

Operating System Principles:
Deadlocks – Problems and Solutions
CS 111
Operating Systems

Outline

- The deadlock problem
 - Approaches to handling the problem
- Handling general synchronization bugs
- Simplifying synchronization

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

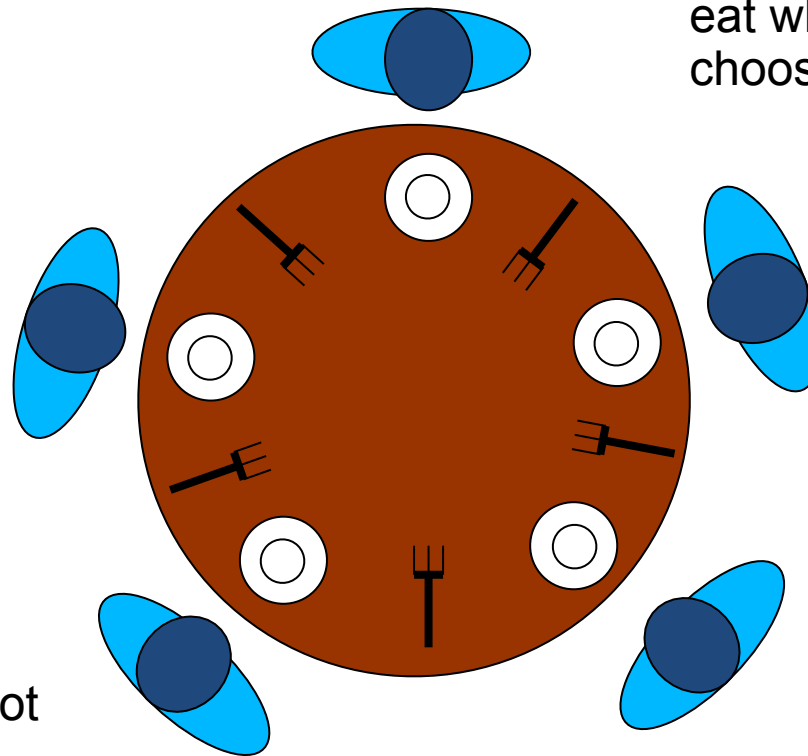
The Dining Philosophers Problem

Five philosophers
Five plates of pasta
Five forks

Philosophers try to
eat whenever they
choose to

A philosopher needs
two forks to eat
pasta, but must pick
them up one at a time

Philosophers will not
negotiate with
one-another



Ensure that
philosophers will not
deadlock while
trying to eat

The problem
demands an
absolute solution

Dining Philosophers and Deadlock

- This problem is the classical illustration of deadlocking
- It was created to illustrate deadlock problems
- It is a very artificial problem
 - It was carefully designed to cause deadlocks
 - Changing the rules eliminates deadlocks
 - But then it couldn't be used to illustrate deadlocks
 - Actually, one point of it is to see how changing the rules solves the problem

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 May Not Be Obvious

- Process resource needs are ever-changing
 - Depending on what data they are operating on
 - Depending on where in computation they are
 - Depending on what errors have happened
- Modern software depends on many services
 - Most of which are ignorant of one another
 - Each of which requires numerous resources
- Services encapsulate much complexity
 - We do not know what resources they require
 - We do not know when/how they are serialized

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

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 anything away from anyone, you're stuck

Deadlock Condition 4: Circular Waiting

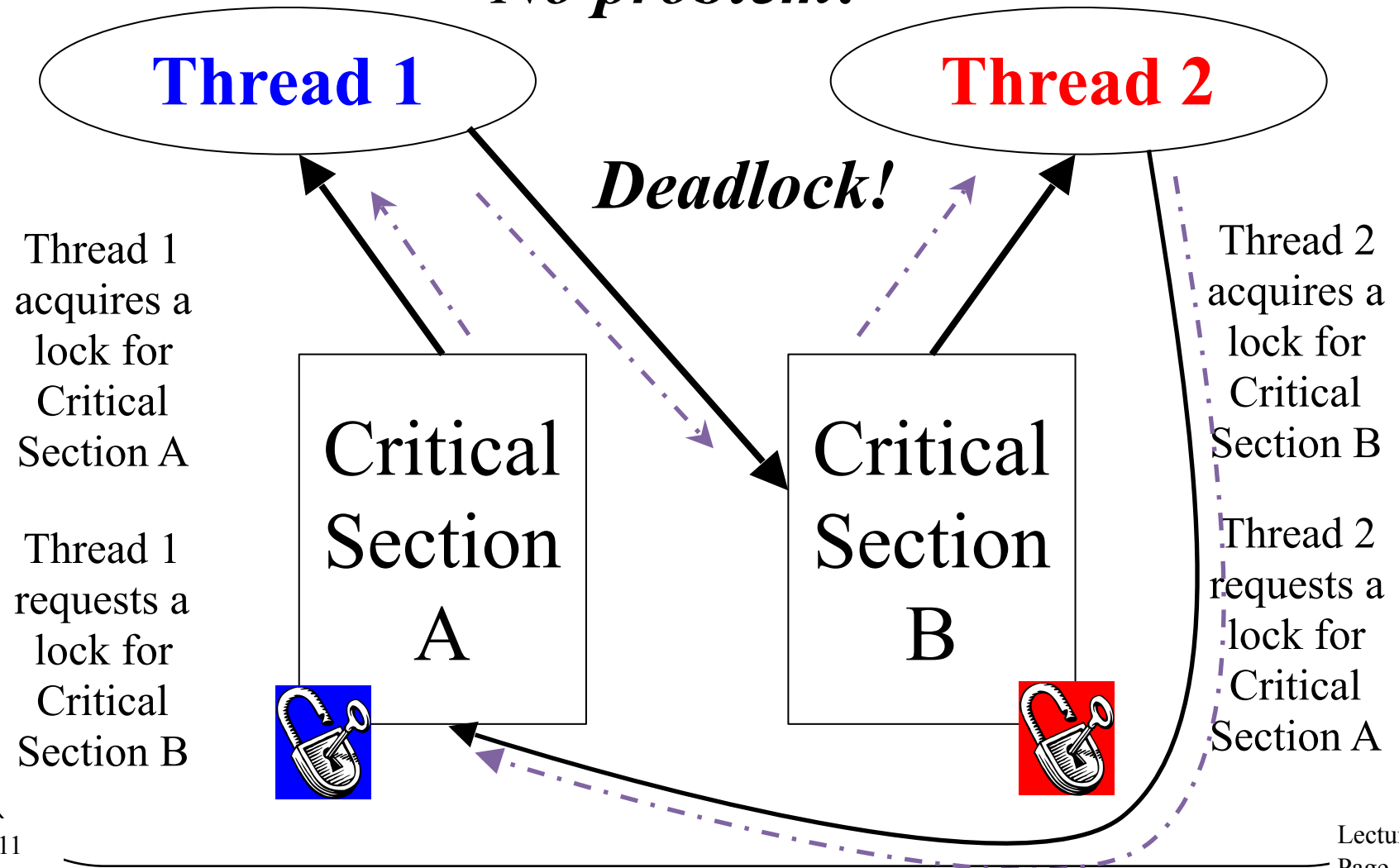
- A waits on B which waits on A
- In graph terms, there's a cycle in a graph of resource requests
- Could involve a lot more than two entities
- But if there is no such cycle, someone can complete without anyone releasing a resource
 - Allowing even a long chain of dependencies to eventually unwind
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

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

A Wait-For Graph

Hmmmm . . .

No problem!



Deadlock Avoidance

- Use methods that guarantee that no deadlock can occur, by their nature
- Advance reservations
 - The problems of under/over-booking
- 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 until 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
 - Memory reservations
 - Disk quotas, Quality of Service contracts
- Once granted, system must guarantee reservations
 - Allocation failures only happen at reservation time
 - One hopes before the new computation has begun
 - Failures will not happen at request time
 - System behavior is 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 to not 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

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 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

Note: deadlock solutions solve deadlocks – they don't necessarily solve all your other problems!

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:

- unlock buffer
- write lock list head
- search for desired buffer
- lock desired buffer
- remove from list
- unlock list head

Because we can't lock the list head while we hold the buffer lock

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

- To detect all deadlocks, need to identify all resources that can be locked
- Must 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?

Dealing With General Synchronization Bugs

- 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 that will never happen
 - E.g., the sleep/wakeup race we talked about earlier
- Priority inversion hangs
 - Like the Mars Pathfinder case
- None of these is a true deadlock
 - Wouldn't be found by a deadlock detection algorithm
 - But all 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)

Making Synchronization Easier

- Locks, semaphores, mutexes are hard to use correctly
 - Might not be used when needed
 - Might be used incorrectly
 - Might lead to deadlock, livelock, etc.
- We need to make synchronization easier for programmers
 - But how?

One Approach

- We identify shared resources
 - Objects whose methods may require serialization
- We write code to operate on those objects
 - Just write the code
 - Assume all critical sections will be serialized
- Compiler generates the serialization
 - Automatically generated locks and releases
 - Using appropriate mechanisms
 - Correct code in all required places

Monitors – Protected Classes

- Each monitor object has a semaphore
 - Automatically acquired on any method invocation
 - Automatically released on method return
- Good encapsulation
 - Developers need not identify critical sections
 - Clients need not be concerned with locking
 - Protection is completely automatic
- High confidence of adequate protection

Monitors: Use

```
monitor CheckBook {  
    // class is locked when any method is invoked  
    private int balance;  
    public int balance() {  
        return(balance);  
    }  
    public int debit(int amount) {  
        balance -= amount;  
        return( balance)  
    }  
}
```

Monitors: Simplicity vs. Performance

- Monitor locking is very conservative
 - Lock the entire object on any method
 - Lock for entire duration of any method invocations
- This can create performance problems
 - They eliminate conflicts by eliminating parallelism
 - If a thread blocks in a monitor a convoy can form
- No free lunch
 - Fine-grained locking is difficult and error prone
 - Coarse-grained locking creates bottle-necks

Java Synchronized Methods

- Each object has an associated mutex
 - Only acquired for specified methods
 - Not all object methods need be synchronized
 - Nested calls (by same thread) do not reacquire
 - Automatically released upon final return
- Static synchronized methods lock class mutex
- Advantages
 - Finer lock granularity, reduced deadlock risk
- Costs
 - Developer must identify serialized methods

Using Java Synchronized Methods

```
class CheckBook {  
    private int balance;  
    // object is not locked when this method is invoked  
    public int balance() {  
        return(balance);  
    }  
    // object is locked when this method is invoked  
    public synchronized int debit(int amount) {  
        balance -= amount;  
        return( balance)  
    }  
}
```