Operating System

Dr. GuoJun LIU

Harbin Institute of Technology

http://guojunos.hit.edu.cn

Outline

Chapter 06

Concurrency: Deadlock and Starvation

并发性：死锁和饥饿

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. Statute passed by the Kansas State Legislature, early in the 20th century.

-- A TREASURY OF RAILROAD FOLKLORE,
B. A. Botkin and Alvin F. Harlow

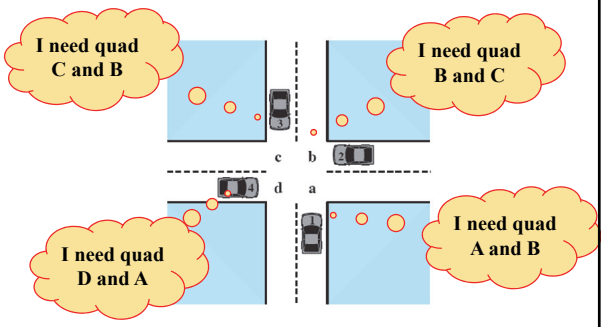
Learning Objectives

- List and explain the conditions for deadlock
- Define deadlock prevention and describe deadlock prevention strategies related to each of the conditions for deadlock
- Explain the difference between deadlock prevention and deadlock avoidance
- Understand two approaches to deadlock avoidance
- Explain the fundamental difference in approach between deadlock detection and deadlock prevention or avoidance
- Analyze the dining philosophers problem

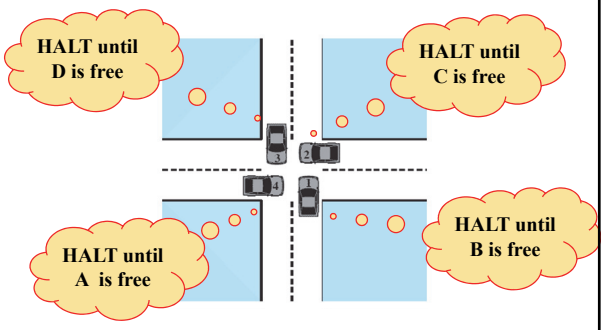
Deadlock

- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent
- No efficient solution

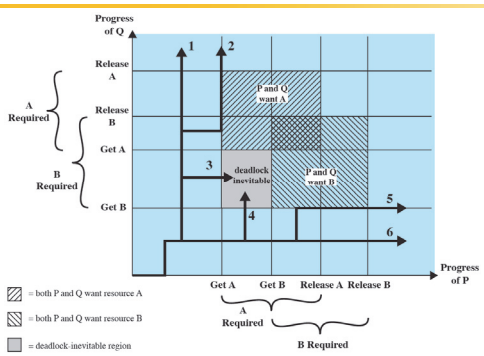
Potential Deadlock



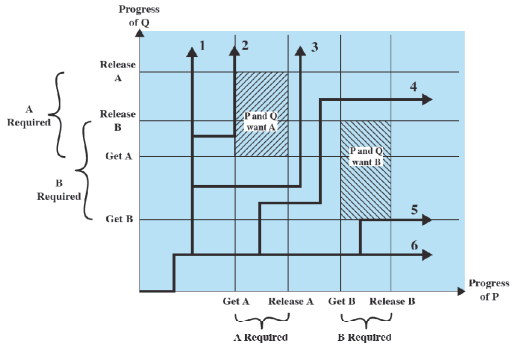
Actual Deadlock



Deadlock Example



No Deadlock Example



Resource Categories

- **Reusable**
  - can be safely used by **only one process at a time** and is **not depleted** by that use
    - processors
    - I/O channels
    - main and secondary memory
    - devices
    - data structures such as files, databases, and semaphores
- **Consumable**
  - one that can be **created** (produced) and **destroyed** (consumed)
    - interrupts
    - signals
    - messages
    - information in I/O buffers

Reusable Resources: Example 1

Process P

Step	Action
p_0 (1)	Request (D)
p_1 (2)	Lock (D)
p_2 (5)	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0 (3)	Request (T)
q_1 (4)	Lock (T)
q_2 (6)	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Example of Two Processes Competing for Reusable Resources

Example 2: Memory Request

- Space is available for allocation of 200 Kbytes, and the following sequence of events occur

P1	P2
…	…
Request 80 Kbytes;	Request 70 Kbytes;
…	…
Request 60 Kbytes;	Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

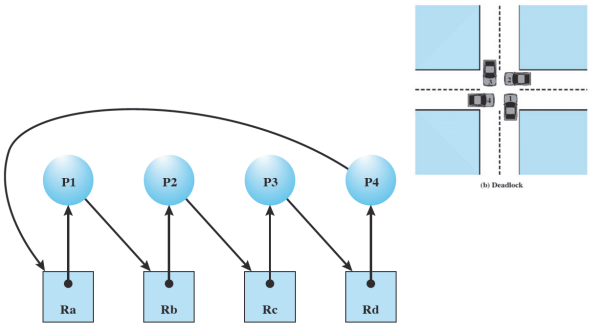
Consumable Resources Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

P1	P2
…	…
Receive(P2);	Receive(P1);
…	…
Send(P2,M1);	Send(P1,M2);

- Deadlock occurs if the Receive is blocking

Resource Allocation Graphs



Conditions for Deadlock

policy decisions

Mutual Exclusion
- only one process may use a resource at a time

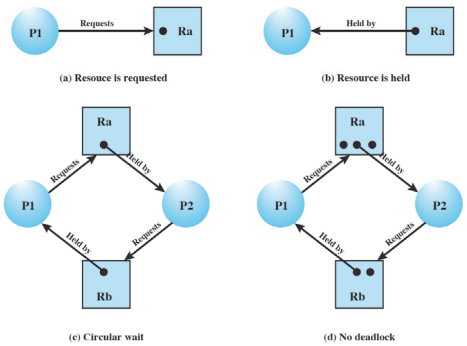
Hold and Wait
- a process may hold allocated resources while awaiting assignment of others

No Preemption
- no resource can be forcibly removed from a process holding it

Circular Wait
- a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

might occur

Resource Allocation Graphs



To summarize

- **Possibility of Deadlock**
  - Mutual exclusion
  - No preemption
  - Hold and wait
- **Existence of Deadlock**
  - Mutual exclusion
  - No preemption
  - Hold and wait
  - Circular wait

Dealing with Deadlock

Prevent Deadlock

- adopt a policy that **eliminates one of the conditions**

Avoid Deadlock

- make the **appropriate dynamic choices** based on the current state of resource allocation

Detect Deadlock

- attempt to **detect** the presence of deadlock and take action to recover

Deadlock Prevention Strategy

- **Design a system in such a way that the possibility of deadlock is excluded**
- **Two main methods**
  - Indirect
    - prevent the occurrence of **one of the three** necessary conditions
  - Direct
    - prevent the occurrence of a **circular wait**

Deadlock Condition Prevention

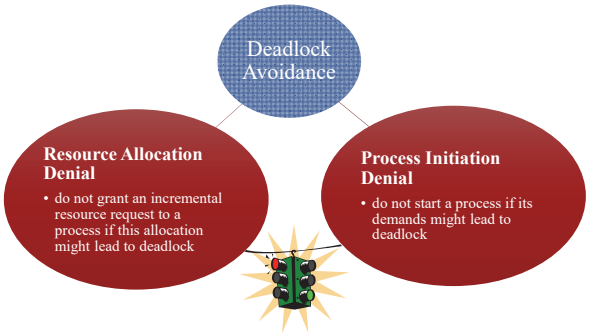
- **Mutual Exclusion**
  - if access to a resource requires mutual exclusion then it must be supported by the OS
- **Hold and Wait**
  - require that a process request all of its required resources **at one time** and blocking the process until all requests can be **granted simultaneously**
- **No Preemption**
  - if a process holding certain resources is denied a further request, that process must **release** its original resources and request them again
  - OS may preempt the second process and require it to release its resources
    - **different priority**
- **Circular Wait**
  - define a **linear ordering** of resource types

Deadlock Avoidance

- **A decision is made dynamically**
  - whether the current resource allocation request will, if granted, potentially lead to a deadlock
- **Requires knowledge of future process requests**

Different from deadlock prevention
allows the three necessary conditions but
makes judicious choices to assure that the
deadlock point is never reached

Two Approaches to Deadlock Avoidance



Resource Allocation Denial

- **Referred to as the banker's algorithm**
- **State of the system**
  - reflects the current allocation of resources to processes
- **Safe state**
  - is one in which there is **at least one sequence** of resource allocations to processes that **does not result in a deadlock**
- **Unsafe state**
  - is a state that is not safe

A system of n processes and m resources

Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j — All resources are either available or allocated.
2. $C_{ij} \leq R_j$, for all i,j — No process can claim more than the total amount of resources in the system.
3. $A_{ij} \leq C_{ij}$, for all i,j — No process is allocated more resources of any type than the process originally claimed to need.

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

P2 runs to completion

Determination of an Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

P1 requests one unit each of R1 and R3

Deadlock Avoidance Logic

```
struct state {
      int resource[m];
      int available[m];
      int claim[n][m];
      int alloc[n][m];
}
```

global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
      < error >;                              /* total request > claim*/
else if (request [*] > available [*])
      < suspend process >;
else {                                        /* simulate alloc */
      < define newstate by:
      alloc [i,*] = alloc [i,*] + request [*];
      available [*] = available [*] - request [*] >;
}
if (safe (newstate))
      < carry out allocation >;
else {
      < restore original state >;
      < suspend process >;
}
```

resource alloc algorithm

Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

P3 runs to completion

Deadlock Avoidance Logic

```
boolean safe (state S) {
   int currentavail[m];
   process rest[<number of processes>];
   currentavail = available;
   rest = {all processes};
   possible = true;
   while (possible) {
      <find a process Pk in rest such that
         claim [k,*] – alloc [k,*] <= currentavail;>
      if (found) {                          /* simulate execution of Pk */
         currentavail = currentavail + alloc [k,*];
         rest = rest - {Pk};
      }
      else possible = false;
   }
   return (rest == null);
}
```

test for safety algorithm (banker's algorithm)

Advantages and Restrictions

- **Advantages**
  - It is **not necessary** to preempt and rollback processes, as in deadlock detection
  - It is **less restrictive** than **deadlock prevention**
- **Restrictions**
  - **Maximum resource requirement** for each process must be stated in advance
  - Processes under consideration must be **independent** and **with no synchronization requirements**
  - There must be a **fixed number** of resources to allocate
  - **No process may exit** while holding resources

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses

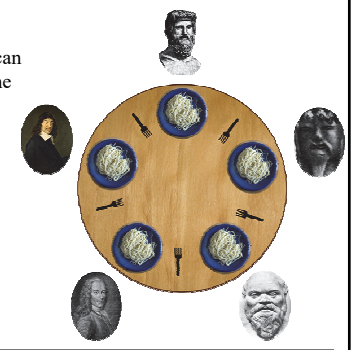
Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems

Deadlock Strategies

- **Deadlock prevention**
  - very **conservative**
  - **limit access** to resources by imposing restrictions on processes
- **Deadlock detection**
  - do the opposite
  - resource requests are granted **whenever possible**

Dining Philosophers Problem

- **Mutual exclusion**
  - **No two philosophers** can use **the same fork** at the **same time**
- **Avoid deadlock and starvation**
  - No philosopher must starve to death

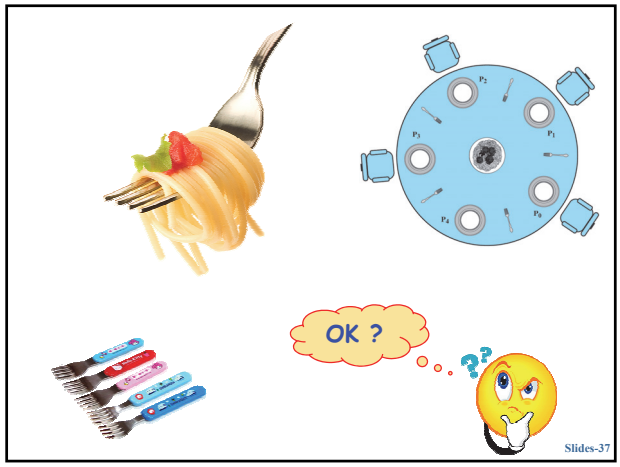


Deadline Detection Algorithms

- **A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur**
- **Advantages:**
  - it leads to **early detection**
  - the algorithm is **relatively simple**
- **Disadvantage**
  - **frequent** checks
  - **consume** considerable processor time

Using Semaphores

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```




```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)          /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);         /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);         /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])     /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```


A Second Solution

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```


Summary

- **Deadlock:**
  - the blocking of a set of processes that either compete for system resources or communicate with each other
  - blockage is **permanent** unless OS takes action
  - may involve reusable or consumable resources
    - Consumable = destroyed
    - Reusable = not depleted or destroyed by use
- **Dealing with deadlock:**
  - **prevention**
    - guarantees that deadlock will not occur
  - **avoidance**
    - analyzes each new resource request
  - **detection**
    - OS checks for deadlock and takes action

Using A Monitor

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);        /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);    /* client releases forks via the monitor */
   }
}
```

```
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```