Deadlocks

Introduction

- In multitasking or multiprogramming system
  - Several processes can compete for finite number of resources
- A process requests resources
  - If not available process placed in wait state
  - If resources never become available
    - Process remains waiting
    - Called *deadlock*
- Illustration
  - Law passed in Kansas around turn of century
    - When two trains approach each other at crossing
      - Each shall come to full stop
      - Neither shall start up again until the other has gone
- We talked about deadlocks earlier
- Will now look at methods to deal with problem
- Note
  - Most contemporary operating systems
    - Do not use deadlock prevention techniques
- As systems become more complex and number of processes increases
  - Problem will need to be addressed

System Model

- System has finite number of resources
  - These distributed among number of competing processes
- Resources partitioned into several types
  - <u>Identical resources</u>
    - May have multiple instances of same resource
      - Printers
      - Telecomm channels
      - Memory space
    - Allocation of any one may be sufficient
    - Note printers may be identical
      - If convenience to user compromised
        - May not be considered identical
          - Printer on 1 and 9th floors of office building
  - <u>Dissimilar resources</u>
    - Second kind of resource
      - Those that are unique for one reason or another
        - Single copy
        - Identical printers for example may not be identical
          - If convenience to user compromised
            - May not be considered identical
        - Printer on 1 and 9th floors of office building

To use resource process must request resource
May request as many resources as it wishes
May not exceed total number

Under normal operation may only utilize resources in following order
Request
If can't be granted immediately
Requesting process must wait
Use
Process operates on or uses resource
Release
When finished give up resource

Request and release are system calls

Deadlock Characterization

Set of processes in deadlock state
Every process in set is waiting for event
Can be caused only be another process in set
Events of concern
Resource acquisition and release

Necessary Conditions

For deadlock to occur following must hold simultaneously
Note these are necessary not sufficient
Note also these are not independent

Mutual Exclusion
A least one resource held in non-sharable mode
Hold and Wait
Must be process holding resource and waiting for additional
Being held by other processes
No Preemption
Resources cannot be preempted
Circular Wait
Set of processes {P0...Pn} such that
P0 waiting for resource held by P1
P1 waiting for resource held by P2
...

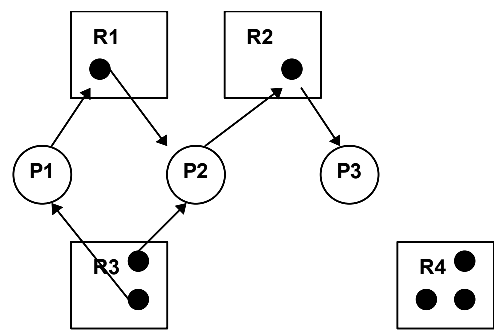
Resource Allocation Graph

Can understand deadlock formally using
Resource Allocation Graph
Resource allocation graph
Directed graph
Set of

Vertices V
- Partitioned into two sets
  - Set of processes {Pn}
  - Set of resources {Rm}

Edges E
- Connecting
  - {Pn} to {Rm}
  - {Rm} to {Pn}
- Directed edge form Pi to Rj
  - Pi $\rightarrow$ Rj
    - Signifies Pi requested Rj and is currently waiting
    - Called *request* edge
- Directed edge form Rj to Pi
  - Rj $\rightarrow$ Pi
    - Signifies Rj allocated to Pi
    - Called *assignment* edge

Graphically
- Process
  - Circle
- Resource
  - Rectangle
  - Multiple copies
    - Signified by dot in rectangle



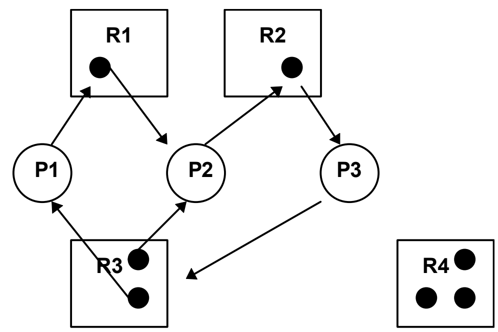
- Consider following RAG
  - We have the following situation
    - Sets
      - P = {P1, P2, P3}
      - R = {R1, R2, R3}
      - E = {P1$\rightarrow$ R1, P2 $\rightarrow$R2, R1$\rightarrow$P2, R2$\rightarrow$ P3, R3 $\rightarrow$P1, R3 $\rightarrow$P2}

    - Resource Instances
      - 1 of R1
      - 1 of R1
      - 2 of R3
      - 3 of R4

    - Process States
      - P1
        - Holding 1 R3
        - Waiting for R1
      - P2
        - Holding 1 R1 and 1 R3
        - Waiting for R2

P3
Holding 1 R2

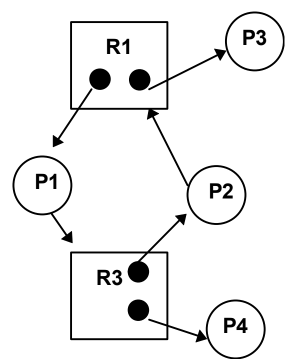
Using techniques from graph theory
Can show if contains no cycles
No process in system is deadlocked
If cycle exists
Potential for deadlock exists
Does not guarantee
If single instance of each resource
Cycle
Implies deadlock has occurred
Becomes necessary and sufficient condition
If multiple instances
Cycle
Does not necessarily imply deadlock
Necessary but not sufficient condition



Let's look at two examples
First has cycle and deadlock

Second has cycle and no deadlock

Handling Deadlocks



Let's now look at some ways of dealing with
Deadlock problem

Several ways
Use protocol to ensure deadlock will never happen
Allow system to enter deadlock state and recover
Ignore problem
Solution used by most operating systems
Including UNIX

Ensuring No Deadlock

Can use
Deadlock prevention
Ensure one of necessary conditions cannot occur
Deadlock avoidance
Requires additional information
Which resources process will request and use
During lifetime

Deadlock Prevention

- Let's look at prevention first
  - Easiest solution

- *Mutual Exclusion*
  - Must hold for non sharable resources
    - Single printer
  - Sharable resources
    - Mutual exclusion not required
    - Read only files
  - Cannot prevent deadlocks by denying mutual exclusion
    - Some resources inherently non-sharable

- *Hold and Wait*
  - To prevent hold and wait condition
    - Must guarantee when process requests resource
      - Does not hold any other resources
  - Protocol 1
    - Request and be allocated all resources
      - Before execution
  - Protocol 2
    - Can only request resources when have none
      - Can request resources and be allocated
      - To request additional
        - Must give up what have
  - Two main disadvantages
    - Resource utilization low
      - Allocated but not used for long time
    - Starvation possible
      - Process needing popular resources may have to wait indefinitely

- *No Preemption*
    - To prevent no preemption condition
      - Protocol 1
        - If holding resources and need more that are not available
          - Process must wait
        - All resources currently being held
          - Preempted
          - Added to list of resources for which process is waiting
        - Process restarted when it can
          - Regain old resources
          - Acquire new ones it requested

      - Protocol 2
        - If process requisite resources

- Check
  - If available
    - Allocate
  - else if with another process waiting for resources
    - If with another waiting process
      - Preempt
      - Allocate to requesting process
  - else
    - wait

Circular Wait

- To prevent circular wait
  - Place total ordering on all resources
  - Require each process to request resources
    - Increasing order of enumeration
- Let R = {R1, R2, ...Rm} be set of resource types
  - Assign each type unique integer number
    - Allows ordering relation to be applied and evaluated
- Protocol 1
  - Initially request any desired resources
  - Additional resource requests
    - Only in increasing order of enumeration
  - If multiple copies of single resource needed
    - Must request all at once
- Protocol 2
  - Initially request any desired resources
  - Additional resource requests
    - If request Rj
      - Must release any resources {Ri} such that i <= j

Deadlock Avoidance

- Deadlock prevention algorithms
  - Prevent deadlocks by restraining requests
  - Restraints ensure
    - At least one of necessary conditions cannot occur
  - Consequence
    - Low utilization of resources
- Deadlock avoidance
  - Requires additional information
    - About how resources requested
      - Consider system
        - Resources
          - Printer
          - Tape drive
        - Processes

  - P1 and P2
- Need
  - P1
    - Printer then tape drive
  - P2
    - Tape drive then printer
- Knowledge
  - Knowing need in advance
    - Permits scheduling to ensure no deadlock

Various algorithms
- Require differing amounts of information
- Let’s walk through simple one to get idea

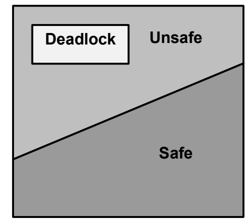
Declare in Advance
- Simplest most useful model requires each process to declare
  - In advance
  - Maximum number of resources
    - Of each type it may need
- Given such information
  - Possible to construct algorithm
    - To ensure system will never enter deadlock state
- Such a scheme defines basis for deadlock avoidance
  - Avoidance algorithm
    - Examine resource allocation state
      - Defined by number of
        - Available and allocated resources
        - Max number of demands by processes

Safe State
- Resource allocation state is *safe*
  - System can
    - Allocate resources to each process
    - In some order
    - Avoid a deadlock

- Formally
  - System is in *safe state*
    - If there exists a *safe sequence*
  - Sequence of processes <P1, P2...Pn> is safe sequence
    - For current allocation state
    - If
      - For each Pi
        - Resources that Pi can still request can be satisfied by
          - Currently available resources plus
          - Resources held by all Pj such that j < i
        - Observe if needed resources not available

Pi can wait until Pj have finished
Can then have all needed resources
When Pi finishes Pi+1 obtain needed resources
If no such sequence exists
System state is unsafe

Observe
Safe state is not deadlock state
Deadlock state is unsafe state
Not all unsafe states are deadlock states
Unsafe state may lead to deadlock
Three spaces illustrated as



Example

Consider following system

12 I/O Ports
3 Processes
Let max and current needs be given as

	Max Needs	Current Needs
P0	10	5
P1	4	2
P2	9	2

We have total allocation of 9 with 3 ports free
At time t0
System in safe state
Sequence <P1, P0, P2>
Safe sequence
Can satisfy P1
Block P0
Until P1 finished
Block P2
Until P1 finished
At time t1
System can go to unsafe state
Let P2 requests additional port
Only P1 can be allocated all resources
When it returns them
Only 4 total available
P0 allocated 5 ports
Max need of 10
May request 5 more

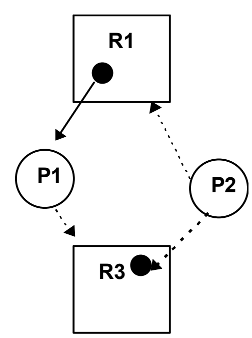
- Not available so block
- P2 may request additional 6
  - Not available so block
    - Deadlock

Avoidance Algorithms

Resource Allocation Graph Algorithms

- If we have system with one instance of each resource
  - Can use variant on resource-allocation graph to avoid deadlocks
    - Introduce new edge type - *claim* edge
- Claim edge Pi →Rj
  - Indicates Process Pi may claim resource Rj sometime in future
  - Edge has semantics similar to request edge
    - Direction same
    - Notation is dashed line

  - Requires that resources be claimed a priori in system
    - Before process starts executing
      - All claim edges must be present in resource-allocation graph
        - Restriction may be relaxed to allow addition of claim edge
          - If all other edges from process are claim edges
- Protocol
  - When Pi requests Rj
    - Claim edge converted to request edge
  - Similarly when resource Rj released by Pi
    - Request edge converted to claim edge
  - Claim edge can only be converted into request edge
    - If conversion does not result in cycle
      - If no cycle exists
        - Allocation will leave system in safe state



  - Observe
    - If P2 requests and is allocated R3
      - Although available
      - Cannot allocate
      - Will create cycle and thus unsafe state
    - If P1 requests R3
      - We have a deadlock

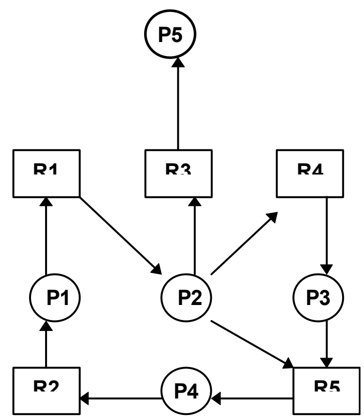
Deadlock Detection

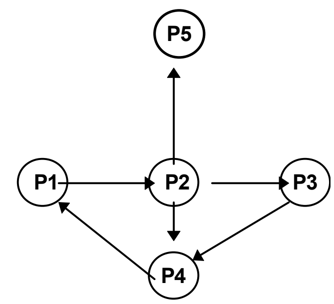
- If system does not employ prevention or avoidance algorithm
  - Deadlock may occur
- In such environment
  - System must provide
    - Algorithm to determine if deadlock has occurred

- Algorithm to recover from deadlock

Detection in Single Instance Environment

- As with avoidance
  - Can use variation on resource allocation graph
    - Called *wait-for* graph
- Algorithm
  - Start with resource-allocation graph
  - Remove nodes of type resource
    - Collapse appropriate edges
    - Will result in graph with only processed
  - Deadlock exists if and only if the graph contains a cycle





Deadlock Recovery

- When deadlock algorithm detects deadlock exists
  - Several possible alternatives
    - Inform user
      - Difficult in embedded system
    - Let system recover automatically
- Automatic recovery
  - Two general schemes
    - Abort
      - All processes
      - One at a time
    - Preempt resources

Process Termination

- All deadlocked processes
  - Will clearly break deadlock
    - At great expense
    - Processes may have computed for long time
    - All results may be lost

- One process at a time
  - Until deadlock cycle eliminated
  - Involves considerable temporal overhead
    - As each process aborted
      - Must rerun deadlock detection algorithm
  - Extreme care must be taken
    - Aborting process may leave resources in
      - Unknown or unusable state
  - Must also determine which process to abort
    - Similar to CPU scheduling problem
    - Want to abort processes in terms of increasing cost
    - Potential factors
      - Process priority
      - Time since start and remaining run time
      - Resource mix and quantity
      - Resource demand to complete
      - Number of processes to be terminated

Resource Preemption

- Method requires
  - Successive preemption of resources
  - Allocation to other processes
    - Until deadlock cycle broken
- If preemption used
  - Must consider three issues
    1. Selecting a victim
       - Must determine order of preemption to minimize cost
       - Factors include
         - Number or resources deadlocked process holding
         - Amount of elapsed execution time for deadlocked process

    2. Rollback
       - If resource preempted
         - What should be done with associated process
           - Cannot continue
       - Often cannot determine completely safe state
       - Simplest solution is complete rollback
         - Abort process and restart
       - Can try to roll back as far as necessary to break deadlock
         - Entails maintaining information on all running processes

    3. Starvation
       - How to ensure starvation will not occur
         - Want to ensure resources not always preempted from same process

Summary

- Deadlock occurs when two or more processes
  - Waiting for event that can only be caused by one of waiting processes
- 3 major methods for addressing
  - Use protocol to ensure will never enter deadlock state
  - Allow system to enter deadlock state and recover
  - Ignore problem

- Deadlock can only occur
  - If and only if 4 conditions occur simultaneously
- We prevent deadlock
  - Ensuring one condition will not occur

- If system does not employ protocol to ensure deadlock does not occur
  - Then detection and recovery scheme must be employed

- If deadlock detected
  - Can recover by global or selective termination
    - Process
    - Resources