Announcements

• Midterm coming up (4/30 5-7pm)

• Practice midterm available on Canvas
  ♦ Try it out to test and get familiar with Canvas online exam
  ♦ Use it to prepare for the midterm
  ♦ Need Thur’s lecture for some of the problems
Semaphore

A synchronization variable that takes on non-negative integer values
- Invented by Edsger Dijkstra in the mid 60’s

Two primitive operations
- **wait**(semaphore): an atomic operation that waits for semaphore to become greater than 0, then decrements it by 1
- **signal**(semaphore): an atomic operation that increments semaphore by 1
Two usages of semaphores

• For mutual exclusion:
  ♦ to ensure that only one process is accessing shared info at a time.
  ♦ Semaphores or binary semaphores?

• For condition synchronization:
  ♦ to permit processes to wait for certain things to happen
  ♦ Semaphores or binary semaphores?
[lec6] Producer & Consumer

Problem

- **Producer**: creates copies of a resource
- **Consumer**: uses up (destroys) copies of a resource.
- **Buffers**: fixed size, used to hold resource produced by producer before consumed by consumer

![Diagram of Producer and Consumer problem]

Producer → \( N = 4 \) → Consumer

2 empty slots 2 occupied slots
Readers-Writers problem

- A data object is shared among multiple processes
- Allow concurrent reads (but no writes)
- Only allow exclusive writes (no other writes or reads)
Producer & Consumer – semaphore, counting is tricky

Init: FULL = 0; EMPTY = N; Mutex = 1;

Producer
while (1) {
    produce an item;
    wait(EMPTY);
    acq(lock); 
    insert(item to pool);
    rel(lock);
    signal(FULL);
}

Consumer
While (1) {
    wait(FULL);
    acq(lock);
    remove(item from pool);
    rel(lock);
    signal(EMPTY);
    consume the item;
}
Producer & Consumer -- is there something simpler than semaphore?

**Producer**

```c
while (1) {
    produce an item;
    acquire(mutex);
    if (pool is Full) {
        wait(NotFULL);
    }
    record if pool was empty;
    insert(item)
    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}```

**Consumer**

```c
While (1) {
    acquire(mutex)
    if (pool is Empty) {
        wait(NotEMPTY)
    }
    record if pool was full
    remove(item)
    if (pool was Full)
        signal(NotFULL)
    release(mutex)
    consume the item;
}```
Producer & Consumer -- is there something simpler than semaphore?

Producer

while (1) {
    produce an item;
    acquire(mutex);
    if (pool is Full) {
        release(mutex);
        wait(NotFULL);
        acquire(mutex)
    }
    record if pool was empty;
    insert(item)
    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}

Consumer

While (1) {
    acquire(mutex)
    if (pool is Empty) {
        release(mutex)
        wait(NotEMPTY)
        acquire(mutex)
    }
    record if pool was full
    remove(item)
    if (pool was Full)
        signal(NotFULL)
    release(mutex)
    consume the item;
}
Producer & Consumer -- is there something simpler than semaphore?

**Producer**

```c
while (1) {
    produce an item;
    acquire(mutex);
    if (pool is Full) {
        wait(NotFULL);
    }
    record if pool was empty;
    insert(item)

    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}
```

**Consumer**

```c
While (1) {
    acquire(mutex)
    if (pool is Empty {

    } record if pool was full
    remove(item)

    if (pool was Full)
        signal(NotFULL)
    release(mutex)
    consume the item;
}
```

The simplification implies NotFull is tied to mutex
Mutual Exclusion provided by OS or language/compiler

- **Semaphore**
  - Powerful, but kind of low level
  - can we have a high level abstraction?

- **Locks and condition variables**
  - Lock alone is not flexible enough
  - Need some mechanism to sleep and wake up

- **Monitor**
Conditional Variables

• An explicit queue that processes/threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition)
  ◆ Also called wait (Java, C++), sleep (Nachos, C#)

• Some other process/thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition)
  ◆ Wake up one: wake (Nachos, C#), notify (Java), notify_one (C++)
  ◆ Wake up all: wakeAll (Nachos, C#), notifyAll (Java), notify_all (C++)
Conditional Variables

• Used in conjunction with locks
• Used inside critical section to wait for certain conditions
• Contrast with Semaphore:
  ◆ Has no counting bundled
  ◆ More intuitive to many people

• Usage
  ◆ On creation, specify which mutex it is associated with
Conditional Variables

- **Wait (condition)**
  - Block on “condition”

- **Signal (condition)**
  - Wakeup one or more processes blocked on “condition”

- **Conditions are like semaphores but:**
  - signal is no-op if none blocked
  - There is no counting!

![Diagram showing lock L, condition variables x(L) and y(L), queues associated with x, y, and a queue of waiting processes trying to enter CSes protected by lock L.]

- Lock: L
- Condition variables: x(L) y(L)
- Queues associated with x, y, and condition
- Queue of waiting processes trying to enter CSes protected by lock L
- Shared data
- Operations
“Wow, I like condition variables”

- One problem – what happens on wakeup?
  - Only one thing can be inside critical section
  - But wakeup implies both signaler and waiter may be in critical section, who should go on?

```
Scheduler

dispatch

Runnin

CondQ

Blocked

Ready

Create a process

terminate

Wait for resource

```

CSE 120 – Lecture 7 – Conditional Variables and Concurrency Bugs

15
Signal Semantics

• signal() places a waiter on the ready queue, but signaler continues inside lock
  ♦ Known as “Mesa” style
  ♦ Easy to implement
  ♦ Another early-time semantics is Hoare style (signaler gives up lock, waiter runs immediately)

• What can happen when the awaken process gets a chance to run?
  ♦ E.g. pool is full, producer 1 waits; consumer signals it; p1 in ready queue; consumer release(lock); p2 comes along…

• Condition not necessarily true when waiter runs again
  ♦ Returning from wait() is only a hint that something changed
  ♦ Must recheck conditional case
Producer & Consumer -- use condition variables – problem?

Producer

while (1) {

produce an item;

acquire(mutex);
if  (pool is Full) {
    release(mutex);
    wait(NotFULL);
    acquire(mutex)
}
record if pool was empty;
insert(item)

if (pool was empty)
    signal(NotEMPTY)
release(mutex)
}

Consumer

While (1) {
    acquire(mutex)
    if (pool is Empty {
        release(mutex)
        wait(NotEMPTY)
        acquire(mutex)
    }
    record if pool was full
    remove(item)
    if (pool was Full)
        signal(NotFULL)
    some other work
    release(mutex)
    consume the item;
}
Producer & Consumer – use condition variables – how to fix?

**Producer**

```c
while (1) {
    produce an item;
    acquire(mutex);
    while (pool is Full) {
        wait(NotFULL);
    }
    record if pool was empty;
    insert(item)
    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}
```

**Consumer**

```c
While (1) {
    acquire(mutex)
    while (pool is Empty { 
        wait(NotEMPTY)
    }
    record if pool was full
    remove(item)
    if (pool was Full)
        signal(NotFULL)
    release(mutex)
    consume the item;
}
```

*Is this busy waiting?*
Be Careful About Pitfalls: CVs Cannot Be “Tested”

- Do not use a CV as a predicate
- Need to use a separate flag
Be Careful About Pitfalls: CVs Require Holding Lock

- Do not release the lock before using the CV
  - Using a CV requires a thread to hold the lock
- Purpose of a CV is to enable threads to block while in a critical section

```java
lock.acquire();
...
lock.release();
cv.wait();
lock.acquire();
...
lock.release();
```

```java
lock.acquire();
...
lock.acquire();
cv.wait();
...
lock.acquire();
```
Be Careful About Pitfalls: Need Lock When Testing Flag

- Testing a flag needs to be done while holding the lock
- It is a shared variable that can lead to race conditions

```java
... if (nonempty) {
    lock.acquire();
    cv.wait();
    lock.release();
} ...
```

```java
lock.acquire();
... if (nonempty) {
    cv.wait();
} ...
lock.release();
```
Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- 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
- If curious, read more in backup slides
Synchronization Primitives

Summary

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

- Condition variables
  - Used by threads as a synchronization point to wait for events
  - Used with locks or inside monitors

- 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
Concurrency Bugs

- **Blocking bugs**
  - Concurrency bugs that cause one or more thread to stuck (cannot make progress)
  - E.g., deadlock

- **Non-blocking bugs**
  - Concurrency bugs that do not block any thread’s execution but results in undesired behavior
  - E.g., data race
Deadlock

• Synchronization is a live gun – we can easily shoot ourselves in the foot
  ♦ Incorrect use of synchronization can block all processes
  ♦ You have likely been intuitively avoiding this situation already
• More generally, processes that allocate multiple resources generate dependencies on those resources
  ♦ Locks, semaphores, monitors, etc., just represent the resources that they protect
• If one process tries to allocate a resource that a second process holds, and vice-versa, they can never make progress
• We call this situation **deadlock**, and we’ll look at:
  ♦ Definition and conditions necessary for deadlock
  ♦ Representation of deadlock conditions
  ♦ Approaches to dealing with deadlock
Deadlock Example
Deadlock Definition

- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized

- Definition:
  - Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.

```
Thread 1
lockA->Acquire();
...
lockB->Acquire();

Thread 2
lockB->Acquire();
...
lockA->Acquire();
```
Deadlock with Join

Thread A

...  
B.join();  
...  

Thread B

...  
A.join();  
...
Deadlock can be described using a resource allocation graph (RAG). The RAG consists of a set of vertices \( P = \{P_1, P_2, \ldots, P_n\} \) of processes/threads and \( R = \{R_1, R_2, \ldots, R_m\} \) of resources.

- A directed edge from a process to a resource, \( P_i \rightarrow R_j \), means that \( P_i \) has requested \( R_j \).
- A directed edge from a resource to a process, \( R_i \rightarrow P_j \), means that \( R_j \) has been allocated by \( P_i \).
- Each resource has a fixed number of units.

If the graph has no cycles, deadlock **cannot exist**.
If the graph has a cycle, deadlock **may exist**.
Resource-Allocation Graph (Cont.)

- Process/thread
- Resource type with 4 instances
- $P_i$ requests instance of $R_j$
- $P_i$ is holding an instance of $R_j$
Resource Allocation Graph – is there a deadlock?
Resource Allocation Graph with a cycle – is there a deadlock?
Resource Allocation Graph with a cycle – is there a deadlock?
Conditions for Deadlock

- Deadlock can exist if and only if the following four conditions hold simultaneously:
  1. **Mutual exclusion** – At least one resource must be held in a non-sharable mode
  2. **Hold and wait** – There must be one process holding one resource and waiting for another resource
  3. **No preemption** – Resources cannot be preempted (critical sections cannot be aborted externally)
  4. **Circular wait** – There must exist a set of processes \([P_1, P_2, P_3, \ldots, P_n]\) such that \(P_1\) is waiting for \(P_2\), \(P_2\) for \(P_3\), etc.

Eliminating *any* condition eliminates deadlock!
Four Possible Strategies to Deal With Deadlocks

1. Ignore the problem
   - It is user’s fault
   - used by most operating systems, including UNIX
2. Detection and recovery (by OS)
   - Fix the problem after occurring
3. Dynamic avoidance (by OS, programmer help)
   - Careful allocation
4. Prevention (by programmer, practically)
   - Negate one of the four conditions
2. Detection and Recovery

- Detection and recovery
  - Allow deadlocks to happen but detect them and recover
- To do this, we need two algorithms
  - One to determine whether a deadlock has occurred
  - Another to recover from the deadlock
2. Deadlock Detection

- Detection
  - Traverse the resource graph looking for cycles
  - If a cycle is found, preempt resource (force a thread to release)

- Expensive
  - Many threads and resources to traverse

- Only invoke detection algorithm depending on
  - How often or likely deadlock is
  - How many threads are likely to be affected when it occurs
2. Deadlock Recovery

Once a deadlock is detected, we have two options...

1. Abort processes
   ♦ Abort all deadlocked threads
     » Threads need to start over again
   ♦ Abort one thread at a time until cycle is eliminated
     » System needs to rerun detection after each abort

2. Preempt resources (force their release)
   ♦ Need to select thread and resource to preempt
   ♦ Need to rollback thread to previous state
   ♦ Need to prevent starvation
3. Deadlock Avoidance

• Avoidance
  ♦ Provide information in advance about what resources will be needed by threads to guarantee that deadlock will not happen
  ♦ System only grants resource requests if it knows that the thread can obtain all resources it needs in future requests
  ♦ Avoids circularities (wait dependencies)

• Tough
  ♦ Hard to determine all resources needed in advance
  ♦ Good theoretical problem, not as practical to use
4. Deadlock Prevention

• Remove any of the four conditions of deadlocks

• Remove mutual exclusion
  ♦ E.g., make resources sharable, not always possible

• Remove hold and wait
  ♦ E.g., try to lock all needed resources at the beginning. If successful, use the resources & release them. Otherwise, release all resources and start over

• Preemption
  ♦ E.g., if a request from a thread holding resources cannot be satisfied, preempt the thread and release all resources

• No circular wait
  ♦ E.g., impose some order of requests for all resources
Deadlock Summary

- Deadlock occurs when processes/threads are waiting on each other and cannot make progress
  - Cycles in Resource Allocation Graph (RAG)
- Deadlock requires four conditions
  - Mutual exclusion, hold and wait, no resource preemption, circular wait
- Four approaches to dealing with deadlock:
  - Ignore it – Living life on the edge
  - Avoidance – Carefully control allocation
  - Detection and Recovery – Look for a cycle, preempt or abort
  - Prevention – Make one of the four conditions impossible
Other Blocking Bugs: Forgetting to Release Lock

```c
void mptctl_simplified(unsigned long arg) {
    mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
    MPT_ADAPTER *iocp = NULL;

    // first fetch
    if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
        return -EFAULT;

    // dependency lookup
    if (mpt_verify_adapter(khdr.iocnum, &iocp) < 0 || iocp == NULL)
        return -EFAULT;

    // dependency usage
    mutex_lock(&iocp->ioctl_cmds.mutex);
    struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;

    // second fetch
    if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
        return -EFAULT;

    mptctl_do_fw_download(kfwdl.iocnum, .......);
    mutex_unlock(&iocp->ioctl_cmds.mutex);
}
```

Fig. 1: A dependency lookup double-fetch bug, adapted from __mptctl_ioctl in file drivers/message/fusion/mptctl.c

actual bug in Linux driver!
Non-Blocking Bugs

• Atomicity-Violation Bugs
  ♦ The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution).
  ♦ Real example in MySQL

Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;

Not Atomic!
Non-Blocking Bugs

• Order-Violation Bugs
  ♦ The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)

Thread 1::
void init() {
  ...
  mThread =
  PR_CreateThread(mMain, ...);
  ...
}

Thread 2::
void mMain(...) {
  ...
  mState = mThread->State;
  ...
}
Next time...

- Read Chapters 7, 8, 32
Monitors

• A monitor is a programming language construct that controls access to shared data
  ♦ Synchronization code added by compiler, enforced at runtime
• 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…
  - Can have a condition variable inside a monitor
Account Example

Monitor `account` {
  double balance;

  double `withdraw(amount)` {
    balance = balance – amount;
    return balance;
  }
}

When first thread exits, another can enter. Which one is undefined.

* Hey, that was easy!
* But what if a thread wants to wait inside the monitor?
  » Such as “mutex(empty)” by reader in bounded buffer?
Monitors, Monitor Invariants and Condition Variables

- A **monitor invariant** is a *safety property* associated with the monitor, expressed over the monitored variables. It holds whenever a thread enters or exits the monitor.
- A **condition variable** is associated with a *condition* needed for a thread to make progress once it is in the monitor.

Monitor M {
  ... monitored variables
  Condition c;

  void enterMonitor (...) {
    if (extra property not true) wait(c);  // waits outside of the monitor's mutex
    do what you have to do
    if (extra property true) signal(c);  // brings in one thread waiting on condition
  }
}
Monitors and Java

• A lock and condition variable are in every Java object
  ♦ Later added 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 method
      ▪ Identical to the Modula-2 “LOCK (m) DO” construct
  ♦ The compiler generates code to acquire the object’s lock at
    the start of the method and release it just before returning
    » The lock itself is implicit, programmers do not worry about it
Monitors and Java

• Every object can be treated as a condition variable
  ♦ Half of Object’s methods are for synchronization!
• Take a look at the Java Object class:
  ♦ Object.wait(*) is wait (Condition.sleep in Nachos)
  ♦ Object.notify() is signal (Condition.wake)
  ♦ Object.notifyAll() is broadcast (Condition.wakeAll)
Modern Languages

• Modern languages provide some form of locks and condition variables for synchronization and coordination
  ♦ C, C++, C#, Java, Go, Rust, …
  ♦ Most common form of synchronization you will encounter

• Typically locks are explicit
  ♦ Programmers have to use acquire and release explicitly
    » C++ and Rust have “release on return” language semantics
    » A half-way monitor implementation…
  ♦ Even Java eventually added separate classes (Lock, Condition) for flexibility