Concurrency Solutions and Deadlock
CS 111
Operating Systems
Peter Reiher

Outline

- Concurrency issues
  - Asynchronous completion
- Other synchronization primitives
- Deadlock
  - Causes
  - Solution approaches

Asynchronous Completion

- The second big problem with parallelism
  - How to wait for an event that may take a while
  - Without wasteful spins/busy-waits
- Examples of asynchronous completions
  - Waiting for a held lock to be released
  - Waiting for an I/O operation to complete
  - Waiting for a response to a network request
  - Delaying execution for a fixed period of time

Using Spin Waits to Solve the Asynchronous Completion Problem

- Thread A needs something from thread B
  - Like the result of a computation
- Thread B isn't done yet
- Thread A stays in a busy loop waiting
- Sooner or later thread B completes
- Thread A exits the loop and makes use of B's result
- Definitely provides correct behavior, but . . .

Well, Why Not?

- Waiting serves no purpose for the waiting thread
  - "Waiting" is not a "useful computation"
- Spin waits reduce system throughput
  - Spinning consumes CPU cycles
  - These cycles can't be used by other threads
  - It would be better for waiting thread to "yield"
- They are actually counter-productive
  - Delays the thread that will post the completion
  - Memory traffic slows I/O and other processors

Another Solution

- *Completion blocks*
- Create a synchronization object
  - Associate that object with a resource or request
- Requester blocks awaiting event on that object
  - Yield the CPU until awaited event happens
- Upon completion, the event is "posted"
  - Thread that notices/causes event posts the object
- Posting event to object unblocks the waiter
  - Requester is dispatched, and processes the event

Blocking and Unblocking

- Exactly as discussed in scheduling lecture
- Blocking
  - Remove specified process from the "ready" queue
  - Yield the CPU (let scheduler run someone else)
- Unblocking
  - Return specified process to the "ready" queue
  - Inform scheduler of wakeup (possible preemption)
- Only trick is arranging to be unblocked
  - Because it is so embarrassing to sleep forever
- Complexities if multiple entities are blocked on a resource – Who gets unblocked when it's freed?

A Possible Problem

- The sleep/wakeup race condition

Consider this sleep code:

```
void sleep( eventp *e ) {
  while(e->posted == FALSE) {
      add_to_queue( &e->queue,
      myproc );
      myproc->runstate |= BLOCKED;
      yield();
  }
}
```

And this wakeup code:

```
void wakeup( eventp *e) {
      struct proce *p;

      e->posted = TRUE;
      p = get_from_queue(&e->
queue);
      if (p) {
         p->runstate &= ~BLOCKED;
         resched();
      }  /* if !p, nobody's
waiting */
}
```

What's the problem with this?

A Sleep/Wakeup Race

- Let's say thread B is using a resource and thread A needs to get it
- So thread A will call `sleep()`
- Meanwhile, thread B finishes using the resource
  - So thread B will call `wakeup()`
- No other threads are waiting for the resource

The Race At Work

Thread A

```
void sleep( eventp *e ) {

 while(e->posted == FALSE) {
```

CONTEXT SWITCH!

Nope, nobody's in the queue!

CONTEXT SWITCH!

```
  add_to_queue( &e->queue, myproc );
  myproc->runsate |= BLOCKED;
  yield();
 }
}
```

Thread B

Yep, somebody's locked it!

```
void wakeup( eventp *e) {
 struct proce *p;

 e->posted = TRUE;
 p = get_from_queue(&e-> queue);
 if (p) {

  }  /* if !p, nobody's waiting */
 }
```

The effect?

Thread A is sleeping

But there's no one to wake him up

Solving the Problem

- There is clearly a critical section in `sleep()`
  - Starting before we test the posted flag
  - Ending after we put ourselves on the notify list
- During this section, we need to prevent
  - Wakeups of the event
  - Other people waiting on the event
- This is a mutual-exclusion problem
  - Fortunately, we already know how to solve those

Lock Contention

- The riddle of parallel multi-tasking:
  - If one task is blocked, CPU runs another
  - But concurrent use of shared resources is difficult
  - Critical sections serialize tasks, eliminating parallelism
- What if everyone needs to share one resource?
  - One process gets the resource
  - Other processes get in line behind him
  - Parallelism is eliminated; B runs after A finishes
  - That resource becomes a bottle-neck

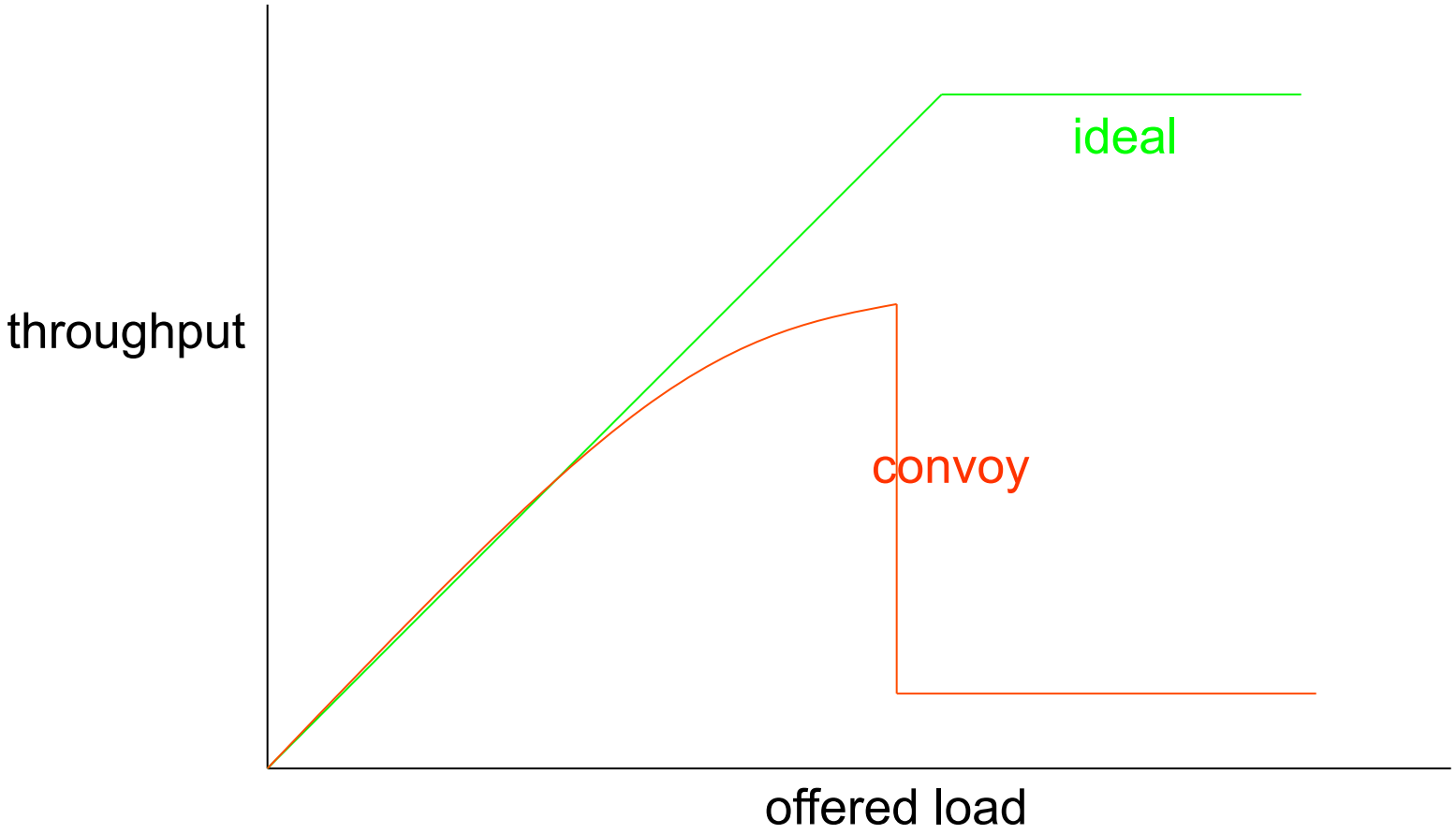
What If It Isn't That Bad?

- Say each thread is only somewhat likely to need a resource
- Consider the following system
  - Ten processes, each runs once per second
  - One resource they all use 5% of time (5ms/sec)
  - Half of all time slices end with a preemption
- Chances of preemption while in critical section
  - Per slice: 2.5%, per sec: 22%, over 10 sec: 92%
- Chances a 2nd process will need resource
  - 5% in next time slice, 37% in next second
- But once this happens, a line forms

Resource Convoys

- All processes regularly need the resource
  - But now there is a waiting line
  - Nobody can "just use the resource", must get in line
- The delay becomes much longer
  - We don't just wait a few μ−sec until resource is free
  - We must wait until everyone in front of us finishes
  - And while we wait, more people get into the line
- Delays rise, throughput falls, parallelism ceases
- Not merely a theoretical transient response

Resource Convoy Performance



Avoiding Contention Problems

- Eliminate the critical section entirely
  - Eliminate shared resource, use atomic instructions
- Eliminate preemption during critical section
  - By disabling interrupts … not always an option
- Reduce lingering time in critical section
  - Minimize amount of code in critical section
  - Reduce likelihood of blocking in critical section
- Reduce frequency of critical section entry
  - Reduce use of the serialized resource
  - Spread requests out over more resources

Lock Granularity

- How much should one lock cover?
  - One object or many
  - Important performance and usability implications
- Coarse grained - one lock for many objects
  - Simpler, and more idiot-proof
  - Results in greater resource contention
- Fine grained - one lock per object
  - Spreading activity over many locks reduces contention
  - Time/space overhead, more locks, more gets/releases
  - Error-prone: harder to decide what to lock when
  - Some operations may require locking multiple objects (which creates a potential for deadlock)

Other Important Synchronization Primitives

- Semaphores
- Mutexes
- Monitors

Semaphores

- Counters for sequence coord. and mutual exclusion
- Can be binary counters or more general
  - E.g., if you have multiple copies of the resource
- Call `wait()` on the semaphore to obtain exclusive access to a critical section
  - For binary semaphores, you wait till whoever had it signals they are done
- Call `signal()` when you're done
- For sequence coordination, signal on a shared semaphore when you finish first step
  - Wait before you do second step

Mutexes

- A synchronization construct to serialize access to a critical section
- Typically implemented using semaphores
- Mutexes are one per critical section
  - Unlike semaphores, which protect multiple copies of a resource

Monitors

- An object oriented synchronization primitive
  - Sort of very OO mutexes
  - Exclusion requirements depend on object/methods
  - Implementation should be encapsulated in object
  - Clients shouldn't need to know the exclusion rules
- A monitor is not merely a lock
  - It is an object class, with instances, state, and methods
  - All object methods protected by a semaphore
- Monitors have some very nice properties
  - Easy to use for clients, hides unnecessary details
  - High confidence of adequate protection