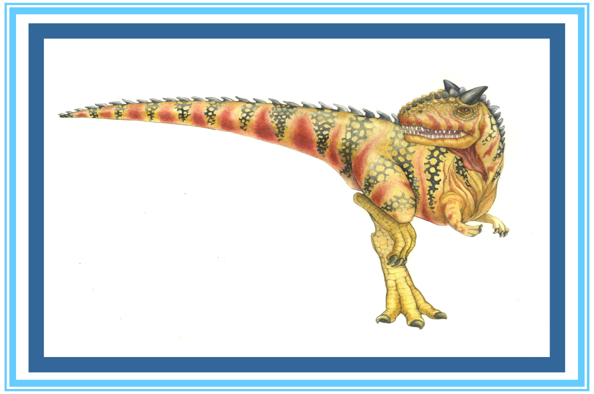
# Chapter 7: Deadlocks_2

School of Computing, Gachon Univ.
Jungchan Cho



Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

가천대학교 AI·소프트웨어학부
Gachon University

# Chapter 7: Deadlocks

- Deadlock Concept
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance

# Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state
  - Deadlock Prevention
  - Deadlock Avoidance
- Allow the system to enter a deadlock state and then **recover**
  - Deadlock Detection
  - Deadlock Recovery

- **Ignore** the problem and pretend that deadlocks never occur in the system (**Ostrich Algorithm**)
  - actually used by most operating systems, including Linux and Windows
  - up to the application developer to handle deadlocks

# Chapter 7: Deadlocks

- Deadlock Concept
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance

# Deadlock Prevention

- For deadlock to occur, each of the four necessary conditions (mutual exclusion, hold and wait, no preemption, circular wait) must hold
- **Deadlock prevention**
  - Ensure ***at least one*** of above conditions cannot hold – deadlock is **impossible!!**


## 1. *No* Mutual Exclusion

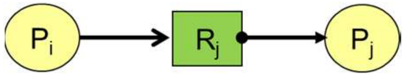
- Make all resources sharable at same time (e.g., read-only file is OK)
- *Practically impossible*. Some resources are non-sharable (e.g., mutex lock, printers)

> - 4 **necessary conditions** for Deadlock
>   1. **Mutual exclusion**
>   2. **Hold and wait**
>   3. **No preemption**
>   4. **Circular wait**

# Deadlock Prevention (Cont.)

- 4 **necessary conditions** for Deadlock
  1. **Mutual exclusion**
  2. **Hold and wait**
  3. **No preemption**
  4. **Circular wait**

## 2. No Hold and Wait




- must guarantee that whenever a process requests a resource, it does not hold any other resources
- **Total allocation**: process requests and be allocated all its resources before it begins execution
  - e.g., A process copies data from DVD drive to a file on disk, sort the file, and then prints the results to a printer
  - The process must initially request the DVD drive, disk file, and printer
    - If all resources are not available must wait
    - hold all resources until process terminates; then release all resources

# Deadlock Prevention (Cont.)

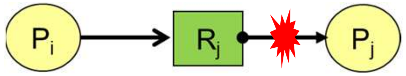
- 4 **necessary conditions** for Deadlock
  1. **Mutual exclusion**
  2. **Hold and wait**
  3. **No preemption**
  4. **Circular wait**

- Alternative protocol: process can request resources only when it has none
- Problems of *No* hold and wait
  - Low resource utilization
    - Resources may be allocated but unused for a long period – total allocation
  - Starvation
    - A process that needs several popular resources may have to wait indefinitely – at least one popular resource it needs is always allocated to some other process

# Deadlock Prevention (Cont.)

- 4 **necessary conditions** for Deadlock
  1. **Mutual exclusion**
  2. **Hold and wait**
  3. **No preemption**
  4. **Circular wait**

## 3. *Allow* Preemption




- A lock can be taken away (preempted) from current owner
- Often applied to resources whose **state** can be easily saved and restored
- Example: preemptive CPU scheduling - RR
  - P1 is using (holding) CPU, P2 is waiting in ready queue
  - P1's time quantum has expired – preempt resource (CPU) from P1.
  - A preempted process can restart only if it can restore previous state: Save P1's register to PCB1, restore P2's register from PCB2
- Problem: OK for CPU but Basically ***impossible*** for some resources
  - E.g., Printers, mutex locks, semaphores

# Deadlock Prevention (Cont.)

## 4. No Circular Wait

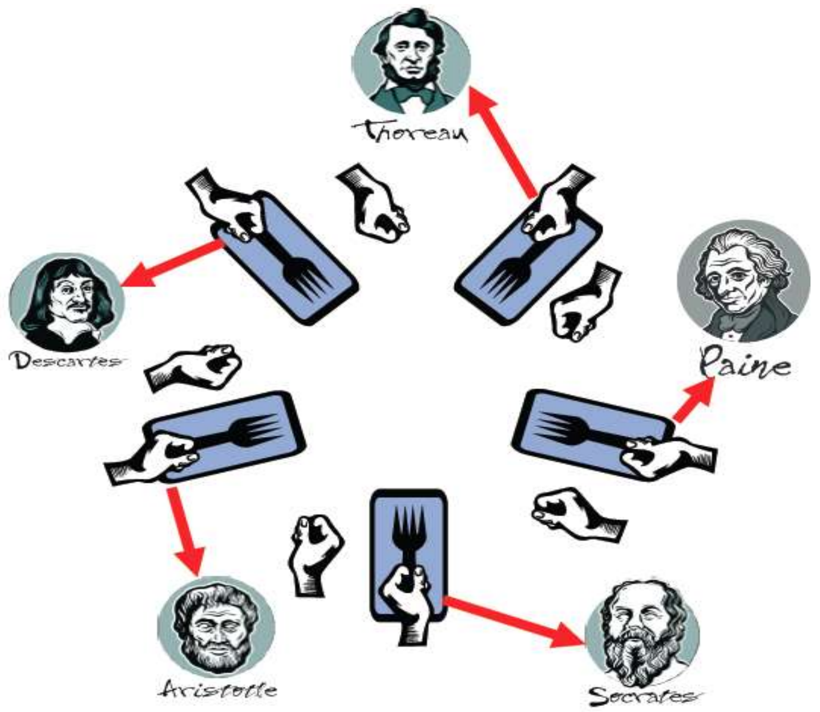
- impose a **total ordering** of all resource types, and require that each process requests resources in an ***increasing order*** of enumeration
- Example

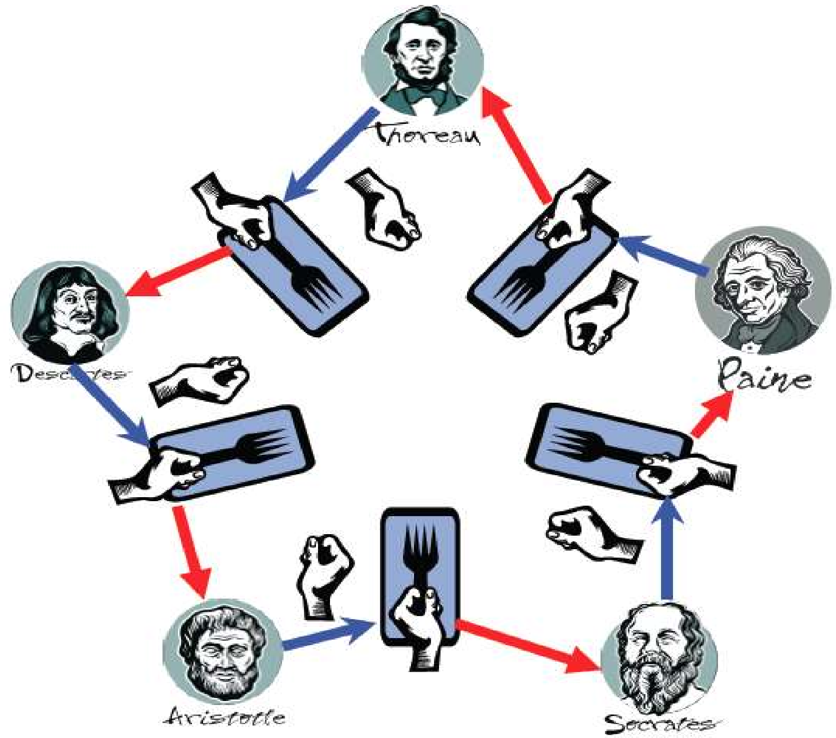
$$F(disk\ drive) = 10$$
$$F(printer) = 20$$
$$F(tape\ drive) = 30$$

- Each process can request resources only in an increasing order of enumeration - a process requests $R_i$. Then, it request instances of resource type $R_j$ if and only if $F(R_j) > F(R_i)$.
- Circular wait can never happen

# Example of Denying Circular Wait

- Dining Philosophers

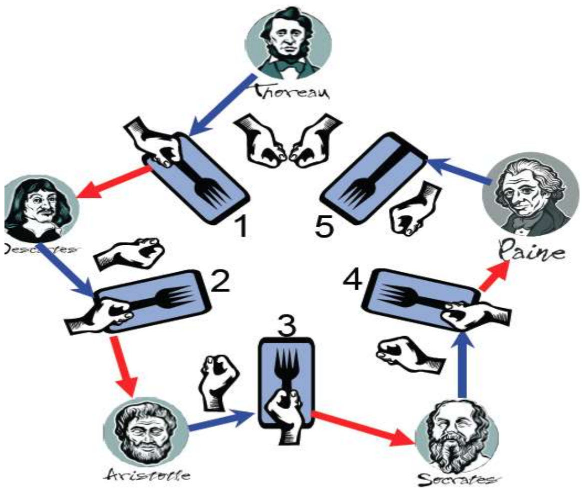


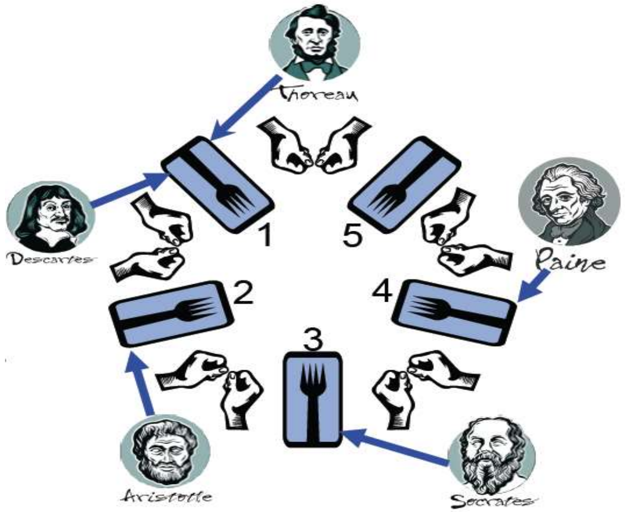
# Example of Denying Circular Wait

- Dining Philosophers

```
void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```




```
void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```



# Deadlock Prevention

- Summary of prevention schemes
  - Deny a (one) necessary condition for deadlocks among the 4 necessary conditions
  - Serious resource waste
    - Low device utilization, Reduced system throughput
  - May incur high cost to redesign the system
- **Deadlock avoidance**!

# Chapter 7:  Deadlocks

- Deadlock Concept
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance

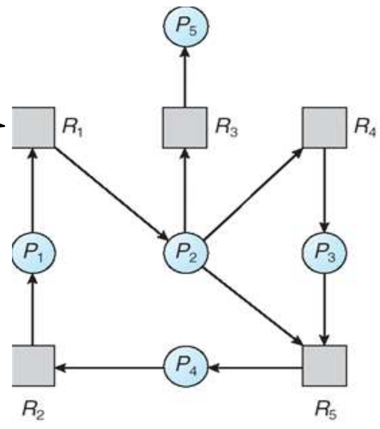
# Deadlock Avoidance

- Require additional **information** about how resources are to be requested
  - System knowledge of complete sequence of requests and releases of resources
    - Example: P1 requests first hard disk then printer. Then release both resources, P2 requests first printer then hard drive.
  - System can decide whether the process should wait in order to avoid possible deadlock
- Construct algorithm so that **circular-wait** can ***never*** exist
- What **Information** (**state**) is required from each process?
  - **Process** declares ***maximum number of resources*** of each type it needs
    - P1: I want to use 2 hard disk instances

# System Model

- What information in **state**?
  - For each **resource type state**,
    - 1. *Current* amount **available** instances
    - 2. *Current* amount **allocated** to each process
    - 3. *Future* amount **needed** by each process (maximum) (declared by each process – prev. slide)

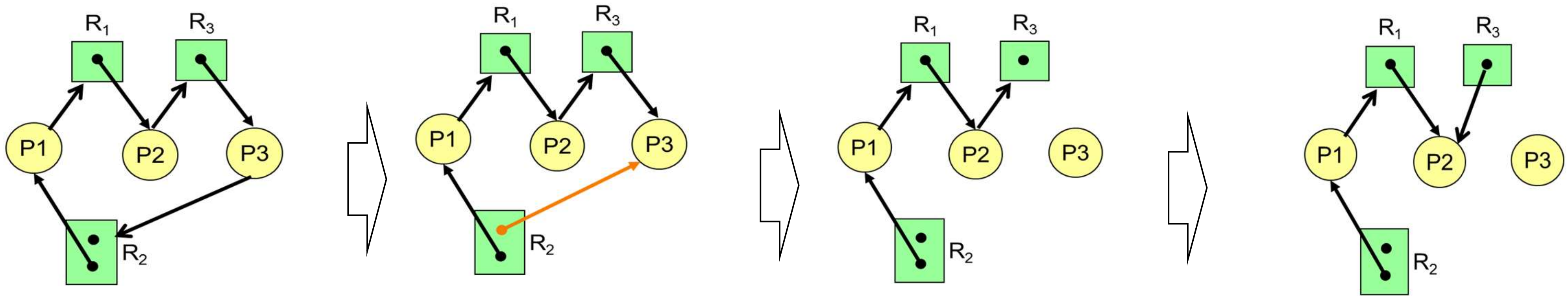
**State ($R_1$)**
available: 0
allocated: $P_2(1)$
max needed: $P_1(1)$




- Assumptions
  - Assume OS knows: **current state** of the system!, i.e.,
    - 1. Number of available instances of each resource
    - 2. For each process, *current* amount of each resource it owns
    - 3. For each process, *maximum* amount of each resource it needs in future

# System Model (cont.)

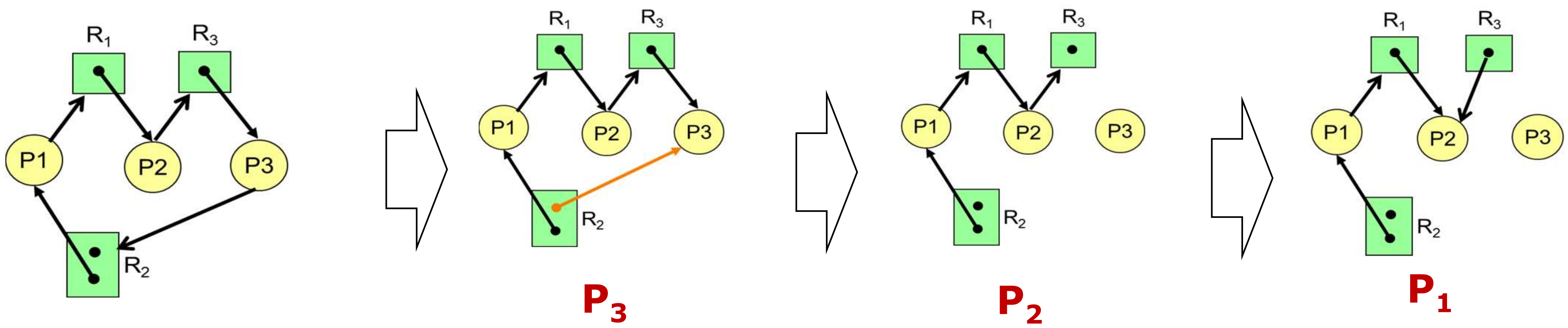
- Assume worst case...
  A process will never release its resources until it's done! (non-preemptive)
  processes are independent
  - While one blocks, others can finish if they have "enough resources"
    - After finishing, each process releases the resources it owns within finite time
    - request – use – **release!!**
- Example



# Safe Sequence

- **Safe** = there's *definitely* a way to finish the processes ***without*** deadlock

- **Safe Sequence**
  - **Sequence (order)** with which the system can allocate resources to each process and avoid a deadlock



# Safe sequence

- Sequence <$P_1$, $P_2$, …, $P_n$> is ***safe*** if
  for each $P_i$, the resources that $P_i$ can request can be satisfied by currently available resources + resources held by other $P_j$, with $j < i$.
  - <$P_1$, …, $P_j$, … , $\mathbf{P_i}$, $\mathbf{P_{i+1}}$, … >
  - If $\mathbf{P_i}$ 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.
- The system is in **safe state** *only if* there exists a **safe sequence**

# Example: Safe State

- We consider a system that has 12 tape instances and 3 processes: $P_0$, $P_1$, $P_2$.
  - **$P_0$ needs 10 tapes, $P_1$ needs 5 and $P_2$ needs 9.**
  - **Currently (at time $t_0$), $P_0$ has 5, $P_1$ has 2 and $P_2$ has 2**

How many tape instances are available?

Q : Is the system in a safe state at $t_0$?

= Is there **any** execution order that can finish without deadlock ?

A process might request its maximum resources at any time

A process will never release its resources until it's done

| | Max needs | Current holdings |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 5 | 2 |
| $P_2$ | 9 | 2 |

가천대학교 AI·소프트웨어학부 Gachon University

# Example: Safe State (cont.)

Q : Is the system in a safe state at $t_0$?

= Is there **any** execution order that can finish without deadlock?

12 tape instances

| | Max needs | Current holdings |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 5 | 2 |
| $P_2$ | 9 | 2 |

Solution ? :

**Search for an order $P_i$, $P_{i+1}$, $P_{i+2}$, ... such that:**

for each $P_i$, the resources that $P_i$ can request can be satisfied by currently available resources + resources held by other $P_j$, with $j < i$.

# Example: Safe State (cont.)

- Same example as before: 12 tape instances (available = 3)

| | Max needs | Current holdings |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 5 | 2 |
| $P_2$ | 9 | 2 |

- Q : Is the system in a safe state at $t_0$?
  - A: Yes, since <$P_1$,$P_0$,$P_2$> satisfied safe state condition
- Q : At $t_1$, if P2 requires and is allocated one more tape by the system, is the system is in a safe state?
  - A: No, since there is no safe sequence
    - See how many instances are available.

# Deadlock Avoidance

- **Safe state vs. unsafe state**

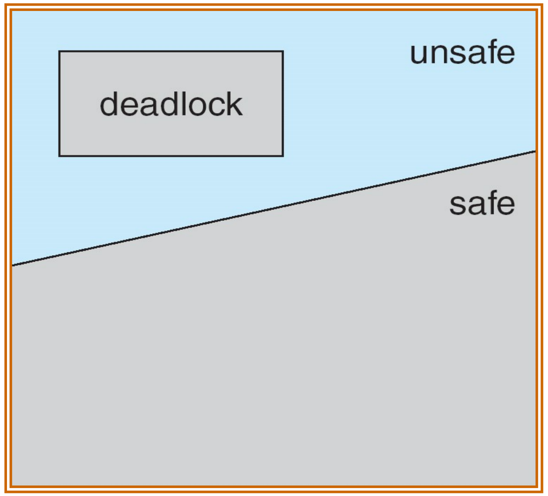
Definition: safe state

A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock.
A system is in a safe sate only if
there exist one or more **safe sequences**
Unsafe state: a state that is not safe

- Meaning of safe state
  - Can guarantee that the system does not get into the deadlock state
- Meaning of unsafe state
  - May not avoid getting into the deadlock state
    - conversely, Deadlock → unsafe state
  - Does not necessarily mean that deadlock will occur in the future

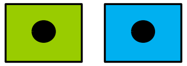
# Basic Facts ; Safe, Unsafe, Deadlock

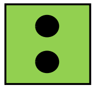
- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.




- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

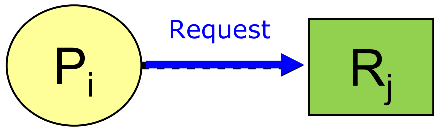
# Deadlock Avoidance Methods

- Single instance of a resource type 
  ⇒ Use a <u>Resource-Allocation Graph</u>

- Multiple instances of a resource type 
  ⇒ Use the <u>Banker's algorithm</u>

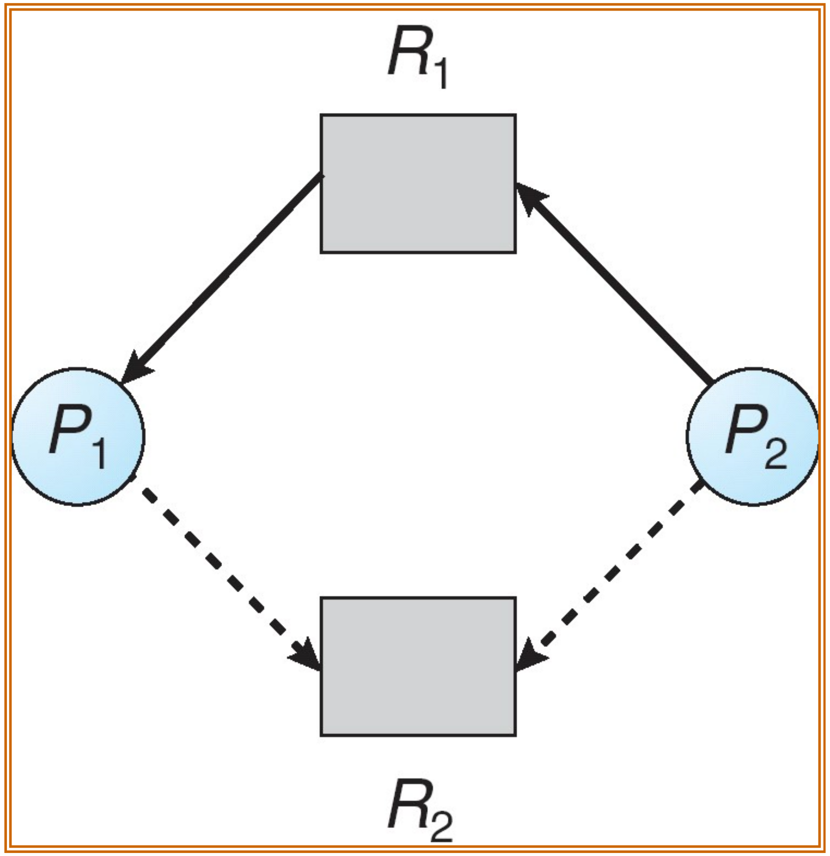
# Resource-Allocation Graph Algorithm

- ***Claim*** *edge* $P_i \rightarrow R_j$ indicates that process $P_i$ may request resource $R_j$ (at some time in the future); represented by a dashed line.




- Claim edge is converted to a request edge when a process actually requests a resource.
- An assignment edge $P_i \leftarrow R_j$ is reconverted to a claim edge when a resource is released by a process.

# Resource-Allocation Graph Algorithm




$P_i$ requests resource $R_j$ - request is granted *only if*

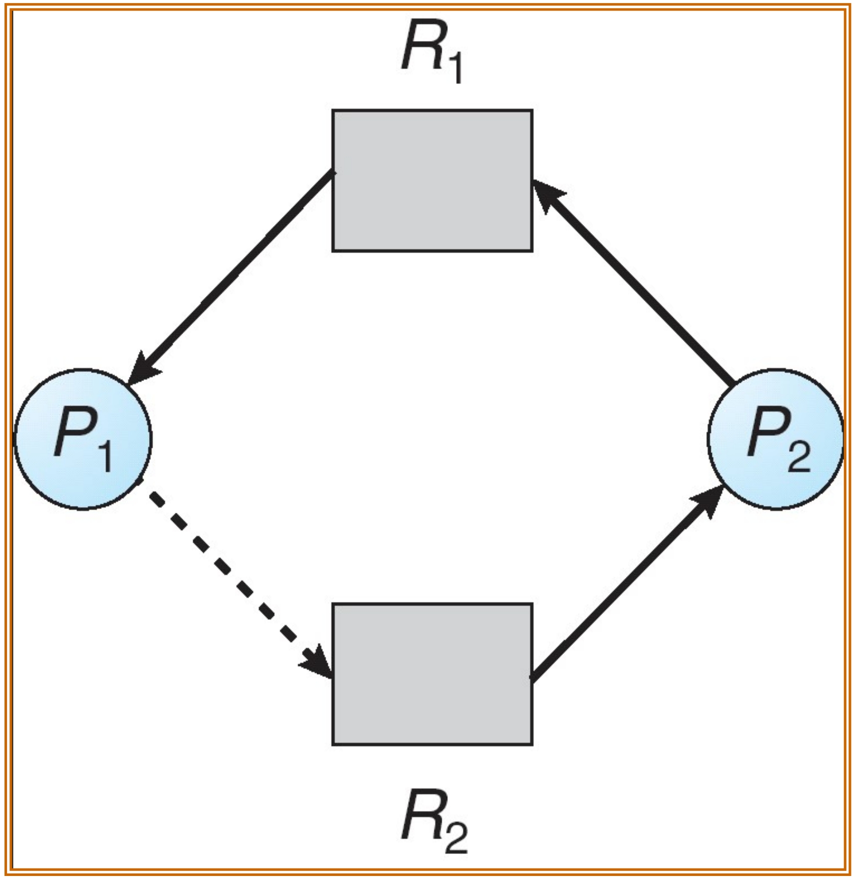
converting request edge $P_i \rightarrow R_j$ to assignment edge $R_j \rightarrow P_i$ does not result in **cycle**

Example: allocation sequence

$P_1$, $P_2$ → YES

**Safe state!**

# Resource-Allocation Graph Algorithm




- Allocation sequence
  - $P_2$, $P_1$ → NO
  - **Unsafe state!**

# Banker's Algorithm

- A resource allocation and deadlock avoidance algorithm
- <u>Multiple</u> instances

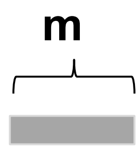
- Assumptions
  - Each process must *a priori* claim *maximum use*.
  - When a process requests a resource, it may have to *wait*.
  - When a process gets *all* its resources, it must *return* them in a finite amount of time.

- Consists of two sub-algorithms
  - **Safety Algorithm**
  - **Resource-Request Algorithm**

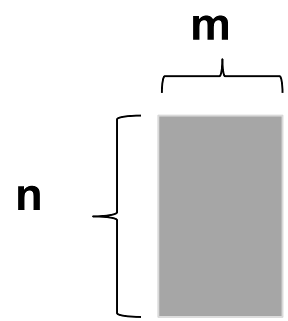
# Data Structures for the Banker's Algorithm

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:*** $n$ x $m$ 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* ].**

# Safety Algorithm

Is it currently **safe state**?

- Algorithm for finding out whether a system is in a **safe state**

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

   *Work* = *Available*

   *Finish* [*i*] = *false* for *i* = 0, 1, 2, …, *n-1.*

2. Find and *i* such that both:

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

   (b) $Need_i \leq Work$

   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*] == true for all *i*, then the system is in a safe state.

# Example of Bankers Algorithm

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).

- Snapshot at time $T_0$:

| | *Allocation* | *Max* | *Available* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 |
| $P_1$ | 2 0 0 | 3 2 2 | |
| $P_2$ | 3 0 2 | 9 0 2 | |
| $P_3$ | 2 1 1 | 2 2 2 | |
| $P_4$ | 0 0 2 | 4 3 3 | |

- Q : The system is in a safe state ?

# Example of Bankers Algorithm

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).

The content of the matrix.
*Need* is defined to be *Max* – *Allocation*

- Snapshot at time $T_0$:

| | Allocation | Max | Need | Work |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| $P_1$ | 2 0 0 | 3 2 2 | 1 2 2 | |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | |

- Q : The system is in a safe state ?

# Example of Bankers Algorithm

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).

- Snapshot at time $T_0$:

| | Allocation | Max | Need | Work |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| ~~$P_1$~~ | ~~2 0 0~~ | ~~3 2 2~~ | ~~1 2 2~~ | 5 3 2 |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | |

- Q : The system is in a safe state ?

# Example of Bankers Algorithm

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).

- Snapshot at time $T_0$:

| | Allocation | Max | Need | Work |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| ~~$P_1$~~ | ~~2 0 0~~ | ~~3 2 2~~ | ~~1 2 2~~ | 5 3 2 |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | 7 4 3 |
| ~~$P_3$~~ | ~~2 1 1~~ | ~~2 2 2~~ | ~~0 1 1~~ | |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | |

- Q : The system is in a safe state ?

# Example of Bankers Algorithm

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).

- Snapshot at time $T_0$:

| | Allocation | Max | Need | Work |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| ~~$P_1$~~ | ~~2 0 0~~ | ~~3 2 2~~ | ~~1 2 2~~ | 5 3 2 |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | 7 4 3 |
| ~~$P_3$~~ | ~~2 1 1~~ | ~~2 2 2~~ | ~~0 1 1~~ | 7 4 5 |
| ~~$P_4$~~ | ~~0 0 2~~ | ~~4 3 3~~ | ~~4 3 1~~ | |

- Q : The system is in a safe state ?

# Example of Bankers Algorithm

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).
- Snapshot at time $T_0$:

| | Allocation | Max | Need | Work |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| ~~$P_0$~~ | ~~0 1 0~~ | ~~7 5 3~~ | ~~7 4 3~~ | 3 3 2 |
| ~~$P_1$~~ | ~~2 0 0~~ | ~~3 2 2~~ | ~~1 2 2~~ | 5 3 2 |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | 7 4 3 |
| ~~$P_3$~~ | ~~2 1 1~~ | ~~2 2 2~~ | ~~0 1 1~~ | 7 4 5 |
| ~~$P_4$~~ | ~~0 0 2~~ | ~~4 3 3~~ | ~~4 3 1~~ | 7 5 5 |

- Q : The system is in a safe state ?

# Example of Bankers Algorithm

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).

- Snapshot at time $T_0$:

| | Allocation | Max | Need | Work |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| ~~$P_0$~~ | ~~0 1 0~~ | ~~7 5 3~~ | ~~7 4 3~~ | 3 3 2 |
| ~~$P_1$~~ | ~~2 0 0~~ | ~~3 2 2~~ | ~~1 2 2~~ | 5 3 2 |
| ~~$P_2$~~ | ~~3 0 2~~ | ~~9 0 2~~ | ~~6 0 0~~ | 7 4 3 |
| ~~$P_3$~~ | ~~2 1 1~~ | ~~2 2 2~~ | ~~0 1 1~~ | 7 4 5 |
| ~~$P_4$~~ | ~~0 0 2~~ | ~~4 3 3~~ | ~~4 3 1~~ | 7 5 5 |
| | | | | 10 5 7 |

- Q : The system is in a safe state ?

Yes, since the sequence < $P_1$, $P_3$, $P_4$, $P_0$, $P_2$> satisfies safety criteria.

# Example: $P_1$ requests (1,0,2)

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).
- Snapshot at time $T_0$:

| | Allocation | Max | Need | Available |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| $P_1$ | 2 0 0 | 3 2 2 | 1 2 2 | |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | |

- **Question.** What will happen if process **$P_1$ requests** one additional instance of resource type A and two instances of resource type C?

**To decide whether the request is granted we use Resource-Request algorithm**

# Resource-Request Algorithm for Process $P_i$

Can we remain in **safe state** after **request**?

*Request* = request vector for process $P_i$.

$Request_i[\,j\,] = k$ : $P_i$ wants $k$ instances of resource type $R_j$.

1. If $\boldsymbol{Request_i} \le \boldsymbol{Need_i}$ go to step 2.
   Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $\boldsymbol{Request_i} \le \boldsymbol{Available}$, go to step 3.
   Otherwise $P_i$ must wait, since resources are not available.

3. Pretend to allocate requested resources to $P_i$ by modifying the state as follows:

   $Available = Available - Request_i;$

   $Allocation_i = Allocation_i + Request_i;$

   $Need_i = Need_i - Request_i;$

   *If safe $\Rightarrow$ the resources are allocated to Pi.*

   *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

# Example: $P_1$ requests (1,0,2) (Cont.)

| | Allocation | Max | Need | Available |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| $P_1$ | 2 0 0 | 3 2 2 | 1 2 2 | |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | |

Request 1 = (1,0,2)

1. Check that Request 1 ≤ Need 1
that is, (1,0,2) ≤ (1,2,2) ⇒ true.

2. Check that Request 1 ≤ Available
that is, (1,0,2) ≤ (3,3,2) ⇒ true.

3. Pretend to allocate requested resources to $P_i$ by modifying the state as:

*$Available = Available – Request_i$;*
*$Allocation_i = Allocation_i + Request_i$;*
*$Need_i = Need_i – Request_i$;*

| | Allocation | Max | Need | Available |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | **2 3 0** |
| $P_1$ | **3 0 2** | 3 2 2 | 0 2 0 | |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | |

# Example: $P_1$ requests (1,0,2) (Cont.)

- Pretend to allocate requested resources to $P_i$ by modifying the state as:

| | *Allocation* | *Need* | *Work* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| ~~$P_0$~~ | ~~0 1 0~~ | ~~7 4 3~~ | 2 3 0 |
| | | | 5 3 2 |
| ~~$P_1$~~ | ~~3 0 2~~ | ~~0 2 0~~ | 7 4 3 |
| ~~$P_2$~~ | ~~3 0 2~~ | ~~6 0 0~~ | 7 4 5 |
| ~~$P_3$~~ | ~~2 1 1~~ | ~~0 1 1~~ | 7 5 5 |
| ~~$P_4$~~ | ~~0 0 2~~ | ~~4 3 1~~ | 10 5 7 |

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.

  ➔ The new system state is safe.
  ➔ We can immediately grant the request for process P1.

- Q: Can request for (3,3,0) by P4 be granted?
- Q: Can request for (0,2,0) by P0 be granted?

# Conclusion

- Deadlock Concept: A set of blocked processes each holding a shared resource and waiting to acquire a resource held by another process in the set. – Dining Philosopher's Problem
- 4 necessary conditions for Deadlock: Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait
- Deadlock Prevention: Make sure 4 necessary conditions for Deadlock does not hold
- Deadlock Avoidance: Always keeps safe state
  - Resource allocation graph
  - Banker's Algorithm

- Appendix
  - Methods for Handling Deadlocks
    - Deadlock Detection
    - Recovery from Deadlock

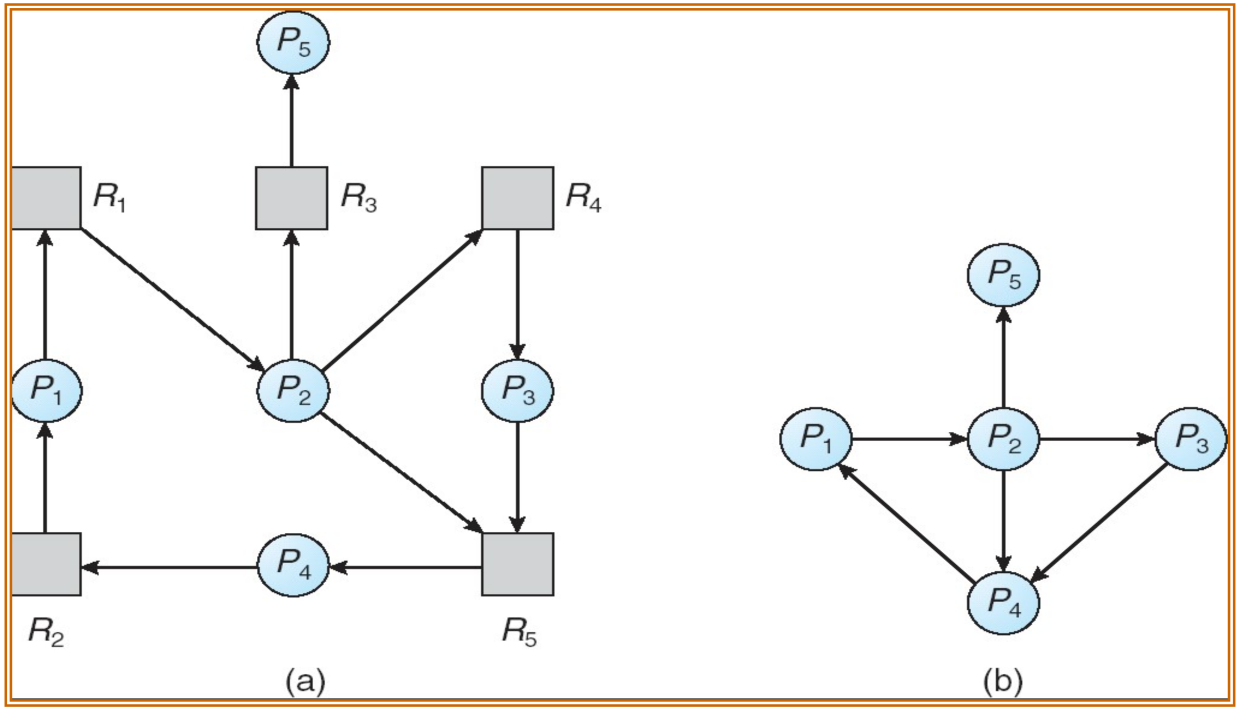
# Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Examine the state of the system to determine whether a deadlock has o ccurred
    (i.e., Check to see if a deadlock has occurred!)
  - Cases
    - single instance per resource type
    - multiple instance per resource type
- Recovery scheme
  - Recover from the deadlock

# Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
    - *Remove the in-between resource nodes*
  - $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$.
- Periodically invoke an algorithm that searches for a cycle in the graph.

# Resource-Allocation Graph and Wait-for Graph




Resource-Allocation Graph

Corresponding wait-for graph

We can see dependencies between process

# Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- Invoking detection-algorithm for every resource request vs.
  Invoking detection-algorithm at defined interval
  - High overhead vs. detection accuracy
    - In a periodic detection, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

# Deadlock Recovery

- Introduction
  - Recovery after detection of deadlocked processes
  - Eliminates detected deadlocks in the system
- **Deadlock recovery** methods
  - **Process termination**
    - Terminate (abnormally) part of the deadlocked processes
    - Terminated processes are restarted or rolled back afterwards
  - **Resource preemption**
    - Elect resources to be preempted to eliminate the deadlock
    - Preemption of the resources from the processes that currently owns them

# Deadlock Recovery: Process Termination

- Abort all deadlocked processes
  - Simple, but at great expense
  - May terminate unnecessary processes

- **Partial Termination**: Abort one process at a time until the deadlock cycle is eliminated. (Based on which process is the min cost)
  - Choose optimal set of processes to be terminated
    - Minimum overall termination cost
  - Complex : a deadlock-detection algorithm must be invoked