Synchronization and Deadlock
CS 111
Operating Systems
Peter Reiher

Outline

- Deadlocks:
  - What are they and why are they important?
  - Deadlock avoidance, prevention, detection and recovery
- Related synchronization problems

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

Deadlocks and Different Resource Types

- Commodity Resources
  - Clients need an amount of it (e.g. memory)
  - Deadlocks result from over-commitment
  - Avoidance can be done in resource manager
- General Resources
  - Clients need a specific instance of something
    - A particular file or semaphore
    - A particular message or request completion
  - Deadlocks result from specific dependency relationships
  - Prevention is usually done at design time

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, 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

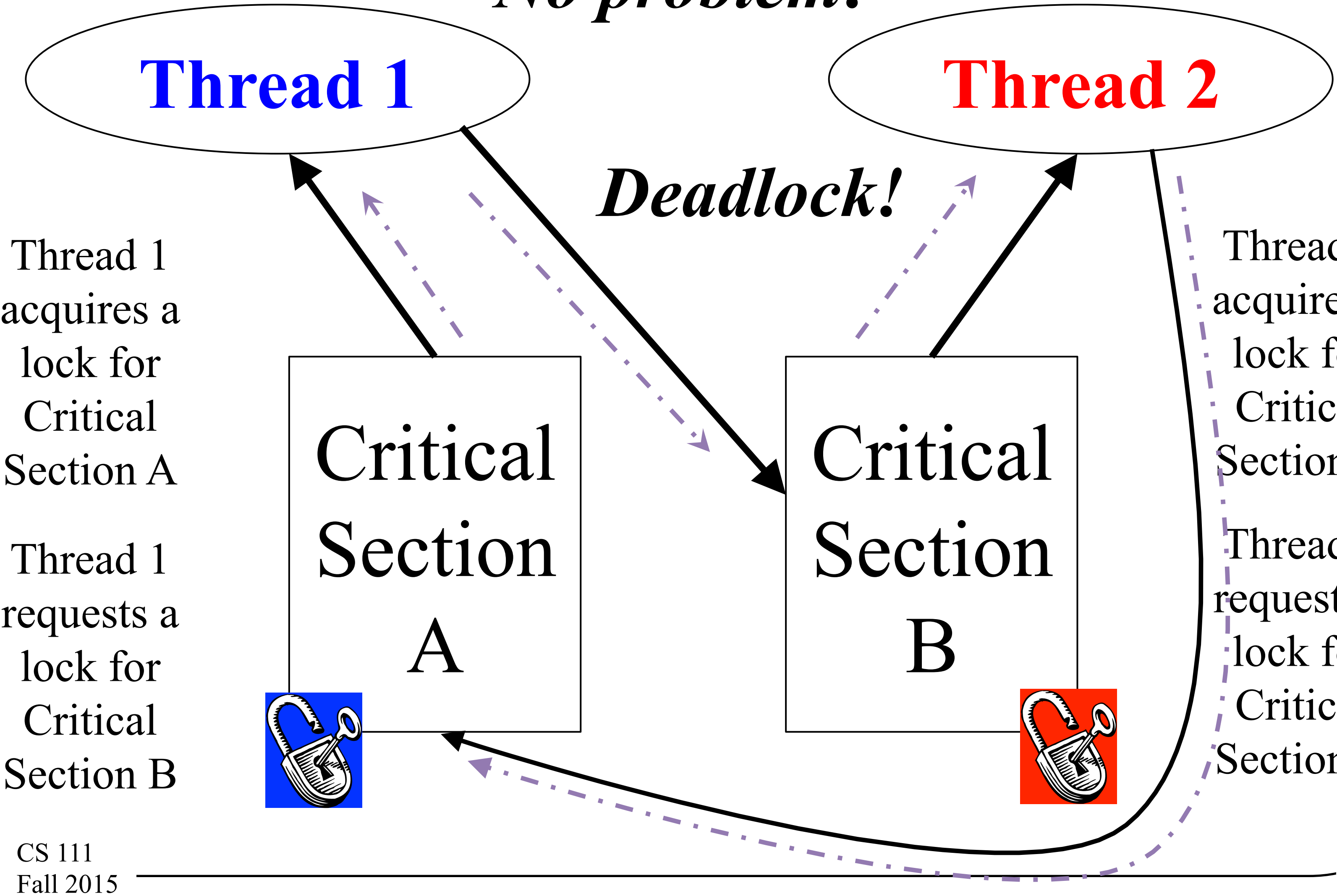
- A waits on B which waits on A
- In graph terms, there's a cycle in a graph of resource requests
- Could involve a lot more than two entities
- But if there is no such cycle, someone can complete without anyone releasing a resource
  - Allowing even a long chain of dependencies to eventually unwind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

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

A Wait-For Graph

Hmmmm . . .

No problem!



Thread 1

Thread 2

Deadlock!

Critical Section A

Critical Section B

Thread 1 acquires a lock for Critical Section A

Thread 1 requests a lock for Critical Section B

Thread 2 acquires a lock for Critical Section B

Thread 2 requests a lock for Critical Section A

Deadlock Avoidance

- Use methods that guarantee that no deadlock can occur, by their nature
- Advance reservations
  - The problems of under/over-booking
  - The Bankers' Algorithm
- 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

The Banker's Algorithm

- One algorithm for resource reservations
- Assumptions:

1. All critical resources are known and quantifiable
   - E.g., money or memory
   - No other resources can cause deadlocks
2. All clients reserve for their maximum requirement
   - They will never need more than this amount
3. If a client gets his maximum, he will complete
   - Upon completion, he frees all his resources
   - Those resources then become available for others

The Rules of the Banker's Algorithm

- Given a resource "state" characterized by:
  - Total size of each pool of resources
  - Reservations granted to each client for each resource
  - Current allocations of each resource to each client
- A state is "safe" if . . .
  - Enough resources are allocated to at least one client to allow him to finish
  - After any client frees its resources, resulting state is safe
  - And so on, until all clients have completed
- A proposed allocation can be granted if the resulting state would still be "safe"

Why Isn’t the Banker’s Algorithm Used?

- Quantified resources assumption
  - Not all resources are measurable in units
  - Other resource types can introduce circular dependencies
- Eventual completion assumption
  - All resources are released when client completes
  - In modern systems many tasks run for months
- Likelihood of resource “convoy” formation
  - Reduced parallelism, reduced throughput
- Many systems choose simpler “don't overbook” policy

Commodity Resource Management in Real Systems

- Advanced reservation mechanisms are common
  - Memory reservations
  - 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

Isn’t Rejecting App Requests Bad?

- It’s not great, but it’s better than failing later
- With advance notice, app may be able to adjust service not to need the unavailable resource
- If app is in the middle of servicing a request, we may have other resources allocated
  - And the request half-performed
  - If we fail then, all of this will have to be unwound
  - Could be complex, or even impossible

Why Not Just Wait?

- If reservation fails, why not hold on to what I’ve got and ask again later?
- What would happen in our deadlock example?
  - Nobody would ever make progress
  - That’s what would generally happen in deadlock if you just wait
- Making your clients wait indefinitely is a bad idea

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

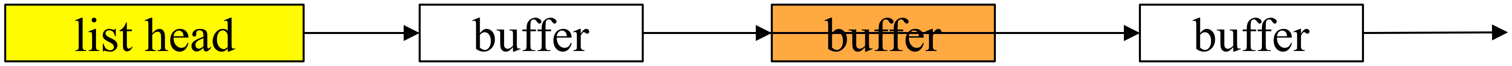
When Can The OS "Seize" a Resource?

- 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 a *lock dance*
  - – Release R2, allocate R1, reacquire R2

Lock Dances



list head must be locked for searching, adding & deleting

individual buffers must be locked to perform I/O & other operations

To avoid deadlock, we must always lock the list head before we lock an individual buffer.

To find a desired buffer:

read lock list head

search for desired buffer

lock desired buffer

unlock list head

return (locked) buffer

To delete a (locked) buffer from list

unlock buffer

write lock list head

search for desired buffer

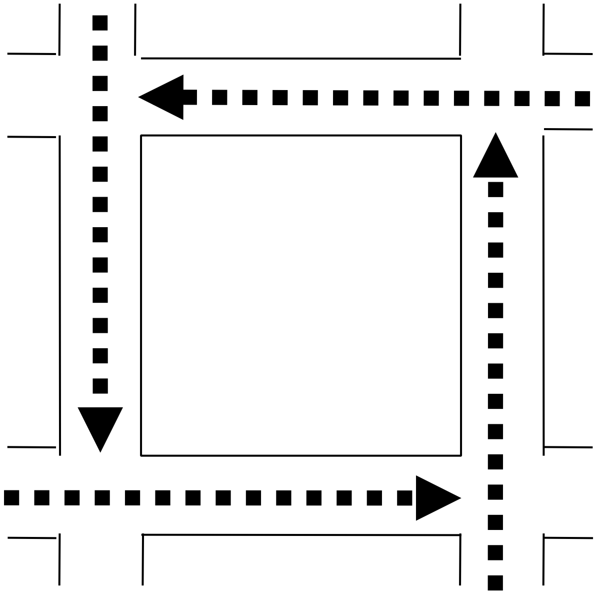
lock desired buffer

remove from list

unlock list head

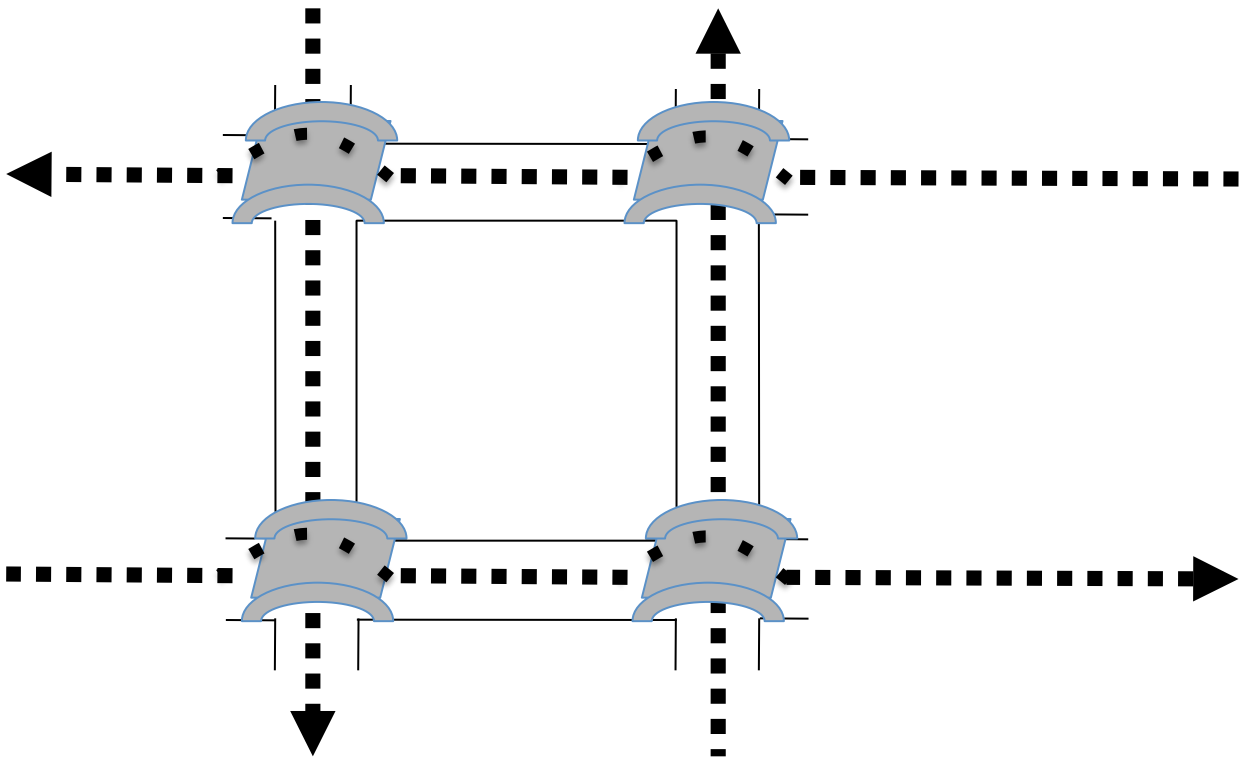
An Example of Breaking Deadlocks

- The problem – urban traffic gridlock
  - “Resource” is the ability to pass through intersection
  - Deadlock happens when nobody can get through



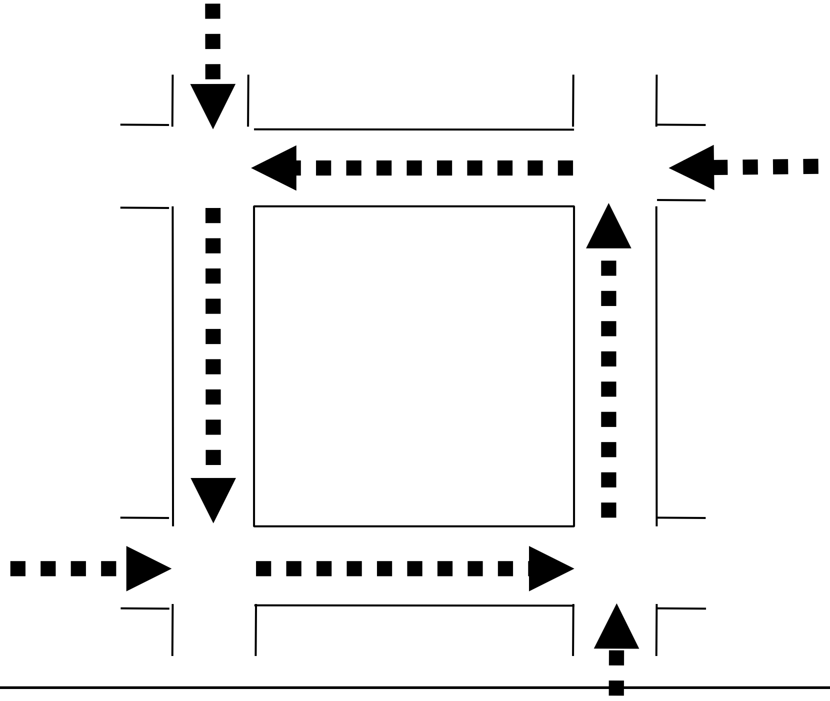
Using Attack Approach 1 To Prevent Deadlock

- Avoid mutual exclusion
- Build overpass bridges for east/west traffic



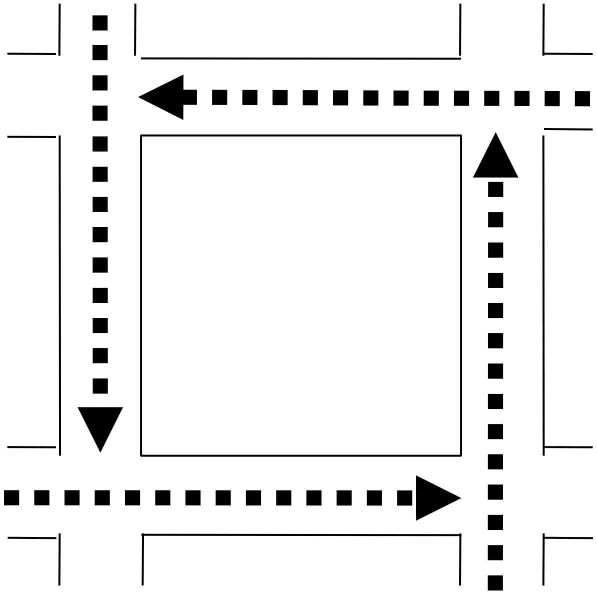
Using Attack Approach 2 To Prevent Deadlock

- Make it illegal to enter the intersection if you can’t exit it
  - Thus, preventing “holding” of the intersection



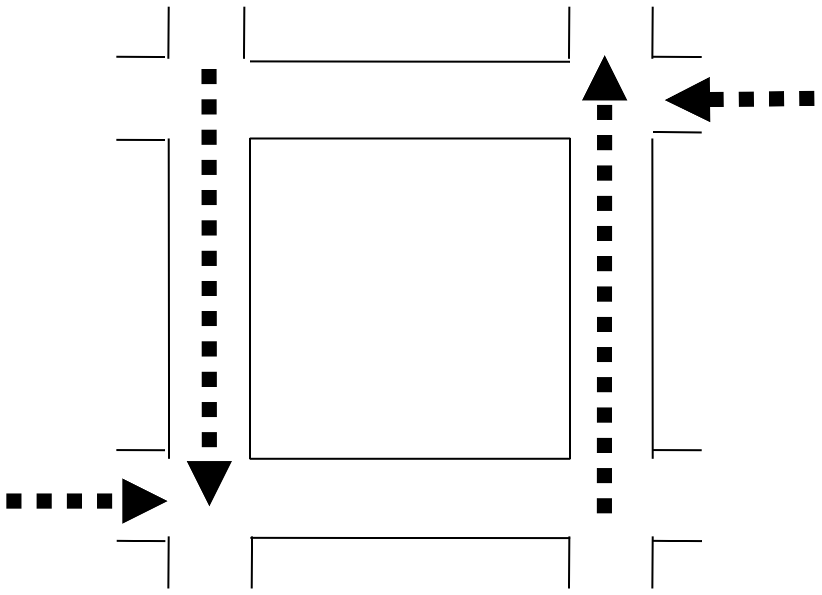
Using Attack Approach 3 To Prevent Deadlock

- Allow preemption
  - Force some car to pull over to the side



Using Attack Approach 4 To Prevent Deadlock

- Avoid circular dependencies by decreeing a totally ordered right of way
  - E.g., North beats West beats South beats East



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 deadlocks?

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 resource
  - Not always clear cut what you should do if you detect one
- Service/application *health monitoring* is better
  - 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

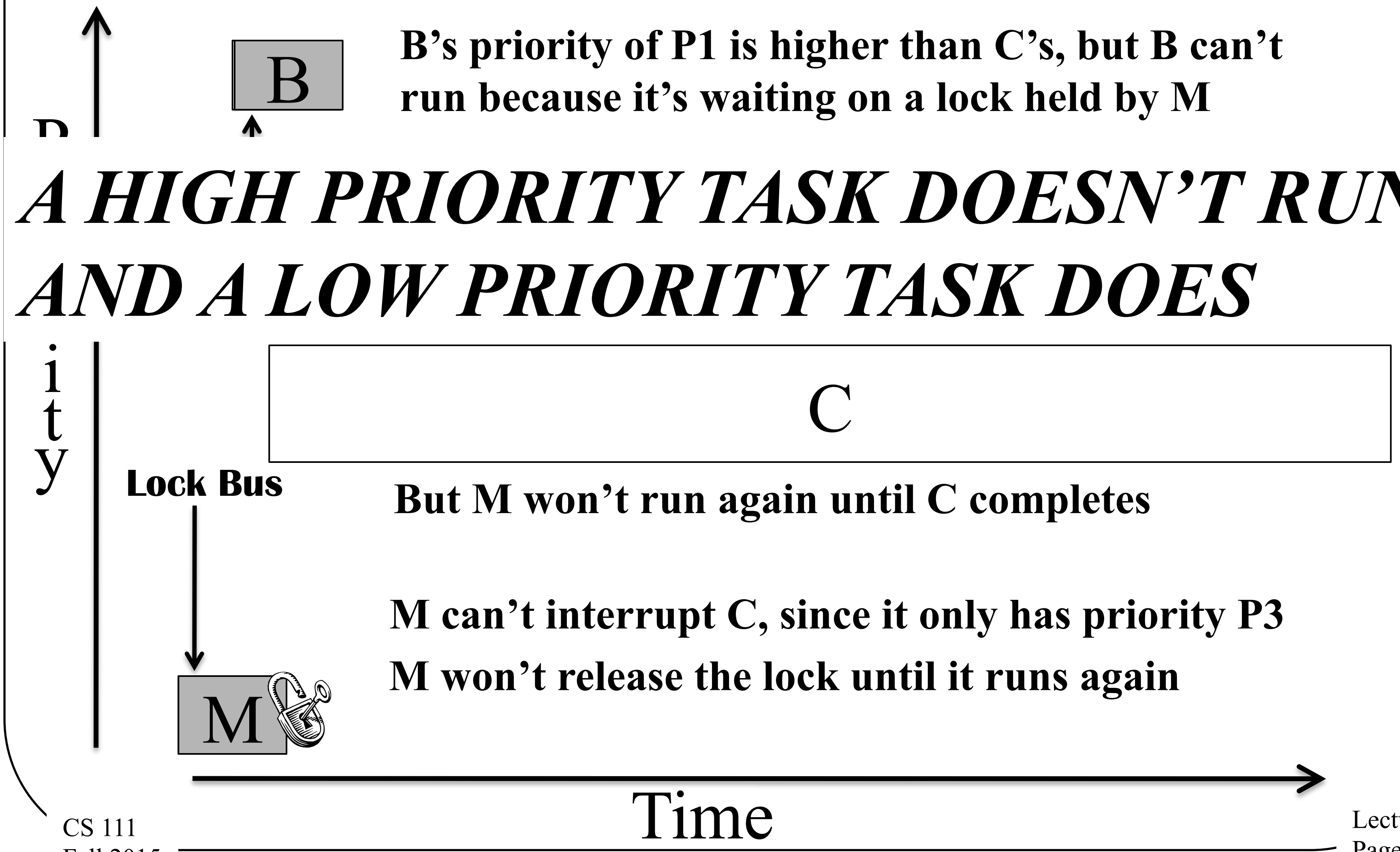
- A high priority bus management task (at P1) needed to run frequently
  - For brief periods, during which it locked the bus
- A low priority meteorological task (at P3) ran occasionally
  - Also for brief periods, during which it locked the bus
- A medium priority communications task (at P2) ran rarely
  - 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

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



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

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

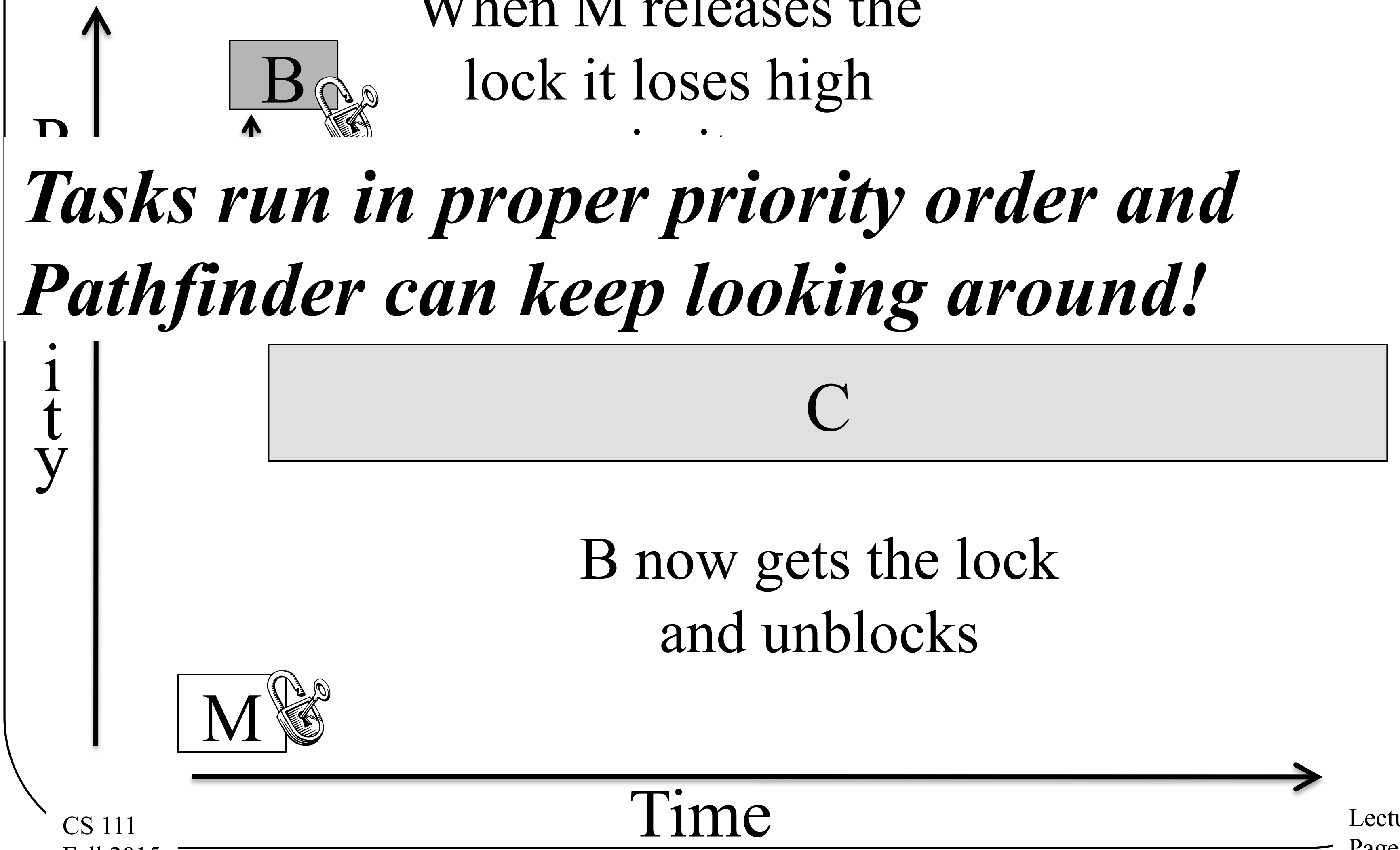
The Ultimate Effect

- 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 blocked by it
  - So the communications task wouldn’t preempt it
  - *Priority inheritance*: a general solution to this kind of problem

The Fix in Action



When M releases the lock it loses high

Tasks run in proper priority order and Pathfinder can keep looking around!

B now gets the lock and unblocks