Overview of Presentation

- Overview of dead-locks
- Deadlock avoidance (advance reservations)
- Deadlock prevention (four necessary conditions)
- Related subjects
  - Other types of hangs
  - Detection and recovery
  - Priority inversion

Why Study Deadlocks?

- A major peril in cooperating parallel processes
- They result in catastrophic system failures
- They generally result from careless design
- Finding them through debugging is very painful
- It is much easier to prevent them at design time
- An ounce of prevention is worth a pound of cure
- If you understand them, you can avoid them

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

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 (e.g. files, mutexes)
  - deadlocks result from specific dependency network
  - Prevention often requires help from clients

Avoidance – Advance reservations

- Require advance reservations for commodities
- Resource manager tracks outstanding promises
- Only grants reservations if resources are available
- Over-subscriptions are detected before client runs
  - before client has yet allocated any resources
- Refused reservation failures must still be handled
  - But these do not result in deadlocks
- Dilemma: over-booking vs under-utilization

Achieving better resource utilization

- Problem: reservations overestimate requirements
  - 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?
- 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

Bankers' Algorithm - Assumptions

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

Bankers' 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 to allow at least one client to finish
  - after that 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"

Bankers' Algorithm – limitations

- Quantified resources assumption
  - not all resources are measurable in units
  - other resources can introduce circular dependencies
- Eventual completion assumption
  - all resources are released when client completes
  - completion is a transaction or batch notion
  - In a time sharing system many tasks run for months
- Likelihood of resource "convoy" formation
  - reduced parallelism, reduced throughput

Practical Commodity Management

- Advanced reservations are definitely useful
  - e.g. Unix setbreak system call
- System should guarantee all reservations
  - allocation failures only happen at reservation time
  - failures will not happen at request time
  - system behavior more predictable, easier to handle
- Clients must deal with reservation failures
  - hang onto resources and try again later doesn't cut it
  - they must complete (or abort) with current resources
  - If they can't they should have reserved more up front!

Prudent Advance 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
- 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

General Resource deadlocks

- Necessary condition #1: mutual exclusion
  - If P_1 is using resource, P_2 cannot use it
- Necessary condition #2: block holding resources
  - Process already has R_1 blocks to wait for R_2
- Necessary condition #3: circular dependencies
  - P_1 has R_1 and needs R_2 while P_2 has R_2 and needs R_1
- Necessary condition #4: no preemption/revocation
  - P_1 has R_1 and it can only be freed when P_1 completes

Attack #1 – Mutual Exclusion

- Deadlock requires mutual exclusion
  - P_1 having the resource precludes P_2 from getting it
- You can't deadlock over a shareable resource
- Whenever possible, make resources sharable
  - maintain with atomic instructions
- Even reader/writer locking can help
  - readers become a non-problem
  - writers may be attacked in other ways

Attack #2: hold and block

- Deadlock requires you to block holding resources
- Allocate all of your resources in a single operation
  - you hold nothing while blocked
  - when you return, you have all or nothing
- Disallow blocking while holding resources
  - you must release locks prior to blocking
  - reacquire them after you resume (lock dance)
- Non-blocking requests
  - a request that can't be satisfied immediately will fail

Attack #3: circular dependencies

- Global resource ordering
  - all requesters allocate resources in same order
  - first allocate R_1 and then R_2 afterwards
  - someone else may have R_2 but he doesn't need R_1
- assumes we know how to order the resources
  - some objects have natural ordering (e.g. processes)
  - relationships may order (e.g. parents before children)
- may require a lock dance
  - release R_2, allocate R_1, reacquire R_2

Attack #4: no preemption/revocation

- deadlock can be averted by resource confiscation
  - time-outs and lock breaking
  - resource can be reallocated to new client
  - old client gets "stale handle" error and must restart
- current owner can be killed
  - all resources will be reclaimed from the corpse
  - this is an extreme measure, use it sparingly

Who can prevent deadlocks?

- advance reservations
  - operating system, basic APIs, resource managers
- eliminating mutual exclusion
  - application developer
- eliminating blocks while holding resources
  - operating system, basic APIs
- eliminating circular dependencies
  - application developer, resource managers
- implementing revocation
  - operating system, basic APIs, resource managers

Divide and Conquer!

- You don't have to pick one solution for all resources
- You have to solve the problem for each resource
- Solve individual problems any way that you can
  - resource hoards for key system services
  - reservations for commodity resources
  - sharable resources where feasible
  - ordered allocation where feasible
  - lock breaking when nothing else will work
- OS is only responsible for deadlocks in OS services
  - applications are responsible for their own behavior

Closely related forms of "hangs"

- live-lock
  - process not blocked, but won't free R_1 until it acquires R_2
- sleeping beauty
  - process is blocked, awaiting a particular message
  - message must be sent by another process
  - for some (unknown) reason, the message is never sent
- program goes rogue
  - program continues to run, but ceases to do useful work
- none of these are true deadlocks
  - but they all leave you just as hung

Deadlock detection vs "hang" detection

- Deadlock detection seldom makes sense
  - it is extremely complex to implement
  - only detects "true deadlocks" in enumerated resources
- Service/application "health monitoring" does
  - monitor progress or submit test transactions
  - if progress is too slow, declare service to be "hung"
  - this is very easy to implement
  - detects and cures a wide range of problems

Deadlock/Hang Recovery

- recovery should be automatic
- determine which service has failed
  - presumably the service whose audit timed out
  - this may be a symptom of some other failure
- primary recovery
  - kill and restart the failed application
  - switch-operation over to a hot-stand-by
- determine efficacy of primary recovery
  - retries, and escalations

When does formal detection make sense?

- Problem: Priority Inversion (a demi-deadlock)
  - low priority process P_1 has mutex M_1 and is preempted
  - high priority process P_2 blocks for mutex M_1
  - process P_2 is effectively reduced to priority of P_1
- Solution: mutex priority inheritance
  - detect when the 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

Deadlock – wrap-up

- deadlocks are a real and common hazard
  - we must be wary of the danger, and prevent them
- there are different types of resources
  - they are subject to different types of deadlocks
- there are several techniques for averting deadlock
  - attack the four necessary conditions for deadlock
- deadlock detection is generally not practical
  - hang detection is more useful
  - recovery mechanisms are also very important