Synchronization, Critical Sections and Concurrency
CS 111
Operating Systems
Peter Reiher

Outline

- Parallelism and synchronization
- Critical sections and atomic instructions
- Using atomic instructions to build higher level locks
- Asynchronous completion
- Lock contention
- Synchronization in real operating systems

Benefits of Parallelism

- Improved throughput
  - Blocking of one activity does not stop others
- Improved modularity
  - Separating compound activities into simpler pieces
- Improved robustness
  - The failure of one thread does not stop others
- A better fit to modern paradigms
  - Cloud computing, web based services
  - Our universe is cooperating parallel processes

The Problem With Parallelism

- Making use of parallelism implies concurrency
  - Multiple actions happening at the same time
  - Or perhaps appearing to do so
- True parallelism is incomprehensible
  - Or nearly so
  - Few designers and programmers can get it right
  - Without help . . .
- Pseudo-parallelism may be good enough
  - Identify and serialize key points of interaction

Why Are There Problems?

- Sequential program execution is easy
  - First instruction one, then instruction two, ...
  - Execution order is obvious and deterministic
- Independent parallel programs are easy
  - If the parallel streams do not interact in any way
  - Who cares what gets done in what order?
- Cooperating parallel programs are hard
  - If the two execution streams are not synchronized
    - Results depend on the order of instruction execution
    - Parallelism makes execution order non-deterministic
    - Understanding possible outcomes of the computation becomes combinatorially intractable

Solving the Parallelism Problem

- There are actually two interdependent problems
  - Critical section serialization
  - Notification of asynchronous completion
- They are often discussed as a single problem
  - Many mechanisms simultaneously solve both
  - Solution to either requires solution to the other
- But they can be understood and solved separately

The Critical Section Problem

- A *critical section* is a resource that is shared by multiple threads
  - By multiple concurrent threads, processes or CPUs
  - By interrupted code and interrupt handler
- Use of the resource changes its state
  - Contents, properties, relation to other resources
- Correctness depends on execution order
  - When scheduler runs/preempts which threads
  - Relative timing of asynchronous/independent events

The Asynchronous Completion Problem

- Parallel activities move at different speeds
- One activity may need to wait for another to complete
- The *asynchronous completion problem* is how to perform such waits without killing performance
  - 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 real time

Critical Sections

- What is a critical section?
- Functionality whose proper use in parallel programs is critical to correct execution
- If you do things in different orders, you get different results
- A possible location for undesirable non-determinism

Critical Sections and Re-entrant Code

- Consider a simple recursive routine:

```
int factorial(x) { tmp =
  factorial( x-1 ); return x*tmp}
```

- Consider a possibly multi-threaded routine:

```
void debit(amt) {tmp = bal-amt;
  if (tmp >=0) bal = tmp)}
```

- Neither would work if `tmp` was shared/static
  - Must be dynamic, each invocation has its own copy
  - This is not a problem with read-only information
- What if a variable has to be writeable?
  - Writable variables should be dynamic or shared
- And proper sharing often involves critical sections

Basic Approach to Critical Sections

- Serialize access
  - Only allow one thread to use it at a time
  - Using some method like locking
- Won't that limit parallelism?
  - Yes, but . . .
- If true interactions are rare, and critical sections well defined, most code still parallel
- If there are actual frequent interactions, there isn't any real parallelism possible
  - Assuming you demand correct results

Recognizing Critical Sections

- Generally includes updates to object state
  - May be updates to a single object
  - May be related updates to multiple objects
- Generally involves multi-step operations
  - Object state inconsistent until operation finishes
    - This period may be brief or extended
  - Preemption leaves object in compromised state
- Correct operation requires *mutual exclusion*
  - Only one thread at a time has access to object(s)
  - Client 1 completes its operation before client 2 starts

Critical Section Example 1: Updating a File

Process 1

```
remove("database");
fd = create("database");
write(fd,newdata,length);
close(fd);
```

Process 2

```
fd = open("database",READ);
count = read(fd,buffer,length);
```

```
remove("database");
fd = create("database");
                          fd = open("database",READ);
                          count = read(fd,buffer,length);
write(fd,newdata,length);
close(fd);
```

- Process 2 reads an empty database
  - This result could not occur with any sequential execution

Critical Section Example 2: Re-entrant Signals

First signal

```
load r1,numsigs // = 0
add r1,=1   // = 1
store r1,numsigs // =1

     load r1,numsigs // = 0
     add r1,=1   // = 1

     store r1,numsigs // =1
```

Second signal

```
load r1,numsigs // = 0
add r1,=1   // = 1
store r1,numsigs // =1

load r1,numsigs // = 0
add r1,=1   // = 1
store r1,numsigs // =1
```

So `numsigs` is 1, instead of 2

numsigs 1

r1 1

The signal handlers share `numsigs` and `r1` ...

Critical Section Example 3: Multithreaded Banking Code

Thread 1

```
load r1, balance   // = 100
load r2, amount1 // = 50
add r1, r2              // = 150
store r1, balance  // = 150
```

Thread 2

```
load r1, balance    // = 100
load r2, amount2 // = 25
sub r1, r2              // = 75
store r1, balance   // = 75
```

```
load r1, b
load r2, a
add r1, r
```

The $25 debit was lost!!!

```
load r1, balance    // = 100
load r2, amount2 // = 25
sub r1, r2              // = 75
store r1, balance   // = 75
```

CONTEXT SWITCH!!!

```
store r1, balance  // = 150
```

amount1	50	balance	150	amount2	25
		r1	75		
		r2	50		

Are There Real Critical Sections in Operating Systems?

- Yes!
- Shared data for multiple concurrent threads
  - Process state variables
  - Resource pools
  - Device driver state
- Logical parallelism
  - Created by preemptive scheduling
  - Asynchronous interrupts
- Physical parallelism
  - Shared memory, symmetric multi-processors

These Kinds of Interleavings Seem Pretty Unlikely

- To cause problems, things have to happen exactly wrong
- Indeed, that's true
- But you're executing a billion instructions per second
- So even very low probability events can happen with frightening frequency
- Often, one problem blows up everything that follows

Can’t We Solve the Problem By Disabling Interrupts?

- Much of our difficulty is caused by a poorly timed interrupt
  - Our code gets part way through, then gets interrupted
  - Someone else does something that interferes
  - When we start again, things are messed up
- Why not temporarily disable interrupts to solve those problems?

Problems With Disabling Interrupts

- Not an option in user mode
  - Requires use of privileged instructions
- Dangerous if improperly used
  - Could disable preemptive scheduling, disk I/O, etc.
- Delays system response to important interrupts
  - Received data isn't processed until interrupt serviced
  - Device will sit idle until next operation is initiated
- Doesn't help with multicore processors
  - Other processors can access the same memory
- Generally harms performance
  - To deal with rare problems

So How Do We Solve This Problem?

- Avoid shared data whenever possible
  - No shared data, no critical section
  - Not always feasible
- Eliminate critical sections with *atomic instructions*
  - Atomic (uninteruptable) read/modify/write operations
  - Can be applied to 1-8 contiguous bytes
  - Simple: increment/decrement, and/or/xor
  - Complex: test-and-set, exchange, compare-and-swap
  - What if we need to do more in a critical section?
- Use atomic instructions to implement locks
  - Use the lock operations to protect critical sections

Atomic Instructions – Test and Set

A C description of a machine language instruction

```
bool TS( char *p) {
  bool rc;
  rc = *p;            /* note the current value          */
  *p = TRUE;          /* set the value to be TRUE        */
  return rc;          /* return the value before we set it */
}

if !TS(flag) {
      /* We have control of the critical section! */
}
```

Atomic Instructions – Compare and Swap

Again, a C description of machine instruction

```
bool compare_and_swap( int *p, int old, int new ) {
  if (*p == old) {    /* see if value has been changed     */
      *p = new;       /* if not, set it to new value       */
      return( TRUE);  /* tell caller he succeeded          */
  } else              /* value has been changed       */
    return( FALSE);   /* tell caller he failed             */
}

if (compare_and_swap(flag,UNUSED,IN_USE) {
      /* I got the critical section! */
} else {
      /* I didn't get it.  */
}
```

Solving Problem #3 With Compare and Swap

Again, a C implementation

```
int current_balance;
writecheck( int amount ) {
  int oldbal, newbal;
  do {
      oldbal = current_balance;
      newbal = oldbal - amount;
      if (newbal < 0) return (ERROR);
  } while (!compare_and_swap( &current_balance, oldbal, newbal))
...
}
```

Why Does This Work?

- Remember, `compare_and_swap()` is atomic
- First time through, if no concurrency,
  - `oldbal == current_balance`
  - `current_balance` was changed to `newbal` by `compare_and_swap()`
- If not,
  - `current_balance` changed after you read it
  - So `compare_and_swap()` didn't change `current_balance` and returned FALSE
  - Loop, read the new value, and try again

Will This Really Solve the Problem?

- If the compare & swap fails, we loop back and try again
  - If there is a conflicting thread isn't it likely to simply fail again?
- Only if preempted during a four instruction window
  - By someone executing the same critical section
- Extremely low probability event
  - We will very seldom go through the loop even twice

Limitation of Atomic Instructions

- They only update a small number of contiguous bytes
  - Cannot be used to atomically change multiple locations
    - E.g., insertions in a doubly-linked list
- They operate on a single memory bus
  - Cannot be used to update records on disk
  - Cannot be used across a network
- They are not higher level locking operations
  - They cannot "wait" until a resource becomes available
  - You have to program that up yourself
    - Giving you extra opportunities to screw up

Implementing Locks

- Create a synchronization object
  - Associated it with a critical section
  - Of a size that an atomic instruction can manage
- Lock the object to seize the critical section
  - If critical section is free, lock operation succeeds
  - If critical section is already in use, lock operation fails
    - It may fail immediately
    - It may block until the critical section is free again
- Unlock the object to release critical section
  - Subsequent lock attempts can now succeed
  - May unblock a sleeping waiter

Using Atomic Instructions to Implement a Lock

- Assuming C implementation of test and set

```
bool getlock( lock *lockp) {
 if (TS(lockp) == 0 )
    return( TRUE);
 else
    return( FALSE);
}
void freelock( lock *lockp ) {
 *lockp = 0;
}
```

Associating the Lock With a Critical Section

- Assuming same lock as in last example

```
while (!getlock(crit_section_lock))
{
    yield(); /*or spin on lock */
}
critical_section(); /*Access critical section */
freelock(crit_section_lock);
```

- Remember, while you’re in the critical section, no one else will be able to get the lock
  - Better not stay there too long
  - And definitely don’t go into infinite loop

Criteria for Correct Locking

- How do we know if a locking mechanism is correct?
- Four desirable criteria:
  1. Correct mutual exclusion
     - Only one thread at a time has access to critical section
  2. Progress
     - If resource is available, and someone wants it, they get it
  3. Bounded waiting time
     - No indefinite waits, guaranteed eventual service
  4. And (ideally) fairness
     - E.g. FIFO

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

Unblocking and Synchronization Objects

- Easy if only one thread is blocked on the object
- If multiple blocked threads, who should we unblock?
  - Everyone who is blocked?
  - One waiter, chosen at random?
  - The next thread in line on a FIFO queue?
- Depends on the resource
  - Can multiple threads use it concurrently?
  - If not, awaking multiple threads is wasteful
- Depends on policy
  - Should scheduling priority be used?
  - Consider possibility of starvation

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

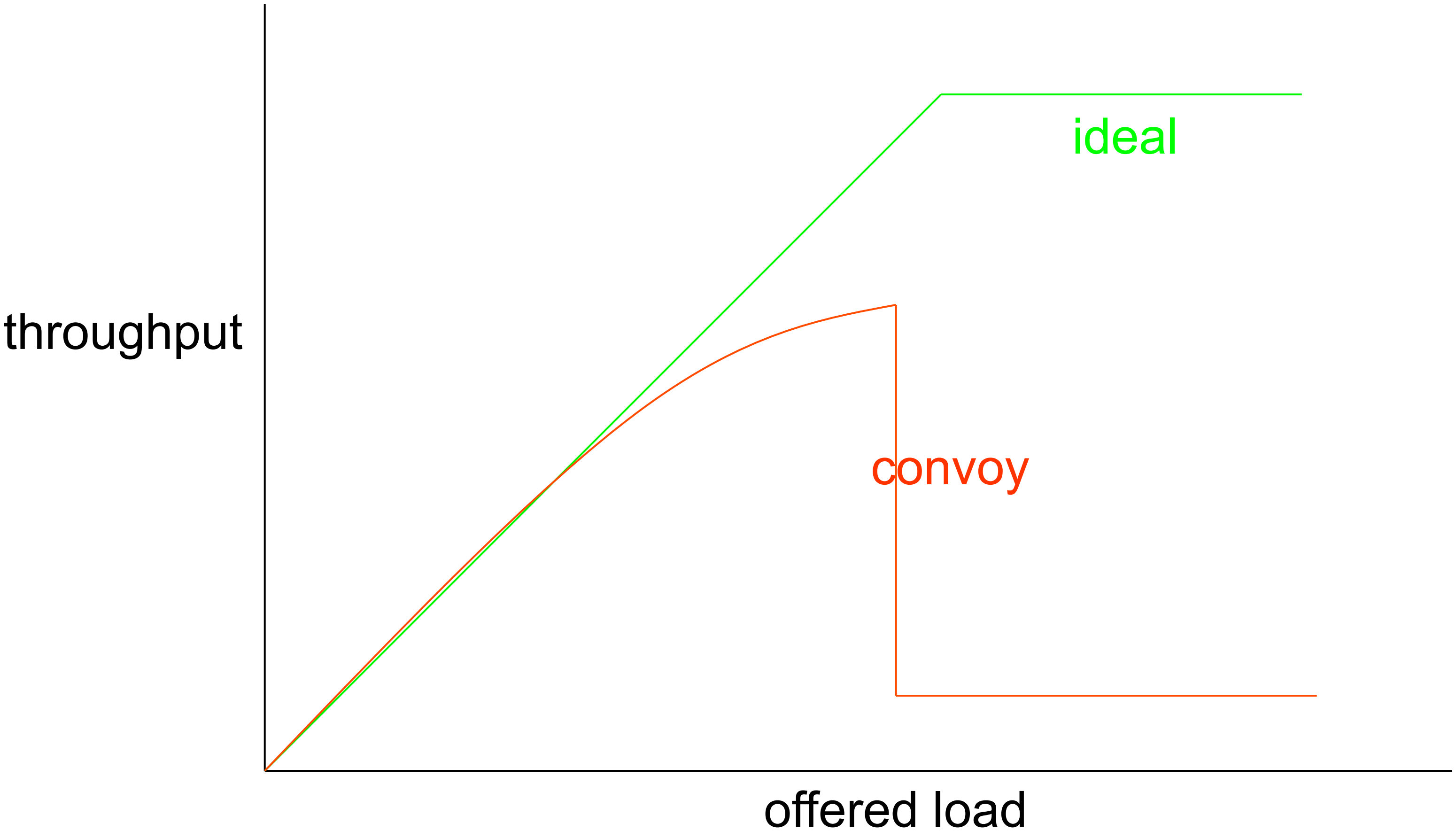
What If It Isn't That Bad?

- 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 each use 5% of their 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”
  - Instead, they must get in line
- The delay becomes <u>much</u> 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

An Approach Based on Smarter Locking

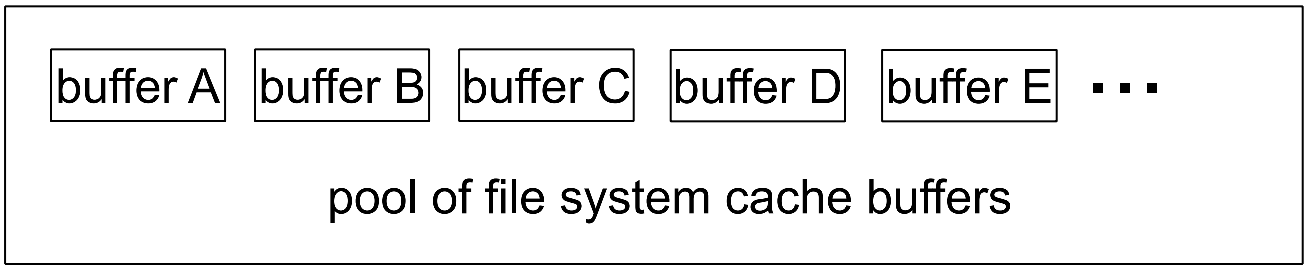
- Reads and writes are not equally common
  - File read/write: reads/writes > 50
  - Directory search/create: reads/writes > 1000
- Writers generally need exclusive access
- Multiple readers can generally share a resource
- Read/write locks
  - Allow many readers to share a resource
  - Only enforce exclusivity when a writer is active

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)

Lock Granularity: Pools Vs. Elements

- Consider a pool of objects, each with its own lock



- Most operations lock only one buffer within the pool
- Some operations require locking the entire pool
  - Two threads both try to add block A to the cache
  - Thread 1 looks for block B while thread 2 is deleting it
- The pool lock could become a bottle-neck
  - Minimize its use, reader/writer locking, sub-pools ...

Synchronization in Real World Operating Systems

- How is this kind of synchronization handled in typical modern operating systems?
- In the kernel itself?
- In user-level OS features?

Kernel Mode Synchronization

- Performance is a major concern
  - Many different types of exclusion are available
    - Shared/exclusive, interrupt-safe, SMP-safe
    - Choose type best suited to the resource and situation
  - Implementations are in machine language
    - Carefully coded for optimum performance
    - Extensive use of atomic instructions
- Imposes a greater burden on the callers
  - Most locking is explicit and advisory
  - Caller expected to know and follow locking rules

User Mode Synchronization

- Simplicity and ease of use of great importance
  - Conservative, enforced, one-size-fits-all locking
    - E.g., exclusive use, block until available
  - Implicitly associated with protected system objects
    - E.g., files, processes, message queues, events, etc.
    - System calls automatically serialize all operations
- Explicit serialization is only rarely used
  - To protect shared resources in multi-threaded apps
  - Simpler behavior than kernel-mode
  - Typically implemented via system calls into the OS

Case Study: Unix Synchronization

- Internal use is very specific to particular Unix implementation
  - Linux makes extensive use of semaphores internally
- But all Unix systems provide some user-level synchronization primitives
  - Including Linux

Unix User Synchronization Mechanisms

- Semaphores
  - Mostly supporting a Posix standard interface
  - sem_open, sem_close, sem_post, sem_wait
- Mutual exclusion file creation (O_EXCL)
- Advisory file locking (flock)
  - Shared/exclusive, blocking/non-blocking
- Enforced record locking (lockf)
  - Locks a contiguous region of a file
  - Lock/unlock/test, blocking/non-blocking
- All blocks can be aborted by a timer

Unix Asynchronous Completions

- Most events are associated with open files
  - Normal files and devices
  - Network or inter-process communication ports
- Users can specify blocking or non-blocking use
  - Non-blocking returns if no data is yet available
  - Poll if a logical channel is ready or would block
  - Select the first of *n* channels to become ready
- Users can also yield and wait
  - E.g., for the termination of a child process
- Signal will awaken a process from any blockage
  - E.g., alarm clock signal after specified time interval

Completion Events

- Available in Linux and other Unix systems
- Used in multithreaded programs
- One thread creates and starts a completion event
- Another thread calls a routine to wait on that completion event
- The thread that completes it makes another call
  - Which results in the waiting thread being woken