Concurrency Solutions and Deadlock
CS 111
Operating System Principles
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!

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 */
 }
```

Nope, nobody's in the queue!

CONTEXT SWITCH!

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

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

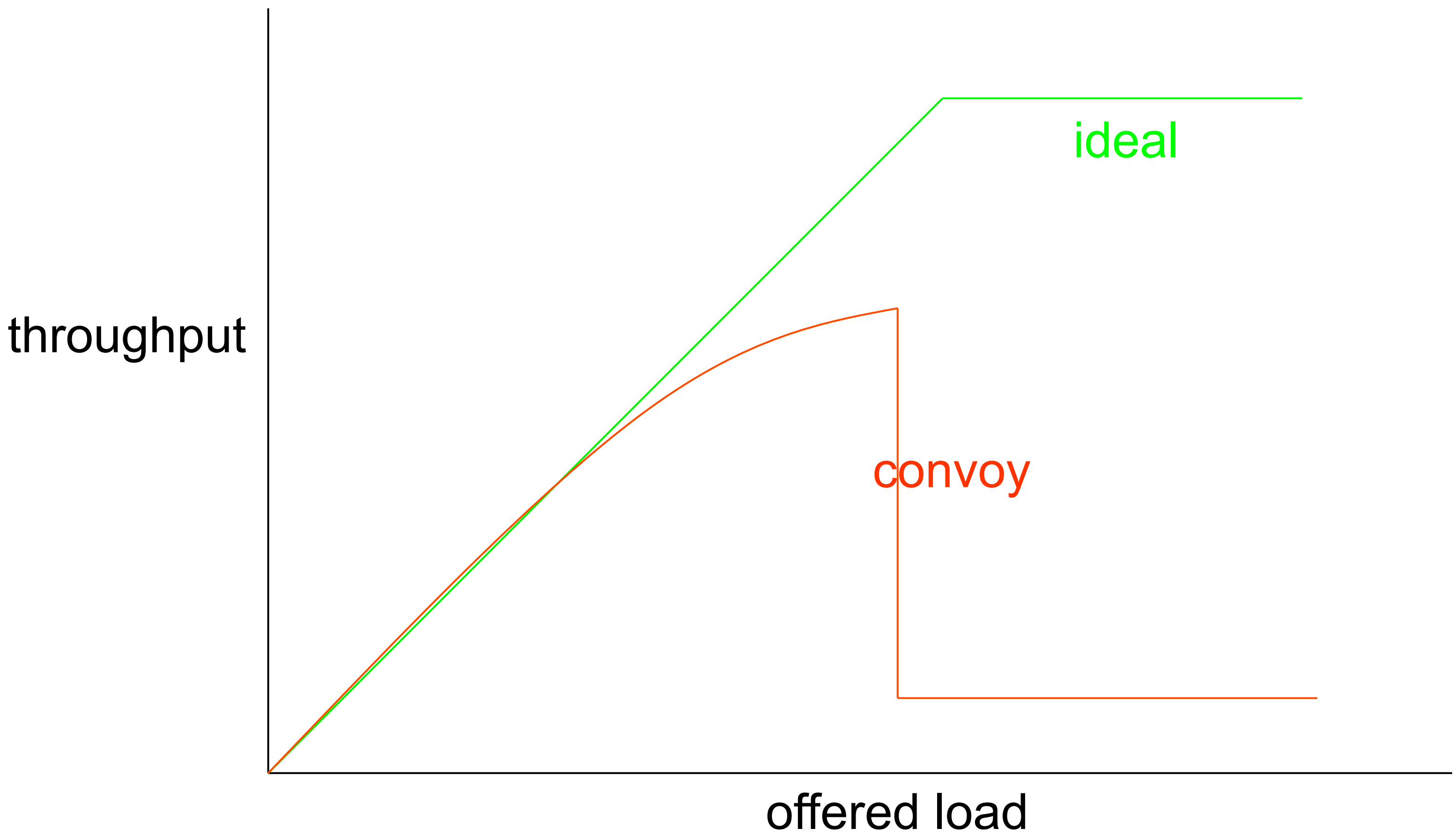
What If It Isn't <u>That</u> Bad?

- 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 <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

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

Deadlock

- What is a deadlock?
- A situation where two entities have each locked some resource
- Each needs the other's locked resource to continue
- Neither will unlock till they lock both resources
- Hence, neither can ever make progress

Why Are Deadlocks Important?

- A major peril in cooperating parallel processes
  - They are relatively common in complex applications
  - They result in catastrophic system failures
- Finding them through debugging is very difficult
  - They happen intermittently and are hard to diagnose
  - They are much easier to prevent at design time
- Once you understand them, you can avoid them
  - Most deadlocks result from careless/ignorant design
  - An ounce of prevention is worth a pound of cure

Types of Deadlocks

- Commodity resource deadlocks
  - E.g., memory, queue space
- General resource deadlocks
  - E.g., files, critical sections
- Heterogeneous multi-resource deadlocks
  - E.g., P1 needs a file P2 holds, P2 needs memory which P1 is using
- Producer-consumer deadlocks
  - E.g., P1 needs a file P2 is creating, P2 needs a message from P1 to properly create the file

Four Basic Conditions For Deadlocks

- For a deadlock to occur, all of these conditions must hold:

1. Mutual exclusion
2. Incremental allocation
3. No pre-emption
4. Circular waiting

Deadlock Conditions: 1. Mutual Exclusion

- The resources in question can each only be used by one entity at a time
- If multiple entities can use a resource, then just give it to all of them
- If only one can use it, once you've given it to one, no one else gets it
  - Until the resource holder releases it

Deadlock Condition 2: Incremental Allocation

- Processes/threads are allowed to ask for resources whenever they want
  - As opposed to getting everything they need before they start
- If they must pre-allocate all resources, either:
  - They get all they need and run to completion
  - They don't get all they need and abort
- In either case, no deadlock

Deadlock Condition 3: No Pre-emption

- When an entity has reserved a resource, you can't take it away from him
  – Not even temporarily
- If you can, deadlocks are simply resolved by taking someone's resource away
  – To give to someone else
- But if you can't take it away from anyone, you're stuck

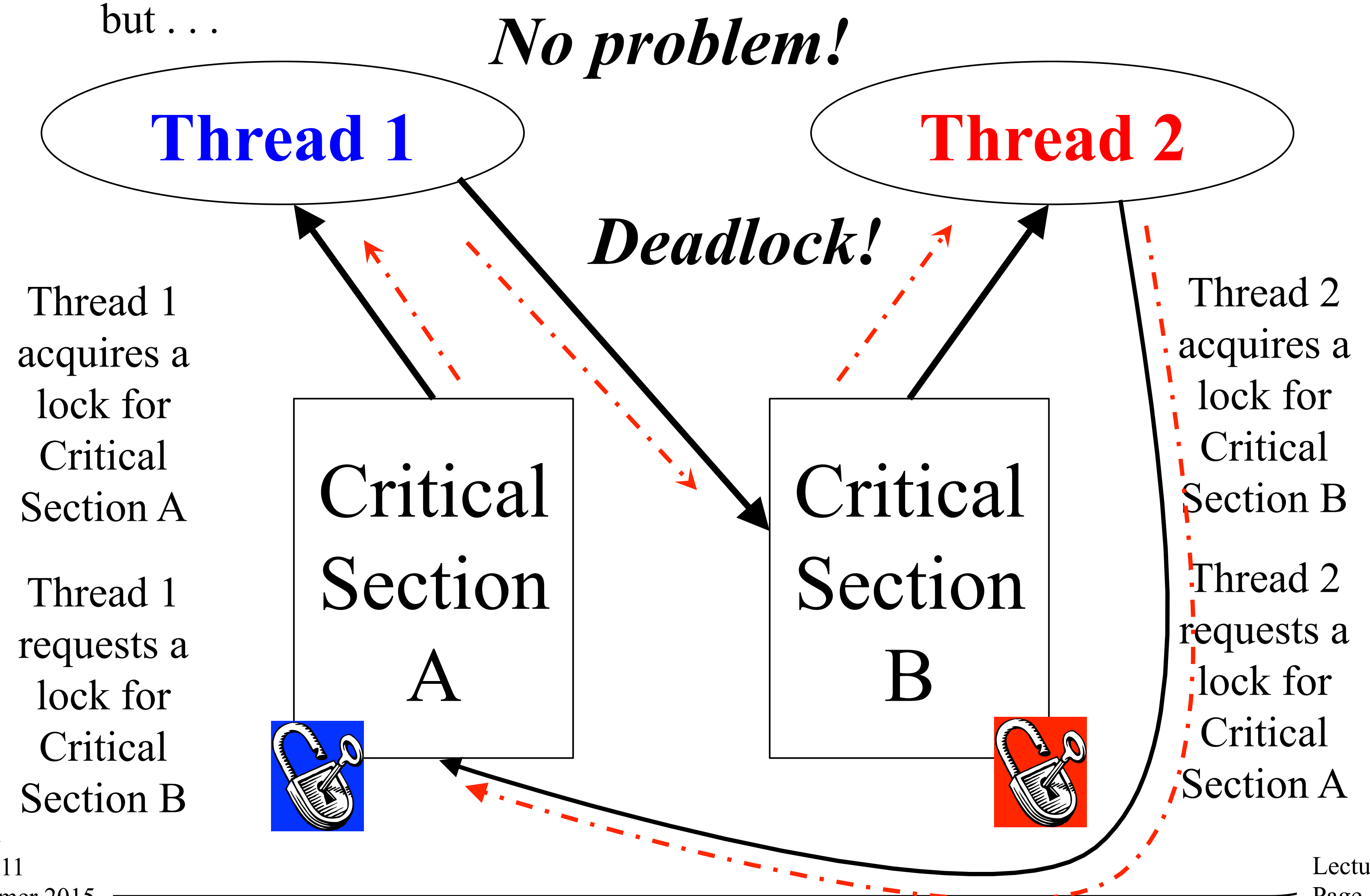
Deadlock Condition 4: Circular Waiting

- A waits on B which waits on A
- In graph terms, there's a cycle in a graph of resource requests
- Could involve a lot more than two entities
- But if there is no such cycle, someone can complete without anyone releasing a resource
  - Allowing even a long chain of dependencies to eventually unwind
  - Maybe not very fast, though . . .

A Wait-For Graph

We can’t give him the lock right now, but . . .

Hmmmm . . .



Deadlock Avoidance

- Use methods that guarantee that no deadlock can occur, by their nature
- Advance reservations
  - The problems of under/over-booking
- Practical commodity resource management
- Dealing with rejection
- Reserving critical resources

Avoiding Deadlock Using Reservations

- Advance reservations for commodity resources
  - Resource manager tracks outstanding reservations
  - Only grants reservations if resources are available
- Over-subscriptions are detected early
  - Before processes ever get the resources
- Client must be prepared to deal with failures
  - But these do not result in deadlocks
- Dilemma: over-booking vs. under-utilization

Overbooking Vs. Under Utilization

- Processes generally cannot perfectly predict their resource needs
- To ensure they have enough, they tend to ask for more than they will ever need
- Either the OS:
  - Grants requests till everything's reserved
    - In which case most of it won't be used
  - Or grants requests beyond the available amount
    - In which case sometimes someone won't get a resource he reserved

Handling Reservation Problems

- Clients seldom need all resources all the time
- All clients won't need max allocation at the same time
- Question: can one safely over-book resources?
  - For example, seats on an airplane
- What is a "safe" resource allocation?
  - One where everyone will be able to complete
  - Some people may have to wait for others to complete
  - We must be sure there are no deadlocks

Commodity Resource Management in Real Systems

- Advanced reservation mechanisms are common
  - Unix `brk()` and `sbrk()` system calls
  - Disk quotas, Quality of Service contracts
- Once granted, system must guarantee reservations
  - Allocation failures only happen at reservation time
  - Hopefully before the new computation has begun
  - Failures will not happen at request time
  - System behavior more predictable, easier to handle
- But clients must deal with reservation failures

Dealing With Reservation Failures

- Resource reservation eliminates deadlock
- Apps must still deal with reservation failures
  - Application design should handle failures gracefully
    - E.g., refuse to perform new request, but continue running
  - App must have a way of reporting failure to requester
    - E.g., error messages or return codes
  - App must be able to continue running
    - All critical resources must be reserved at start-up time

System Services and Reservations

- System services must never deadlock for memory
- Potential deadlock: swap manager
  - Invoked to swap out processes to free up memory
  - May need to allocate memory to build I/O request
  - If no memory available, unable to swap out processes
  - So it can't free up memory, and system wedges
- Solution:
  - Pre-allocate and hoard a few request buffers
  - Keep reusing the same ones over and over again
  - Little bit of hoarded memory is a small price to pay to avoid deadlock
- That's just one example system service, of course

Deadlock Prevention

- Deadlock avoidance tries to ensure no lock ever causes deadlock
- Deadlock prevention tries to assure that a particular lock doesn't cause deadlock
- By attacking one of the four necessary conditions for deadlock
- If any one of these conditions doesn't hold, no deadlock

Four Basic Conditions For Deadlocks

- For a deadlock to occur, these conditions must hold:

1. Mutual exclusion
2. Incremental allocation
3. No pre-emption
4. Circular waiting

1. Mutual Exclusion

- Deadlock requires mutual exclusion
  - P1 having the resource precludes P2 from getting it
- You can't deadlock over a shareable resource
  - Perhaps maintained with atomic instructions
  - Even reader/writer locking can help
    - Readers can share, writers may be handled other ways
- You can't deadlock on your private resources
  - Can we give each process its own private resource?

2. Incremental Allocation

- Deadlock requires you to block holding resources while you ask for others

1. Allocate all of your resources in a single operation
   – If you can’t get everything, system returns failure and locks nothing
   – When you return, you have <u>all or nothing</u>
2. Non-blocking requests
   – A request that can't be satisfied immediately will fail
3. Disallow blocking while holding resources
   – You must release all held locks prior to blocking
   – Reacquire them again after you return

Releasing Locks Before Blocking

- Could be blocking for a reason not related to resource locking
- How can releasing locks before you block help?
- Won't the deadlock just occur when you attempt to reacquire them?
  - When you reacquire them, you will be required to do so in a single all-or-none transaction
  - Such a transaction does not involve hold-and-block, and so cannot result in a deadlock

3. No Pre-emption

- Deadlock can be broken by resource confiscation
  – Resource “leases” with time-outs and “lock breaking”
  – Resource can be seized & reallocated to new client
- Revocation must be enforced
  – Invalidate previous owner's resource handle
  – If revocation is not possible, kill previous owner
- Some resources may be damaged by lock breaking
  – Previous owner was in the middle of critical section
  – May need mechanisms to audit/repair resource
- Resources must be designed with revocation in mind

When Can The OS “Seize” a Resource?

- When it can revoke access by invalidating a process’ resource handle
  - If process has to use a system service to access the resource, that service can no longer honor requests
- When is it not possible to revoke a process’ access to a resource?
  - If the process has direct access to the object
    - E.g., the object is part of the process’ address space
    - Revoking access requires destroying the address space
    - Usually killing the process

4. Circular Dependencies

- Use *total resource ordering*
  - All requesters allocate resources in same order
  - First allocate R1 and then R2 afterwards
  - Someone else may have R2 but he doesn't need R1
- Assumes we know how to order the resources
  - Order by resource type (e.g. groups before members)
  - Order by relationship (e.g. parents before children)
- May require complex and inefficient releasing and re-acquiring of locks

Which Approach Should You Use?

- There is no one universal solution to all deadlocks
  - Fortunately, we don't need one solution for all resources
  - We only need a solution for each resource
- Solve each individual problem any way you can
  - Make resources sharable wherever possible
  - Use reservations for commodity resources
  - Ordered locking or no hold-and-block where possible
  - As a last resort, leases and lock breaking
- OS must prevent deadlocks in all system services
  - Applications are responsible for their own behavior

One More Deadlock "Solution"

- Ignore the problem
- In many cases, deadlocks are very improbable
- Doing anything to avoid or prevent them might be very expensive
- So just forget about them and hope for the best
- But what if the best doesn't happen?

Deadlock Detection and Recovery

- Allow deadlocks to occur
- Detect them once they have happened
  - Preferably as soon as possible after they occur
- Do something to break the deadlock and allow someone to make progress
- Is this a good approach?
  - Either in general or when you don't want to avoid or prevent

Implementing Deadlock Detection

- Need to identify all resources that can be locked
- Need to maintain wait-for graph or equivalent structure
- When lock requested, structure is updated and checked for deadlock
  - In which case, might it not be better just to reject the lock request?
  - And not let the requester block?

Deadlock Detection and Health Monitoring

- Deadlock detection seldom makes sense
  - It is extremely complex to implement
  - Only detects "true deadlocks" for a known resources
  - Not always clear cut what you should do if you detect one
- Service/application "health monitoring" makes more sense
  - Monitor application progress/submit test transactions
  - If response takes too long, declare service "hung"
- Health monitoring is easy to implement
- It can detect a wide range of problems
  - Deadlocks, live-locks, infinite loops & waits, crashes

Related Problems Health Monitoring Can Handle

- Live-lock
  - Process is running, but won't free R1 until it gets message
  - Process that will send the message is blocked for R1
- Sleeping Beauty, waiting for "Prince Charming"
  - A process is blocked, awaiting some completion
  - But, for some reason, it will never happen
- Neither of these is a true deadlock
  - Wouldn't be found by deadlock detection algorithm
  - Both leave the system just as hung as a deadlock
- Health monitoring handles them

How To Monitor Process Health

- Look for obvious failures
  - Process exits or core dumps
- Passive observation to detect hangs
  - Is process consuming CPU time, or is it blocked?
  - Is process doing network and/or disk I/O?
- External health monitoring
  - “Pings”, null requests, standard test requests
- Internal instrumentation
  - White box audits, exercisers, and monitoring

What To Do With "Unhealthy" Processes?

- Kill and restart "all of the affected software"
- How many and which processes to kill?
  - As many as necessary, but as few as possible
  - The hung processes may not be the ones that are broken
- How will kills and restarts affect current clients?
  - That depends on the service APIs and/or protocols
  - Apps must be designed for cold/warm/partial restarts
- Highly available systems define restart groups
  - Groups of processes to be started/killed as a group
  - Define inter-group dependencies (restart B after A)

Failure Recovery Methodology

- Retry if possible ... but not forever
  - Client should not be kept waiting indefinitely
  - Resources are being held while waiting to retry
- Roll-back failed operations and return an error
- Continue with reduced capacity or functionality
  - Accept requests you can handle, reject those you can't
- Automatic restarts (cold, warm, partial)
- Escalation mechanisms for failed recoveries
  - Restart more groups, reboot more machines

Priority Inversion and Deadlock

- Priority inversion isn't necessarily deadlock, but it's related
  - A low priority process P1 has mutex M1 and is preempted
  - A high priority process P2 blocks for mutex M1
  - Process P2 is effectively reduced to priority of P1
- Solution: mutex priority inheritance
  - Check for problem when blocking for mutex
  - Compare priority of current mutex owner with blocker
  - Temporarily promote holder to blocker's priority
  - Return to normal priority after mutex is released

Priority Inversion on Mars

- A real priority inversion problem occurred on the Mars Pathfinder rover
- Caused serious problems with system resets
- Difficult to find

The Pathfinder Priority Inversion

- Special purpose hardware running VxWorks real time OS
- Used preemptive priority scheduling
  - So a high priority task should get the processor
- Multiple components shared an "information bus"
  - Used to communicate between components
  - Essentially a shared memory region
  - Protected by a mutex

A Tale of Three Tasks

- A high priority bus management task (at P1) needed to run frequently
  - For brief periods, during which it locked the bus
- A low priority meteorological task (at P3) ran occasionally
  - Also for brief periods, during which it locked the bus
- A medium priority communications task (at P2) ran rarely
  - But for a long time when it ran
  - But it didn’t use the bus, so it didn’t need the lock
- P1>P2>P3

What Went Wrong?

- Rarely, the following happened:
  - The meteorological task ran and acquired the lock
  - And then the bus management task would run
  - It would block waiting for the lock
    - Don't pre-empt low priority if you're blocked anyway
- Since meteorological task was short, usually not a problem
- But if the long communications task woke up in that short interval, what would happen?

The Priority Inversion at Work

B's priority of P1 is higher than C's, but B can't run because it's waiting on a lock held by M

A HIGH PRIORITY TASK DOESN'T RUN AND A LOWER PRIORITY TASK DOES

B

C

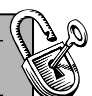
i
t
y

Lock Bus

But M won't run again until C completes

M can't interrupt C, since it only has priority P3

M won't release the lock until it runs again

M 

Time

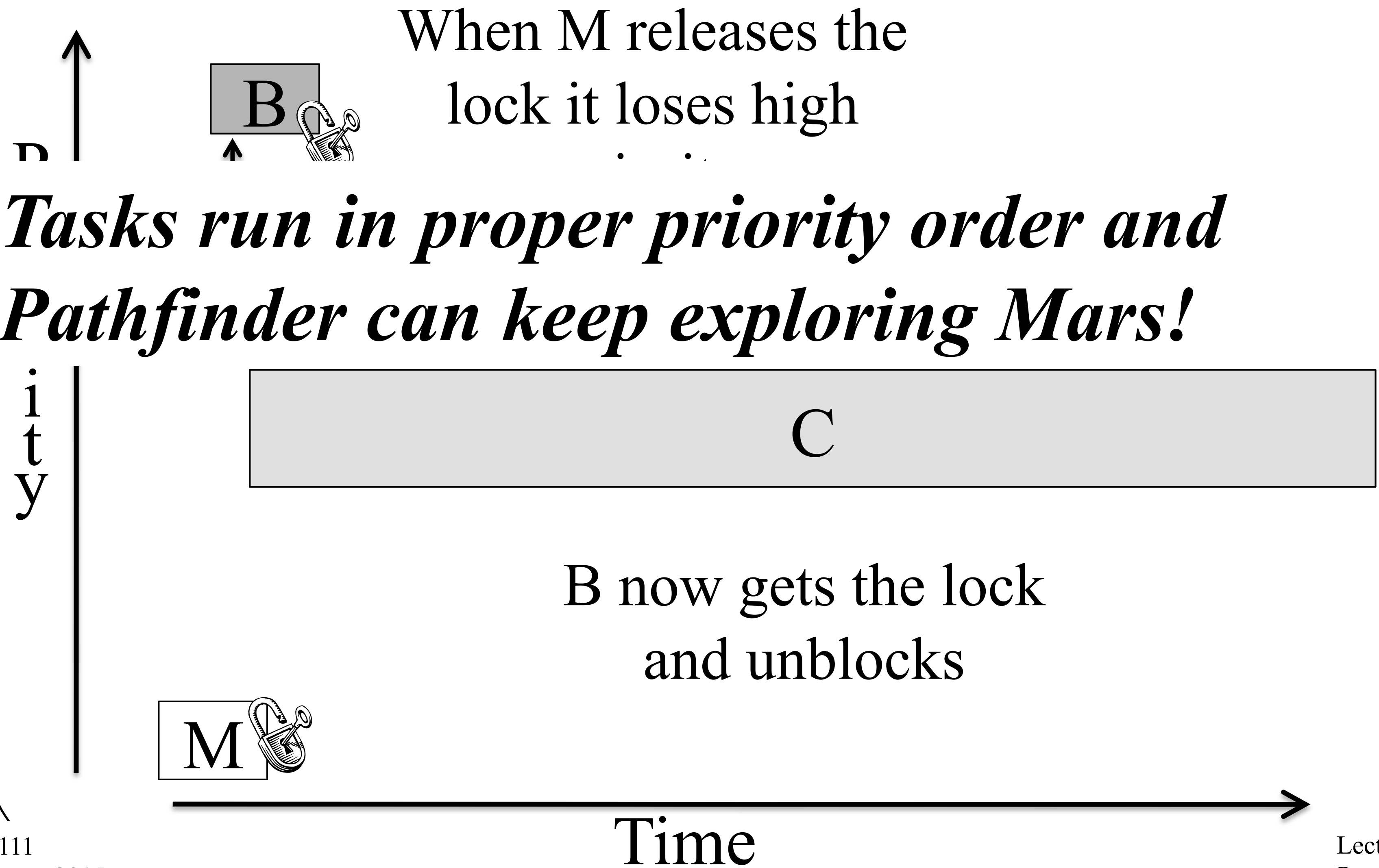
The Ultimate Effect

- A watchdog timer would go off every so often
  - At a high priority
  - It didn't need the bus
  - A health monitoring mechanism
- If the bus management task hadn't run for a long time, something was wrong
- So the watchdog code reset the system
- Every so often, the system would reboot

Solving the Problem

- This was a priority inversion
  - The lower priority communications task ran before the higher priority bus management task
- That needed to be changed
- How?
- Temporarily increase the priority of the meteorological task
  - While the high priority bus management task was block by it
  - So the communications task wouldn't preempt it
  - *Priority inheritance*: a general solution to this kind of problem

The Fix in Action



Tasks run in proper priority order and Pathfinder can keep exploring Mars!