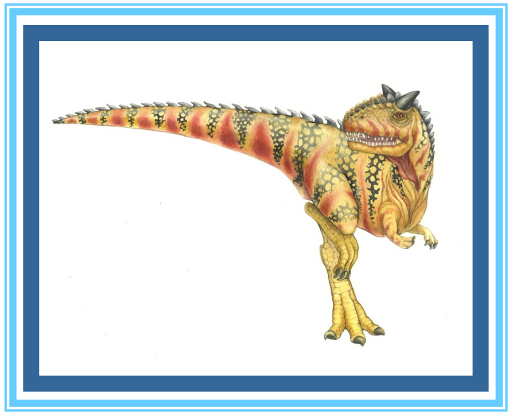
Spring 2022 COMP 3511 Review #6



Coverages

- Deadlock
- Memory Management – contiguous memory allocation

DEADLOCK

The Deadlock Problem

- A set of blocked processes each holding resource(s) while waiting to acquire more resource(s) held by another process in the set.

- Example 1
  - A system has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example 2
  - semaphores *A* and *B*, initialized to 1

```
     P0               P1
wait (A);          wait(B)
wait (B);          wait(A)
```

Deadlock Characterization

- If Deadlock occurs, **four** conditions must hold simultaneously
- Mutual exclusion
  - only one process at a time can use a resource.
- Hold and wait
  - a process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption
  - a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait
  - there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Methods for Handling Deadlocks

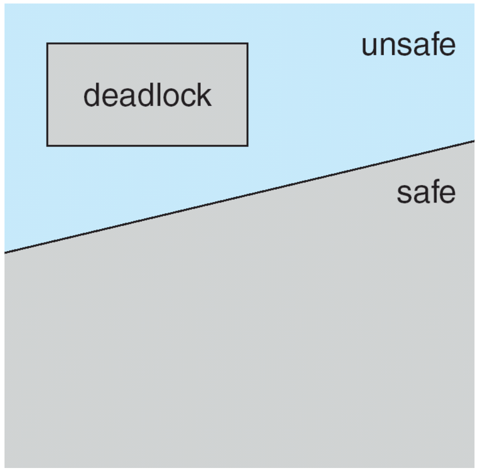
- Ensure that the system will ***never*** enter a deadlock state:
  - **Deadlock prevention**: it provides a set of methods to ensure at least one of the necessary conditions cannot hold
  - **Deadlock avoidance**: this requires additional information given in advance concerning which resources a process will request and use during its lifetime. Within such knowledge, the OS can decide for each resource request whether or not a process should wait
- **Deadlock detection** - allow the system to enter a deadlock state, periodically detect deadlock and then recover from it
- Many OSes just ignore the problem and pretend that deadlocks never occur in the system

Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible
- **No Preemption** –
  - If a process holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. This might not be practically feasible

Deadlock Avoidance

- **Avoidance** ⇒ ensure that a system never enters an unsafe state.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

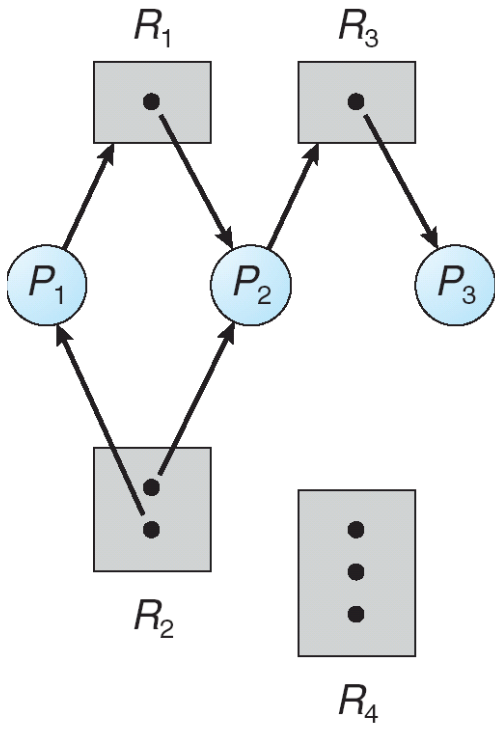


- System is in safe state if there exists a safe sequence of all processes: sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with j<i.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

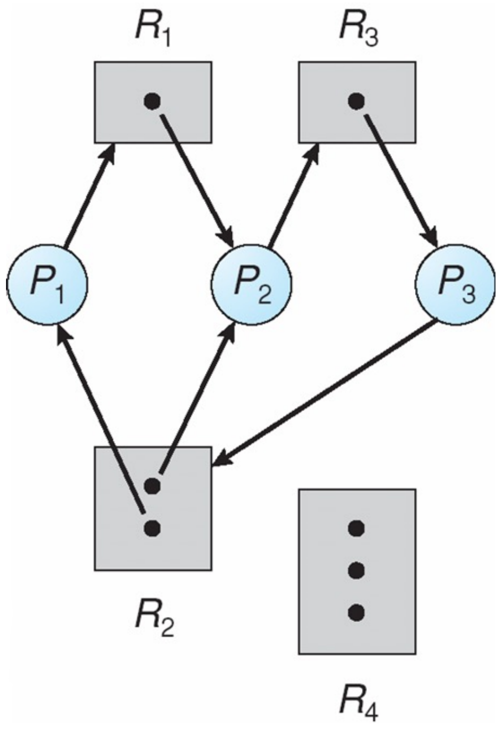
Resource-Allocation Graph

- A set of vertices *V* and a set of edges *E*.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows: request, use, release
- Request edge – directed edge $P_i \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

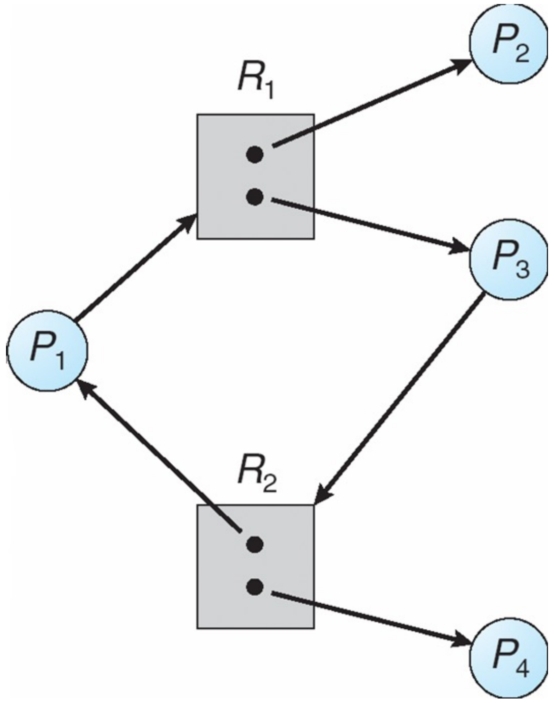
Resource Allocation Graph: Examples



A resource allocation graph with no cycle no deadlock



A resource allocation graph with a deadlock - cycle



A resource allocation graph with a cycle but no deadlock

Basic Facts & Limitation

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if each resource has only one instance, then deadlock.
  - if several instances per resource type, possibility of deadlock.
- Resource-Allocation-Graph is useful to detect deadlock if there is only one instance of each resource type
- Resource-Allocation-Graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type
- Thus, the banker's algorithm is used to tackle a more general deadlock avoidance problem

Banker’s Algorithm Terminologies

- We use capital letters (e.g. A, B, C) to represent resources
  - Each resource can have multiple instances
- We use P0, P1, P2, to represent processes
- The key idea of banker’s algorithm
  - Each process must declare a priori maximum usage
  - When a process requests a resource it may have to wait – check to see if this allocation results in a safe state or not
  - When a process gets all its resources it must return them in a finite amount of time after use
  - This is analogous to banking load system, which has a maximum amount, total, that can be loaned at one time to a set of businesses each with a credit line.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Usage of Banker's Algorithm

- Safety algorithm
  - Finding out whether a system is in a safe state
- Resource-Request algorithm
  - Determine whether a request can be safely granted
  - Please note that resource-request algorithm will invoke safety algorithm
  - Nearly the **SAME** as Safety, except extra initialization and error checking
- Detection algorithm
  - Very similar to banker's algorithm
  - Initialization conditions are different, no need to compute the Need matrix…

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively

    Initialize:

    (a) $\boldsymbol{Work = Available}$

    (b) for $\boldsymbol{i = 0, 1, \ldots, n-1}$

    if $\boldsymbol{Allocation_i = 0}$ **or** $\boldsymbol{Need_i = 0}$**:** $\boldsymbol{Finish[i] = \text{true}; Work = Work + Allocation_i}$

    else: $\boldsymbol{Finish[i] = \text{false}}$

2. Find an index *i* such that both:

    (a) $\boldsymbol{Finish[i] == false}$

    (b) $\boldsymbol{Need_i \leq Work}$

    If no such *i* exists, go to step 4.

3. $\boldsymbol{Work = Work + Allocation_i}$
    $\boldsymbol{Finish[i] = true}$
    go to step 2.

4. If $\boldsymbol{Finish\ [i] == true}$ for all $\boldsymbol{i}$, then the system is in a safe state, otherwise, it is unsafe

Safety Algorithm - Example

- Consider the following snapshot of a system, is this system safe?

	Allocation					Max					Available			
	A	B	C	D		A	B	C	D		A	B	C	D
P0	0	0	1	2		0	0	3	2		2	1	2	0
P1	2	0	0	0		2	7	5	0					
P2	0	0	3	4		6	6	5	6					
P3	2	3	5	4		4	3	5	6					
P4	0	3	3	2		0	6	5	2					

Safety Algorithm - Example

What is the content of the matrix Need denoting the number of resources needed by each process?

Max – Allocation = Need (matrix)

	Need			
	A	B	C	D
P0	0	0	2	0
P1	0	7	5	0
P2	6	6	2	2
P3	2	0	0	2
P4	0	3	2	0

Step 1

Sequence = <>

Initialize "Finish" and "Work"

	Allocation					Need				Finish	Work
	A	B	C	D		A	B	C	D		(2,1,2,0)
P0	0	0	1	2		0	0	2	0	false	
P1	2	0	0	0		0	7	5	0	false	
P2	0	0	3	4		6	6	2	2	false	
P3	2	3	5	4		2	0	0	2	false	
P4	0	3	3	2		0	3	2	0	false	

Step 2

Sequence = <P0>

Find an index *i* such that both:

(a) ***Finish*[*i*] == *false***

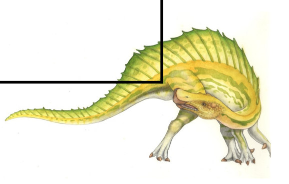
(b) ***Need*$_i$ ≤ *Work***

i=0 satisfies the conditions

***Work* = *Work* + *Allocation*$_i$**

Finish*[*i*] = *true

	Allocation					**Need**				**Finish**	**Work**
	A	**B**	**C**	**D**		**A**	**B**	**C**	**D**		~~**(2,1,2,0)**~~ **(2,1,3,2)**
P0	0	0	1	2		0	0	2	0	~~false~~ true	
P1	2	0	0	0		0	7	5	0	false	
P2	0	0	3	4		6	6	2	2	false	
P3	2	3	5	4		2	0	0	2	false	
P4	0	3	3	2		0	3	2	0	false	



Step 3

Sequence = <P0,P3>

Find an index *i* such that both:

(a) ***Finish*[*i*] == *false***

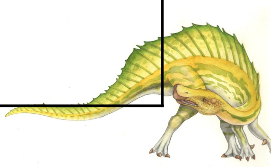
(b) ***Need$_i$* ≤ *Work***

i=3 satisfies the conditions

Work* = *Work* + *Allocation$_i$

Finish*[*i*] = *true

	Allocation					Need				Finish	Work
	A	B	C	D		A	B	C	D		~~(2,1,2,0)~~ ~~(2,1,3,2)~~ (4,4,8,6)
P0	0	0	1	2		0	0	2	0	true	
P1	2	0	0	0		0	7	5	0	false	
P2	0	0	3	4		6	6	2	2	false	
P3	2	3	5	4		2	0	0	2	~~false~~ true	
P4	0	3	3	2		0	3	2	0	false	



Step 4

Sequence = <P0,P3,P4>

Find an index *i* such that both:

(a) ***Finish*[*i*] == *false***

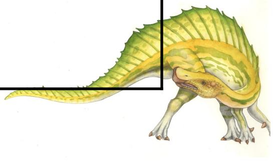
(b) ***Need**$_i$* **≤ *Work***

i=4 satisfies the conditions

***Work = Work + Allocation**$_i$*

Finish*[*i*] = *true

	Allocation					Need				Finish	Work
	A	B	C	D		A	B	C	D		~~(2,1,2,0)~~ ~~(2,1,3,2)~~ ~~(4,4,8,6)~~ (4,7,11,8)
P0	0	0	1	2		0	0	2	0	true	
P1	2	0	0	0		0	7	5	0	false	
P2	0	0	3	4		6	6	2	2	false	
P3	2	3	5	4		2	0	0	2	true	
P4	0	3	3	2		0	3	2	0	~~false~~ true	



Step 5

Sequence = <P0,P3,P4,P1>

Find an index *i* such that both:

(a) ***Finish*[*i*] == *false***

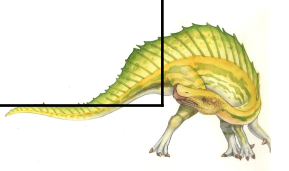
(b) $\boldsymbol{Need_i \leq Work}$

i=1 satisfies the conditions

$\boldsymbol{Work = Work + Allocation_i}$

Finish*[*i*] = *true

	Allocation					Need				Finish	Work
	A	B	C	D		A	B	C	D		~~(2,1,2,0)~~ ~~(2,1,3,2)~~ ~~(4,4,8,6)~~ ~~(4,7,11,8)~~ (6,7,11,8)
P0	0	0	1	2		0	0	2	0	true	
P1	2	0	0	0		0	7	5	0	~~false~~ true	
P2	0	0	3	4		6	6	2	2	false	
P3	2	3	5	4		2	0	0	2	true	
P4	0	3	3	2		0	3	2	0	true	



Step 6

Sequence = <P0,P3,P4,P1,P2>

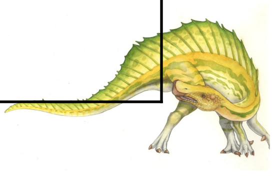
Find an index *i* such that both:

(a) ***Finish*[*i*] == *false***
(b) ***Need*$_i$ ≤ *Work***

i=2 satisfies the conditions

***Work* = *Work* + *Allocation*$_i$**
Finish*[*i*] = *true

	Allocation					Need				Finish	Work
	A	**B**	**C**	**D**		**A**	**B**	**C**	**D**		~~(2,1,2,0)~~ ~~(2,1,3,2)~~ ~~(4,4,8,6)~~ ~~(4,7,11,8)~~ ~~(6,7,11,8)~~ (6,7,14,12)
P0	0	0	1	2		0	0	2	0	true	
P1	2	0	0	0		0	7	5	0	true	
P2	0	0	3	4		6	6	2	2	~~false~~ true	
P3	2	3	5	4		2	0	0	2	true	
P4	0	3	3	2		0	3	2	0	true	



Safety Algorithm: Final Result

- Is the system in a safe state? Why?
  - From the previous step, all Finish[i] become true
  - The allocation should be safe right now, with a sequence of process execution: <P0, P3, P4, P1, P2>
  - Note: **There may be more than one possible sequence of process execution**. It is because in **step 2 of the safety algorithm, we may have more than one possible choices**

	Resources available after each process finished			
	A	**B**	**C**	**D**
P0	2	1	3	2
P3	4	4	8	6
P4	4	7	11	8
P1	6	7	11	8
P2	6	7	14	12

Another Example of Banker's algorithm (Safety)

- Consider the following snapshot of a system:

	Allocation				Max			
	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0
P_1	1	2	3	1	1	6	5	2
P_2	1	3	6	5	2	3	6	6
P_3	0	6	3	2	0	6	5	2
P_4	0	0	1	4	0	6	5	6

- Using the Banker's algorithm, determine whether each of the following states is safe or not. If the system is safe, specify one execution sequence in that all the processes may complete. Otherwise, briefly justify why the state is unsafe.
- See part (a) and part (b) and the next few slides

Another Example of Banker's algorithm (Safety)

- (a) Available = (0, 3, 0, 1).

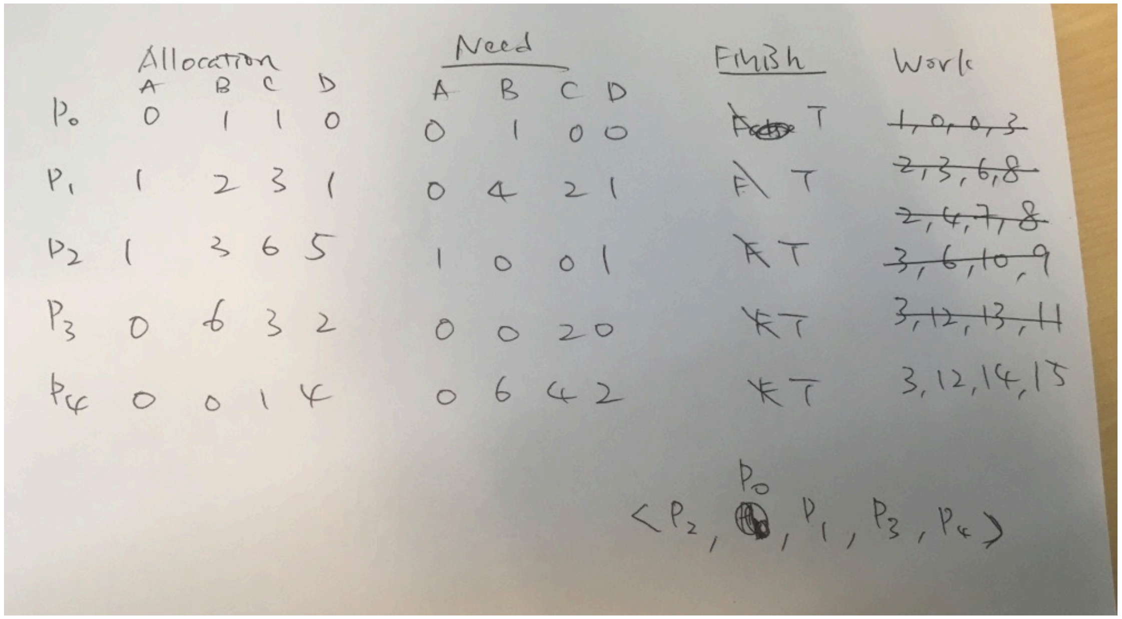
- Answer: It is unsafe.
- Only P0 can be finished. With available vector (0, 4, 1, 1), which cannot satisfy remaining processes' needs.

	Allocation			
	A	B	C	D
P_0	0	1	1	0
P_1	1	2	3	1
P_2	1	3	6	5
P_3	0	6	3	2
P_4	0	0	1	4

	Need			
	A	B	C	D
P_0	0	1	0	0
P_1	0	4	2	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

Another Example of Banker's algorithm (Safety)

- (b) Available = (1, 0, 0, 3)
- In this example, we pick the smallest index if we have >1 choices
- Answer: Yes. The order is P2, P0, P1, P3, P4



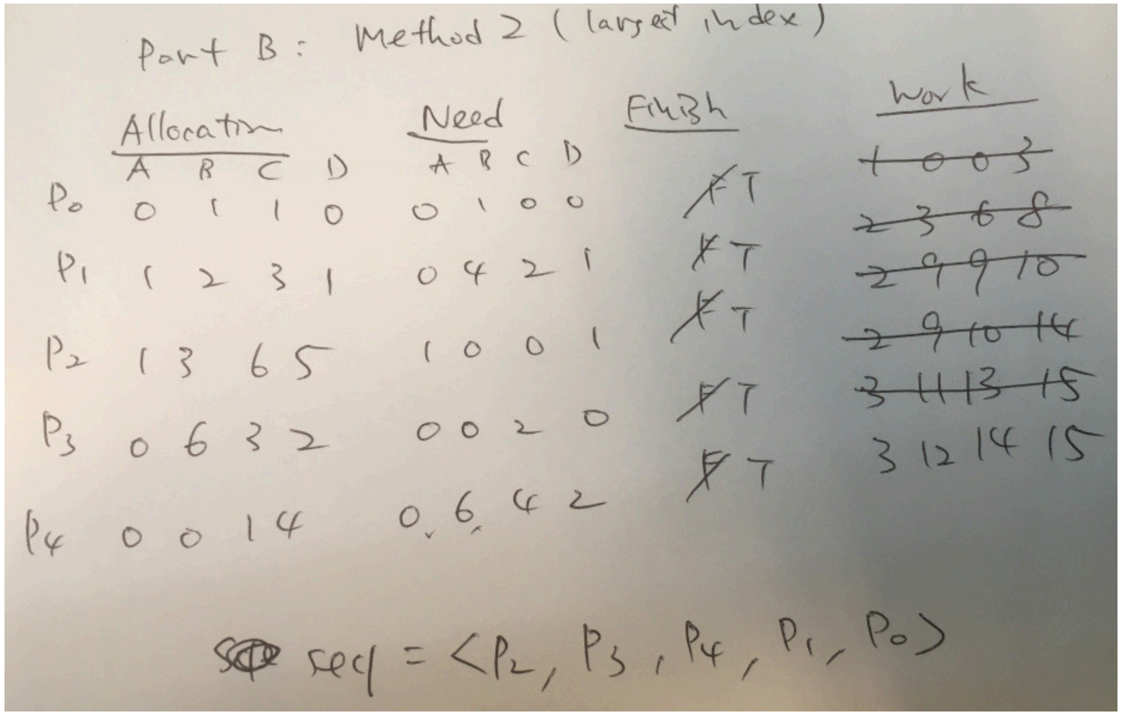
	Allocation				Need				Finish	Work
	A	B	C	D	A	B	C	D		
P_0	0	1	1	0	0	1	0	0	~~F~~ T	~~1, 0, 0, 3~~
P_1	1	2	3	1	0	4	2	1	~~F~~ T	~~2, 3, 6, 8~~ ~~2, 4, 7, 8~~
P_2	1	3	6	5	1	0	0	1	~~F~~ T	~~3, 6, 10, 9~~
P_3	0	6	3	2	0	0	2	0	~~F~~ T	~~3, 12, 13, 11~~
P_4	0	0	1	4	0	6	4	2	~~F~~ T	3, 12, 14, 15

P_0

$\langle P_2$, [illegible], P_1, P_3, $P_4 \rangle$

Another Example of Banker's algorithm (Safety)

- (b) Available = (1, 0, 0, 3).
- In this example, we pick the largest index if we have >1 choices
- Answer: An alternative solution: The order is P2, P3, P4, P1, P0
- Banker's algorithm may have more than one possible solutions.



Part B: Method 2 (largest index)

	Allocation				Need				Finish
	A	B	C	D	A	B	C	D	
P_0	0	1	1	0	0	1	0	0	~~F~~ T
P_1	1	2	3	1	0	4	2	1	~~F~~ T
P_2	1	3	6	5	1	0	0	1	~~F~~ T
P_3	0	6	3	2	0	0	2	0	~~F~~ T
P_4	0	0	1	4	0	6	4	2	~~F~~ T

Work

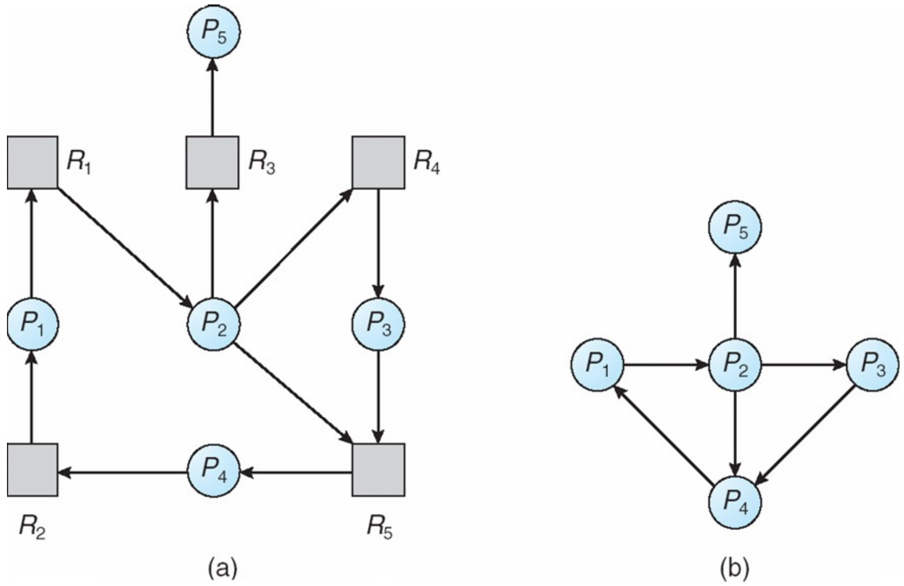
~~1 0 0 3~~
~~2 3 6 8~~
~~2 9 9 10~~
~~2 9 10 14~~
~~3 11 13 15~~
3 12 14 15

seq = <P_2, P_3, P_4, P_1, P_0>

Deadlock Detection

Maintain wait-for graph if each resource has a single instance

Periodically invoke an algorithm that searches for a cycle in the graph. If there is a *cycle* => a deadlock



Resource-Allocation Graph Corresponding wait-for graph

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \dots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

Very similar to banker's algorithm
Initialization conditions are different, no need to compute the Need matrix…

Detection Algorithm - Example

- Five processes: P_0 through P_4;
- Three resource types:
  A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Detection Algorithm - Example

- P_2 requests an additional instance of type C.

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

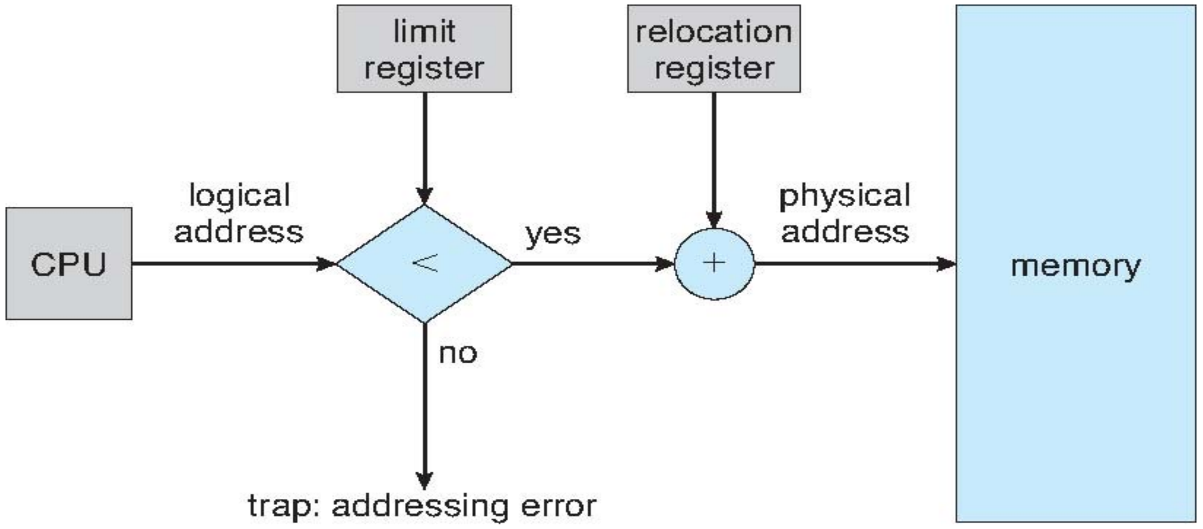
- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Memory Management Part 1

CONTIGUOUS MEMORY ALLOCATION

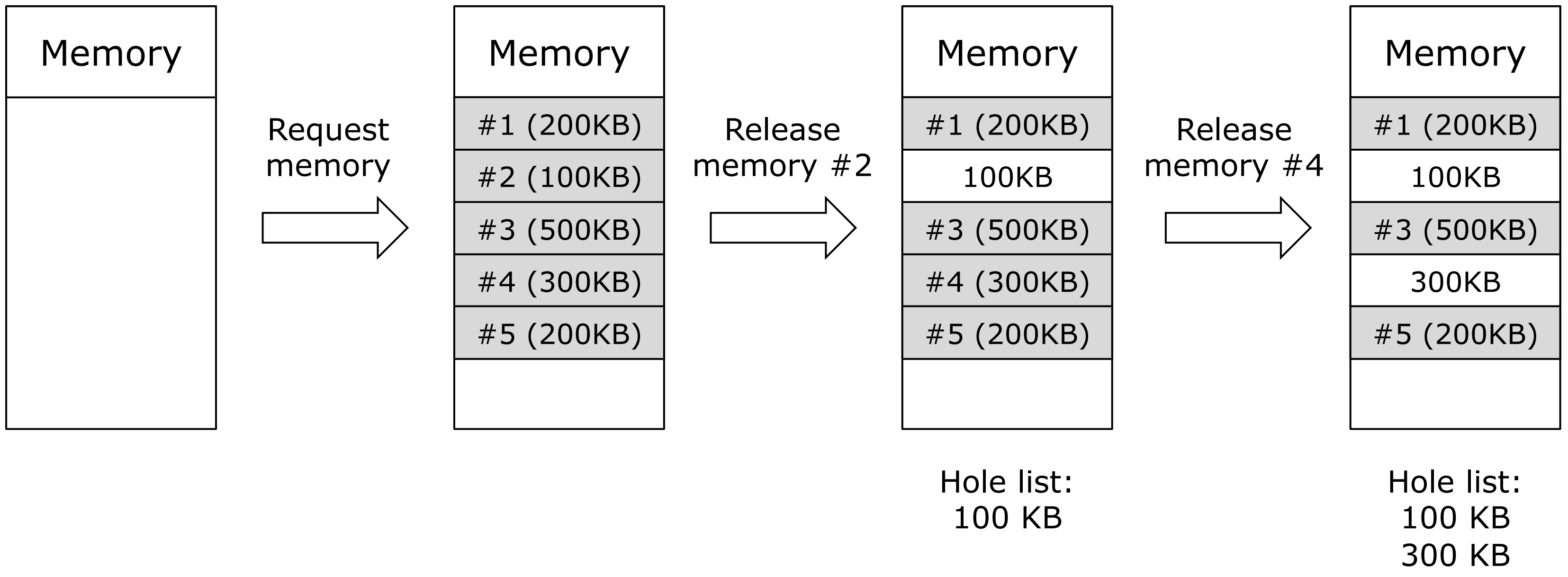
Contiguous Memory Allocation

- In a multi-programmed OS, several user processes reside in memory at the same time.
- In contiguous memory allocation, each process contained in a single section of memory that is contiguous to the section containing the next process.
- Relocation register and limit register are used to protect user processes from each other and from OS
  - Relocation register contains value of the smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address dynamically



Memory Hole

- Hole – block of available memory; holes of various size are scattered throughout memory



Dynamic Storage-Allocation Problem

How to satisfy a request of (variable) size *n* from a list of free holes?

- First-fit: Allocate the *first* hole that is big enough
  - No need to searching the entire list
- Best-fit: Allocate the *smallest* hole that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole – intended not use it
- Worst-fit: Allocate the *largest* hole
  - Must also search entire list
    - Really?
      - Find the maximum element in a set
      - Remove the maximum element from a set
      - Insert a new element into a set
  - Produces the largest leftover hole – intended to reuse the remaining hole

Example Question

- Given five memory partitions of 200 KB, 600 KB, 500 KB, 100 KB, and 400 KB (in order)
- How would each of the first-fit, best-fit, and worst-fit algorithms place processes of 150 KB, 350 KB, 450 KB, and 550 KB (in order)?
- Which algorithm makes the most efficient use of memory?

First-Fit Example

	200KB	600KB	500KB	100KB	400KB
150KB	√				
350KB		√			
450KB			√		
550KB wait					

Best-Fit Example

	200KB	600KB	500KB	100KB	400KB
150KB	√				
350KB					√
450KB			√		
550KB		√			

Worst-Fit Example

	200KB	600KB	500KB	100KB	400KB
150KB		√			
350KB			√		
450KB wait					
550KB wait					

- Best-Fit turns out to be the best in this case