Chapter 8: Deadlocks
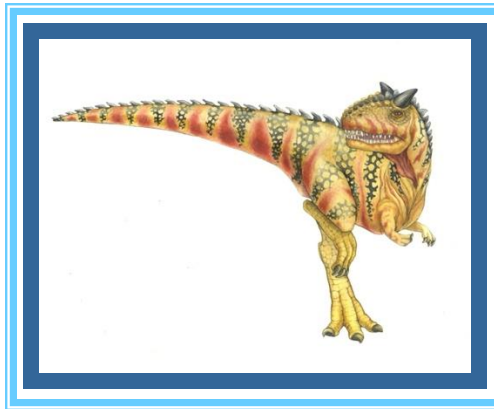
Chapter 8: Deadlocks

- Deadlock Examples
- Deadlock Characterization *— necessary conditions!*
- Resource Allocation Graph *→ indicate deadlocks!*
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance *→ no deadlocks!*
  - Deadlock Detection and Recovery from Deadlock

  Detect, recover!

Chapter Objectives

no sufficient condition (handwritten)

in today [lecture! (handwritten)

- Illustrate how deadlock can occur when mutex locks are used. *(EP 2 deadlock)* (handwritten)

- Define the **four necessary conditions** that characterize deadlock.

- Identify a deadlock situation in a resource allocation graph.

- Evaluate the four different approaches for preventing deadlocks.

- Apply Banker's algorithm for deadlock avoidance.

- Apply the deadlock detection algorithm.

- Evaluate approaches for recovering from deadlock.

Deadlock in Multithreaded Application

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

☐ The order in which the threads run depends on how they are scheduled by the CPU scheduler

bad luck → deadlock!

☐ This example illustrates a fact that it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.

W_1 , W_2 → no problem!

RR → bad luck!
($W_2(0)$) require second lock!
Deadlock will be happened!

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
        acquire(lock2);
            withdraw(from, amount);
            deposit(to, amount);
        release(lock2);
    release(lock1);
}
```

difficult to debug!
10:2 之间 deadlock: lucky!
But bad luck!
depends on cpu scheduling!
↓
very difficult to debug!
↓
unconfident to determine

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

System Model

(handwritten: use mathematical tools)

- System consists of resources

- Resource types $R_1, R_2, \ldots, R_m$ *(handwritten: 之利用 resources!)*

    CPU cycles, memory space, files, I/O devices, semaphores

    (handwritten: different number!)

    (handwritten: Assume here different resource!)

- Each resource type R_i has W_i instances.

- Each process P_i utilizes a resource as follows:
    - **request**
    - **use** *(handwritten: → 3 actions!)*
    - **release**

(handwritten: 沒 W_i 人能進入 (害调)!)

Deadlock Characterization

Please remember it

Deadlock involving multiple processes can arise if the following **four** conditions hold **simultaneously** – they are **necessary** but **not sufficient** conditions

- **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 resource(s) held by other processes

力 force !

- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task

repeatedly form a cycle!

- **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.

Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.   *Graph!*

- 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

- **request edge** – directed edge $P_i \rightarrow R_j$

- **assignment edge** – directed edge $R_j \rightarrow P_i$

Draw process we edges

Maybe in final exam!

- Process

- Resource Type with 4 instances

request resources!

- P_i requests instance of R_j

P_i → R_j
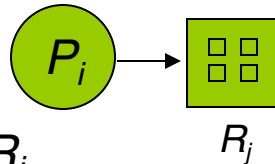
- P_i is holding an instance of R_j
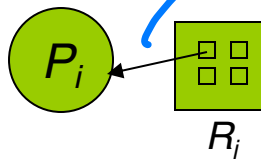
using the resource instance
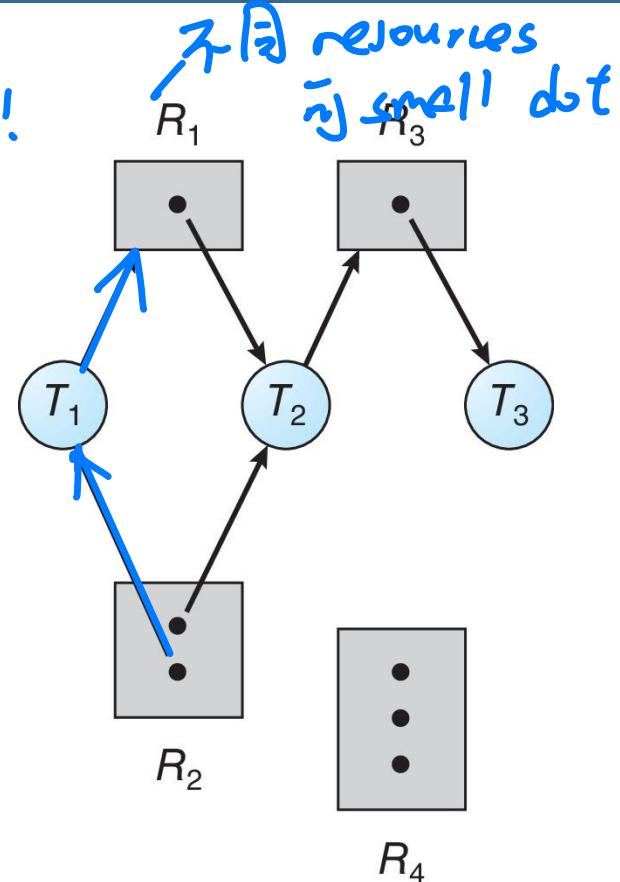
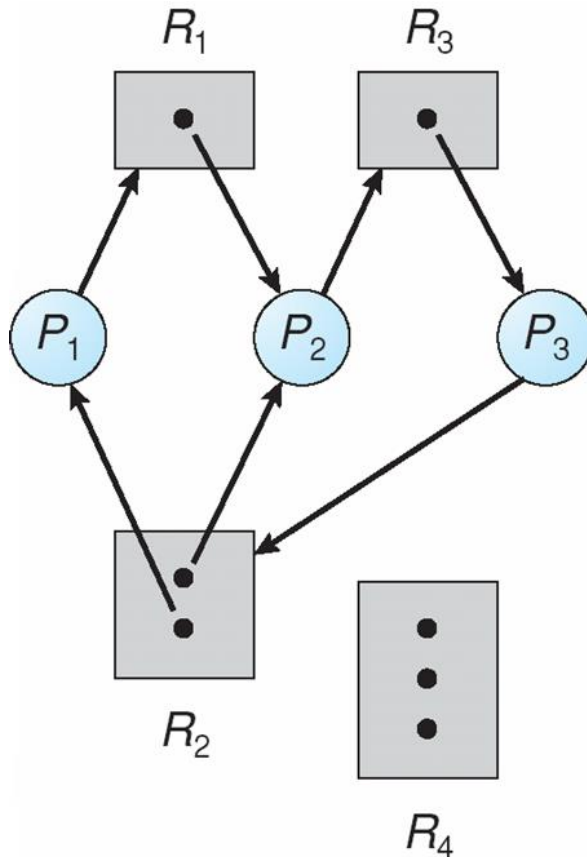P_i ← R_j

Resource Allocation Graph Example

- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3

understand how to draw it!

가로 resources 안 small dot

Cycles exist

Cycle !!!

☐ P1 → R1 → P2 → R3 → P3 → R2 → P1

☐ P2 → R3 → P3 → R2 → P2

☐ Processes P1, P2, and P3 are deadlocked

No process can be progressed

3개 모두 check !!!

cycle → necessary condition, not sufficient condition!

Circular wait only within process produce deadlock?

A cycle exists

- P1 → R1 → P3 → R2 → P1

- However, there is no deadlock. Observe that thread P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.
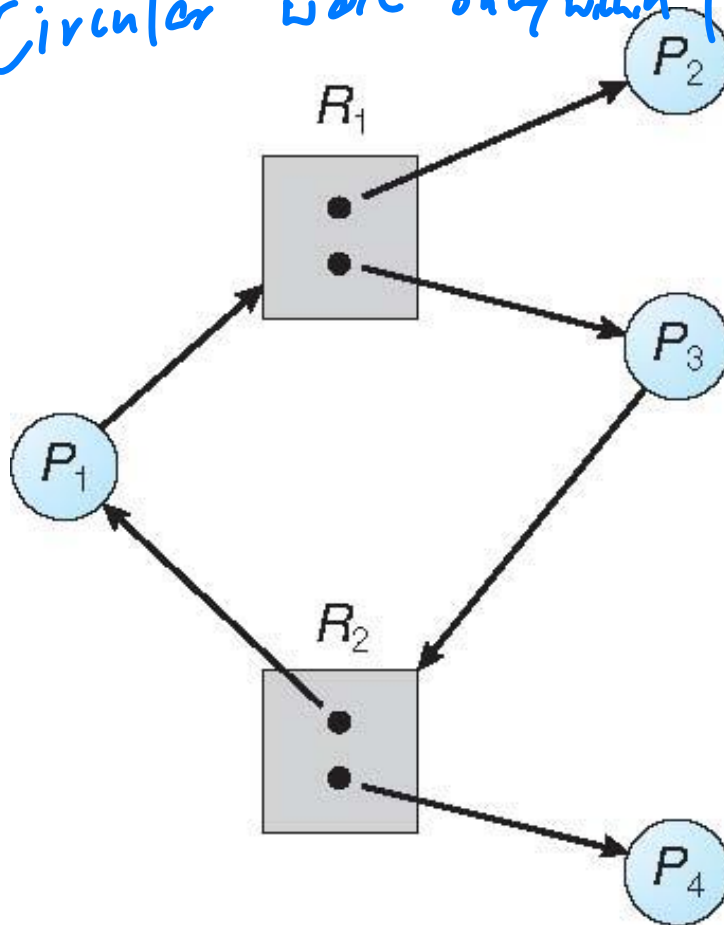
→ if P4 is released!
↓
no deadlock!

Basic Facts

- If a graph contains **no cycles** $\Rightarrow$ **no deadlock**

- If a graph contains a cycle $\Rightarrow$ the system may or may not be in a deadlocked state
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

[handwritten annotations] → only one instance → deadlock :(

Extreme case!

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:

  [handwritten: free]

  [handwritten: ①]
  - **Deadlock prevention**: it provides a set of methods to ensure at least one of the necessary conditions cannot hold *[handwritten: OS → judge, avoid the deadlock!]*

  - **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 a process should wait or not

  [handwritten: A&B satisfy 44 conditions, "check conditions → check whether be granted or not!]

- **Deadlock detection** - allow the system to enter a deadlock state. periodically detect if there is a deadlock and then recover from it

  [handwritten: not common in yr system!]

- Many commercial operating systems, esp., for desktops, laptops, and smart phones ignore the deadlock problem because of the overhead and pretend that deadlocks never occur in the system

  - It will cause the system's performance to deteriorate, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state – restart the system manually

Deadlock Prevention

how to design

Restrain the ways request can be made

取消 1 condition(s)

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); but it must hold for non-sharable resources *(not practical!)* 重复使用

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources *hold resource, request new resource!*
  - Require each process to request and be allocated all its resources before it begins execution, or request resources only when the process has none
  - The **disadvantages** - low resource utilization, and possible starvation

- **No Preemption** – *(can do this but not recommend to do this!) low performance!*
  - ↳ *waiting!*
  - If a process that is holding some resources requests another that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources added to list of resources for which the process is waiting
  - Process will be restarted only when it can regain all of its old resources, as well as the new ones that it is currently requesting
  - This can only be applied to resources whose state can be easily saved and restored such as registers, memory space and database transactions. It cannot generally be applied to resources such as locks and semaphores *(difficult)*

change to wait ↓ release All current resources!

Deadlock Prevention (Cont.)

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration – R = {<R_1, R_2, …, R_m>}
  - This requires that a process cannot request a resource R_j before requesting a resource R_i if j > i

- This can be proved by contradiction
  - Let the set of processes involved in a circular wait be P = {<P_0, P_1, …, P_n>}, where P_i is waiting for a resource R_i, which is held by process P_{i+1}, so that P_n is waiting for a resource R_n held by P_0.
  - Since process P_{i+1} is holding resource R_i while requesting resource R_{i+1}, we must have $R_i < R_{i+1}$ for all i.
  - This implies $R_0 < R_1 < R_2 … < R_n < R_0$
  - $R_0 < R_0$, this is impossible, therefore there can be no circular wait

[Handwritten annotations:]
increamental!
有 resource J, 不引 i) resource land 2!!
only request > then number bigger then its currently occupied!
no break!
usucly break the deadlock! !!:

Circular Wait

- Invalidating the circular wait condition is most common.

- Simply assign each resource (i.e. mutex locks) a unique number.

- Resources must be acquired **in order**.

- If:

  first_mutex = 1 R_1
  second_mutex = 5 R_2

  code for **thread_two** could not be written as follows:

```c
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

[handwritten annotations] ⌐ ⌐ request second one → then request first ones

X lower the performance for lot!
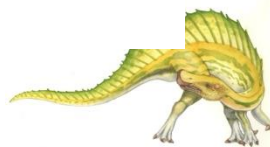
Deadlock Avoidance

+ Deadlock detection!

Requires that the system has some additional **a priori** information available

For instance, with the knowledge of complete sequence of request and release for each process, system can decide for each request whether the process should wait to avoid a possible future deadlock.

☐ The simplest and most useful model requires that each process declares the **maximum number** of resources of each type that it may need

☐ The **deadlock-avoidance** algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist

3 types of information

☐ **Resource-allocation state** is defined by the number of (1) available and (2) allocated resources, and (3) the maximum demands of the processes

→ resources

have been allocated

when/how to to detect deadlock

Deadlock Prevention: usually break circular wait.

Safe State

- When a process requests an available resource, system must decide whether such an allocation will leave the system in a **safe state**

  check whether the system is safe before allocating resources?

- System is in safe state if there exists a safe sequence $<P_1, P_2, …, P_n>$ consisting of **all processes** in the systems such that for each P_i, the resources that P_i can still request (based on prior declaration) can be satisfied by currently available resources plus resources held by **all** P_j, with $j < i.$ That is:

  deallocate the resource after 注释 *unsafe!*

  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished

    maybe deadlock!

  - 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

- If no such sequence exists, then the system state is said to be **unsafe**.
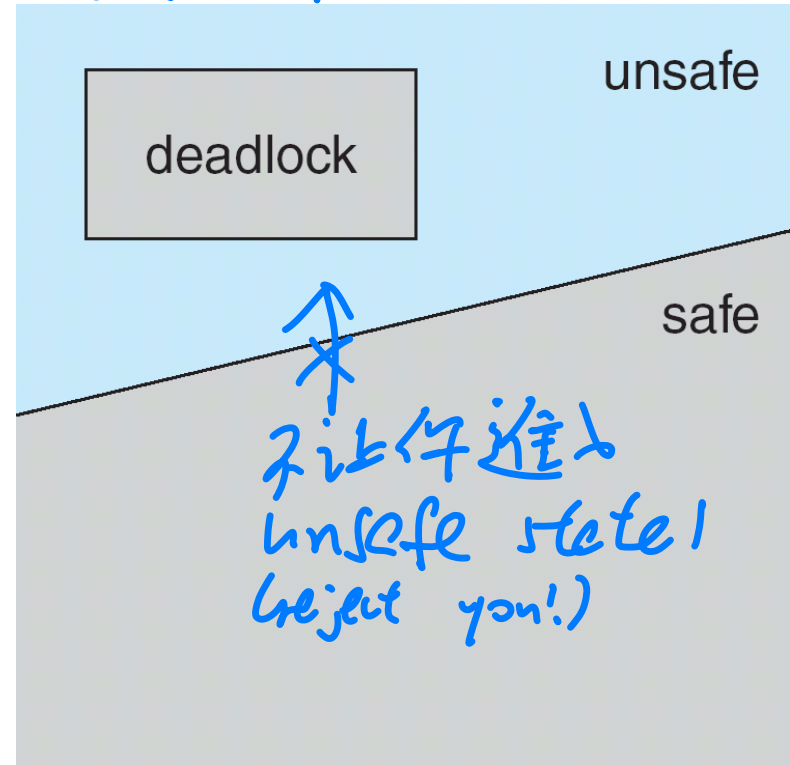
Basic Facts

- 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

  - In this scheme, if a process requests a resource that is currently available, it may still have to wait (if the allocation leads to unsafe state).

  - The resource utilization may be lower than it would be otherwise

[handwritten: check whether in the safe state!]

[handwritten diagram labels: unsafe, deadlock, safe]

[handwritten: 不让你进入 unsafe state! (reject you!)]

[handwritten: :(may have some sources under utilization!]

Avoidance algorithms

- Single instance of a resource type

  - Use a resource-allocation graph

- Multiple instances of a resource type

  - Use the Banker's algorithm

    ↳ two part
    - safety algorithm
    - request allocation algorithm
    (safe → approve request)

Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_i$ indicates that process P_i may request resource R_i; represented by a dashed line

$$P \rightarrow R$$

- Claim edge converts to request edge when a process requests a resource

- Request edge converts to an assignment edge when the resource is allocated to the process

- When a resource is released by a process, assignment edge reconverts to claim edge

- Resources must be claimed a priori in the system

Resource-Allocation Graph Algorithm

☐ Suppose that process P_i requests a resource R_j

☐ The request can be granted only if converting the request edge to an assignment edge **does not result** in the formation of a cycle in the resource allocation graph

↳ circular wait
⇒ reject!

request not happened

approved!

Resource-Allocation Graph
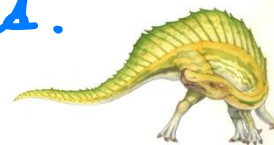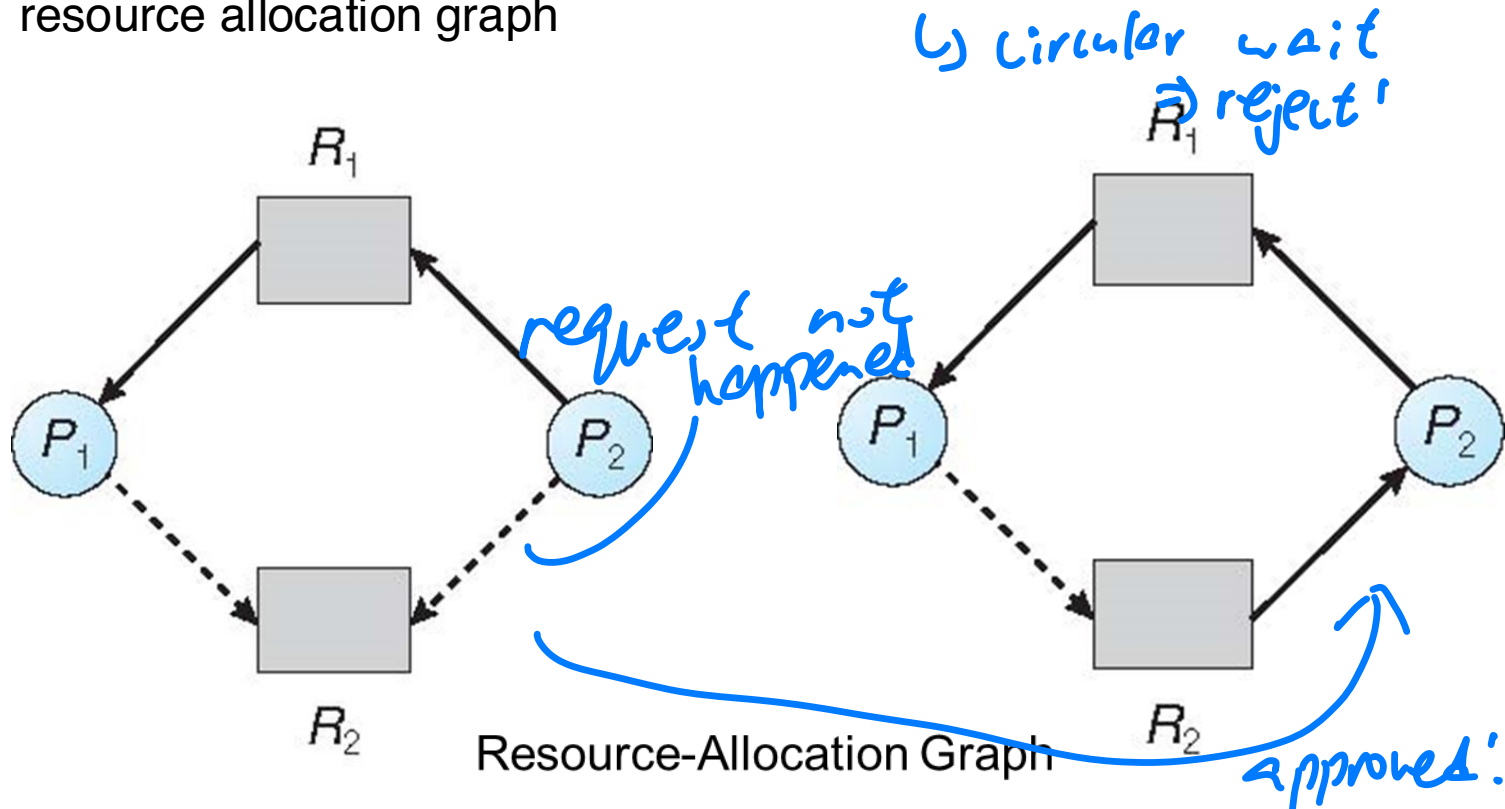
Banker's Algorithm

Very important!!!

☐ **Multiple instances** — more reflat real word cars!

☐ Each process must declare a priori maximum usage what resource you need during the whole cycle!

☐ When a process requests a resource, it may have to wait – check to see if this allocation results in a safe state or not *(Safety Algorithm); is in the safe state* *check whether the resource*

☐ When a process gets all its resources it must return them in a finite amount of time after use *finished!*

☐ This is analogous to banking loan system, which has a maximum amount, total, that can be loaned at one time to a set of businesses each with a credit line. *X good example :(*

One by one!. =#of proiess

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 *how many resources 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**: $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j

Still need!

☐ **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

Relationship: *Gap!.*

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

Safety Algorithm

1. Let **Work** and **Finish** be vectors of length *m* and *n*, respectively. Initialize: *[temp data structure]*

   Work = Available

   Finish [*i*] = false for *i* = 0, 1, …, *n*- 1 *[still need resource]*

2. Find an *i* such that both: *[determine whether can finish : P_i still waiting]*

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

   (b) **Need*ᵢ* ≤ Work** *[how many resource still require]*

   If no such *i* exists, go to step 4 *[surely complete!]*

3. **Work = Work + Allocation*ᵢ*;**
   Finish[*i*] = true
   go to step 2 *[return available resource to the system]* *[All can finish!!!]*

4. If **Finish [*i*] == true** for all *i*, then the system is in a safe state, otherwise unsafe

[for loop] *[not safe → some process cannot finish]*
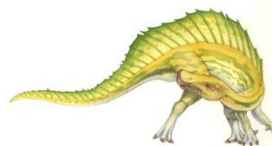
Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If **Request$_i$[j] = k** then process P_i wants **k** instances of resource type R_j

1. If **Request$_i$ ≤ Need$_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim ✓

2. If **Request$_i$ ≤ Available**, go to step 3. Otherwise, P_i <u>must wait</u>, since resources are not available

3. Pretend to have allocated requested resources to P_i by modifying the state as follows:

   Available = Available – Request; ✓ ← *Error!*

   Allocation$_i$= Allocation$_i$ + Request$_i$;

   Approved! <u>**Need$_i$ = Need$_i$ – Request$_i$;**</u> ✓

   ☐ Run safety algorithm: If safe ⇒ the resources can be allocated to P_i

   ☐ If unsafe ⇒ P_i must wait, and the old resource-allocation state is restored

Example of Banker's 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:

maximum resource required!

col. the need matrix

	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	

☐ The content of the matrix **Need** is defined to be **Max − Allocation**

:need!

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

:= Max − Allocation

☐ The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria

Example (Cont.)

☐ 5 processes P_0 through P_4; 3 resource types:

A (10 instances), B (5 instances), and C (7 instances)

A = 10,5,7

Snapshot at time T_0:

A = 7,5,5

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
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

✓

A =
5, 3. 2 ✓

X

✓ *A = 7,4,3*

✓ *A = 7,4,5*

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies the safety criteria

Example: P_1 Request (1,0,2)

- Check that Request $\leq$ Available, that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

  (handwritten: A B C above (3,3,2))

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement

- Can request for (3,3,0) by P_4 be granted? – resource not available

  (handwritten: – still reject by the system!)

- Can request for (0,2,0) by P_0 be granted? – state is not safe

  (handwritten: request!)

  (handwritten: because cannot find safe sequence!)
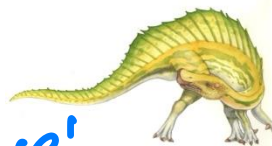
Deadlock Detection

[handwritten: 3rd Algorithm ... no deadlock!]

If a system does not use either a deadlock-prevention, or deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide

[handwritten: detect! => prevent!]

[handwritten: 증가 => serve performance problem!]

- ☐ An algorithm that examines the state of the system to determine whether a deadlock can occur

- ☐ An algorithm to recover from the deadlock

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes _wait_
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

$P_1 \rightarrow P_2$

- Periodically invoke an algorithm that searches for a *cycle* in the graph. If there is a cycle, there exists a deadlock

$O(n^2)$

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances for each resource type

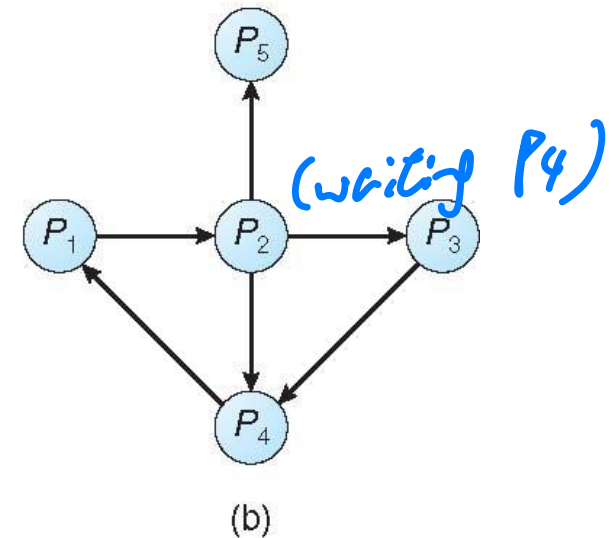Assume single instance!

(a)

(b)

(waiting P4)
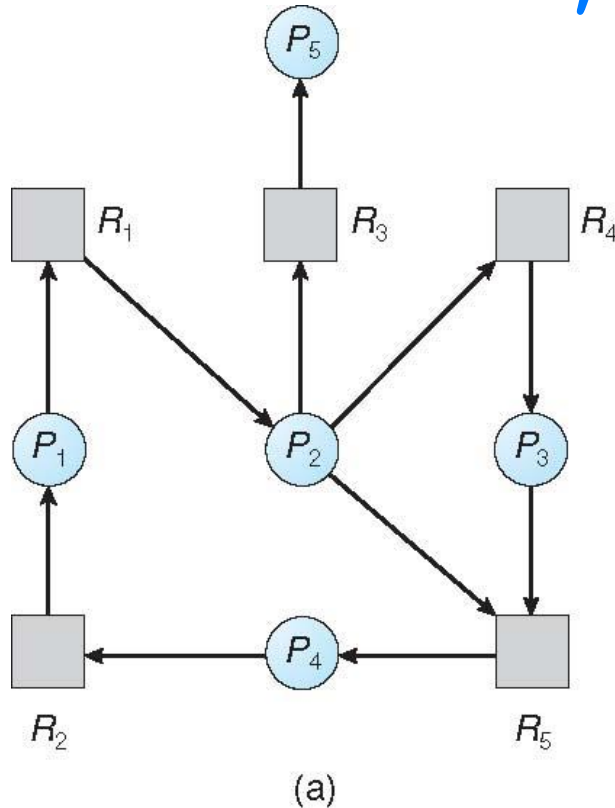
Resource-Allocation Graph   Corresponding Wait-for Graph

Circle ⟺ deadlock!

4° benker algorithm 一整一样！
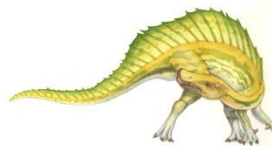
- **Available**: A vector of length m indicates the number of available resources of each type

- **Allocation**: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process

- **Request**: An $n \times m$ matrix indicates the current request of each process. If **Request** $[i][j] = k$, then process P_i is requesting k instances of resource type R_i.

Detection Algorithm

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

   (a) **Work = Available**

   (b) For **i = 1,2, …, n**, if **Allocation$_i$ ≠ 0**, then
       Finish[i] = **false**; otherwise, **Finish**[i] = **true**

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

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

   (b) **Request$_i$ ≤ Work**

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

3. **Work = Work + Allocation$_i$** ✓
   Finish[i] = true
   go to step 2

   deadlock detection!

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

This algorithm requires an order of O($m \times n^2$) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

☐ 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**

Example (Cont.)

- P_2 requests an additional instance of type C

<u>Request</u>

	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	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

Detection-Algorithm Usage

- When, and how often, to invoke depends on: *(use deadlock detection algorithm)*
    - How often a deadlock is likely to occur? *periodic*
    - How many processes will be affected by a deadlock when it occurs
        - one for each disjoint cycle

        lots of overhead!
        能不用就不用!

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph; we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

    use when interiaction is low !

- Invoking the deadlock detection algorithm for every resource request will incur considerable overhead in computation.
    - A less expensive alternative is to invoke the algorithm at defined intervals – for example, once per hour, or whenever CPU utilization drops below 40%

Recovery from Deadlock: Process Termination

☐ Abort all deadlocked processes: This clearly breaks the deadlock cycle, but at great expense

☐ Abort one process at a time until the deadlock cycle is eliminated: This also incurs considerable overhead, since after each process is aborted, the deadlock-detection algorithm needs to run

[handwritten: $O(mn^2)$]

[handwritten: brief of tradeoffs!]

☐ In which order should we choose to abort? – many factors:

  1. Priority of the process
  2. How long process has computed, and how much longer to complete?
  3. Resources the process has used
  4. Resources the process needs to complete
  5. How many processes will need to be terminated?
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

semiphone₂ (handwritten)

To successively preempt some resources from processes and give these resources to other processes ==until the deadlock cycle is broken==

pick lucky process, → release its resources! (handwritten)

❑ Selecting a victim – minimize cost (which resources and which processes are to be preempted) *may cause sterictation* (handwritten)

❑ Rollback – return to some safe state, restart process from that state

❑ Starvation – the same process may always be picked as victim, including the number of rollback in cost factor might help to reduce the starvation

End of Chapter 8