slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

CSCI 3753 Operating Systems
Summer 2019

Christopher Godley

PhD Student

Department of Computer Science

University of Colorado Boulder

Deadlock

Deadlock: General Solution?

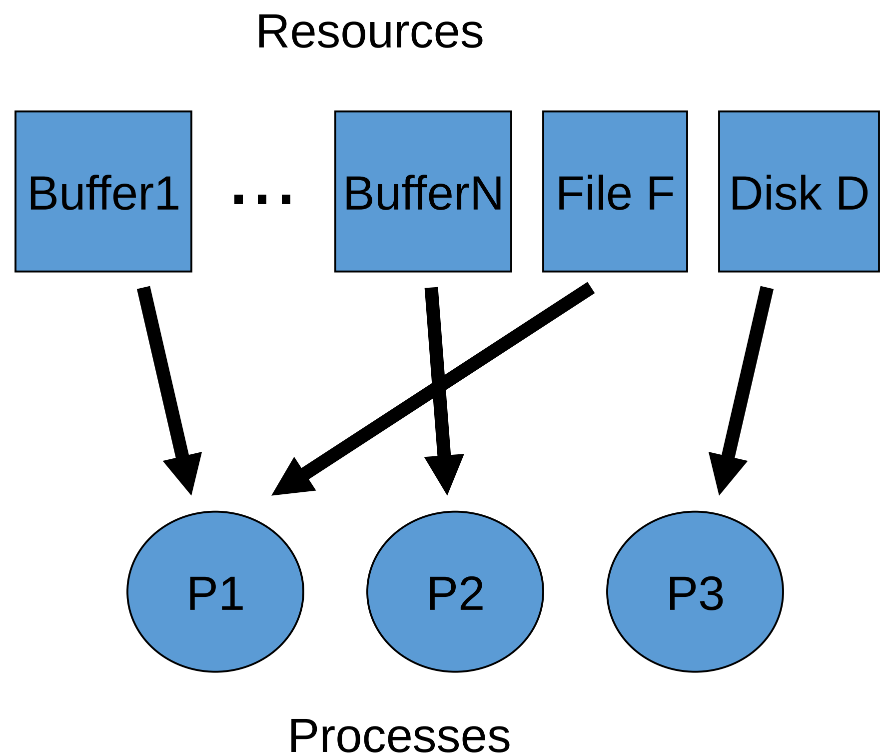
- Want a general solution to deadlock that is not restricted to the solutions for the 3 classic problems of DP, R/W, and BB P/C

- **A set of processes is in a deadlock state when every process in the set is waiting for an event (e.g. release of a resource) that can only be caused by another process in the set**
  - You have a circular dependency

- multithreaded and multi-process applications are good candidates for deadlock
  - thread-thread deadlock within a process
  - process-process deadlock

Modeling Deadlock

- Develop a model so we can see circular dependency
  - to use a resource, a process must
    1. request() a resource -- must *wait* until it's available
    2. use() or hold() a resource
    3. release() a resource
  - thus, we have resources and processes
  - Most of the following discussion will focus on reusable resources



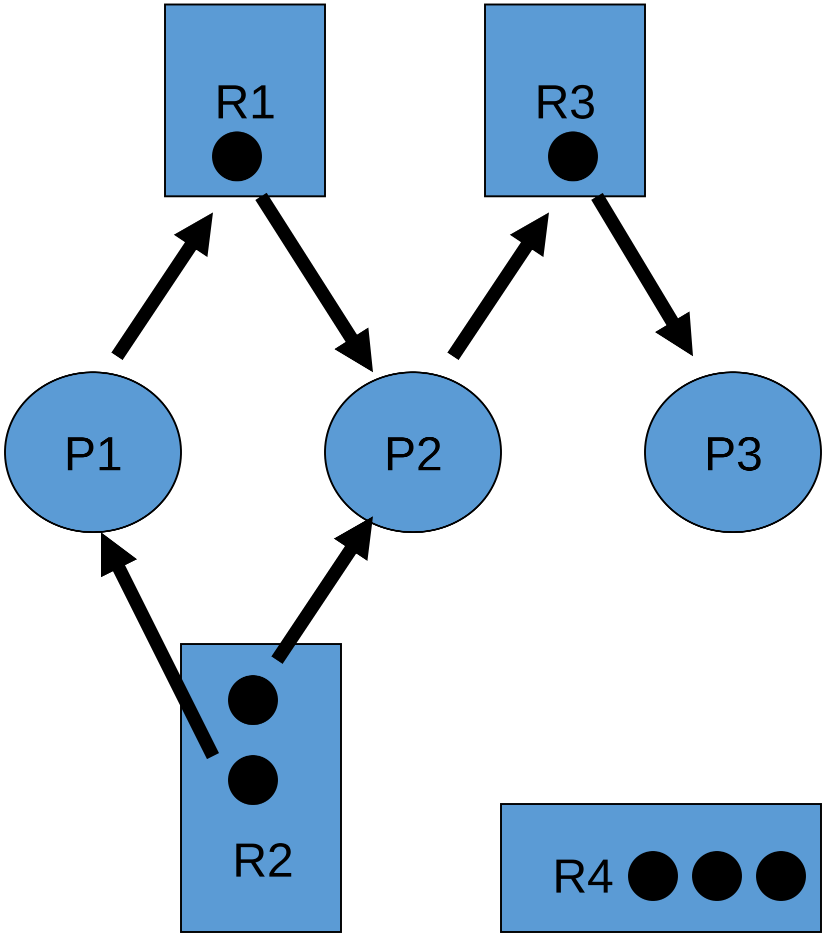
P1 holds Buffer 1 and File F
P2 holds Buffer N
P3 holds Disk D

Modeling Deadlock using Directed Graph

- a ***resource allocation graph*** can be used to model deadlock
  - try to represent deadlock by a *directed graph* D(V,E), consisting of
    - vertices V: namely processes and resources
    - and edges E:
      - a **request**() for a resource R_j by a process P_i is signified by a directed arrow from **process P_i → R_j**
      - a process P_i will **hold**() a resource R_j via a **directed arrow R_j → P_i**

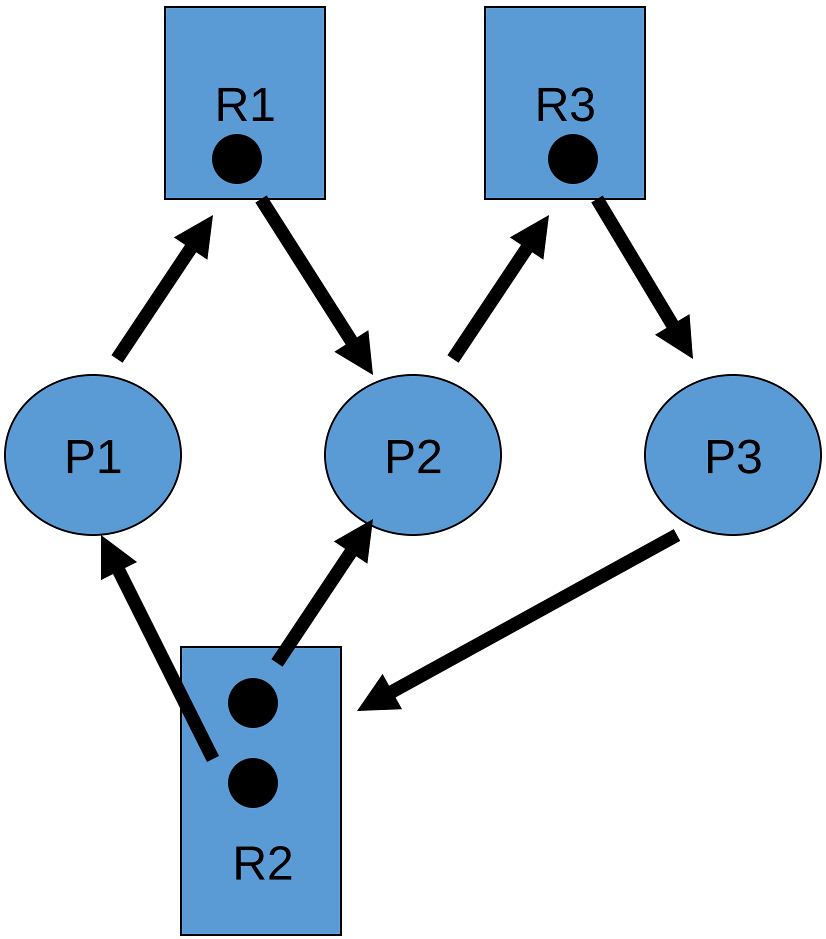
Modeling Deadlock

- Example 1:
  - P1 wants resource R1 but that is held by P2
  - P2 wants resource R3 but that is held by P3
    - Also, P1 holds an *instance* of resource R2, and
    - P2 holds an instance of R2
    - There is no deadlock
      - if the graph contains no cycles or loops, then there is no deadlock



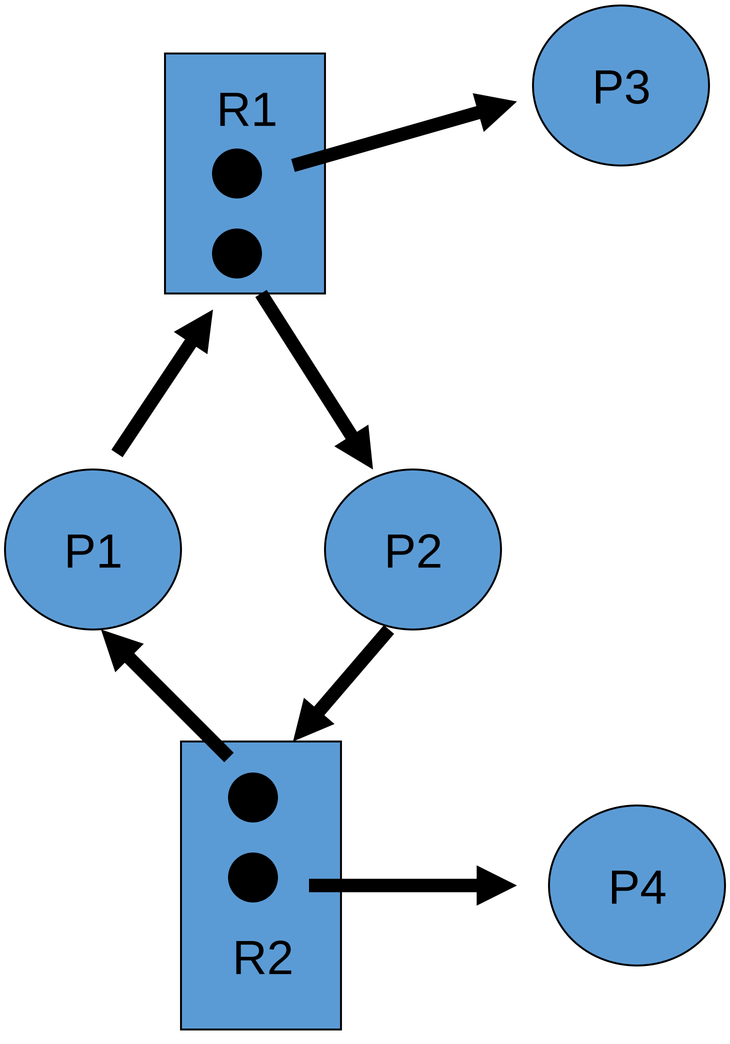
Modeling Deadlock

- Example 2:
  - same graph as before, except now P3 requests instance of R2
  - Deadlock occurs!
    - P3 requests R2, which is held by P2, which requests R3, which is held by P3 - this is a loop
      - $P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$
      - If P1 could somehow release an instance of R2, then we could break the deadlock
    - But P1 is part of a second loop:
      - $P_3 \rightarrow R_2 \rightarrow P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$
      - So P1 can't release its instance of R2
  - if the graph contains cycles or loops, then there *may be the possibility* of deadlock
    - but does a loop guarantee that there is deadlock?



Modeling Deadlock

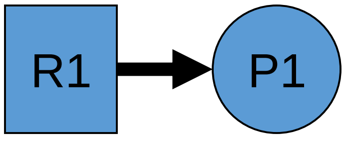
- Example 3:
  - there is a loop:
    - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$
  - In this case, there is no deadlock
    - either P3 can release an instance of R1, or P4 can release an instance of R2
      - this breaks any possible deadlock cycle
  - if the graph contains cycles or loops, then there *may be the possibility* of deadlock, but this is not a guarantee of deadlock



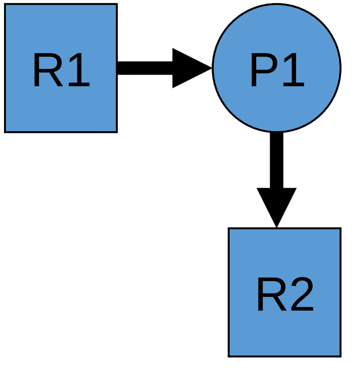
Necessary Conditions for Deadlock

The following 4 conditions must hold simultaneously for deadlock to arise:

1. Mutual exclusion
   - at least 1 resource is held in a non-sharable mode. Other requesting processes must wait until the resource is released



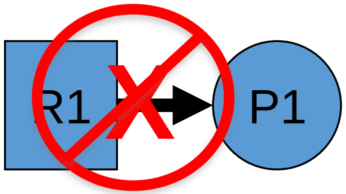
2. Hold and wait
   - a process holds a resource while requesting (and waiting for) another one



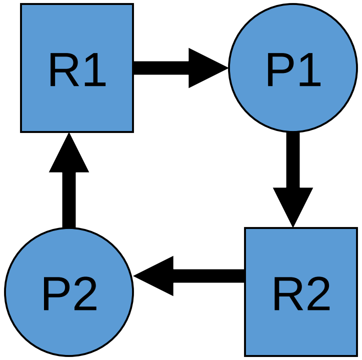
Necessary Conditions for Deadlock

The following 4 conditions must hold simultaneously for deadlock to arise: (continued)

3. No preemption:
   - resources cannot be preempted and can only be released voluntarily by the process holding them, after the process is finished. No OS intervention is allowed. A process cannot withdraw its request.



4. Circular wait
   - A set of n waiting processes $\{P_0, \ldots, P_{n-1}\}$ must exist such that Pi waits for a resource held by $P_{(i+1)\%n}$



Solutions to Handling Deadlocks

1. Prevention by OS
   - provide methods to guarantee that at least 1 of the 4 necessary conditions for deadlock does not hold
2. Avoidance by OS
   - the OS is given advanced information about process requests for various resources
   - this is used to determine whether there is a way for the OS to satisfy the resource requests and avoid deadlock

Solutions to Handling Deadlocks (2)

3. Detection and Recovery by OS
   - Analyze existing system resource allocation, and see if there is a **sequence of releases** that satisifies every process' needs.
   - If not, then deadlock is detected, so must recover – drastic action needed, such as killing the affected processes!

Solutions to Handling Deadlocks (3)

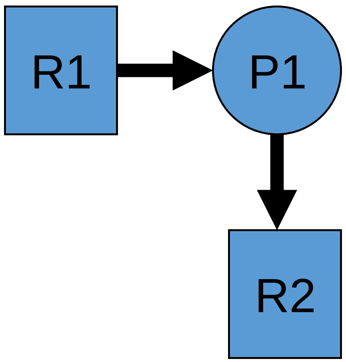
4. Application-level solutions (OS Ignores and Pretends)
   - the most common approach, e.g. UNIX and Windows, based on the assumption that **deadlock is relatively infrequent**
   - it's up to the application programmer to implement mechanisms that prevent, avoid, detect and deal with application-level deadlock
   - **Map your problem to known deadlock-free solutions: e.g. Bounded Buffer P/C, Readers/Writers problems, Dining Philosophers, …**

Deadlock Prevention: Mutual Exclusion

- Prevent the *mutual exclusion* condition #1 from coming true
  - Many resources are non-sharable and must be accessed in a mutually exclusive way
    - example: a printer should print a file X to completion before printing a file Y. a printer should not print half of file X, and then print the first half of file Y on the same paper
  - thus, it is unrealistic to prevent mutual exclusion

Deadlock Prevention: Hold and Wait

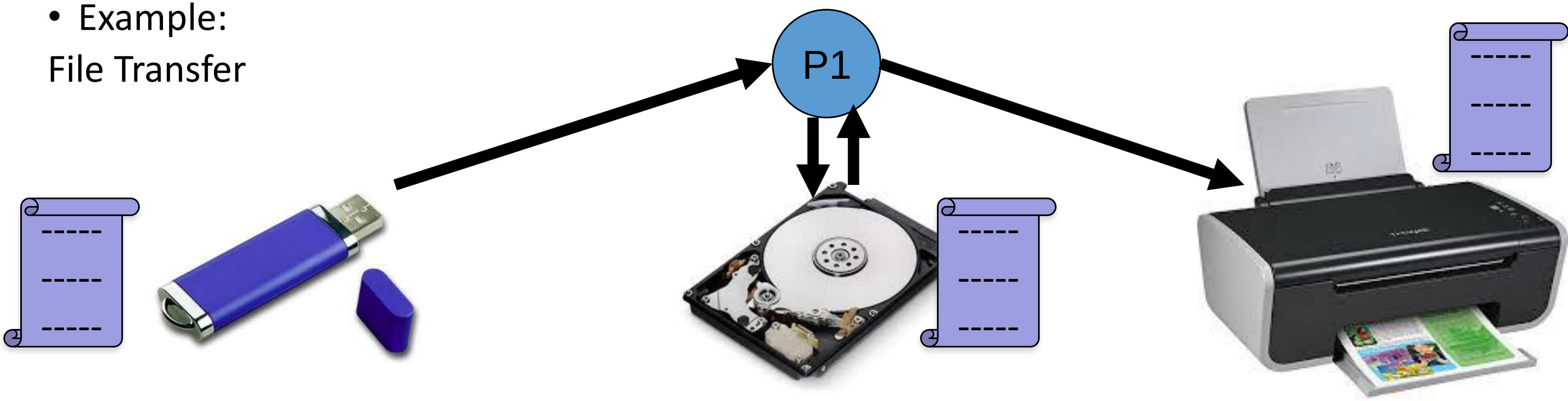
- Prevent the hold and wait condition #2 from coming true
  - prevent a process from holding resources and requesting others
  - *Solution I:* request all resources at process creation
  - *Solution II*: release all held resources before requesting a set of new ones simultaneously
  - *Solution III*: only allow a process to hold one resource at a time



Deadlock Prevention: Hold and Wait

- Example:

File Transfer

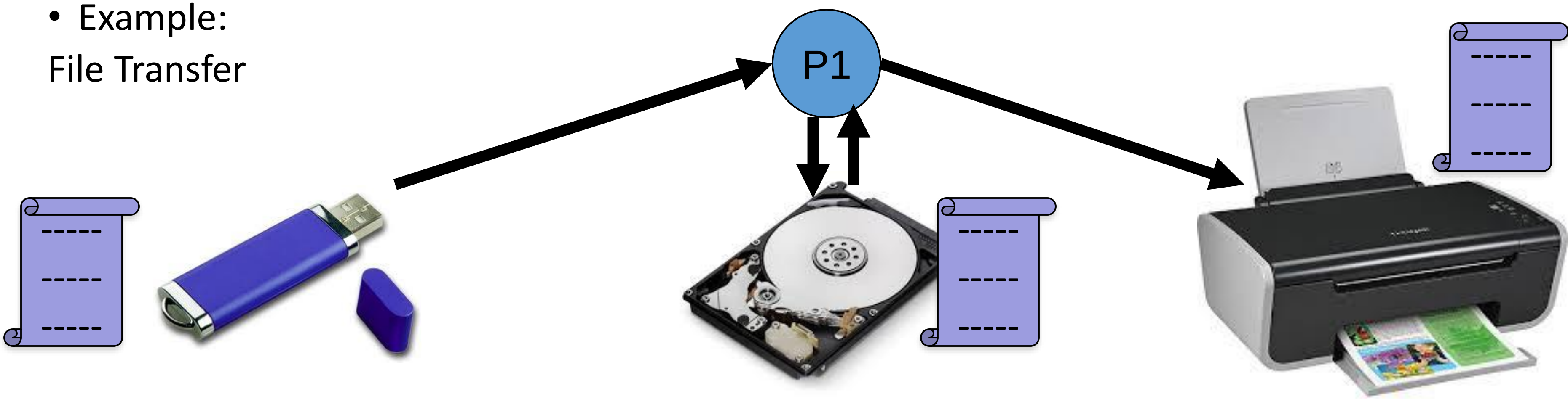


- a process reads file from USB drive and writes it to hard drive, retrieves the file, then sends the file to the printer
- Solution I: request the USB drive, hard drive, and printer at process creation

Deadlock Prevention: Hold and Wait

- Example:

File Transfer

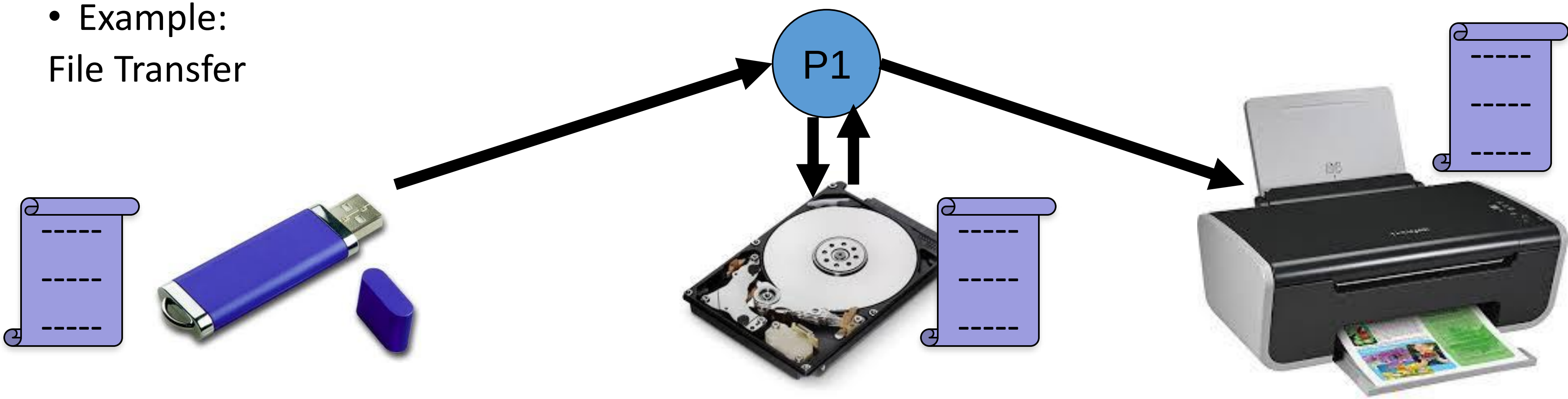


– Solution II: divide task into self-contained stages that release all & then request all resources
  - obtain the USB and hard drive together for the file transfer, then release both together
  - next obtain the hard drive and printer together for the printing operation, then release both together

Deadlock Prevention: Hold and Wait

- Example:

File Transfer



– Solution III:
- Request the USB drive then release
- Request the hard drive then release
- Request the hard drive again then release
- Request the printer then release

Deadlock Prevention: Hold and Wait

- Disadvantages of Hold-and-wait solutions
  - Solution I: don’t know in advance all resources needed
  - Solutions I & II: poor resource utilization
    - a process that is holding multiple resources for a long time may only need each resource for a short time during execution
  - Solution II: possible starvation
    - a process that needs several popular resources simultaneously may have to wait a very long time

Deadlock Prevention: Hold and Wait

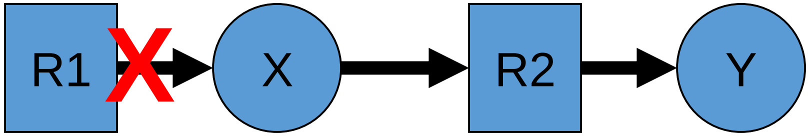
- Disadvantages of Hold-and-wait solutions
  - Solution III: Some processing may require holding more than one resource at a time
    - e.g. writing a file to a printer may require locking both the file and the printer
    - Reading a file from a drive may require locking both the file and the drive

Deadlock Prevention: Hold and Wait

- Example: Dining Philosophers Problem prevented hold-and-wait – How?
  - Enforced a rule that either a philosopher picked up both chopsticks or none at all, i.e. all-or-nothing
  - Hence no holding one chopstick while waiting on the other chopstick

Deadlock Prevention: No Preemption

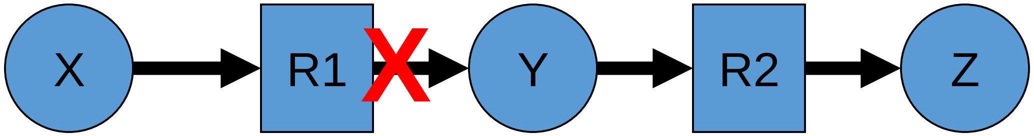
- Prevent the “No Preemption” condition #3 from coming true
  - allow resources to be preempted



  - Policy I:
    - If a Process X requests a held resource, then all resources currently held by X are released.
    - X is restarted only when it can regain all needed resources

Deadlock Prevention: No Preemption

- Policy II:



  - If a process X requests a resource held by process Y, then preempt the resource from process Y, but only if Y is waiting on another resource
  - Otherwise, X must wait.
  - the idea is if Y is holding some resources but is waiting on another resource, then Y has no need to keep holding its resources since Y is suspended

Deadlock Prevention: No Preemption

- Disadvantages:
  - **these policies don't apply to all resources**, e.g. printers should not be prempted while in the middle of printing, disks should not be preempted while in the middle of writing a block of data
  - can result in unexpected behavior of processes, since an **application developer may not know a priori which policy is being used**

Deadlock Prevention: Circular Wait

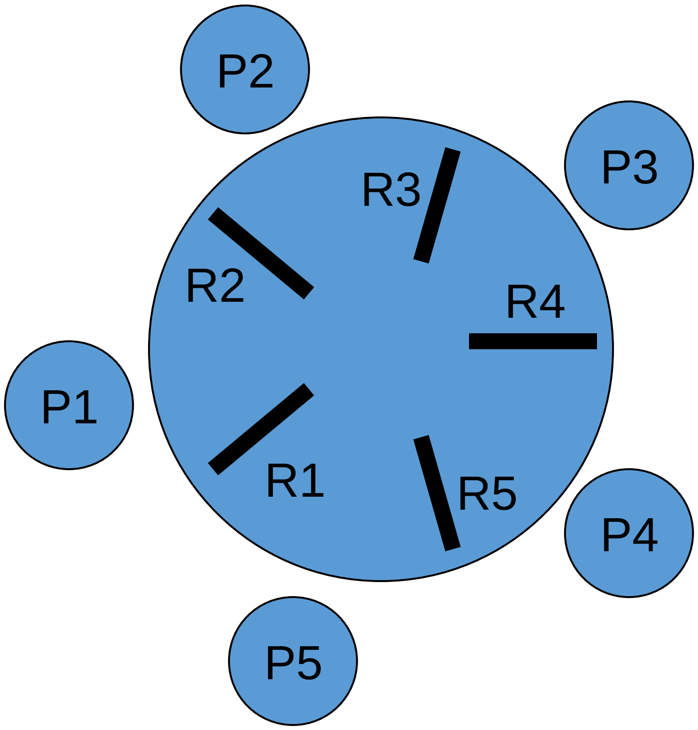
- Prevent the *circular wait* condition #4 from coming true
  - Solution I: a process can only hold 1 resource at a time
    - disadvantage: in some cases, a process needs to hold multiple resources to accomplish a task
  - Solution II: impose a total ordering of all resource types and require each process to request resources in increasing order
    - this prevents a circular wait

Deadlock Prevention: Circular Wait

- Solution II example:
  - Order all resources into a list: R1, R2, ..., Rm, where R1 < R2 < ... < Rm
    - tape drive = R1, disk drive = R2, printer = R10, temporary buffer = R22
  - Impose the rule that a process holding R_i can only request R_j if $R_j > R_i$
  - If a process P holds some R_k and requests R_j such that $R_j < R_k$, then the process must release all such R_k, acquire R_j, then reacquire R_k

Deadlock Prevention: Circular Wait

- Applying ordering of resources to break circular waiting in the Dining Philosophers Problem
  - R1 < R2 < R3 < R4 < R5
  - Deadlock happened when all processes first requested their right chopsticks, then requested their left chopsticks
  - Here, P1 to P4 can all request their right then left chopsticks
  - *But Process P5 requests its left (R1) then right (R5) chopstick due to ordering*
    - thus, P5 blocks on R1, not R5, which breaks any possibility of a circular deadlock



Deadlock Prevention: Circular Wait

- Disadvantages of ordering resources:
  - can lead to poor performance, due to releasing and then reacquiring resources
  - Difficult to implement in a dynamic resource environment
    - Need to come up with a global scheme for numbering resources

Deadlock Avoidance

- Goal: analyze the system state to see if there is a way to avoid deadlock.
- At startup, each process provides OS with information about all of its requests and releases for resources R_i
  - e.g. batch jobs know a priori which resources they'll request, when, and in which order
- OS decides whether deadlock will occur at run time

Deadlock Avoidance

- Disadvantage: need a priori info
- Simple strategy:
  - each process specifies a maximum claim
  - knowing all individual future requests and releases is difficult
  - Having each process estimate its maximum demand for resources is easier and not completely unreasonable
- A *resource allocation state* is defined by
  - # of available resources
  - # of allocated resources to each process
  - maximum demands by each process

Deadlock Avoidance

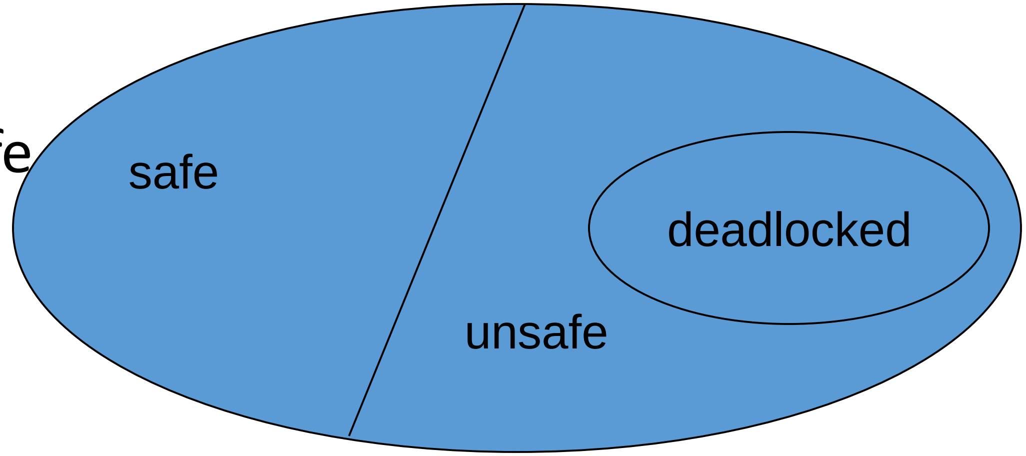
- A system is in a *safe* state if there exists a *safe sequence* of processes <P1, ..., Pn> for the current resource allocation state
  - A sequence of processes is safe if for each P_i in the sequence, the resource requests that P_i can still make can be satisfied by:
    - currently available resources + all resources held by all previous processes in the sequence P_j, $j<i$
  - If resources needed by P_i are not available, P_i waits for all P_j to release their resources

Deadlock Avoidance

- Intuition for a safe state: given that the system is in a certain state, we want to find at least one "way out of trouble"
  - i.e. find a sequence of processes that, even when they demand their maximum resources, won't deadlock the system
  - this is a worst-case analysis
    - it may be that during the normal execution of processes, none ever demands its maximum in a way that causes deadlock
  - to perform a more optimal (less than worst-case) analysis is more complex, and also requires a record of future accesses

Deadlock Avoidance

- A safe state provides a safe "escape" sequence
- A deadlocked state is unsafe
- An unsafe state is not necessarily deadlocked
- A system may transition from a safe to an unsafe state if a request for resources is granted
  - ideally, check with each request for resources whether the system is still safe



Deadlock Avoidance

- Example 1:
  - 12 instances of a resource
  - At time t0, P0 holds 5, P1 holds 2, P2 holds 2
  - Available = 3 free instances

processes	max needs	Allocated/12
P0	10	5
P1	4	2
P2	9	2

Deadlock Avoidance

- Example 1:
  - 12 instances of a resource
  - At time t0, P0 holds 5, P1 holds 2, P2 holds 2
  - Available = 3 free instances

processes	max needs	Allocated/12
P0	10	5
P1	4	2
P2	9	2

- Even in worst case of max demands there is no deadlock
  - The system can find an ordered way of granting max requests such that deadlock is avoided
  - At any given stage, don't grant max requests to tasks whose max requests can't be satisfied
  - Do grant max requests to remaining tasks
    - After requesting and getting its max, each such task finishes and releases its resources, adding to the pool of available resources, etc.

Deadlock Avoidance

- Example 1:
  - 12 instances of a resource
  - At time t0, P0 holds 5, P1 holds 2, P2 holds 3
  - Available = 3 free instances

processes	max needs	Allocated/12
P0	10	5
P1	4	2
P2	9	3

Deadlock Avoidance

- Example 1 (cont):
  - Is the system in a safe state? Can I find a safe sequence?
  - Yes, I claim the sequence <P1, P0, P2> is safe.
    - P1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource
    - Then P1 releases all of its resources, so 5 free
    - Next, P0 requests its max (currently has 5, so needs 5 more) and holds 10, so that now 0 free
    - Then P0 releases all its held resources, so 10 free
    - Next P2 requests its max of 9, leaving 3 free and then releases them all

Deadlock Avoidance

- Example 1 (cont):
  - Is the system in a safe state? Can I find a safe sequence?
  - Yes the sequence <P1, P0, P2> is safe, and is able in the worst-case to request maximum resources for each process in the sequence, and release all such resources for the next process in the sequence
  - Can this system avoid deadlock? Yes, we can find a safe sequence.

Intuition: Deadlock Avoidance

- Example 2 shows a system moving from a safe to an unsafe state where there *could* be deadlock
  - No safe sequence of max requests could be found among all the possibilities
  - We found a case where there would be deadlock

Dijkstra’s Banker’s Algorithm

- *Generalizes* deadlock avoidance to multiple resources
  - Determines whether the system is in a safe state
- Before granting a request, run Banker’s Algorithm pretending as if request was granted
  - Does the worst-case analysis find such a hypothetical system is in a safe state?
  - If so, grant request.
  - If not, delay requestor, and wait for more resources to be freed.

Dijkstra's Banker's Algorithm

- As before, try to find a safe sequence
  - each process declares its maximum demands
    - must be less than total resources in system
- Advantages:
  - Generalization to multiple types of resources
  - Works for multiple instances of resources, unlike resource allocation graph
  - Adapts at run time to individual requests
  - Individual requests don't need to be known a priori, except for maximum demands, which is not completely unrealistic

Dijkstra's Banker's Algorithm

- Disadvantages:
  - Must know in advance the maximum resource usage for each process
  - Overly conservative in avoiding deadlock in the worst case

Banker's Algorithm Definitions

- Available resources in a vector or list **Available[j]**, j=1,...,m resource types
  - Available[j] = k means that there are k instances of resource R_j available
- Maximum demands in a matrix or table **Max[i,j]**, where i=1,...,n processes, and j=1,...,m resource types
  - Max[i,j] = k means that process i's maximum demands are for k instances of resource R_j
- Allocated resources in a matrix **Alloc[i,j]**
  - Alloc[i,j] = k means that process i is currently allocated k instances of resource R_j
- Needed resources in a matrix **Need[i,j]** = **Max[i,j]** - **Alloc[i,j]**

Banker’s Algorithm Definitions

- An example of the Alloc[i,j] matrix:

Resources

$$
\text{Processes}\quad
\begin{array}{r}
\\ \\ \\ \\ \text{Row } i \\ \\ \\ \\ \\
\end{array}
\begin{array}{c}
\begin{array}{cccccc} & & \text{Column } j & & & \end{array} \\
\left[\begin{array}{cccccc}
 & & 1 & & & \\
 & & 7 & & & \\
 & & 12 & & & \\
8 & 0 & 2 & 17 & 0 & 1 \\
 & & 0 & & & \\
 & & 4 & & & \\
 & & 0 & & & \\
 & & 1 & & &
\end{array}\right]
\end{array}
$$

$Alloc_i$ is shorthand for row i of matrix Alloc[i,j], i.e. the resources allocated to process i

Banker's Algorithm Definitions

- Some terminology:
  - let X and Y be two vectors. Then we say X ≤ Y if and only if X[i] ≤ Y[i] for all i.
  - Example:

$$V1 = \begin{bmatrix} 1 \\ 7 \\ 3 \\ 2 \end{bmatrix} \quad V2 = \begin{bmatrix} 0 \\ 3 \\ 2 \\ 1 \end{bmatrix} \quad V3 = \begin{bmatrix} 0 \\ 10 \\ 2 \\ 1 \end{bmatrix}$$

then V2 ≤ V1, but V3 ≰ V1, i.e. V3 is not less than or equal to V1

Banker's Algorithm

- Is the system in a safe state? Find a safe sequence:

1. Let Work and Finish be vectors length m and n respectively. Initialize Work = Available, and Finish[i]=false for i=0,...,n-1
2. Find a process i such that both
   - Finish[i]==false, and
   - $Need_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Alloc_i$

   Finish[i] = true

   Go to step 2.
4. If Finish[i]==true for all i, then the system is in a safe state

Intuition: if all prior processes give up all their resources, is there enough to meet the max needs of next process in the sequence?

Banker’s Algorithm Example

- Example 3:
  - 3 resources (A,B,C) with total instances available (10,5,7)
  - 5 processes
  - At time t0, the allocated resources **Alloc[i,j]**, max needs **Max[i,j]**, and available resources **Avail[j]**, are:

Alloc[i,j]

$$\begin{array}{c c} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} \end{array}$$

Max[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{bmatrix} \end{array}$$

Avail[j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 3 & 3 & 2 \end{bmatrix} \end{array}$$

Need[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix} \end{array}$$

Banker's Algorithm Example

- Is the system in a safe state? Is there a safe sequence?
  - Yes, the claim is that *<P1, P3, P4, P2, P0>* is a safe sequence
  - Find a process *i* such that $Need_i \leq$ Work (=Avail)
    - Which processes satisfy this condition?

Alloc[i,j]

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max[i,j]

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Avail[j]

A	B	C
3	3	2

Need[i,j]

	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Banker’s Algorithm Example

- claim is that *<P1, P3, P4, P2, P0>* is safe
  - P1 and P3 have $Need_i \leq$ Work (=Avail)
  - Choose P1: $Need_1$ = <1,2,2> ≤ Available = <3,3,2>
  - P1 takes all that it needs, then releases all held resources, which equals its maximum demands
  - Work = Work + $Alloc_1$ = <3,3,2> + <2,0,0> = <5,3,2>

Alloc[i,j]

$$\begin{array}{c c} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} & \left[\begin{array}{ccc} 0 & 1 & 0 \\ \mathbf{2} & \mathbf{0} & \mathbf{0} \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{array}\right] \end{array}$$

Max[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \left[\begin{array}{ccc} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{array}\right] \end{array}$$

Avail[j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \left[\begin{array}{ccc} 3 & 3 & 2 \end{array}\right] \end{array}$$

Need[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \left[\begin{array}{ccc} 7 & 4 & 3 \\ \mathbf{1} & \mathbf{2} & \mathbf{2} \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{array}\right] \end{array}$$

Banker's Algorithm Example

- claim is that *<P1, P3, P4, P2, P0>* is safe
  - Find process i with $\text{Need}_i \le \text{Work} = <5,3,2>$
  - Choose P3: $\text{Need}_3 = <0,1,1> \le \text{Available} = <5,3,2>$
  - P3 takes its max, releases all
  - $\text{Work} = \text{Work} + \text{Alloc}_3 = <5,3,2> + <2,1,1> = <7,4,3>$

Alloc[i,j]

$$\begin{array}{c c} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} \end{array}$$

Max[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{bmatrix} \end{array}$$

Avail[j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 3 & 3 & 2 \end{bmatrix} \end{array}$$

Need[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix} \end{array}$$

Banker's Algorithm Example

- claim is that *<P1, P3, P4, P2, P0>* is safe
  - Find process i with $Need_i \leq Work = <7,4,3>$
  - P0, P2 and P4 all satisfy this condition, so we can take them in any order.
  - Therefore, <P1, P3, P4, P2, P0> is safe!

Alloc[i,j]

$$\begin{array}{c|ccc} & A & B & C \\ \hline P0 & 0 & 1 & 0 \\ P1 & 2 & 0 & 0 \\ P2 & 3 & 0 & 2 \\ P3 & 2 & 1 & 1 \\ P4 & 0 & 0 & 2 \end{array}$$

Max[i,j]

$$\begin{array}{ccc} A & B & C \\ \left[\begin{matrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{matrix}\right] \end{array}$$

Avail[j]

$$\begin{array}{ccc} A & B & C \\ \left[\begin{matrix} 3 & 3 & 2 \end{matrix}\right] \end{array}$$

Need[i,j]

$$\begin{array}{ccc} A & B & C \\ \left[\begin{matrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{matrix}\right] \end{array}$$

Deadlock Avoidance

- Questions about Banker's Algorithm:
  - If it finds a safe sequence, are there other safe sequences, i.e. is the safe sequence that is found unique?
    - The safe sequence is not necessarily unique. There may be others.
    - Simple case: if P1 and P2 both can have their max needs satisfied by available resources, then <P1, P2> and <P2,P1> are both safe sequence
    - In Banker's example 3, another safe sequence was <P3,P1,P0,P2,P4>

Complexity of the algorithm

- Complexity of Banker's Algorithm is order $O(m*n^2)$
  - If there are n processes, then in the worst case the processes are ordered such that each iteration of the banker's algorithm must evaluate all remaining processes before the last one satisfies $Need_i \leq Work$
    - the 1st time, n processes have to compare if their $Need_i \leq Work$
    - the 2nd time, n-1 processes have to be compared
    - Thus, in the worst case, there are $n + (n-1) + (n-2) + ... + 2 + 1 = n(n+1)/2$ comparisons, which is proportional to n^2 complexity
    - Each vector comparison requires m individual comparisons, since there are m resource types, so total complexity is $O(m*n^2)$

When does the system use Banker’s Alg?

- Banker’s Algorithm determines whether the system is in a safe state
- Suppose we have a *new* request, and the current system is in a safe state
  - Should we grant the new request?
  - Yes, if it leaves the system in a safe state.

Resource-Request Algorithm

Let $Request_i$ be a new request vector for resources for process P_i

1. If $Request_i \leq Need_i$, go to step 2. Else, the new request exceeds the maximum claim. Exit.
2. If $Request_i \leq$ Available, go to step 3. Else, process P_i must wait because there aren't enough available resources
3. Temporarily modify Available[j], Need[i,j], and Alloc[i,j]
   - Avail -= $Request_i$
   - $Alloc_i$ += $Request_i$
   - $Need_i$ -= $Request_i$

Execute the Banker's algorithm. If system is in a safe state, grant the request and update Avail, Need, and Alloc.

Deadlock Avoidance

- Recall Example 3:
  - 3 resources (A,B,C) with total instances available (10,5,7)
  - 5 processes
  - At time t0, the allocated resources Alloc[i,j], Max needs Max[i,j], and Available resources Avail[j], are:

Alloc[i,j]

$$\begin{array}{c c} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} \end{array}$$

Max[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{bmatrix} \end{array}$$

Avail[j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 3 & 3 & 2 \end{bmatrix} \end{array}$$

Need[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix} \end{array}$$

where Need[i,j] is computed given Alloc[i,j] and Max[i,j]

Banker's Algorithm found that this was in a safe state, with safe sequence <P1, P3, P4, P2, P0>

Deadlock Avoidance

- Suppose we have a new request from process P1, Request_1 = <1,0,2>.
  - Can the system satisfy this request without becoming unsafe? Execute the Resource-Request Algorithm.
  1. $\text{Request}_1 \leq \text{Need}_1$ = <1,2,2> Good!
  2. $\text{Request}_1 \leq$ Available = <3,3,2> Good!
  3. Temporarily recompute Avail, Need, and Alloc
     - see next slide

Deadlock Avoidance

temporarily modified

Alloc'[i,j]

$$\begin{array}{c} \\ P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \left[\begin{array}{ccc} 0 & 1 & 0 \\ \mathbf{3} & \mathbf{0} & \mathbf{2} \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{array}\right] \end{array}$$

Max[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \left[\begin{array}{ccc} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{array}\right] \end{array}$$

Avail'[j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \left[\begin{array}{ccc} \mathbf{2} & \mathbf{3} & \mathbf{0} \end{array}\right] \end{array}$$

Need'[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \left[\begin{array}{ccc} 7 & 4 & 3 \\ \mathbf{0} & \mathbf{2} & \mathbf{0} \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{array}\right] \end{array}$$

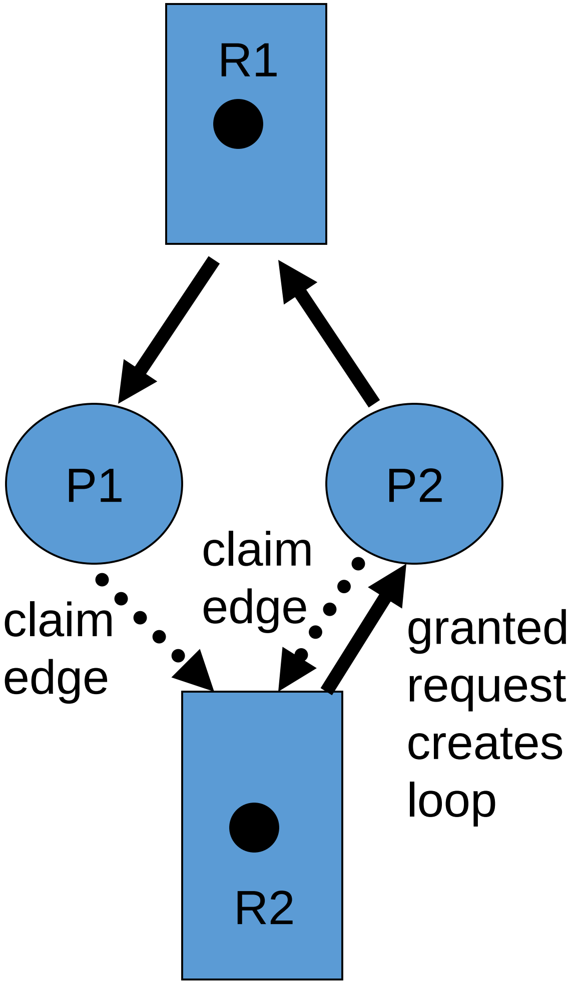
- Execute Banker's Algorithm on this revised state. We find that the sequence <P1,P3,P4,P0,P2> is safe.
- Thus we can grant the Request_1 immediately, and permanently update the matrices and vectors.

Deadlock Avoidance

- Given this new state, suppose we have a 2nd request:
  - from P4, such that $Request_4$ = <3,3,0>. Show that this cannot be granted because
    - $Request_4 \leq$ Available = <2,3,0>.
    - That is, there are not enough available resources.
  - from P0, such that $Request_0$ = <0,2,0>. Show that this cannot be granted because no safe sequence can be found.

Deadlock Avoidance

- Using a Resource Allocation Graph for deadlock avoidance
  - each process identifies possible future claims, drawn as claim edges (dotted lines)
  - If converting a claim edge to a request edge creates a loop, then don't grant request
    - limited applicability - only valid when there is 1 instance of each resource
  - example: Granting P2's request for R2 creates a loop, so don't grant it



Solutions to Handling Deadlocks

1. Prevention by OS
   - provide methods to guarantee that at least 1 of the 4 necessary conditions for deadlock does not hold
2. Avoidance by OS
   - the OS is given advanced information about process requests for various resources
   - this is used to determine whether there is a way for the OS to satisfy the resource requests and avoid deadlock

Solutions to Handling Deadlocks (2)

3. Detection and Recovery by OS
    - Analyze existing system resource allocation, and see if there is a **sequence of releases** that satisifies every process' needs.
    - If not, then deadlock is detected, so must recover – drastic action needed, such as killing the affected processes!

Deadlock Detection

- When/how often should the detection algorithm run?
  - Depends on how often deadlock is likely to occur
  - Depends on how quickly deadlock grows after it occurs, i.e. how many processes get pulled into deadlock and on what time scale
  - Could check at each resource request – this is costly
  - Could check periodically – but what is a good time interval?
  - Could check if CPU utilization suddenly drops – this might be an indication that there's deadlock, and processes are no longer executing, but what's a good threshold?
  - Could check if resource utilization exceeds some threshold, but what's a good threshold?

Deadlock Recovery

- After OS has detected which processes are deadlocked, then OS can:
  - Terminate all processes
  - Terminate one process at a time until the deadlock cycle is eliminated
    - Check if there is still deadlock after each process is terminated. If not, then stop.
  - Preempt some processes – temporarily take away a resource from current owner and give it to another process but don't terminate process
    - e.g. give access to a laser printer – this is risky if you're in middle or printing a documents
  - Rollback some processes to a checkpoint – assuming that processes have saved their state at some checkpoint

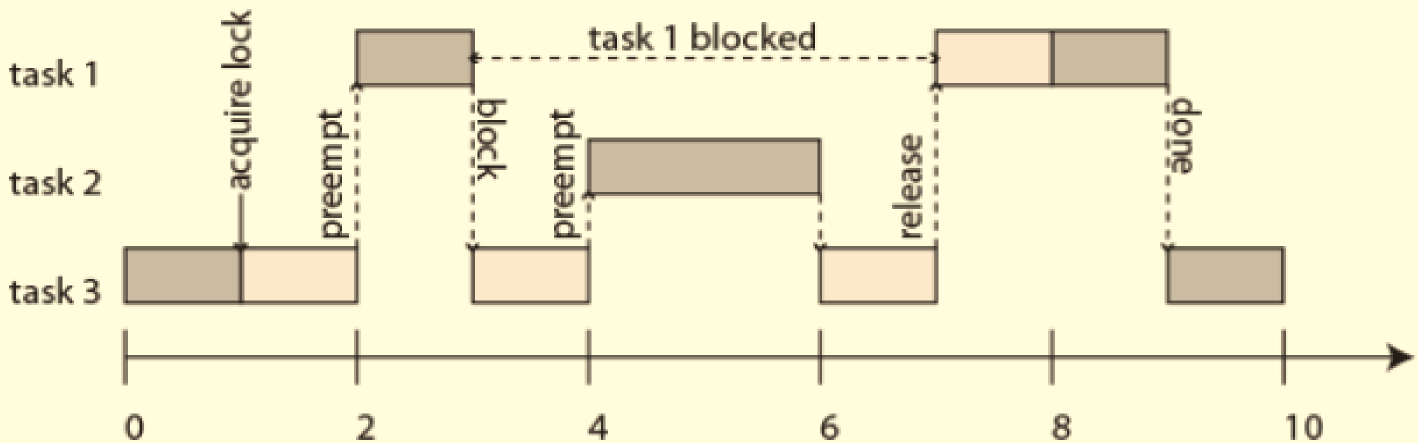
Mars Rover Pathfinder

The Mars Rover Pathfinder landed on Mars on July 4th, 1997. A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data. The problem was diagnosed on the ground as priority inversion, where a low priority meteorological task was holding a lock blocking a high-priority task while medium priority tasks executed.

Source: RISKS-19.49 on the comp.programming.threads newsgroup, December 07, 1997, by Mike Jones (mbj@MICROSOFT.com).

Priority Inversion problem

Priority Inversion: A Hazard with Mutexes



Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.