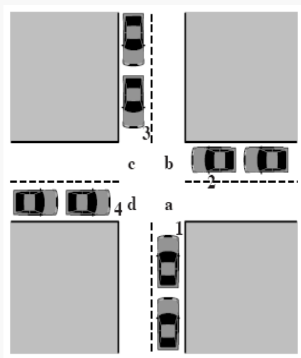
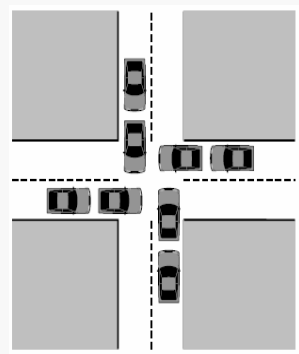
Deadlock

Permanent blocking of a set of processes that either compete for system resources or communicate with each other

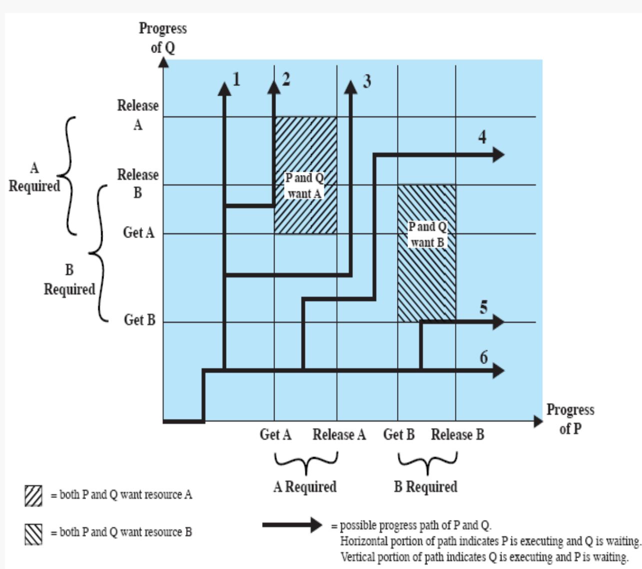
No efficient solution

Involve conflicting needs for resources by two or more processes

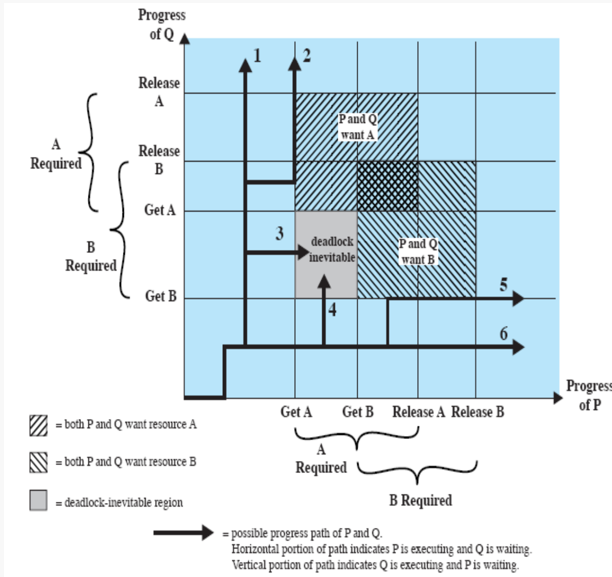




No Deadlock



Deadlock



Reusable Resources

Used by only one process at a time and not depleted by that use

Processes obtain resources that they later release for reuse by other processes

Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Deadlock occurs if each process holds one resource and requests the other

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)

Another Example of Deadlock

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

P1

```
. . .
  Request 80 KiB;
. . .
  Request 60 KiB;
```

P2

```
. . .
  Request 70 KiB;
. . .
  Request 80 KiB;
```

Deadlock occurs if both processes progress to their second request, assuming the processes block until their requests can be granted.

Consumable Resources

Created (produced) and destroyed (consumed)
Interrupts, signals, messages, and information in I/O buffers
Deadlock may occur if a Receive message is blocking
May take a rare combination of events to cause deadlock

Deadlock occurs if receive is blocking:

P1

```
. . .
Receive(P2);
. . .
Send(P2, M1);
```

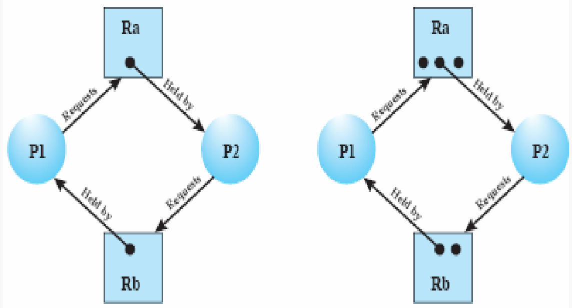
P2

```
. . .
Receive(P1);
. . .
Send(P1, M2);
```

Resource Allocation Graphs

Directed graph that depicts a state of the system of resources and processes

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



(c) Circular wait (d) No deadlock

Conditions for Deadlock

Mutual exclusion

Only one process may use a resource at a time

Hold-and-wait

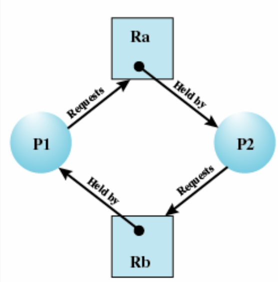
A process may hold allocated resources while awaiting assignment of others

No preemption

No resource can be forcibly removed form a process holding it

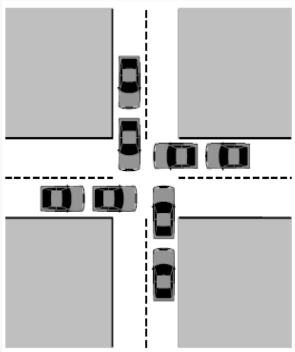
Circular wait

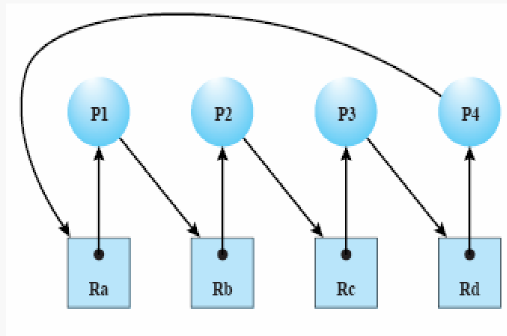
A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



(c) Circular wait

RAG with Deadlock





Possibility of Deadlock

Mutual Exclusion

No preemption

Hold and wait

Circular wait

Deadlock Prevention

Goal

- Design the system so that deadlock is logically impossible

Mutual Exclusion

- Must be supported by the operating system

Hold and Wait

- Require a process request all of its required resources at one time?

No Preemption

- Process must release resource and request again?
- Operating system may preempt a process to require it releases its resources?

Circular Wait

- Define a linear ordering of resource types?

Deadlock Avoidance

Goal

- Deny requests that might lead to the occurrence of deadlock

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

Do not start a process if its demands might lead to deadlock

Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Resource Allocation Denial

Referred to as Dijkstra's Banker's Algorithm

State of the system is the current allocation of resources to process

Safe state is where there is at least one sequence that does not result in deadlock

Unsafe state is a state that is not safe

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix **Q**

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix **A**

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

Determination of a Safe State

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	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**

(a) Initial state

P2 runs to completion:

	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

Determination of a Safe State

P1 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

P3 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**

(d) P3 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**

(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

Deadlock Avoidance

Maximum resource requirement must be stated in advance

Processes under consideration must be independent; no synchronization requirements

There must be a fixed number of resources to allocate

No process may exit while holding resources

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	~~0~~	~~0~~	~~0~~	~~0~~	~~1~~
P4	1	0	1	0	1

Request matrix **Q**

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix **A**

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	1	1

Available vector

Figure 6.10 Example for Deadlock Detection

Strategies once Deadlock Detected

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

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

Strengths and Weaknesses of the Strategies

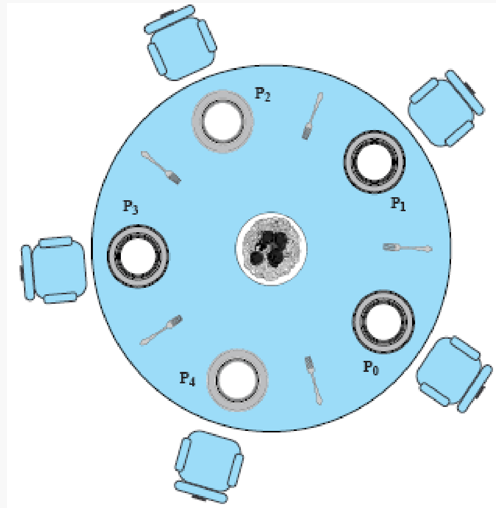
Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

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 on-line handling	•Inherent preemption losses

Dining Philosophers Problem

Concerns:

- deadlock may occur
- starvation (indefinite postponement) may occur



Dining Philosophers Solution I

```
int i;
semaphore fork[5] = {1};
```

```
int main() {
   parbegin(phil(0), phil(1), phil(2), phil(3), phil(4));
}
```

```
void phil(int i) {
   while ( true ) {
      think();
      wait( fork[i] );              // wait 'til get left fork
        wait( fork[(i + 1) % 5] );    // wait 'til get right fork
           eat();
        signal( fork[(i + 1) % 5] ); // put right fork down
      signal( fork[i] );            // put left fork down
   }
}
```

Assume:

- `think()` and `eat()` are guaranteed to return in finite, but not fixed, time

Dining Philosophers Solution II


```
int i;
semaphore fork[5] = {1};
semaphore room = 4;
```

```
void phil(int i) {
   while ( true ) {
      think();
      wait( room );                // cap attendance at 4
        wait( fork[i] );             // wait 'til get left fork
          wait( fork[(i + 1) % 5] );   // wait 'til get right fork
            eat();
          signal( fork[(i + 1) % 5] ); // put right fork down
        signal( fork[i] );           // put left fork down
      signal( room );              // raise cap when leaving
   }
}
```

Really cheesy… essentially cheats by changing the fundamental problem… but it does work.

Dining Philosophers Solution III


```
int i;
semaphore fork[5] = {1};
```

```
void phil(int i) {
   int j = i % 2;
   while ( true ) {
      think();
      wait( fork[i+j] );             // go for preferred fork
        wait( fork[(i+1-j) % 5] );   // go for opposite fork
          eat();
        signal( fork[(i+1-j) % 5] ); // put down opposite fork
      signal( fork[i+j] );           // put down preferred fork
   }
}
```

Basically:

- makes alternating philosophers left-handed
- no artificial limit on # of philosophers competing at once
- does it work?