks

- Condition variables are also used without monitors in conjunction with blocking locks
- A monitor is “just like” a module whose state includes a condition variable and a lock
  - Difference is syntactic; with monitors, compiler adds the code
- It is “just as if” each procedure in the module calls acquire() on entry and release() on exit
  - But can be done anywhere in procedure, at finer granularity
- With condition variables, the module methods may wait and signal on independent conditions
Using Condition Variables and Locks

• Alternation of two threads (ping-pong)
• Each executes the following:

```c
Lock lock;
Condition cond;

void ping() {
    acquire(lock);
    while(1) {
        printf("ping\n");
        signal(cond, lock);
        wait(cond, lock);
    }
    release(lock);
}
```

Must acquire lock before you can wait (similar to needing interrupts disabled to call sleep in Nachos)

Wait atomically releases lock and blocks until signal()

Wait atomically acquires lock before it returns
Using Condition Variables and Locks

```c
void ping() {
    acquire(lock);
    while(1) {
        printf("ping\n");
        signal(cond, lock);
    }
    release(lock);
}

void pong() {
    acquire(lock);
    while(1) {
        printf("pong\n");
        signal(cond, lock);
    }
    release(lock);
}
```

- `acquire(lock);` - Gains lock and returns from acquire()
- `signal(cond, lock);` - Signal puts blue in ready state.
- `wait(cond, lock);` - Blocks and releases lock
- `acquire(lock);` - Blocks on lock
- `while(1) {` - Nobody waiting, signal dropped.
  - `printf("ping\n");` - Signal puts blue in ready state.
  - `signal(cond, lock);` - Blocks and releases lock
  - `wait(cond, lock);` - Blocks on lock
Monitors and Java

• A lock and condition variable are in every Java object
  – No explicit classes for locks or condition variables
• Every object is/has a monitor
  – At most one thread can be inside an object’s monitor
  – A thread enters an object’s monitor by
    • Executing a method declared “synchronized”
      – Can mix synchronized/unsynchronized methods in same class
    • Executing the body of a “synchronized” statement
      – Supports finer-grained locking than an entire procedure
• Every object can be treated as a condition variable
  – Object::notify() has similar semantics as Condition::signal()
Synchronization Summary

• Semaphores
  – `wait()`/`signal()` implement blocking mutual exclusion
  – Also used as atomic counters (counting semaphores)
  – Can be inconvenient to use

• Monitors
  – Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
    • Only one thread can execute within a monitor at a time
  – Relies upon high-level language support

• Condition variables
  – Used by threads as a synchronization point to wait for events
  – Inside monitors, or outside with locks
Next Time

• Read Chapter 5 (Scheduling)
• PeerWise question & reviews due tomorrow.
• Check Web site for course announcements
  – http://www.cs.ucsd.edu/classes/su09/cse120