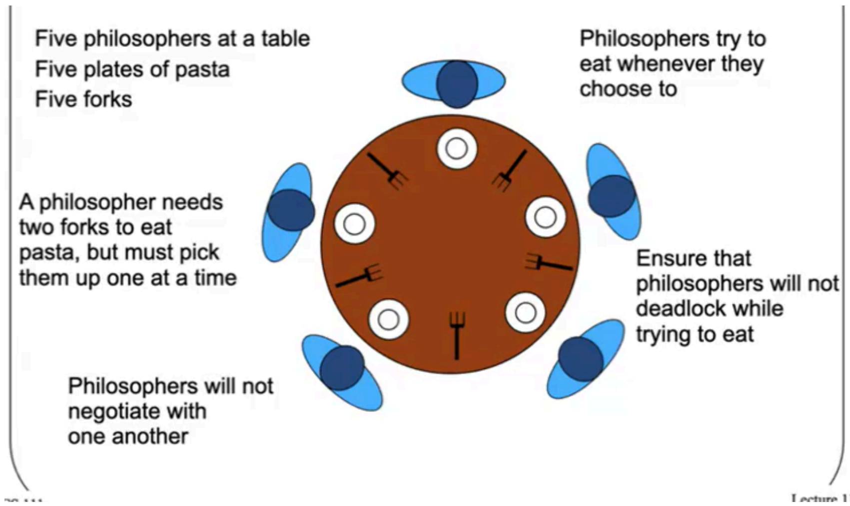
Deadlock

Deadlock
- Each resource holds its lock and does not release it since it needs a lock that someone else holds it



- Problem was carefully designed to cause deadlocks

Deadlock Issues
- Major problem for parallel interpreters
  - Relatively common in complex apps and results in large failures

- Difficult to find through debugging
  - Happen intermittently and are much easier to prevent at design time

- Process resource needs continue to change
  - Depends on what data they are operating on
  - Depends on where in computation they are at and what errors have occurred

- Modern software has many services
  - Many services are not aware of one another
  - Each service has complexity that we aren't aware of (resource usage or how they are serialized)

Resource Types
- Commodity Resources
  - Clients need an amount of a certain resource (memory)
  - Deadlocks result from over commitment
  - Avoidance done in resource manager

- General Resource
  - Clients need a very specific instance of something
  - Ex. Particular file

Basic Deadlock Conditions

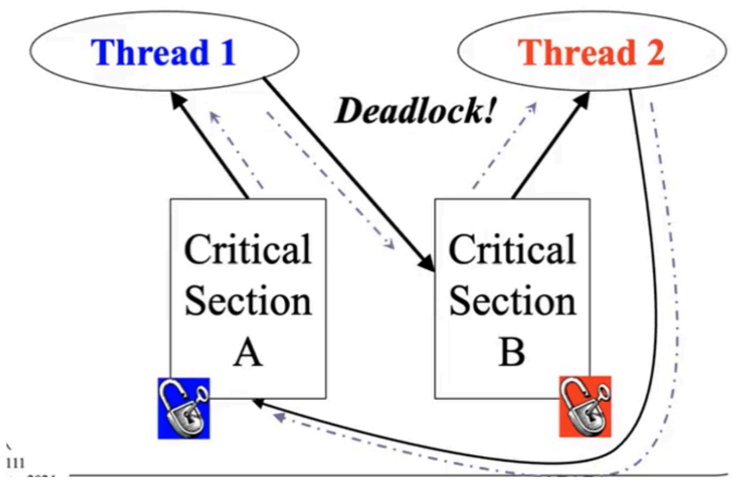
- For deadlock, one of the 4 can occur:

  Mutual Exclusion
  - Resource can only be used by one entity at a time
  - If multiple entities can use a resource, then it can be given to all

  Incremental Allocation
  - Processes are allowed to ask for resources whenever they want
  - However, if all locks were asked for at once, then no deadlock occurs

  No Pre-emption
  - When an entity has reserved a resource, can't take it away
  - Deadlocks can be resolved by taking away resources

  Circular Waiting
  - Each process waits on a resource that someone else holds
  - Wait for graph:



    - 2 Threads, 2 critical sections
    - When thread A gets a lock for section A, draw arrow to thread 1 from section A
    - When thread B gets a lock for section B, draw an arrow to thread 2
    - If thread 1 tries to get the lock for B, then draw arrow from thread 1 to section B (indicates waiting for graph B)
    - If thread 2 tries to get lock for A, then draw arrow from 2 to A
    - This forms a cycle

Deadlock Avoidance

- Use methods that ensure that no deadlock can occur
- Use advance reservations where no object can over reserve resources

  Using reservations
  - Advance reservations for <u>commodity resources</u>
    - Overbooking is ok for commodity resources
    - Only grant reservations if resources available
  - Over subscriptions are detected early before processes ever receive resources

  Overbooking vs Under Utilization
  - Processes cannot perfect predict resource needs ahead of time
  - Ensure they have enough (tend to ask for more than needed)
  - Either the OS
    - Grants requests until everything is reserved
    - Continue granting requests beyond available amount (since not all resources reserved are used)

  Handling Reservation Failures
  - Reservation eliminates deadlock
  - Apps must handle reservation failures
    - App design should handle failures gracefully
    - App must have a way of reporting failure to requester
    - App must continue running (all critical resources were reserved at start up time)

  Rejecting Requests
  - Better to inform the process they can't have a resource than to promise it when it's been reserved for something else
  - Given advance notice, app may be able to adjust service

Deadlock Prevention

- Commodity resource locking doesn;t work on general resource locking where there is a general lock on some resource
- Deadlock avoidance tries to ensure no lock can cause a deadlock
- Deadlock prevention tries to ensure that a particular lock doesn't cause a deadlock
- Targets the four necessary conditions for deadlock, and if any one of these conditions doesn't occur then deadlock can't occur

  Mutual Exclusion
  - Deadlock requires mutual exclusion, where P1 has the resource before P2 gets it
  - Can't deadlock on a shareable resource
    - Use atomic instructions

    - Raider writer locking can help (readers can share, writers handled in other ways)
- Can't deadlock process's private resource (giving each process private resource not always practical)

Incremental Allocation

- Deadlock requires you to block holding resources while you continue to ask for more locks

1. Allocate all resources one operation
    a. Must get all locks at once, otherwise failure
    b. All or nothing

2. Non Blocking requests
    a. A request that can't satisfied immediately will fail

3. Disallow blocking while holding resources
    a. Must release all held locks prior to blocking (can't hold locks while spinning)
    b. Reacquire them after you return to running the process
    c. Doesn't solve the problem when you return, the locks you need might not be free

No Pre-emption

- Deadlock can be broken by resource confiscation (steal locks)
    - Resource leases with time outs and lock breaking
        - Lease: limited time access to a resource (results in issue of partially completed tasks)
        - Lock breaking: Resource can be seized and reallocated

- Revocation must be enforced
    - Invalidate previous owners resource handle
    - Reset the partially completed process after its lock is taken
    - If revocation 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 repair resource

- Resources are designed with revocation
    - If a process crashes, one option is to kill the rest

OS seizing resources

- Process has to use a system service to access the resource, then the service can stop requests
- OS can't revoke a process access if the process has direct access
    - Object is part of address space
    - If the resource isn't controlled by the OS, OS can't handle it

Circular Dependencies

- Total resource ordering
    - All requesters allocate resources in same order
    - Allocate R1 and then R2 after
    - Someone else may have R2 but doesn't need R1

- Assumes we know the exact order of resources
    - Order by resource type (groups before members)
    - Order by relationship (parents before children)

- Lock dance: similar to round robin
    - Release R2, allocate R1, require R2

  Lock Dance:
    - Must acquire lock on list head
    - Individual buffers also have locks
    - In order to get the lock for a buffer, the process needs the lock or the list head

Approaches to solving deadlock

- No one universal solution
- Sole each individual problem in any way
    - Making resources shareable when possible
    - Use reservations for commodity resources
    - Ordered locking or no hold and block where possible
    - Last resort: leasing and lock breaking

Ignoring Deadlock

- If it doesn't occur much, we don't implement any anti-deadlock measure

Implement Deadlock Detection

- Identify all resources that can be locked
    - Not always clear in an OS
    - Hard to determine for app level locks
- Deadlocks outside OS difficult to detect
- Other synchronization problems such as bugs in app code can cause issues

Health Monitoring

- Deadlock detection rarely makes sense since its hard to implement and difficult to fix even when detected
- Service/app level health monitoring is better
    - Monitor app progress
    - If response takes too long, service is marked as “hung”
- Health mentoring easy to implement
- Can detect a variety of problems
    - Deadlocks, live locks, etc

Live-lock

- Process is running but won’t free R1 until it gets message
- Process will send the message is blocked for R1
- Process blocked and waits for completion that never happens

Priority Inversion

- Higher level process holding lock can’t run due to long running lower priority process

Monitoring Process Health

- Look for process exists or core dumps
- Passive observation
    - Process consuming CPU time or is blocked
    - PRocess doing network or disk
- External monitoring
    - Test for responses from a system (if a response isn;t received, problem occurred)
    - Pings, null requests, standard test requests
- Internal instrumentation
    - White box audits and monitoring

Handling Unhealthy Processes

- Kill and restart “all of the affected software”
- Kill as few processes as possible
- Apps designed to react to kills and restarts
- Highly available systems define restart groups
    - Group of processes to be started or killed as a group
    - Defines inter group dependencies

Failure Recovery

- Roll back failed operations and return an error
- Continue with reduced capacity
    - Accept requests that can be handled and reject those who can’t
- Automatic restarts

        - Escalation mechanisms (partial restarts, then escalating) such as restarting more groups

Making Synchronization Easier

- Identify shared resources
    - Identify what methods require serialization
- Write code to operate on those objects, and assume critical sections will be serialized
- Compiler generates the serialization automatically
    - Automatically generated locks and releases

Monitors

- Protected Classes

- Each monitor object has a mutex
    - Automatically acquired on any method invocation
    - Automatically released on method return
- Good encapsulation
    - Developers don't need to identify critical sections
    - Automatic locking when modifying a monitor object

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

Java Synchronized Methods

- Each object has an associated mutex
    - Only acquired for specific mutex
- Static synchronized methods lock class mutex
- Advantages
    - Fine grain locks and reduced deadlock risk
- Costs
    - Developer must identify serialized methods

- Object is only locked when the method attempts to modify it

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)
    }
}
```