CSC369: Operating Systems

Fall 2014

Andrew Petersen

Anonymous ... Prayers?

In moments like these I would hope that many of us do not stray from our daily prayers. Lest we forget..

The Processes Creed:
I believe in Assembly the Father almighty.
I also believe in Andrew his only son, our Lord.
Conceived of the CPU and born of the BIOS.
Who was crucified under stack smashing, locked, starved and freed.
Descended into the kernel, on the third day rose a zombie process.
And sitteth on the right hand of the pthread.
Whence he come again to judge the code efficiency.

I believe in the holy kernel, the holy catholic OS, the communion of parallel processes, the remission of API's, the resurrection of Andrew and thread lifespan; eternal.

- I'm not sure I like where this is going. I'm going to be crucified and zombified?

No Office Hours this Afternoon

- Sorry! I'm downtown for a tribunal right after CSC108 today.

- We do have office hours immediately after class, and I can also be reached by email.

Any AI Questions?

Software Solution: Semaphores

- Semaphores are data structures that provide synchronization. They include:
  - An integer variable *count* accessed only through 2 atomic operations
    - wait (also called P, down, or decrement) - block until semaphore is free, then decrement the variable
    - signal (also called V, up, or increment) - increment the variable and unblock a waiting thread if there are any
  - A queue for waiting threads

Types of Semaphores

- Mutex (or Binary) Semaphore
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
  - Max number of threads is determined by semaphore's initial value (*count*)
- Mutex has *count* = 1, counting has *count* = N

Readers/Writers, Revisited
Implement with Mutexes and CVs

```
//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;


Writer {
    wait(w_or_r);  //lock out others
    Write;
    signal(w_or_r); //up for grabs
}
```

```
Reader {
    wait(mutex); //lock readcount
    // one more reader
    readcount += 1;
    // is this the first reader?
    if(readcount == 1)
        //synch w/ writers
        wait(w_or_r);
    //unlock readcount
    signal(mutex);
    Read;
    wait(mutex); //lock readcount
    readcount -= 1;
    if(readcount == 0)
        signal(w_or_r);
    signal(mutex);
}
```

Discussion Question

- The author states that semaphores are capable of being both locks and CVs.

- Do you agree or disagree?
- Is there any CV behaviour that a semaphore cannot easily replicate?
- Or is there a semaphore behaviour that would make us use them differently from CVs?

Communicating State

- Semaphores allow objects to communicate
  - If a semaphore is free, then the thread can take it and move on.
  - If a semaphore is absent, then the thread can wait and be guaranteed a wake-up.
  - Forces 1-1 communication and requires that critical sections be atomic.
- This form of communication is limited.
  - What if some arbitrary number of threads want to be woken up?
  - What if we realize we can't make progress and want to continue later?
  - What we are looking for is notification that some condition is true.

Condition Variable Ops

- Condition variables support:
  - wait() (suspend the invoking process)
  - signal() (resume one or zero suspended processes)

- These are the same operations as semaphores, but ...
  - On "wait", the the process gives up a mutex
    - On being awakened, the process reacquires the mutex
  - A signal wakes up one process -- if one is waiting

Wasted Signals?

- If no process is suspended, a signal has no effect
  - In contrast, a semaphore signal always changes the state of the semaphore

- Back to the question: "Who goes first?"
  - Suppose process P executes a signal operation that would wake a suspended process Q
    - Either process can continue execution ...
    - But both cannot be simultaneously active in the monitor

Monitor Semantics

- Hoare monitors (original): waiter first
  - signal() immediately switches from the caller to a waiting thread
  - The condition that the waiter was blocked on is guaranteed to hold when the waiter resumes
  - Need another queue for the signaler, if signaler was not done using the monitor
- Mesa monitors (Mesa, Java, Nachos): signaler first
  - signal() places a waiter on the ready queue, but signaler continues inside monitor
  - Condition is not necessarily true when waiter resumes
  - Must check condition again

Hoare vs. Mesa Semantics

- Hoare

```
if (empty)
    wait(condition);
```

- Mesa

```
while(empty)
    wait(condition)
```

- Tradeoffs
  - Hoare monitors make it easier to reason
  - Mesa monitors are easier to implement, more efficient, can support additional operations like broadcast

CV Example

- Bounded buffer example: Want a monitor to control access to a buffer of limited size, N
  - Producers add to the buffer if it is not full
  - Consumers remove from the buffer if it is not empty
- Need two functions – add_to_buffer() and remove_from_buffer()
- Need one lock – only lock holder is allowed to be active in one of the monitor's functions
- Need two conditions – one to make producers wait, one to make consumers wait

Bounded Buffer Example: Variables

```
#define N 100
typedef struct buf_s {
   int data[N];
   int inpos; /* producer inserts here */
   int outpos; /* consumer removes from here */
   int numelements; /* # items in buffer */
   struct lock *mylock; /* access to monitor */
   struct cv *notFull; /* for producers to wait */
   struct cv *notEmpty; /* for consumers to wait */
} buf_t;

buf_t buffer;
void add_to_buff(int value);
int remove_from_buff();
```

Bounded Buffer Example: add

```
void add_to_buf(int value) {
  lock_acquire(buffer.mylock);
    while (nelements == N) {
      /* buffer is full, wait */
      cv_wait(buffer.notFull, buffer.mylock);
    }
    buf.data[inpos] = value;
    inpos = (inpos + 1) % N;
    nelements++;
    cv_signal(buffer.notEmpty, buffer.mylock);
  lock_release(buffer.mylock);
}
```

Higher-level Abstractions

- Locks
  - Very primitive, minimal semantics, difficult to use correctly
- Semaphores
  - Basic, easy to understand, hard to program with
- Condition Variables (to implement monitors)
  - High-level, ideally has language support (Java)
- Event-based synchronization
  - Essentially a polling loop
- Transactions

When Parallelizing, Consider ...

- How critical is the code to be (potentially) parallelized?
- How much parallelism is available?
  - How many dependencies are there between operations?
  - How frequent are collisions?

Discussion Question

Discuss the issues you would consider if asked to parallelize the following python-like pseudocode:

```
L = [...]
sums = [0, 0, 0, ...]
for item in L:
    sums[hash(item)] += func(item)
```

Concurrency Bugs

- The majority of bugs are ones of omission
  - Missing locks or synchronization that lead to race conditions.
- But too much synchronization is also a problem ...

Supplementary Material on Deadlock

Deadlock and Starvation

- Deadlock occurs when each thread in a set is waiting for an event that can only be caused by another thread in the set
  - In other words, "circular dependency"
  - In the case of semaphores, the event is the execution of the signal operation
- A thread is suffering starvation (or indefinite postponement) if it is waiting for a signal that never comes (or is not guaranteed to be sent)

Deadlock Defined

- The permanent blocking of a set of processes that either:
  - Compete for system resources, or
  - Communicate with each other

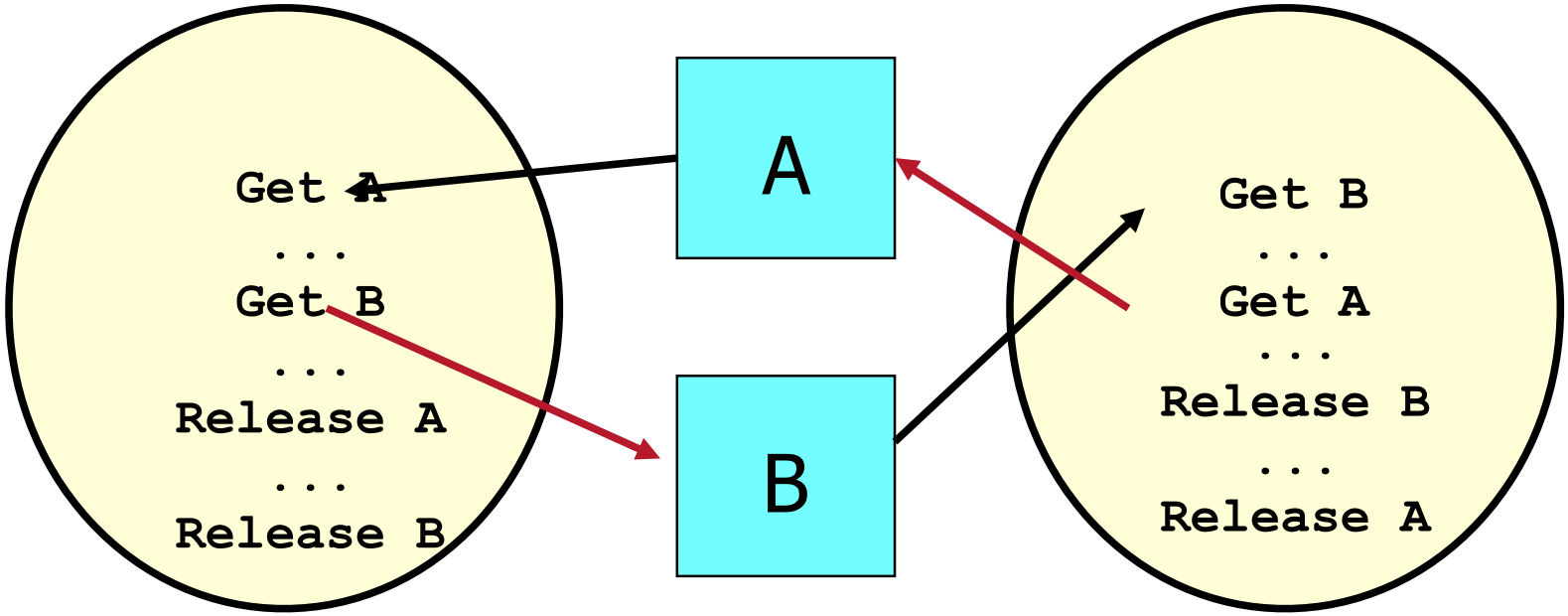
- Each process in the set is blocked, waiting for an event which can only be caused by another process in the set
  - Resources are finite
  - Processes wait if a resource they need is unavailable
  - Resources may be held by other waiting processes

Types of Resources

- Reusable
  - Can be used by one process at a time, released and used by another process
    - Printers, memory, processors, files
    - Locks, semaphores, monitors
- Consumable
  - Dynamically created and destroyed
  - Can only be allocated once
    - e.g. interrupts, signals, messages

Example of Deadlock

- Suppose processes *P* and *Q* need (reusable) resources *A* and *B*:



Conditions for Deadlock

1. Mutual Exclusion
   - Only one process may use a resource at a time
2. Hold and wait
   - A process may hold allocated resources while awaiting assignment of others
3. No preemption
   - No resource can be forcibly removed from a process holding it

- These are necessary conditions

One more condition…

4. Circular wait
   - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain
- Together, these four conditions are necessary and sufficient for deadlock
- Circular wait implies hold and wait
  - Circular wait is a sequence of events
  - Hold and wait is a policy decision

Deadlock Detection & Recovery

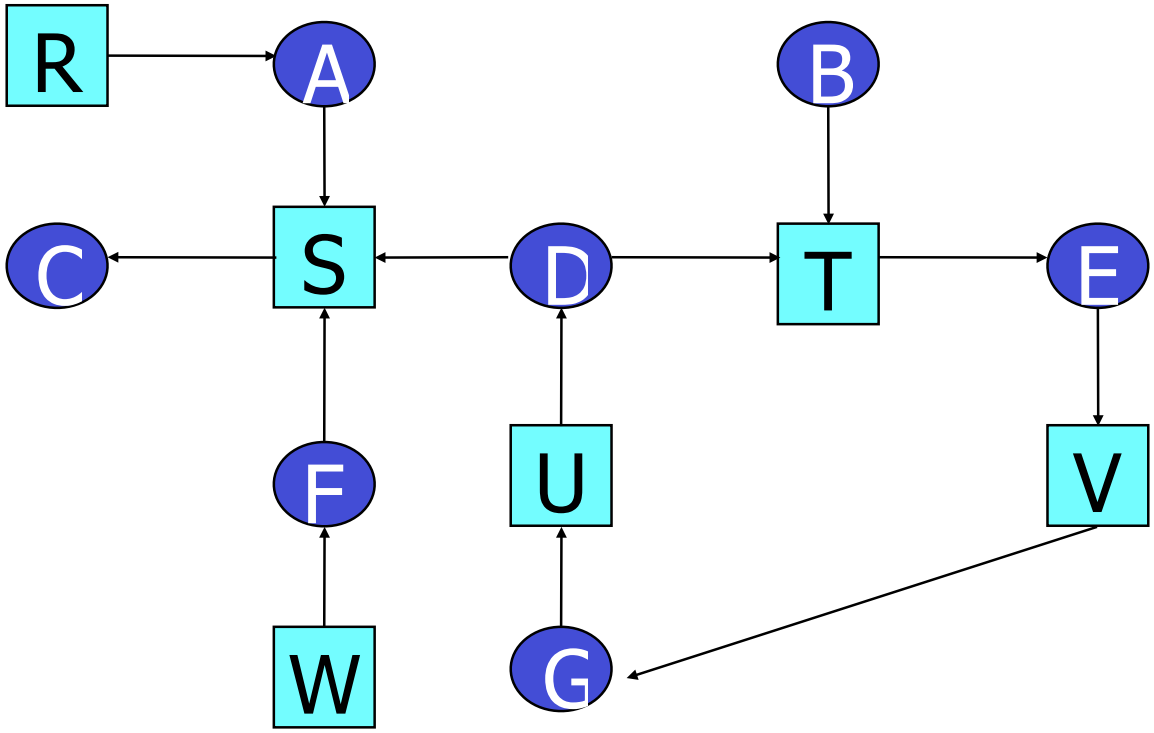
- Prevention and avoidance is awkward and costly
  - The need to be cautious leads to low utilization

- Instead, allow deadlocks to occur, but detect when this happens and find a way to break it
  - Check for circular wait condition periodically
- When should the system check for deadlocks?

Detection

- Finding circular waits is equivalent to finding a cycle in the resource allocation graph
  - Nodes are processes (drawn as circles) and resources (drawn as squares)
  - Arcs from a resource to a process represent allocations
  - Arcs from a process to a resource represent ungranted requests
- Any algorithm for finding a cycle in a directed graph will do
  - Note: with multiple instances of a type of resource, cycles may exist without deadlock

Example Resource Allocation Graph



Deadlock Recovery

- Basic idea is to break the cycle
  - Drastic - kill all deadlocked processes
  - Still drastic - selectively kill deadlocked processes until cycle is broken
    - Re-run detection alg. after each kill
  - Costly - back up and restart deadlocked processes (hopefully, non-determinism will keep deadlock from repeating)
  - Best (but tricky) - selectively preempt resources until cycle is broken
    - Processes must be rolled back

Reality Check

- No single strategy for dealing with deadlock is appropriate for all resources in all situations
- All strategies are costly in terms of computation overhead, or restricting use of resources
- Most operating systems employ the "Ostrich Algorithm"
  - "Ignore the problem and hope it doesn't happen often"

Why does the Ostrich Algorithm Work?

- Recall the causes of deadlock:
  - Resources are finite
  - Processes wait if a resource they need is unavailable
  - Resources may be held by other waiting processes
- Prevention/Avoidance/Detection deal with the last 2 points
- Modern operating systems virtualize most physical resources, eliminating the first problem
  - Some logical resources can't be virtualized (there must be exactly one), such as bank accounts or the process table
    - These are protected by synchronization objects, which are now the only resources that can cause deadlock