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

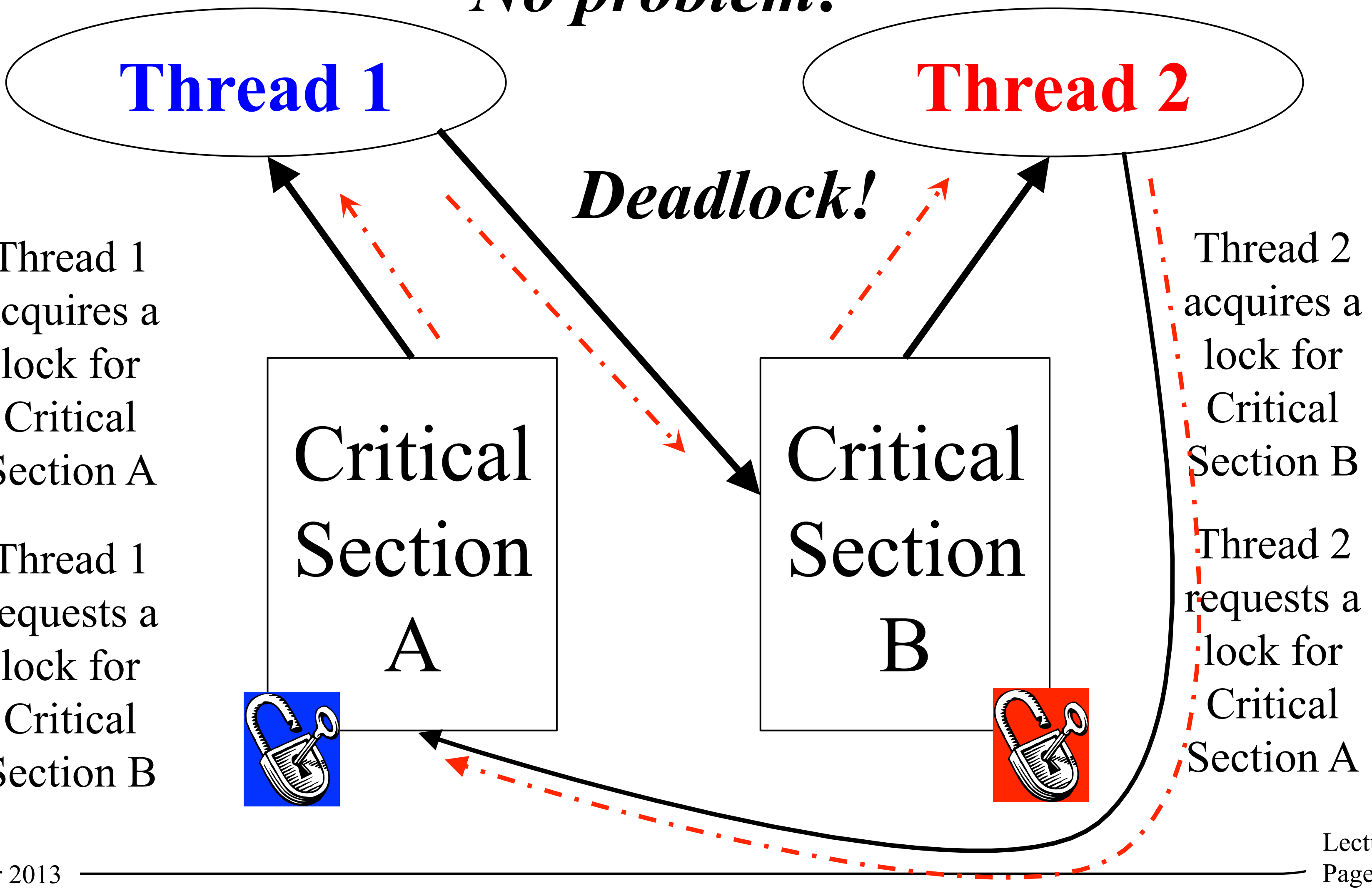
Deadlock Condition 3: No Pre-emption

- 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 . . .
No problem!
Thread 1
Thread 2
Deadlock!
Thread 1 acquires a lock for Critical Section A
Thread 1 requests a lock for Critical Section B
Critical Section A
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
- 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 all or nothing
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?