G22.2250-001
Operating Systems

Lecture 6

CPU Scheduling (cont'd)
Process Deadlocks

October 9, 2007

Outline

Announcements

- Lab 2 was due last Friday
  - Please send me e-mail if you have not yet started this lab
- Today's lecture will end at 6:30pm

- CPU Scheduling (cont'd)
  - Scheduling algorithms – Round Robin
  - Example: Windows XP scheduler
  - Advanced topic: Real-time scheduling
- Process Deadlocks
  - System model
  - Characteristics of deadlocks: Graph representation
  - Methods for handling deadlocks
    - Deadlock prevention
    - Deadlock avoidance

[Silberschatz/Galvin/Gagne: Sections 5.3, 5.6 – 5.7, 7.1 – 7.5]

(Review)

Scheduling Algorithms

- CPU scheduling: Deciding which process to give the CPU to next
  - Process behavior modeled as alternating bursts of CPU and I/O activity
  - Algorithm attempts to meet different system and/or user objectives
    - Response time, waiting time, throughput, priorities, …

Algorithms

- First-come first-served (FCFS)
  - Simple to implement
  - Long waiting times for short jobs, which can be held back by long jobs
- Shortest Job First (SJF) and Shortest Remaining Time First (SRTF)
  - Prioritize jobs with shorter (remaining) CPU time bursts
  - Estimate burst time by exponentially averaging past bursts
- Priority-based, with or without preemption
  - Generalized notion of priority to give different preferences to jobs
  - Two problems: Starvation (fixed by priority ageing) and
    Priority inversion (fixed by priority inheritance)

(Review)

Scheduling Algorithms (4): Round Robin (RR)

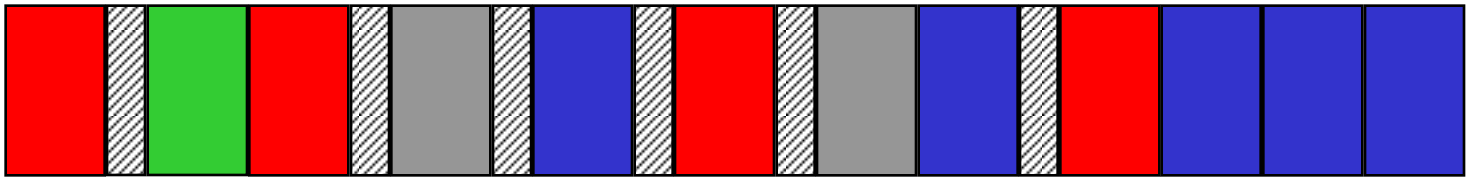
- A strictly preemptive policy

- At a general level
  - choose a fixed time unit, called a quantum
  - allocate CPU time in quanta
  - preempt the process when it has used its quantum
    - Unless the process yields the CPU because of blocking
  - typically, FCFS is used as a sequencing policy
    - each new process is added at the end of the ready queue
    - when a process blocks or is preempted, it goes to the end of the ready queue
  - very common choice for scheduling interactive systems

(Review)

Round-robin Scheduling: Example

- Consider five processes **A**, **B**, C, and **D**
  - With burst times: 4, 1, 2, 5
  - Arriving at times: 0, 0, 2, 3
- Round-robin system with quantum size 1 unit
  - Overhead of context switching a process: 0.2 units
    - Incurred only when a process is preempted or needs to block



Waiting time = ((0 – 0 + 6.2) + (1.2 – 0 + 0) + (3.4 – 2 + 2.6) + (4.6 – 3 + 3.6))/4 = **4.15** units

FCFS = (0 + (4-0) + (5-2) + (7-3))/4 = **3.75** units

Response time = ((0 + (1.2 – 0) + (3.4 – 2) + (4.6 – 3))/4 = **1.05** units

FCFS = (0 + (4-0) + (5-2) + (7-3))/4 = **3.75** units

CPU utilization?

Choice of Quantum Size

- Quantum size *q* is critical
- Affects waiting and turnaround times
  - if *q* is the quantum size and there are *n* processes in the ready queue,
    - the maximum wait is *(n-1) . q* units of time
  - as *q* increases, we approach FCFS scheduling
  - as q decreases
    - ⬇ the rate of context switches goes up, and the overhead for doing them
    - ⬆ the average wait time goes down, and the system approaches one with *1/n* the speed of the original system

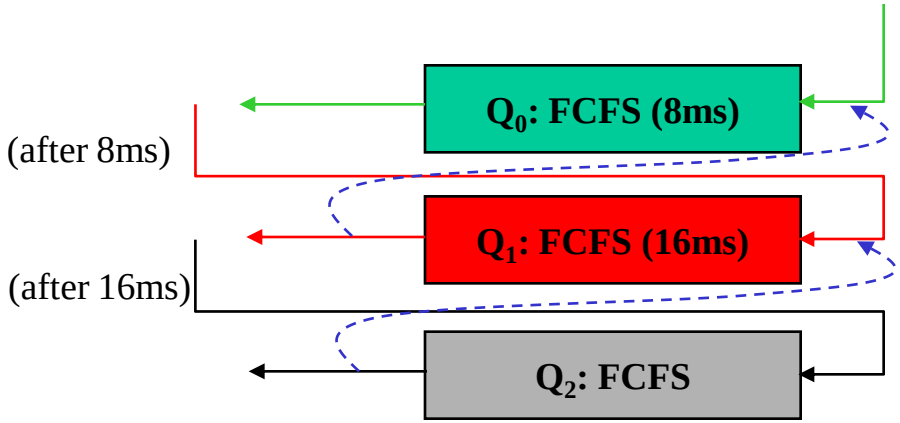
Hybrid Schemes: Multilevel Queue Scheduling

- Processes are partitioned into groups based on static criteria
  - background (batch)
  - foreground (interactive)
- All the processes in a fixed group of the partition share the same scheduling strategy and a distinct family of queues
  - different scheduling algorithm can be used across different groups
    - foreground: Round Robin
    - background: FCFS
- Need to schedule the CPU between the groups as well; for example,
  - fixed-priority: e.g., serve all from foreground, then from background
    - possibility of starvation
  - time slice: each group gets a certain fraction of the CPU
    - e.g., 80% to foreground in RR, 20% to background in FCFS

Generalization: Multilevel Feedback Queues

- Provide a mechanism for jobs to move between queues
  - ageing can be implemented this way
- Complete specification
  - queues: number, scheduling algorithms (within and across queues)
  - promotion and demotion policies
  - which queue should a process enter when it needs service?

- Example: 3 queues: Q_0 (FCFS, 8ms), Q_1 (FCFS, 16ms), Q_2 (FCFS)



Choosing a Scheduling Approach

- Identify metrics for evaluation
  - we have already seen a variety of metrics
    - throughput, wait time, turnaround time, ...
  - the goal is to start with an expectation or specification of what the scheduler should do well
    - for example, we might wish to have a system in which
      - the CPU utilization is maximized, subject to a bound on the response time

- Evaluate how different scheduling algorithms perform
  - deterministic modeling
    - requires accurate knowledge of job and system characteristics
    - practical only for real-time and embedded systems
  - more detailed performance evaluation
    - queueing models, simulation, measurement

- See Section 5.7 for details

Windows XP Scheduler (Section 5.6.2)

- Preemptive, priority based
- 32 priority levels: Higher priority numbers imply higher priority
  - priority level 0: memory management thread
  - 1-15 are variable priority classes
    - processes start off with a base priority (one of these levels)
      - HIGH (13), ABOVE_NORMAL (10), NORMAL (8), BELOW_NORMAL(6), IDLE (1)
    - threads in the process can start at priority = *(base_priority ± 2)*
      - Additional support for TIME_CRITICAL (15) and IDLE (1) threads
      - OS raises priorities of I/O-bound threads (max value is 15)
        - » Amount of boost depends on type of I/O
      - OS lowers priorities of CPU-bound threads (min value is *base_priority-2*)
      - distinction between foreground and background processes in NORMAL class
  - 16-31 are real-time priority classes
    - real-time threads have a fixed priority
    - threads within a particular level processed according to RR

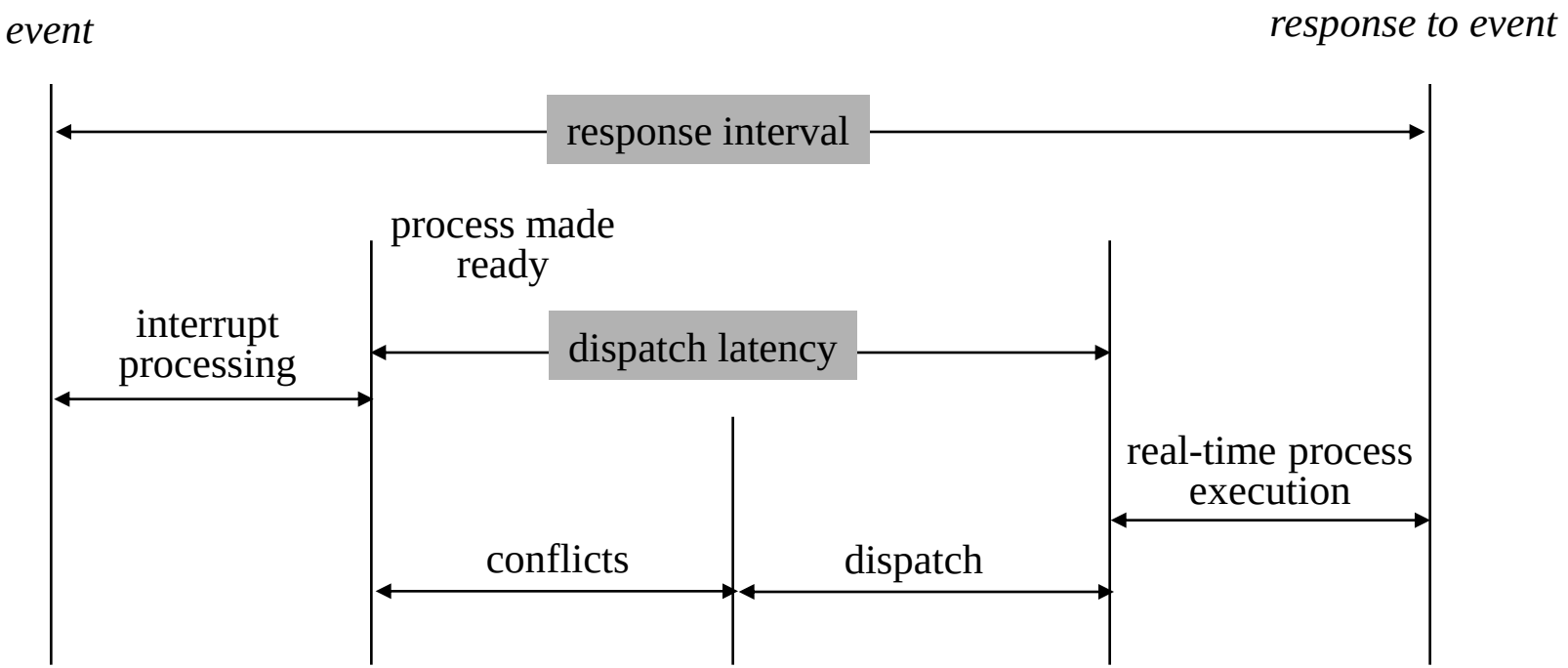
Advanced Topic: Real-Time Scheduling

- Processes have real-time requirements (deadlines)
  - e.g., a video-frame must be processed within certain time
  - growing in importance
    - media-processing on the desktop
    - large-scale use of computers in embedded settings
      - *sensors* produce data that must be processed and sent to *actuators*

- Real-time tasks typically considered along two dimensions
  - aperiodic (only one instance) versus periodic (once per period T)
  - hard real-time (strict deadlines) versus soft real-time
    - hard real-time tasks require *resource reservation*, and (typically) *specialized hardware* and scheduling *algorithms*
      - earliest-deadline first
      - rate-monotonic scheduling
      - details are beyond the scope of this class
    - our focus is on supporting soft real-time tasks in a general environment

Soft Real-Time Scheduling

- Most contemporary, general-purpose OSes deal with soft real-time tasks by being *as responsive as possible*
  - ensure that when a deadline approaches, the task is quickly scheduled
    - minimize latency from arrival of interrupt to start of process execution



Soft Real-Time Scheduling: OS Requirements

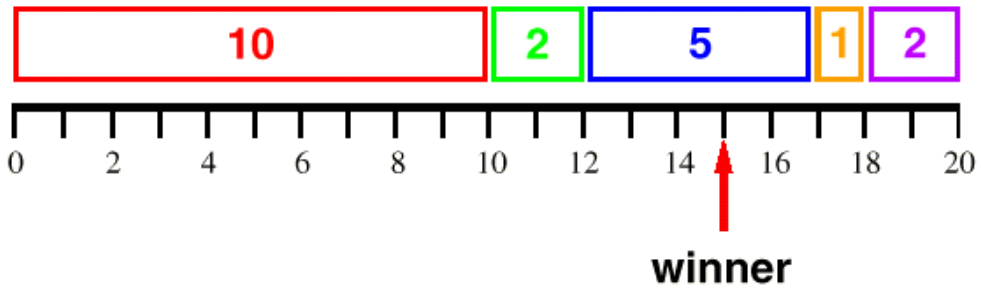
- Minimize interrupt processing costs
  - minimization of intervals during which interrupts are disabled
- Minimize dispatch latency
  - preemptive priority scheduling
    - real-time processes have higher priority than non real-time processes
    - priority of real-time processes does not degrade over time
  - current activity must be preemptible
    - Unacceptable options
      - traditional UNIX approach (waiting for system call completion)
      - preemption at *safe points* (unless enough of these as in Linux 2.6)
    - Acceptable: entire kernel must be preemptible (e.g., Solaris 2)
      - kernel data structures protected by synchronization mechanisms
    - Must cope with the priority inversion problem
      - A lower-priority process holds a resource required by the higher-priority process

Fair-Share Scheduling

- Problems with priority-based systems
  - priorities are absolute: no guarantees when multiple jobs with same priority
  - no encapsulation and modularity
    - behavior of a system module is unpredictable: a function of absolute priorities assigned to tasks in other modules
- *Solution*: Fair-share scheduling
  - each job has a *share:* some measure of its relative importance
    - denotes user's share of system resources as a fraction of the total usage of those resources
    - e.g., if user A's share is twice that of user B
      - then, in the long term, A will receive twice as many resources as B
- Traditional implementations
  - keep track of per-process CPU utilization (a running average)
  - reprioritize processes to ensure that everyone is getting their share
  - are slow!

Example Fair-Share Policy: Lottery Scheduling

- A randomized mechanism for efficient *proportional-share* resource management
  - each process has certain number of lottery tickets (its share)
    - Processes reside in a conventional ready queue structure
  - each allocation is determined by holding a *lottery*
    - Pick a random ticket number
    - Grant resource to process holding the winning ticket



Why Does Lottery Scheduling Work?

- Expected allocation of resources to processes is proportional to the number of tickets that they hold

- Number of lotteries won by a process has a binomial distribution
  - probability p of winning = t/T
  - after n lotteries, E[w] = np and variance = $np(1-p)$

- Number of lotteries to first win has a geometric distribution
  - E[n] = $1/p$, and variance = $(1-p)/p^2$

Outline

Announcements

- Lab 2 was due last Friday
  - Please send me e-mail if you have not yet started this lab
- Today's lecture will end at 6:30pm

- CPU Scheduling (cont'd)
  - Scheduling algorithms – Round Robin
  - Example: Windows XP scheduler
  - Advanced topic: Real-time scheduling
- Process Deadlocks
  - System model
  - Characteristics of deadlocks: Graph representation
  - Methods for handling deadlocks
    - Deadlock prevention
    - Deadlock avoidance

[Silberschatz/Galvin/Gagne: Sections 5.3, 5.6 – 5.7, 7.1 – 7.5]

Process Deadlock

- Example:
  - 2 processes, each holding a different resource in exclusive mode, and each requesting access to the resource held by the other process
    - e.g., processes requiring access to disk and printer
      - one process acquires the disk and waits for the printer
      - the other acquires the printer and waits for the disk
  - neither will make progress!

- Definition

  A deadlock occurs when a set of processes in a system is blocked waiting on requirements that can never be satisfied.
  These processes, while holding some resources, are requesting accesses to resources held by other processes in the same set.
  In other words, the processes are involved in a circular wait.

- Resolving the deadlock requires the intervention of some process outside those involved in the deadlock

Deadlock versus Starvation

- Deadlock: A process waits for a resource that is currently assigned to another process, which is in turn waiting for another resource …
- Starvation: A process waits for a resource that continually becomes available but is never assigned to the waiting process

- Two major differences between deadlock and starvation
  - in starvation, it is not certain that a process will never get the requested resource (i.e., there is a chance it might), while a deadlocked process is *permanently* blocked
  - in starvation, the resource under contention is *continuously available*, whereas this is not true in a deadlock

- Starvation is typically easier to fix than deadlock

System Model for Deadlocks

- Resources
  - different types of resources (e.g., memory space, CPU cycles, file handles)
  - processes request a resource type, not a particular resource
    - any of the resources in that type can be used to satisfy the request
- Processes
  - use resources
    - *request* resource type i
    - *use* resource type i
    - *release* resource type i
  - a process can request multiple instances of a resource
  - OS intervenes on *request* and *release*
- Deadlocks: Caused by processes waiting for events that never happen
  - events of interest: *request* and *release*
  - events can be for different resource types

Conditions for Deadlock

- Deadlocks involve a *set of processes* contending for a *set of resources*

- All of the following conditions must hold for deadlock to occur
  - Mutual Exclusion
    - at least one resource can only be used by one process at any one time
  - Hold and Wait
    - there must exist at least one process that is holding at least one resource, *and* is waiting to acquire additional resources *currently held* by other processes
  - No Preemption
    - processes cannot be forced to give up resources
  - Circular Wait
    - there is a sequence of processes $p_1, p_2, ..., p_n, p_1$
    - such that p_i is waiting for a resource held by p_{i+1}

Conditions for Deadlock: Dining Philosophers

- Conditions
  - Mutual exclusion — each chopstick can only be used by one philosopher
  - Hold and wait — philosophers hold on to a chopstick while requesting another
  - No preemption — not possible to force a philosopher to give up a chopstick
  - Circular wait — philosopher$_i$ waits on philosopher$_{i+1}$ …

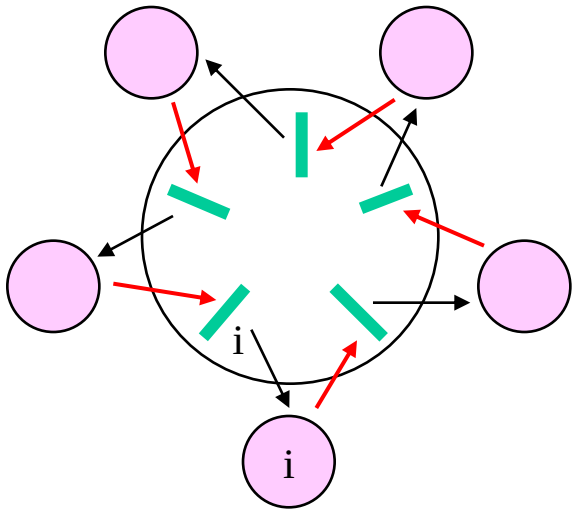
Philosopher$_i$

```
P( chopstick[i] );
P( chopstick[i+1 mod 5] );

EAT

V( chopstick[i] );
V( chopstick[i+1 mod 5] );

THINK
```

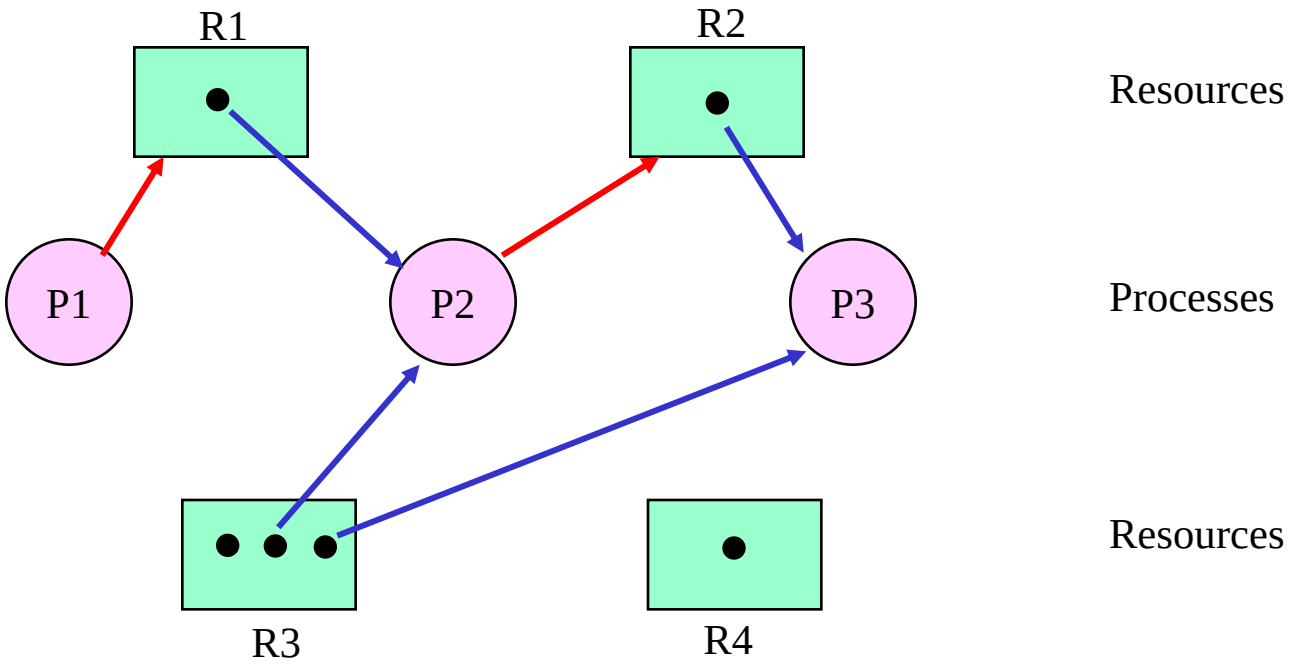


Graph Representations of Deadlocks

A Resource Allocation Graph (RAG)

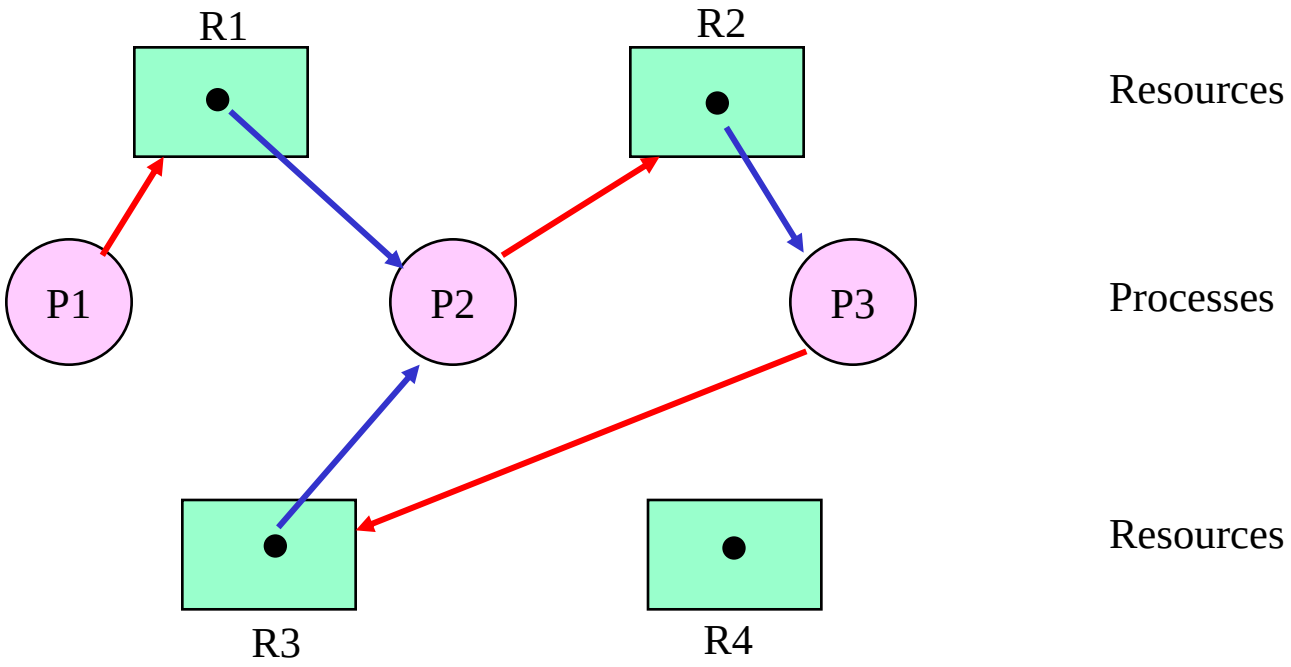
- Two types of nodes
  - Processes and
  - Resources
- Three types of *directed edges* between Processes and Resources
  - request edge: a solid edge from *P* to *r*, indicating that *P* has requested *r*
  - assignment edge: a solid edge from *r* to *P*, indicating that the OS has already allotted resource r to process P
  - claim edge: a dotted edge from a process node *P* to a resource node *r*, indicating that *P* may request *r* at some point in the future
- We shall focus only on requests for exclusive access to a resource
  - handling of mixed access types is slightly complicated

Resource Allocation Graphs: Example



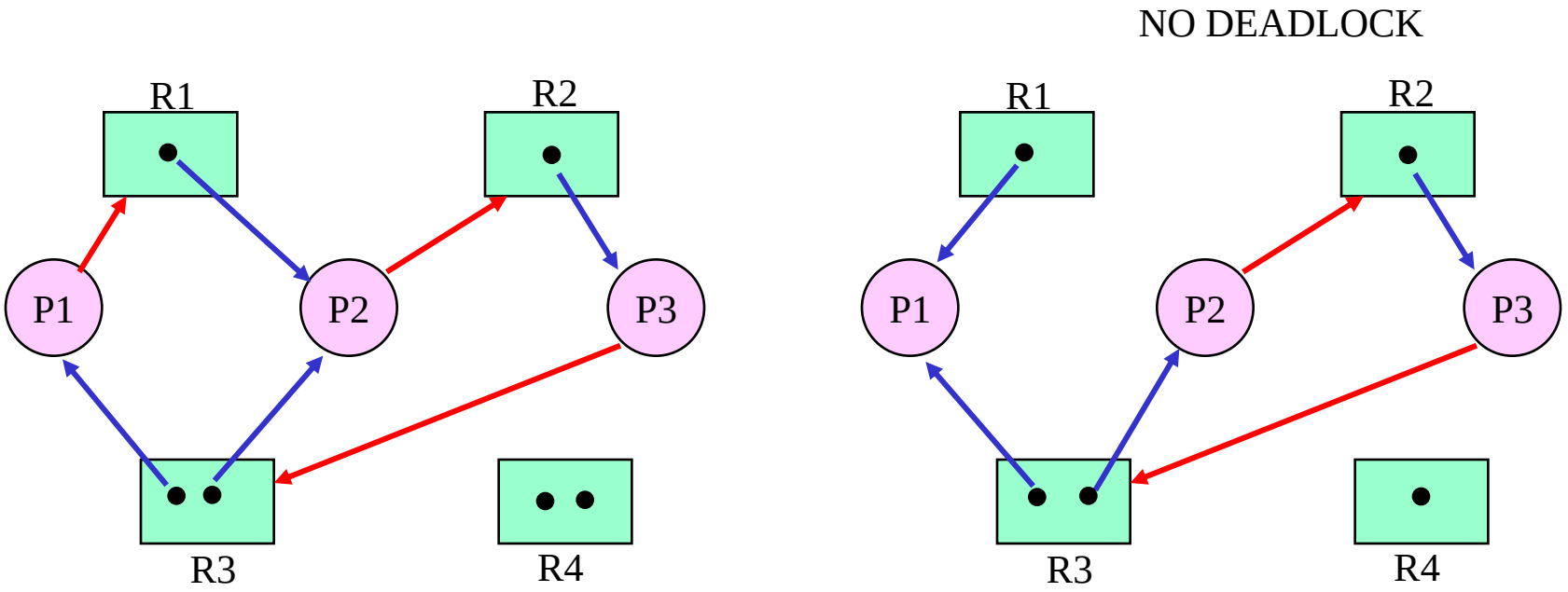
- Assignment edges originate from a *resource instance*
- A request edge is *instantaneously* transformed to an assignment edge if resources are available

Deadlocks in RAGs with Single Resource Instances



- A cycle in the graph is a necessary and sufficient condition for the existence of a deadlock

Deadlocks in RAGs with Multiple Resource Instances



- A cycle in the graph is a necessary but not sufficient condition for the existence of a deadlock
  - if a cycle does not exist: no deadlock
  - if a cycle exists: there may or may not be a deadlock

Methods for Handling Deadlocks

Three approaches with different cost-performance tradeoffs

- Prevention
  - deadlock cannot possibly occur
- Avoidance
  - deadlock can occur, but there are algorithms to avoid it
  - relies on the OS having an advance model of possible resource requests from processes
- Detection and Recovery
  - deadlock may occur, but there are ways of detecting it and recovering
  - this method is preferable when deadlocks happen rarely

Deadlock Prevention

- Approach: Ensure that the necessary conditions for deadlocks are never satisfied

- Prevent one of the following from becoming true
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait

Deadlock Prevention (1): Mutual Exclusion

- Mutual exclusion is not a problem for sharable resources
  - an example is a “read-only” file which is a resource that can be accessed simultaneously

- Problem: Some resources are inherently not sharable
  - so, denying the mutual exclusion condition cannot be enforced in general

Deadlock Prevention (2): Hold-and-Wait

- Approach: Guarantee that when a process requests a resource it does not hold any other resources
  - Choice 1: A process requests and is allocated all of its resources before it begins execution
    - require system calls requesting resources to precede all other system calls
      - Need all resources to be requested with a single call
  - Choice 2: A process releases any resources it is holding before it requests for new ones
  - Choice 3: A process that is "holding" a resource immediately releases it if another of its requests cannot be satisfied currently
- Limitations
  - inefficient
  - lowered resource utilization
  - starvation

Deadlock Prevention (3): No Preemption

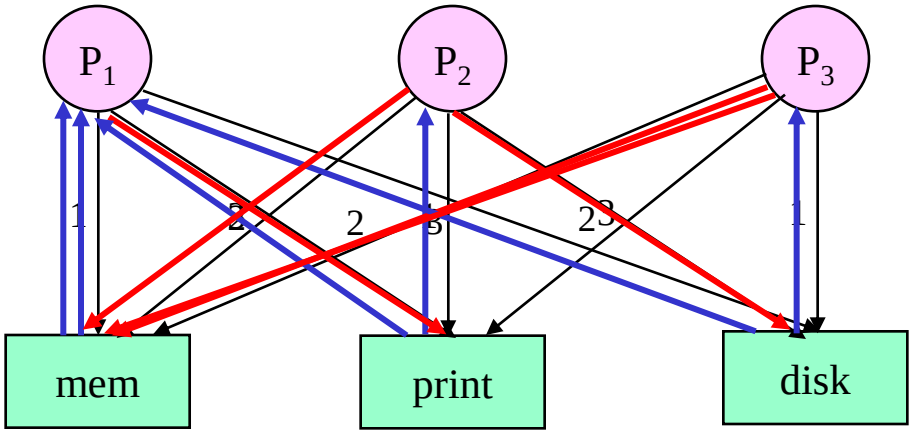
- Approach: Take away resources from a process (preemption) and give them to another waiting process
  - some resources are preemptible
    - e.g., memory space, disk space (on a particular disk)
    - these can be taken away from a process
  - examples of non-preemptible resources?

- Choices
  - protocol 1: if a process is holding some resources and requests other resources that cannot be granted to it, all of its resources are taken away
  - protocol 2: when a process requests additional resources, see if these resources are being held by a process that is itself waiting for new resources. In this situation, preempt the second process

- Limitation: Cost of preemption
  - a process may get preempted even when there is no deadlock

Deadlock Prevention (4): Circular Wait

- Example:
  - Processes need three resources: memory, disk, printer
  - Consider two cases:
    - Case 1: processes pick own order in which to ask for resources
    - Case 2: each process asks first for memory, then disk, then the printer
  - Which of these cases can result in deadlock?



Deadlock Prevention: Circular Wait (cont'd)

- Approach: Impose an order on resource acquisition
  - all the N types of resources in the system are linearly ordered
    - each is given a number, called rank, in the range 1, 2, ... , N
    - the resources of the same type all have the same rank
    - different types of resources get distinct ranks
  - processes are required to sequence their resource requests in strictly increasing order of rank
    - i.e., they ask for all the "smaller" rank resources first

- In our example
  - rank(memory) = 1, rank(disk) = 2, rank(printer) = 3

- Why does this work?

Deadlock Prevention: Circular Wait (cont'd)

- Why it works
  - suppose the circular wait consists of processes P1, P2, ..., Pn, P1
  - suppose P(i) is waiting on a resource held by P(i+1) of rank R(j)
    - P(i+1) must have been granted all resources it needs of rank R(j)
    - it must therefore be waiting for a resource of rank ≥ R(j) + 1
  - since a cycle of strictly increasing ranks cannot exist, there can exist no such cycle.

- Two related points
  - an equivalent strategy is one where a process, when it requests a resource of a particular rank, releases all those with a higher rank
  - typical rank orders are based on natural usage
    - e.g., since storage devices are used "before" printers, they get smaller ranks

Deadlock Prevention: Summary

- Main idea: Prevent one of the four necessary conditions
  - mutual exclusion
  - hold-and-wait: ask for all resources at start
  - no preemption
  - circular wait: resource ranking scheme

- Limitations
  - inefficient
    - static allocation of resources reduces concurrency
    - a process may need to be preempted even when there is no deadlock
  - restrictive
    - requires allocation of future resource requirements before it starts executing

- Alternative approaches?

Deadlock Avoidance

- Deadlock occurs because processes are waiting on each other to release resources

- Main idea of deadlock avoidance:
  - request additional information about how resources are to be requested
  - before allocating request, check if system will enter a deadlock state

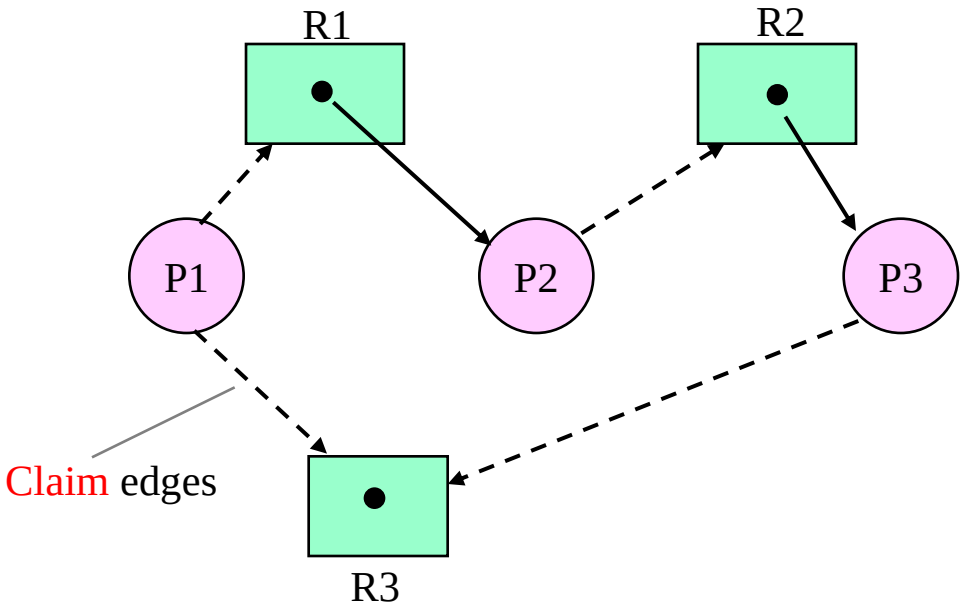
    F (resources available, resources allocated, future requests/releases)
    - if no: grant the request
    - if yes: block the process

- Algorithms differ in amount and type of information
  - simplest (also most useful) model: maximum number of resources
  - other choices
    - sequence of requests and releases
    - alternate request paths

- How can we find out if a system will enter a deadlock state?

Deadlock Avoidance: Notion of a Safe State

- A system is in a safe state iff there exists a safe sequence
- A sequence <P_1, P_2, …, P_n> is a safe sequence for the current allocation if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus resources held by all the P_j, for j<i



<P3, P2, P1> is a safe sequence

P3's future request can be satisfied because R3 is available

P2's future request can be satisfied by P3 yielding R2

P1's future requests can be satisfied by P2 giving up R1 and available R3

Properties of Safe States

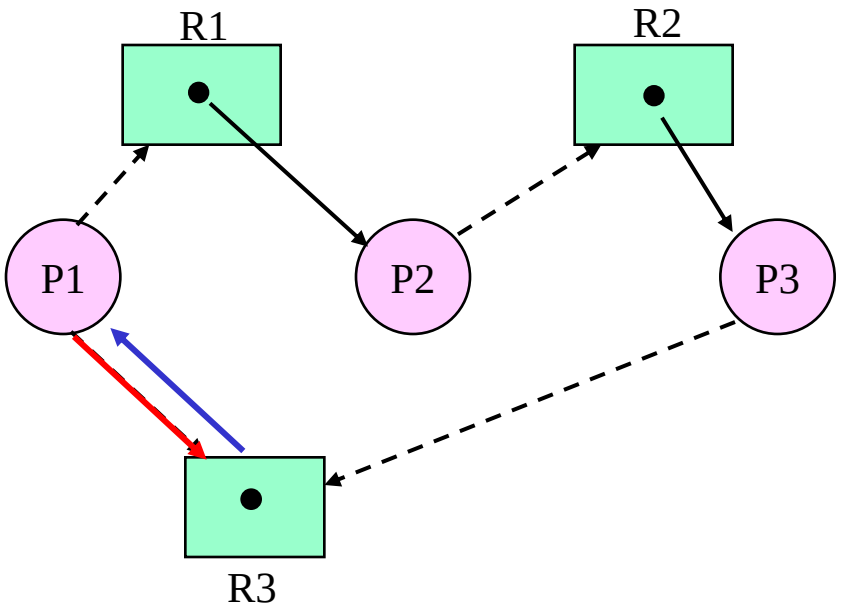
- A safe state is not a deadlock state
- An unsafe state may lead to deadlock
- It is possible to go from a safe state to an unsafe state

- Example: A system with 12 units of a resource
  - Three processes
    - P_1: max need = 10, current need = 5
    - P_2: max need = 4, current need = 2
    - P_3: max need = 9, current need = 2
  - This is a safe state, since a safe sequence $<P_2, P_1, P_3>$ exists
  - P_3 requests an additional unit. Should this request be granted?
  - No, because this would put the system in an unsafe state
    - P_1, P_2, P_3 will then hold 5, 2, and 3 resources (2 units are available)
    - P_2's future needs can be satisfied, but no way to satisfy P_1's and P_3's needs

- Avoidance algorithms prevent the system from entering an unsafe state

Deadlock Avoidance: Single Resource Instances

- Deadlock ≡ Cycle in the resource allocation graph
- A request is granted iff it does not result in a cycle
  - cycle detection: O(V + E) operations



<P3, P2, P1> is a safe sequence

Say P1 requests R3:
should this be granted?

No, because an assignment edge from R3 to P1 would create a cycle in the RAG.
[No safe sequence exists]
Does this always imply a deadlock?

No, because P1 can release R3 before requesting R1

Deadlock Avoidance: Multiple Resource Instances

- Banker's Algorithm
  - upon entering the system, a process declares the maximum number of instances of each resource type that it may need
  - the algorithm decides, for each request, whether granting it would put the system in an unsafe state

resource availability
  Available[1..m]
maximum demand
  Max[1..n, 1..m]
current allocation
  Allocation[1..n, 1..m]
potential need
  Need[1..n, 1..m]

1. If $Request_i \leq Need_i$
   goto Step 2, else flag error
2. If $Request_i \leq Available$
   goto Step 3, else wait
3. Allocate the resources
   $Available := Available - Request_i$;
   $Allocation_i := Allocation_i + Request_i$;
   $Need_i := Need_i - Request_i$;
   Check if this is a safe state.
   If not: undo the allocation and wait

1. $Work := Available$;
   $Finish[i] := false$, for all i;
2. Find an i such that
   a. $Finish[i] = false$, and
   b. $Need_i \leq Work$
   if no such i, goto Step 4
3. $Work := Work + Allocation_i$;
   $Finish[i] := true$;
   goto Step 2;
4. If $Finish[i] = true$ for all i,
   then the system is in a safe state

Banker's Algorithm: Example

- Three resource types and
  three processes (P_1, P_2, P_3)

  Capacity = [2, 4, 3]
  Max = [[1, 2, 2], [1, 2, 1], [1, 1, 1]]
  Allocation = [[1, 2, 0], [0, 1, 1], [1, 0, 1]]
  Available = [0, 1, 1]
  Need = [[0, 0, 2], [1, 1, 0], [0, 1, 0]]

- P_1 requests [0, 0, 1]
  Should this be granted?

- Allocate and check if system is in a safe state

  Allocation = [[1, 2, 1], [0, 1, 1], [1, 0, 1]]
  Available = [0, 1, 0]
  Need = [[0, 0, 1], [1, 1, 0], [0, 1, 0]]

1. *Work := Available;*
   Finish[i] := false, for all *i*;
2. Find an *i* such that
   a. *Finish[i] = false*, and
   b. $Need_i \leq Work$
   if no such *i*, goto Step 4
3. $Work := Work + Allocation_i$;
   Finish[i] := true;
   goto Step 2;
4. If *Finish[i] = true* for all *i*,
   then the system is in a safe state

Initially, *Work* = [0, 1, 0]
$Need_3 \leq Work$, so P_3 can finish
Work = [1, 1, 1]
Now, both P1 and P2 can finish

Limitations of Deadlock Avoidance

- Deadlock avoidance vs. deadlock prevention
  - Prevention schemes work with local information
    - What does this process already have, what is it asking
  - Avoidance schemes work with global information
    - Therefore, are less conservative

- However, avoidance schemes require specification of future needs
  - not generally known for OS processes
  - more applicable to specialized situations
    - programming language constructs (e.g., transaction-based systems)
    - known OS components (e.g., Unix “exec”)

- More general solution: Deadlock detection and recovery