Deadlocks

- 8A Introduction to Deadlocks
- 8B Deadlock Avoidance
- 8C Deadlock Prevention
- 8D Detection and Recovery
- 8E Priority Inversion

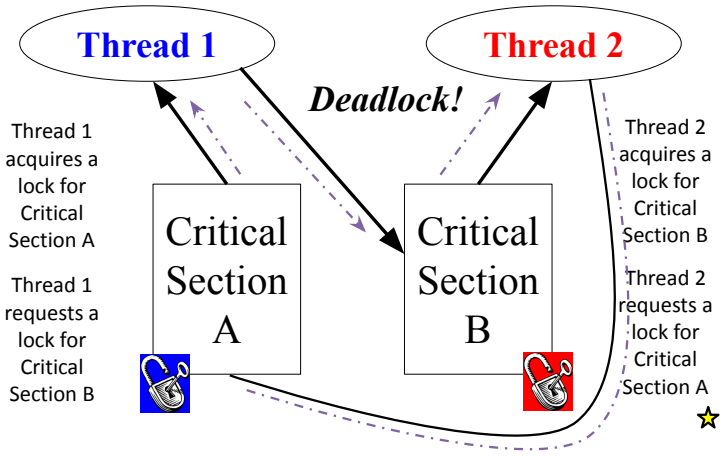
Why Study Deadlocks?

- 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

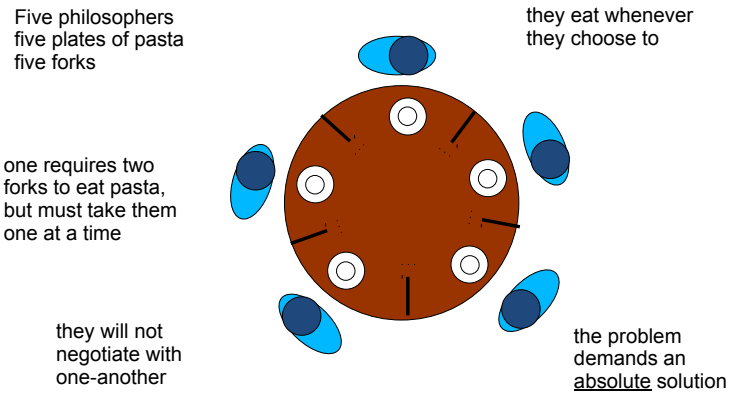
What is a Deadlock?

- Two (or more) processes or threads
  - cannot complete without all required resources
  - each holds a resource the other needs
- No progress is possible
  - each is blocked, waiting for another to complete
- Related problem: livelock
  - processes not blocked, but cannot complete
- Related problem: priority inversion
  - high priority actor blocked by low priority actor

Resource Dependency Graph



The Dining Philosophers Problem



(The Dining Philosophers Problem)

- 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 eliminate deadlocks
  - but then it couldn't be used to illustrate deadlocks

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

Many Types of Deadlocks

- Different deadlocks require different solutions
- 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 needs memory
- Producer-consumer deadlocks
  - e.g. P1 needs a file, P2 needs a message from P1

Basic Approaches

- Avoidance
  - evaluate each proposed action
  - avoid taking actions that would deadlock
- Prevention
  - design system to make deadlock impossible
- Detection and Recovery
  - wait for it to happen
  - try to detect that it has happened
  - take some action to break the deadlock

Commodity vs. General Resources

- 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 the dependency network
  - prevention is usually done at design time

Commodity Resource Problems

- memory deadlock
  - we are out of memory
  - we need to swap some processes out
  - we need memory to build the I/O request
- critical resource exhaustion
  - a process has just faulted for a new page
  - there are no free pages in memory
  - there are no free pages on the swap device

Avoidance – Advance Reservations

- advance reservations for commodities
  - 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

Real Commodity Resource Management

- advanced reservation mechanisms are common
  - Unix setbreak system call to allocate more memory
  - disk quotas, Quality of Service contracts
- once granted, reservations are guaranteed
  - 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 Rejection

- reservations eliminate difficult failures
  - recovering from a failure in mid-computation
  - may involve awkward and complex unwinding
- graceful handling of reservation failures
  - fail new request, but continue running
  - try to reserve essential resources at start-up time
- keep trying until it works ... not so good
  - may impose un-bounded delay on requestor
  - freeing resources or shedding load could help

Pre-reserving critical resources

- system services mustn't 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, can't swap out processes
- 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

Over-Booking vs. Under Utilization

- Problem: reservations overestimate reqts
  - clients seldom need all resources all the time
  - all clients won't need max amount at 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 may have to wait for others to complete
  - we must be sure there are no deadlocks

Deadlock Prevention

- Deadlock has four necessary conditions:
  1. **mutual exclusion**
     P1 cannot use a resource until P2 releases it
  2. **hold and wait**
     process already has R1 blocks to wait for R2
  3. **no preemption**
     R1 cannot be taken away from P1
  4. **circular dependency**
     P1 has R1, and needs R2
     P2 has R2, and needs R1

Attack #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 in other ways
- you can't deadlock if you have private resources
  - can we give each process its own private resource?

Attack #2: hold and block

deadlock requires you to block holding resources

1. allocate all resources in a single operation
   - you hold nothing while blocked
   - when you return, you have all or nothing
2. disallow blocking while holding resources
   - you must release all held locks prior to blocking
   - reacquire them again after you return
3. non-blocking requests
   - if it can't be satisfied immediately, it will fail

Attack #3: non-preemption

- deadlock prevents forwards progress
  - can we *back-out* of the deadlock?
  - reclaim resource(s) from current holders
- use *leases* rather than locks
  - process only has resource for a limited time
  - after which ownership is automatically lost
- forceful resource confiscation
- termination ... with extreme prejudice

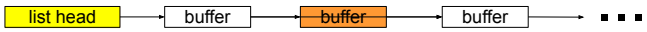
When is Preemption Feasible?

- Is access mediated by the operating system?
  - e.g. all object access is via system calls
  - we can revoke access, and return errors
- Can we force a graceful release of resource?
  - make a *claw-back* call to the current owner
- Does confiscation leave resource corrupted?
  - we can un-map a segment or kill a process
  - can we return resource to a default initial state?
  - is it protected by all-or-none updates?

Attack #4: circular dependencies

- 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 ID (e.g. I-node #, IP-address, mem address)
  - 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" to preserve ordering



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:

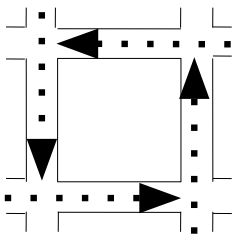
read lock list head
search for desired buffer
lock desired buffer
unlock list head
return (locked) buffer

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

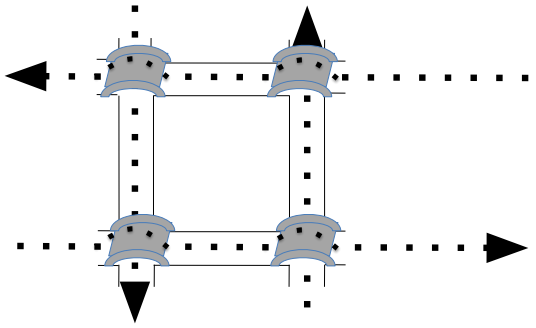
Deadlock – Practical Examples

- the problem – urban gridlock
  - resource: being in the intersection
  - deadlock: nobody can get through



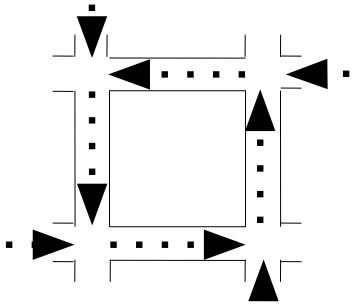
Prevention: Mutual Exclusion

- Build overpass bridges for east/west traffic



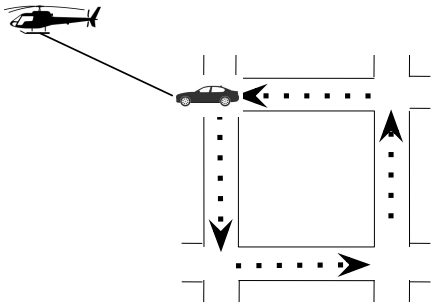
Prevention: Hold and Block

- illegal to enter the intersection if you can't exit
  - thus, preventing "holding" of the intersection



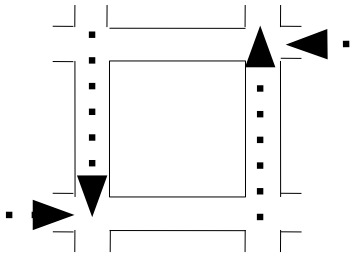
Prevention: Preemption

- Helicopters forcibly remove blocking vehicles



Prevention: Circular Dependencies

- decree a total ordering for right of way
  - e.g., North beats West beats South beats East



Deadlocks: divide and conquer!

- There is no one universal solution to all deadlocks
  - fortunately, we don't need a universal solution
  - 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

Closely related forms of "hangs"

- live-lock
  - process is running, but won't free R1 until it gets msg
  - 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

Deadlock vs. "hang" detection

- deadlock detection seldom makes sense
  - it is extremely complex to implement
  - only detects true deadlocks for known resources
- service/application "health monitoring" does
  - monitor 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

Hang/Failure Detection Methodology

- 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

Automated Recovery

- kill and restart “all of the affected software”
- how will this affect service/clients
  - design services to automatically fail-over
  - components can warm-start, fall back to last check-point, or cold start
- which, and how many processes to kill?
  - define service failure/recovery zones
  - processes to be started/killed as a group
  - progressive levels of increasingly scope/severity

When formal detection makes sense

- Problem: Priority Inversion (a demi-deadlock)
  - preempted low priority process P1 has mutex M1
  - high priority process P2 blocks for mutex M1
  - process P2 is effectively reduced to priority of P1
- Consequences:
  - depends on what high priority process does
    - might go unnoticed
    - might be a minor performance issue
    - might result in disaster

Priority Inversion on Mars

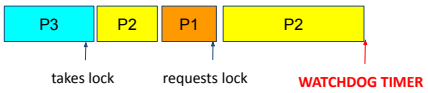
- occurred on the Mars Pathfinder rover
- caused serious problems with system resets
- very difficult to find

The Pathfinder Priority Inversion

- Special purpose h/w, VxWorks real-time OS
- preemptive priority scheduling
  - to ensure execution of most critical tasks
- shared an “information bus”
  - shared memory region
  - used to communicate between components
  - shared data protected by a mutex lock

A Tale of Three Tasks

- P1: critical, high priority bus management task
  - ran frequently for brief periods, holding bus lock
  - watchdog timer made sure that P1 was still running
- P3: low priority meteorological task
  - ran occasionally, for brief periods, briefly holding bus lock
- P2: medium priority communications task
  - ran rarely, for longtime, did not need or hold bus loc
- A very unlikely race condition:
  - P3 had the lock, and was preempted by P2
  - P1 can preempt P2, but blocks until P3 completes
  - P1 is now waiting for (much lower priority) P3
  - watchdog timer concludes P1 has failed, resets system



Solution: Priority Inheritance

- Identify resource that is blocking P1
- Identify current owner of that resource (P3)
- Temporarily raise P3 priority to that of P1
  - P3 now preempts P2, runs to completion
  - P3 releases lock, and loses inherited priority
  - P1, now un-blocked is highest priority process
  - P2 resumes and completes execution
  - P3 resumes execution

Reading and Assignments

Reading:

- Arpaci C35 … introduction to storage
- Arpaci C36 … devices
- Arpaci C37 … disks

Prep for Mid-Term