Operating system

Part VII: Deadlock [死锁]

By KONG LingBo (孔令波)

Goals

- Know the concepts of deadlock
  – Four necessary conditions
- Know the strategies to overcome the deadlock situation
  – Staying Safe
    - **Preventing** Deadlocks
    - **Avoiding** Deadlocks
  – Living Dangerously
    - Let the deadlock happen, then **detect** it and **recover** from it.

Dev ice ma nag em ent

- Deadlock
  - Definition, Model
- Methods for Handling Deadlocks
  - Providing **enough resources**
  - Staying Safe
    - **Preventing** Deadlocks
    - **Avoiding** Deadlocks
  - Living Dangerously
    - Let the deadlock happen, then **detect** it and **r ecover** from it.
    - **Ignore** the risks

However, it also will cause some problems if….

- If there is **no controlled access** to shared data, execution of the processes on the... leave **Cooperation**.
  - **The results will then depen... data were modified ☾ Data i...**
    - i.e. the results are non-determi...

We have discussed this by **synchronization**

- Concurrent processes (or thread... ...d to **share data** (maintained either in shared memory or files) and **resources**
  - If there is no proper policy to assign resources among processes, it may result in that all the processes get blocked ☾ **Deadlock** [死锁]

http://www.vijayforvictory.com/tag/picture

Deadlock: General

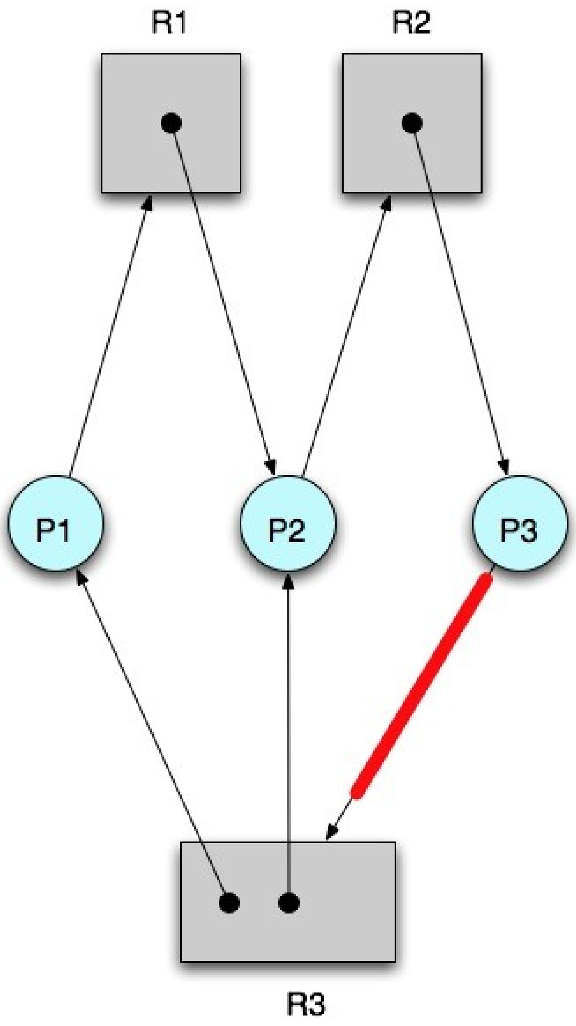
From Wikipedia, the free encyclopedia

- A deadlock is a situation wherein **two or more competing actions are waiting for the other to finish, and thus neither ever does**.
  - It is often seen in a paradox like the "chicken or t

> When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.
>
> — Illogical statute passed by the Kansas Legislature

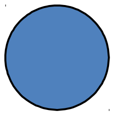
Deadlock in OS

- A set of blocked processes each holding some resourc es and waiting to acquire a resource held by another p rocess in the set.
- None of the processes can proceed or back-off (release resources it owns)

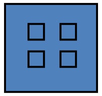


Resource-Allocation Graph (Cont.)

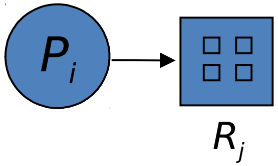
- Process



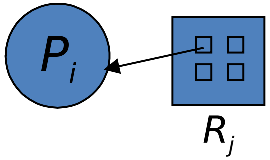
- Resource Type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j



It seems … [Basic Facts, Theorem]

1. If there are enough materials for everyone, no deadlock at all!
2. When a deadlock occurs, four conditions must have been reached at the same time!

① Mutual Exclusion [互斥]

- At least one resource is non-sharable: at most one process at a time can use it

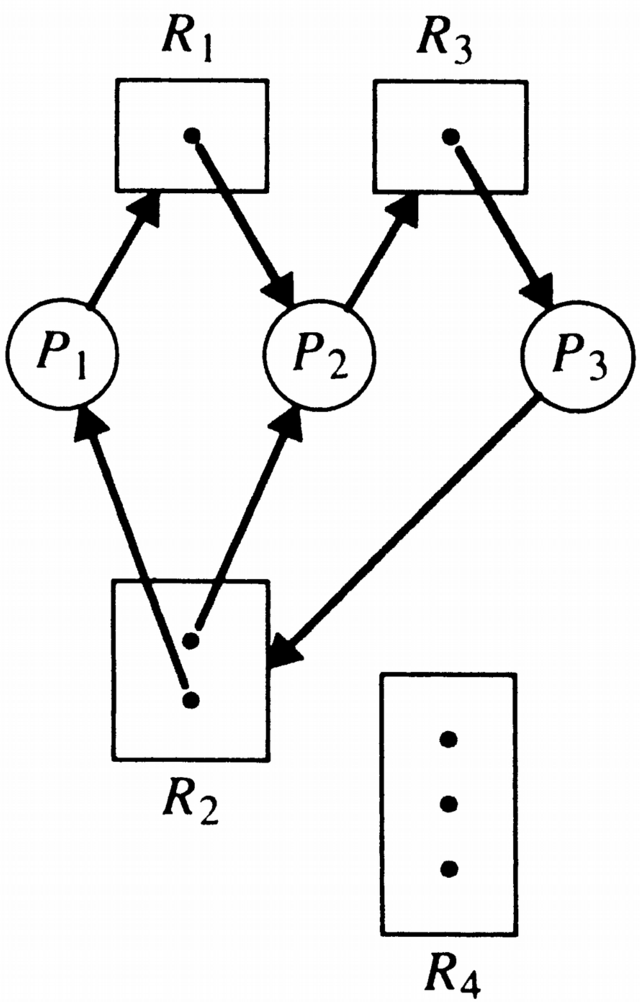
② Hold-and-Wait [占有并等待]

- At least one process is holding one resource while waiting to acquire others, that are being held by other processes

Deadlock can arise only if …

- A deadlock can arise only if all four conditions hold
  - ③ No preemption [非抢占]
    - A resource cannot be preempted (**a process needs to give it up voluntarily**)
  - ④ Circular Wait [循环等待]
    - There exists a set $\{P_0, P_1, \cdots, P_n\}$ of waiting processes such that
      - P_i is waiting for a resource that is held by P_{i+1}, $0 \leqslant i < n$
      - P_n is waiting for a resource that is held by P_0

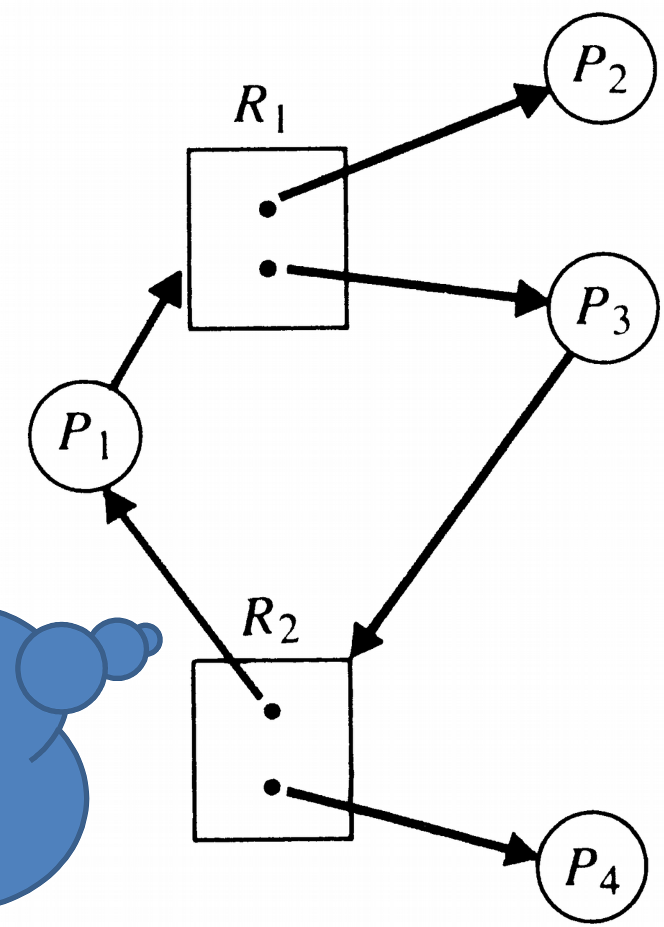
Deadlock



Because you cannot find an execution sequence of the all three processes so that each of them can finishes its work finally.

PPTs from others\www.dcs.ed.ac.uk_teaching_cs3_osslidesdeadlock.ppt

A cycle not sufficient to imply a deadlock:



Because there is a path of resource allocation to satisfy all the requirements

It seems …

1. If the graph contains no cycles, then no process is deadlocked.
2. If there is a cycle, then two situations :
   - If resource types have multiple instances, then deadlock **MAY** exist.
   - If each resource type has 1 instance, then deadlock has occurred.
     - The existence of a cycle is a sufficient and necessary condition for the existence of a deadlock
       - Each process involved in the cycle is deadlocked

Dev
ice
ma
nag
em
ent

- Deadlock
  - Definition, Model
- Methods for Handling Deadlocks
  - Providing **enough resources**
  - Staying Safe
    - **Preventing** Deadlocks
    - **Avoiding** Deadlocks
  - Living Dangerously
    - Let the deadlock happen, then **detect** it and **recover** from it.
    - **Ignore** the risks

Providing enough resources - A useful equation!

- Given:
  - Here are **3** processes: A, B, C. Each of them requires **5** system resources.
- Question:
  - How many resources should the system at least have so that the system is safe?
- Rule:
  - If the number of system resources satisfies the following equation, then the system is safe!

$$\sum (P_{\max} - 1) + 1 \le R_{Total}$$

A useful equation!

$$\sum (P_{\max} - 1) + 1 \le R_{Total}$$

- P_{max}: is the **max** number of the required res ources by process P
- R_{total}: is the total resources the system has

- It could be simplified as follows, where N is the number of processes

$$(P_{\max} - 1) \star N + 1 \le R_{Total}$$

- It is easy to answer the given question:

$$(P_{\max} - 1) * N + 1 \le R_{Total}$$

- According to the above equation
  - N is 3
  - P_{max} is 5

  - So, there should be at least (5-1)*3+1 = 13 reso urces in the system, then the system is in safe!

Variations

- Question:
  - A system has 10 tape drivers, which are shared by m processes, and each process requires 3 tape drivers at most. So, what should “m” be then the system could be in safe?

    A.3 B.4 C.5 D.6
- By that equation, we have
  - (3-1)*m+1 <= 10
  - So, m <= 4.5
  - B is the answer

Ignore .J.['ɔstritʃ]

Most Operating systems do this!!

- the Ostrich[鸵鸟] or **Head-in-the-Sand** algor

- **Of course**, Try to reduce chance of deadlock as far as reasonable
- **And**, accept that deadlocks will occur occasionally
  – example: kernel table sizes - max number of pages, open files etc.
- **Because**, maybe
  – MTBF versus deadlock probability ?
  – cost of any other strategy may be too high
    - overheads and efficiency

MTBF ：mean-time between "failures"

Maybe it' s better

- "**Don' t hide** from the fact…, Be alert!"

Deadlock Prevention（预防）

- Do not allow one of the four conditions to occur.
  - **Mutual Exclusion** [**互斥**]
    - Only one process may use a resource at a time
  - **Hold and Wait** [**持有和等待**]
    - A process may hold allocated resources while awaiting assi gnment of others
  - **No Preemption** [**非抢占**]
    - No resource can be forcibly removed form a process holdi ng it
  - **Circular Wait** [**循环等待**]
    - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in t he chain

Deadlock Prevention - **negating Hold and Wait**

- Two strategies
  1. When the process begins, all the resources re quired by it should be assigned to it
     - inefficient - not all resources needed all t he time
     - processes probably will not know in adva nce what resources they will need
     - may have to wait excessive time to get all resources at once - starvation
       – high priority processes may cause starvation of low priority processes

- Two strategies
  2. When a new request is needed by a processes
     A. it could release existing resources it holds if it fails to get a new resource immediately (try again later)
     B. the process always releases its existing resources and asks for all of them at once

Deadlock Prevention - **negating Circular Wait**

- Ordered resource allocation [资源顺序分配法]
  - Quite popular in management science (such as MBA)
  - Steps:
    - Assign each resource class with a unique number

$$r_1,\ r_2,\ \ldots,\ r_i$$

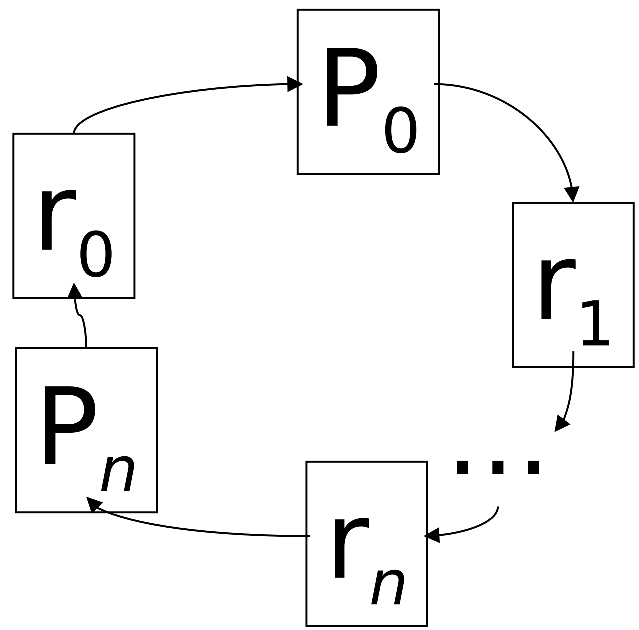
$$1\quad 2\quad\quad i$$

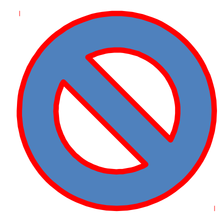
$$F(r_i) = i$$

    - The process should apply for all its required m-class resources following the order of resource class num

- Since all the resourc
en achieved befor
e resources in high
ot be cycle!

This implies … request has been known. You’ve to propose your request at the beginning





NO WAY!

Deadlock Avoidance（避免）

- Deadlock prevention ☾ low device utilization and reduced system throughput.
- Deadlock avoidance
  - Given the complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait.
  - For every request, the system
    - considers the resources currently **available**, the resources currently **allocated**, and the **future (Needed)** requests and releases of each process, and
    - decides whether the current request can be satisfied or must wait to avoid a possible future deadlock.

Example 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*	*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	

Example 2 (cont'　)

- We can compute the matrix ***Need*** as ***Max – Allocation***

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

Available

A B C

3 3 2

SAFE or not is determined by if we could find a sequence!

Example 2 (cont'd)

P3

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

P3 0 0 0

Available		
A	B	C
3	3	2

5 4 3

- Compare the available resources and the needed resources for each process to find if there is some process whose requirement could be satisfied.
  - P1 1 2 2
  - P3 0 1 1
- Randomly select one, here we select P3
- The available now is
  - <3, 3, 2> + <2,1,1> ℗ <5, 4, 3>

Allocation			
P3	2	1	1

P3 ⇨ P1

Example 2 (cont’)

Need

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

Available

A	B	C
5 / 7	4	3

- Similarly, there are two processes whose requirements could be satisfied.
  - P1 1 2 2
  - P4 4 3 1
- Randomly select one, here we select P1
- The available now is
  - <5, 4, 3> + <2,0,0> ℗ <7, 4, 3>

Allocation

P1 2 0 0

P3→P1→P0→P2→P4

Example 2 (cont')

Need

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

Available

A	B	C
7	**4**	**3**

- Similarly, there are three processes whose requirements could be satisfied.
  - P0 7 4 3
  - P2 6 0 0
  - P4 4 3 1
- We can dire
se processe
  - P0 ® P2

So, system is safe at the snapshot time t0!

Example 2:

- How about the snapshot as follows?

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

Example 2 (cont’) P3

Need

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

Available

A	B	C
1	1	1
3	**2**	**2**

- Compare the available resources and the needed resources for each process to find if there is some process whose requirement could be satisfied.
  - P3 0 1 1
- So we select P3
- The available now is
  - <1, 1, 1> + <2,1,1> ℗ <3, 2, 2>

Allocation

P3 2 1 1

Example 2 (cont’) P3 ⇒ P1

Need

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

Available

A	B	C
5	**2**	**2**

- Similarly, we can see P1 could be satisfied.
  - P1 1 2 2
- So we select P1
- The available now is
  - <3, 2, 2> + <2,0,0> ⓟ <5, 2, 2>

Allocation

P1	2	0	0

Example 2 (cont’) P3 ⇨ P1

Need

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

Available

A	B	C
5	**2**	**2**

- After P1, available res is <5, 2, 2>
- Sadly, this time, the rest three processes could not be satisfied < 5, 2, 2>.
  - P0 7 4 3
  - P2 6 0 0
  - P4 4 3 1
- This means it is not safe for the snapshot of example 2!

Example 3:

- 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	

Could the request of P0=<2 1 1> be satisfied or not?

- The logic for that satisfaction is
  - Pretend to satisfy the request first, and check if the change will lead the system into unsafe state or not
    - Still safe, the request could be satisfied
- We need update the resource allocation first – the **available** and the **needed** for P0

	Needed	**Allocation**
	A B C	A B C
P0	**5 3 2**	**2 2 1**
P1	1 2 2	2 0 0
P2	6 0 0	3 0 2
P3	0 1 1	2 1 1
P4	4 3 1	0 0 2

Available

A B C

ou've learned how to verify if the system is safe or not from **Example**

Drawbacks of Banker’ s Algorithm

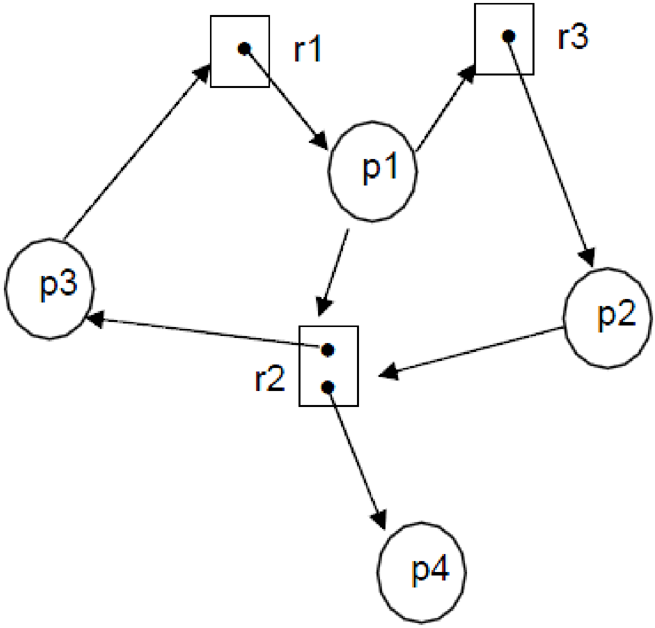
- processes are rarely known in advance how many resources they will need
- the number of processes changes as time pro gresses
- resources once available can disappear
- the algorithm assumes processes will return t heir resources within a reasonable time
- processes may only get their resources after a n arbitrarily long delay
- **practical use is therefore rare**!

Detection & Recovery

- **Check** for deadlock (periodically or sporadically[偶发地 , 零星地]), then **recover**
- Differentiate between
  - Serially reusable resources: A unit must be allocated before being released
  - Consumable resources: Never release acquired resources; resource count is the number currently available

PPTs from others\cms.dt.uh.edu\chap10.p

Borrow Banker's algorithm for several instances



Available = [0 0 0]

$$\text{Allocation} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Request} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\text{Finish} = \begin{bmatrix} False \\ False \\ False \\ False \end{bmatrix}$$

- Find if there is a sequence of the involved processes to finish
  - If all the values in Finish vector are TRUE or not

Recovery

- Two strategies
  1. Process Termination
     - Abort all deadlocked processes.
     - Abort one process at a time until the deadlock cycle is eliminated.
  - In which order should we choose to abort?
     - Priority of the process.
     - How long process has computed, and how much lo nger to completion.
     - Resources the process has used.
     - Resources process needs to complete.
     - How many processes will need to be terminated.
     - Is process interactive or batch?

Recovery

- Two strategies
  2. Resource Preemption
     - Choose a blocked process
     - Preempt it (releasing its resources)
       – Back up each deadlocked process to some previously defined checkpoint
     - Run the detection algorithm
     - Iterate it until the state is not a deadlock state
  – Selection Criteria Deadlocked Processes
     - Least amount of processor time consumed so far
     - Least number of lines of output produced so far
     - Most estimated time remaining
     - Least total resources allocated so far
     - Lowest priority

Combined Approach to Deadlock Handling

- Combine the three basic approaches (**prevention**, **avoidance**, and **detection**), allowing the use of the optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes； Use most appropriate technique for handling deadlocks within each class.
- An example:
  - Internal resources (Prevention through resource ordering)
  - Central memory (Prevention through preemption)
  - Job resources (Avoidance)
  - Swappable space (Pre-allocation)

In conclusion

- Deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does.
- Four necessary conditions
  - Mutual Exclusion [互斥]
  - Hold-and-Wait [占有并等待]
  - No preemption [非抢占]
  - Circular Wait [循环等待]
- Strategies to overcome the deadlock situation
  - Staying Safe
    - **Preventing** Deadlocks
    - **Avoiding** Deadlocks ☾ **Banker's algorithm**!
  - Living Dangerously
    - Let the deadlock happen, then **detect** it and **recover** from it.