# 目录 Contents





# Concurrency: Deadlock and Starvation

## 5.1 Principles Of Deadlock

### 1 What is Deadlock

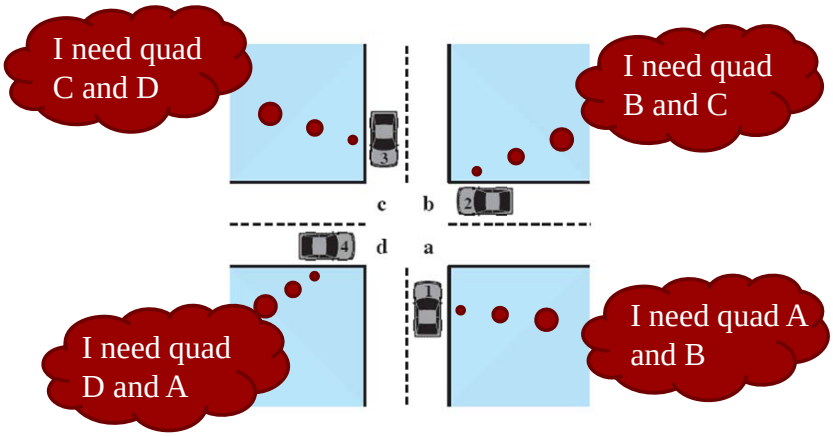
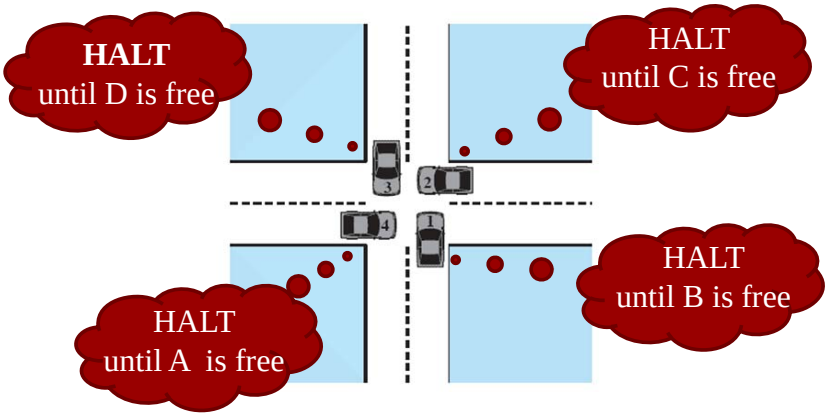



Figure 6.1 Potential Deadlock



## Deadlock Examples




- **Two Process**: Process A; Process B
- **Two Resources**: Graph plotter; CDROM

## The Concept of Deadlock

- **Deadlock**: The **permanent blocking** of a set of processes that either compete for system resources or communicate with each other.
- **Deadlock**: 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.
- **No efficient solution**
  - Involve conflicting needs for resources by two or more processes.

- **Root Cause of Deadlock**: The root cause of deadlocks is **resource competition** between concurrent processes.



## 5.1 Principles Of Deadlock

### 2 Resources

- **Resource Categories**: Reusable and Consumable

- **Reusable Resources(可重用资源)**: used by only one process at a time and is not depleted by that use.
- Processes obtain resources that they later release for reuse by other processes, deadlock occurs if each process holds one resource and requests the other.
- Possible Reusable Resources:
  - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores.

## Deadlock Involving Reusable Resources

| Process P | |
|---|---|
| Step | Action |
| $p_0$ | Request (D) |
| $p_1$ | Lock (D) |
| $p_2$ | Request (T) |
| $p_3$ | Lock (T) |
| $p_4$ | Perform function |
| $p_5$ | Unlock (D) |
| $p_6$ | Unlock (T) |

| Process Q | |
|---|---|
| Step | Action |
| $q_0$ | Request (T) |
| $q_1$ | Lock (T) |
| $q_2$ | Request (D) |
| $q_3$ | Lock (D) |
| $q_4$ | Perform function |
| $q_5$ | Unlock (T) |
| $q_6$ | Unlock (D) |

**Figure 6.4 Example of Two Processes Competing for Reusable Resources**

- Deadlock or not?
- p0 p1 q0 q1 p2 q2



## Deadlock Involving Reusable Resources.

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

P1
…
Request 80 Kbytes;
…
Request 60 Kbytes;

P2
…
Request 70 Kbytes;
…
Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

## Consumable Resources

- **Consumable Resources(可消耗的资源)**: Can be created (produced) and destroyed (consumed).
- When a resource is acquired by a consuming process, the resource ceased to exist.
- Examples of consumable resources:
  - interrupts, signals, messages, and information in I/O buffers.



## Deadlock Involving Consumable Resources

- 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
…
Receive(P2);
…
Send(P2, M1);
```

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

- Deadlock occurs if receive is blocking(i.e., the receiving process is blocked until the message is received).

## 5.1 Principles Of Deadlock

### 3 Resource Allocation Graphs

- introduced by Holt [HOLT72]. The resource allocation graph is a directed graph(有向图) that depicts a state of the system of resources and processes.
  - Two types of **nodes**: round-process and square-resource
  - **Arc**: From resource node to process node—the resource is occupied; From process to resource—the process is applying for the resource and is blocked.


(a) Resouce is requested (b) Resource is held

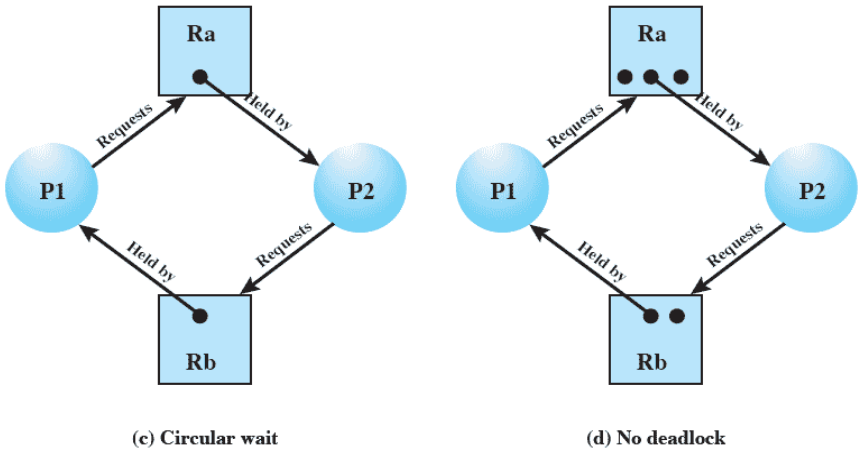


## 5.1 Principles Of Deadlock

### 4 Conditions for Deadlock

- **Mutual exclusion(互斥)**: Only one process may use a resource at a time
- **Hold-and-wait(占有并等待)**: A process may hold allocated resources while awaiting assignment of others
- **No preemption(不可抢占)**: No resource can be forcibly removed from a process holding 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 the chain

## Resource Allocation Graphs Of Deadlock




(c) Circular wait (d) No deadlock



## Possibility and Existence of Deadlock

- Possibility of Deadlock
  - Mutual Exclusion
  - No preemption
  - Hold and wait

  } Necessary Conditions

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

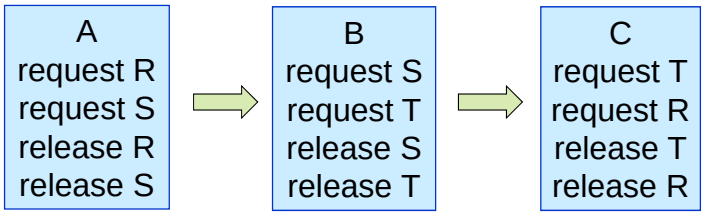
  } Necessary and Sufficient Conditions

a potential consequence

## Inducement Of Deadlock

- Example: Process A B C; Resource R S T
- A request RS; B request ST; C request TR

- Runs Sequentially

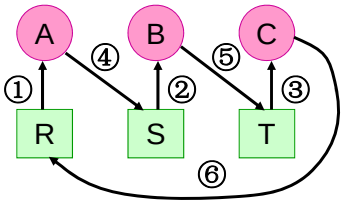


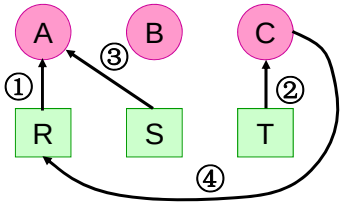

(a) (b) (c)



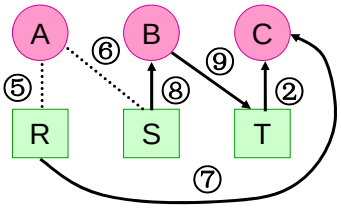
## Inducement of Deadlock.

- Concurrency




- **Inducement of Deadlock**: Improper order of progress



# Joint Progress Diagram(资源轨迹图)

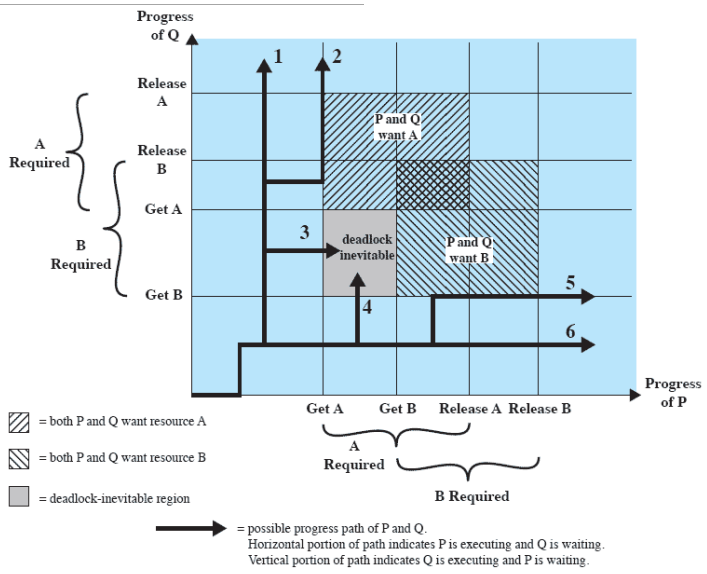



Figure 6.2 Example of Deadlock



# Joint Progress Diagram.

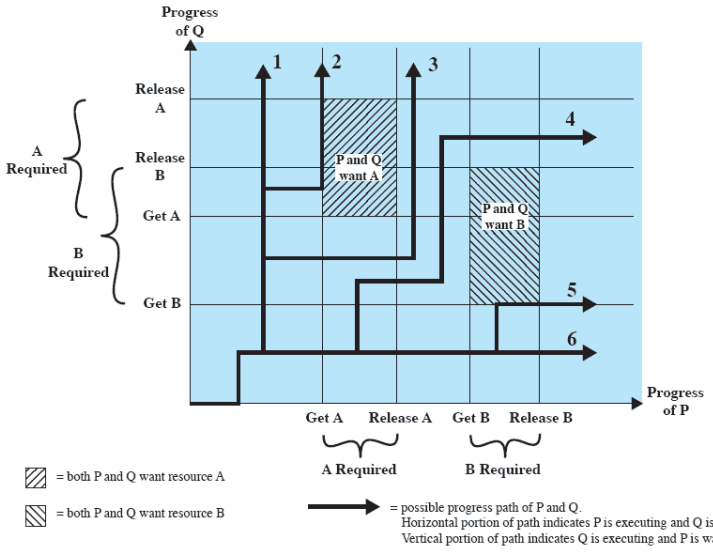



Figure 6.3 Example of No Deadlock [BACO03]

## Dealing With Deadlock

**Deadlock Prevention(死锁预防)**

- adopt a policy that eliminates one of the conditions

**Deadlock Avoidance(死锁避免)**

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

**Deadlock Detection(死锁检测)**

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



## 5.2 Deadlock Prevention

- The strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is excluded.
- **Indirect method**: prevent the occurrence of one of the three necessary conditions
- **Direct method**: prevent the occurrence of a circular wait

### Mutual Exclusion

- If access to a resource requires mutual exclusion then it must be supported by the OS.
- Method:
  - Allow to access resources simultaneously, i.e., **spooling** can be done for resources that need to be exclusive.

## Hold and Wait

- Methods:
  - Require a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously.
- Inefficient and impractical
  - It's difficult to predict the resource requirements of a process, and system cannot be optimized for the time and duration of resource utilization are uncertain.



## No Preemption

- Methods:
  - 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.
- **Practical only** when applied to resources whose state can be easily saved and restored later.

## Circular Wait

- Methods:
  - Define a linear ordering of resource types.

## Conclusion: Deadlock Prevention

- Constrain resource requests to prevent at least one of the four conditions of deadlock
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait

- Leads to inefficient use of resources and inefficient execution of processes



## 5.3 Deadlock Avoidance

### 1 Strategy of Deadlock Avoidance

- Allows the three necessary conditions but makes **judicious choices** to assure that the deadlock point is never reached.
- Avoidance allows more concurrency than prevention.
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.
- Requires knowledge of **future** process request

## Approaches to Deadlock Avoidance

- Do not start a process if its demands might lead to deadlock
  - Process initiation denial
- Do not grant an **incremental resource request** to a process if this allocation might lead to **deadlock**
  - Resource allocation denial



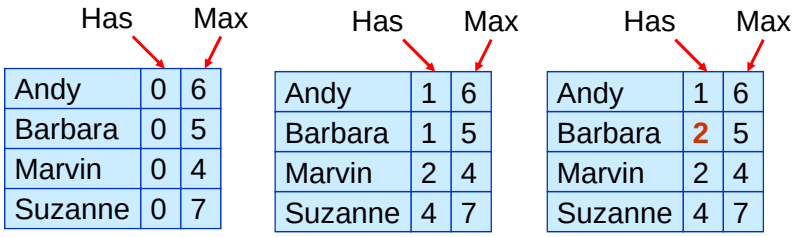
## 5.3 Deadlock Avoidance

### 2 Resource Allocation Denial

- Resource Allocation Denial: referred to as the **banker's algorithm**(银行家算法1965, Dijkstra)
- **State of the system**: is the current allocation of resources to process
- **Safe state**: is one in which there is at least **one sequence of resource allocations to processes** that does not result in a deadlock(i.e., all of the processes can be run to completion).
- **Unsafe state**: is a state that is not safe

## A Banker's Algorithm For A Single Resource

- A small-town banker might deal with a group of customers to whom he has granted lines of credit, total 22. The banker does not necessarily have enough cash on hand to lend every customer the full amount of each one's line of credit at the same time, i.e., only 10 at the time.




(a) Claim matrix free: 10

(b) Allocation matrix free: 2

(c) Try to Allocate free: **1**



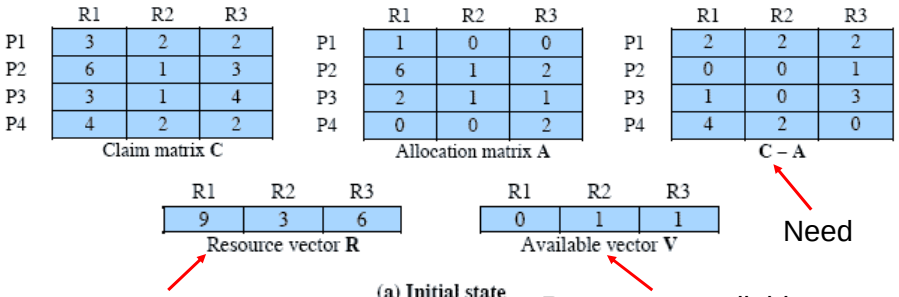
## A Banker's Algorithm For A Single Resource.

- The banker's algorithm considers each request as it occurs, and sees if granting it leads to a **safe state**. If it does, the request is granted; otherwise it is postponed until later.

- To see if a state is safe, the banker checks to see if he has enouah resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid. the state is safe and the initial request can be granted.

## A Banker's Algorithm For Multi-resource

- State of a system consisting of four processes and three resources.




Need

Amount of existing resources

Resources available after allocation

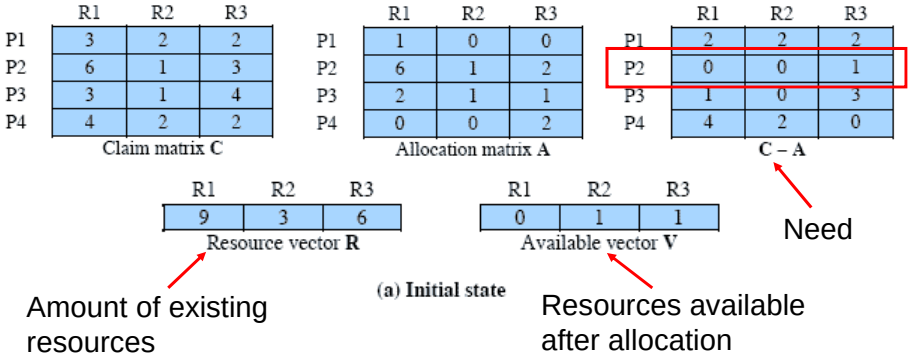
$$\sum_j C_{ij} \le R_i, \quad \text{for all } i, j$$

$$A_{ij} \le C_{ij}, \quad \text{for all } i, j$$



## A Banker's Algorithm For Multi-resource.

- State of a system consisting of four processes and three resources.




Need

Amount of existing resources

Resources available after allocation

- Safe or not ?

$$C_{ij} - A_{ij} \le V_j, \quad \text{for all } j$$

# Determination of A Safe 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**

**(b) P2 runs to completion**

| | 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**

**(c) P1 runs to completion**



# Determination of A Safe State.

| | 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**

**(d) P3 runs to completion**

- Safe: P2, P1, P3, P4
- Note: The safe sequence may not be unique.

## 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**

**p1 ( 1 0 1 )**

**(a) Initial state**

$$Request_i < C_i - A_i, \quad \text{for all } i, j$$

$$Rquest_i \le V_i, \quad \text{for all } i, j$$



## 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**

**p1 ( 1 0 1 )**

**(a) 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**

**(b) P1 requests one unit each of R1 and R3**

## Determination Steps

- Check if the right-hand matrix(C-A) has a row whose unsatisfied number of resources is less than or equal to vector V(available resources). If not, the system will be deadlocked, because no process can run to completion.
- If such a row is found, assume that it gets the required resources and run finished. Mark the process as finished, and add the resources to vector V.
- Repeat the above steps until all processes are marked as finished. The state is safe if all processes are finished, otherwise a deadlock may occurs.



## Deadlock Avoidance Logic

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

(a) global data structures

$R_i < C_i - A_i, \quad \text{for all } i, j$

```
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 >;
}
```

$R_i \le V_i, \quad \text{for all } i, j$

(b) resource alloc algorithm

## 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);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

37

## Conclusion: Deadlock Avoidance

- Advantages: It is not necessary to preempt and rollback processes as in deadlock prevention, so 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

## 5.4 Deadlock Detection

### 1 Deadlock Detection Algorithm

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

- 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



## Deadlock Detection Algorithm.

- Allocation matrix $A$, Available vector and Request matrix $Q$.
- algorithm[Coff 71]
  - **1**. Mark each process that has a row in the allocation matrix of all zeros.
  - **2**. Initialize a temporary vector $W$ to equal to Available vector.
  - **3**. Find an index $i$ such that process $i$ is currently unmarked and the $i$th row of $Q$ is less then or equal to $W$. If no such row is found, terminate the algorithm.
  - **4**. If such a row is found, mark process $i$ and add the corresponding row of the allocation matrix to $W$. Return to step **3**.

## Deadlock Detection


**Figure 6.10 Example for Deadlock Detection**



## 5.4 Deadlock Detection

### 2 Recovery Strategies

- Abort all deadlocked processes.
- Back up each deadlocked process to some previously defined checkpoint, and restart all process.
  - Original deadlock may occur
- Successively abort deadlocked processes until deadlock no longer exists.
- Successively preempt resources until deadlock no longer exists.

simple

sophistication

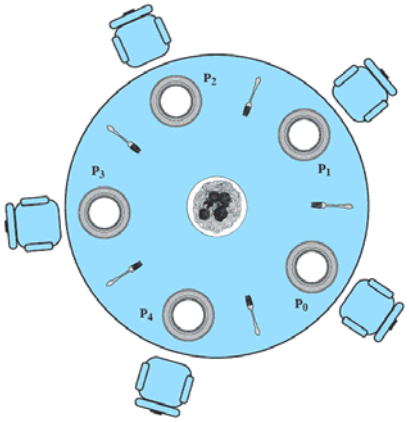
## Recovery Strategies.

- 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



## 5.5 Dining Philosophers Problem

### Dining Philosophers Problem(Dijkstra,1965)




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

Figure 6.11 Dining Arrangement for Philosophers

## Simple Solution Using Semphores

```
/* program dining philosophers
semaphore fork[5] = {1};
int i;
void philosopher(int i)
{   while(true)
    {   think();
        wait(fork[i]);
        wait(fork[(i+1)%5]);
        eat();
        signal(fork[(i+1)%5]);
        signal(fork[i]);
     }
}
```

- This solution may leads to deadlock: If all of the philosophers are hungry at the same time, they all sit down, then all pick up the fork on their left, and they all reach out for the other fork, which is not there. In this position, all philosophers starve.



## Solution Adding An Attendant

```
/* program dining philosophers
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)%5]);
        eat();
        signal(fork[(i+1)%5]);
        signal(fork[i]);
        signal(room);     }
}
```

## Solution Using A Set Of Semaphore

```
#define N  5
#define LEFT  (i-1)%N
#define RIGHT (i+1)%N
#define THINKING  0
#define HUNGRY   1
#define EATING    2
int state[N];
semaphore mutex = 1;
semaphore s[N]={0};
```

```
void philosopher(int i)
{  think( );
   take_forks(i);
   eat( );
   put_forks(i); }
```

```
void take_forks(int i)
{  wait(mutex);
   state[i]=HUNGRY;
   test(i);
   signal(mutex);
   wait(s[i]);  }
```

```
void put_forks(int i)
{  wait(mutex);
   state[i]=THINKING;
   test(LEFT);
   test(RIGHT);
   wait(mutex);  }
```

```
void test(int i)
{  if(state[i]==HUNGRY && state[LEFT] !=EATING
   && state[RIGHT]!=EATING)
   {  state[i]=EATING;
      signal(s[i]);  } }
```



## Solution Using a Monitor

```
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]);
}
```

## Solution 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 */
   }
}
```

**Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor**



## 5.6 Linux Kernel Concurrency Mechanisms

### Atomic Operation

- Includes all the mechanisms found in UNIX
- Atomic operations execute without interruption and without interference

| Atomic Bitmap Operations | |
|---|---|
| `void set_bit(int nr, void *addr)` | Set bit nr in the bitmap pointed to by addr |
| `void clear_bit(int nr, void *addr)` | Clear bit nr in the bitmap pointed to by addr |
| `void change_bit(int nr, void *addr)` | Invert bit nr in the bitmap pointed to by addr |
| `int test_and_set_bit(int nr, void *addr)` | Set bit nr in the bitmap pointed to by addr; return the old bit value |
| `int test_and_clear_bit(int nr, void *addr)` | Clear bit nr in the bitmap pointed to by addr; return the old bit value |
| `int test_and_change_bit(int nr, void *addr)` | Invert bit nr in the bitmap pointed to by addr; return the old bit value |
| `int test_bit(int nr, void *addr)` | Return the value of bit nr in the bitmap pointed to by addr |

## Atomic Operation

| Atomic Integer Operations | |
|---|---|
| `ATOMIC_INIT (int i)` | At declaration: initialize an atomic_t to i |
| `int atomic_read(atomic_t *v)` | Read integer value of v |
| `void atomic__set(atomic_t *v, int i)` | Set the value of v to integer i |
| `void atomic_add(int i, atomic_t *v)` | Add i to v |
| `void atomic_sub(int i, atomic_t *v)` | Subtract i from v |
| `void atomic__inc(atomic_t *v)` | Add 1 to v |
| `void atomic__dec(atomic_t *v)` | Subtract 1 from v |
| `int atomic_sub_and_test(int i, atomic_t *v)` | Subtract i from v; return 1 if the result is zero; return 0 otherwise |
| `int atomic_add_negative(int i, atomic_t *v)` | Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores) |
| `int atomic_dec_and_test(atomic_t *v)` | Subtract 1 from v; return 1 if the result is zero; return 0 otherwise |
| `int atomic_inc_and_test(atomic_t *v)` | Add 1 to v; return 1 if the result is zero; return 0 otherwise |



## Spinlocks

- If the value is 0, the thread sets the value to 1 and enters it's critical section. In the value is nonzero, the thread continually checks the value until it is 0.

**Table 6.4 Linux Spinlocks**

| | |
|---|---|
| `void spin_lock(spinlock_t *lock)` | Acquires the specified lock, spinning if needed until it is available |
| `void spin_lock_irq(spinlock_t *lock)` | Like spin_lock, but also disables interrupts on the local processor |
| `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)` | Like spin_lock_irq, but also saves the current interrupt state in flags |
| `void spin_lock_bh(spinlock_t *lock)` | Like spin_lock, but also disables the execution of all bottom halves |
| `void spin_unlock(spinlock_t *lock)` | Releases given lock |
| `void spin_unlock_irq(spinlock_t *lock)` | Releases given lock and enables local interrupts |
| `void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)` | Releases given lock and restores local interrupts to given previous state |
| `void spin_unlock_bh(spinlock_t *lock)` | Releases given lock and enables bottom halves |
| `void spin_lock_init(spinlock_t *lock)` | Initializes given spinlock |
| `int spin_trylock(spinlock_t *lock)` | Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise |
| `int spin_is_locked(spinlock_t *lock)` | Returns nonzero if lock is currently held and zero otherwise |

## Semophores

**Table 6.5 Linux Semaphores**

| Traditional Semaphores | |
|---|---|
| `void sema_init(struct semaphore *sem, int count)` | Initializes the dynamically created semaphore to the given count |
| `void init_MUTEX(struct semaphore *sem)` | Initializes the dynamically created semaphore with a count of 1 (initially unlocked) |
| `void init_MUTEX_LOCKED(struct semaphore *sem)` | Initializes the dynamically created semaphore with a count of 0 (initially locked) |
| `void down(struct semaphore *sem)` | Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable |
| `int down_interruptible(struct semaphore *sem)` | Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an `up` operation is received. |
| `int down_trylock(struct semaphore *sem)` | Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable |
| `void up(struct semaphore *sem)` | Releases the given semaphore |
| **Reader-Writer Semaphores** | |
| `void init_rwsem(struct rw_semaphore, *rwsem)` | Initalizes the dynamically created semaphore with a count of 1 |
| `void down_read(struct rw_semaphore, *rwsem)` | Down operation for readers |
| `void up_read(struct rw_semaphore, *rwsem)` | Up operation for readers |
| `void down_write(struct rw_semaphore, *rwsem)` | Down operation for writers |
| `void up_write(struct rw_semaphore, *rwsem)` | Up operation for writers |

## Barriers

- In some architectures, compilers and/or the processor hardware may reorder memory accesses in source code to optimize performance.
- Barriers: to enforce the order in which instruction are executed.

**Table 6.6 Linux Memory Barrier Operations**

| | |
|---|---|
| `rmb()` | Prevents loads from being reordered across the barrier |
| `wmb()` | Prevents stores from being reordered across the barrier |
| `mb()` | Prevents loads and stores from being reordered across the barrier |
| `barrier()` | Prevents the compiler from reordering loads or stores across the barrier |
| `smp_rmb()` | On SMP, provides a `rmb()` and on UP provides a `barrier()` |
| `smp_wmb()` | On SMP, provides a `wmb()` and on UP provides a `barrier()` |
| `smp_mb()` | On SMP, provides a `mb()` and on UP provides a `barrier()` |

SMP = symmetric multiprocessor
UP = uniprocessor

## Terminology

- deadlock
- consumable resources; reusable resources
- starvation
- hold and wait
- circular wait
- resource allocation graph
- deadlock prevention
- deadlock avoidance
- deadlock detection
- deadlock recovery
- Banker's algorithm
- spinlock