Concurrency: Deadlock and Starvation

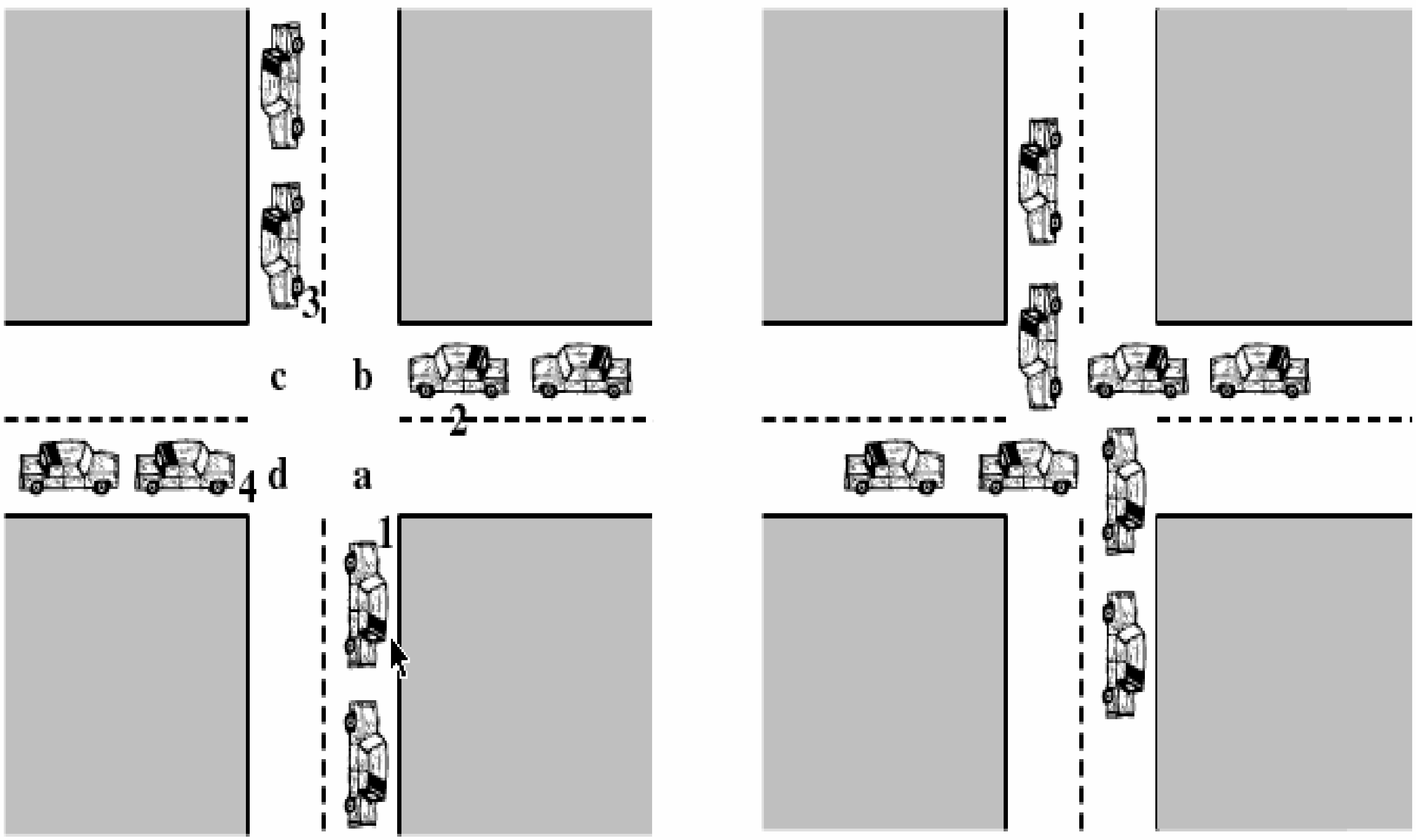
Chapter 6

What is Deadlock

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

- Involve conflicting needs for resources by two or more processes
- No efficient solution

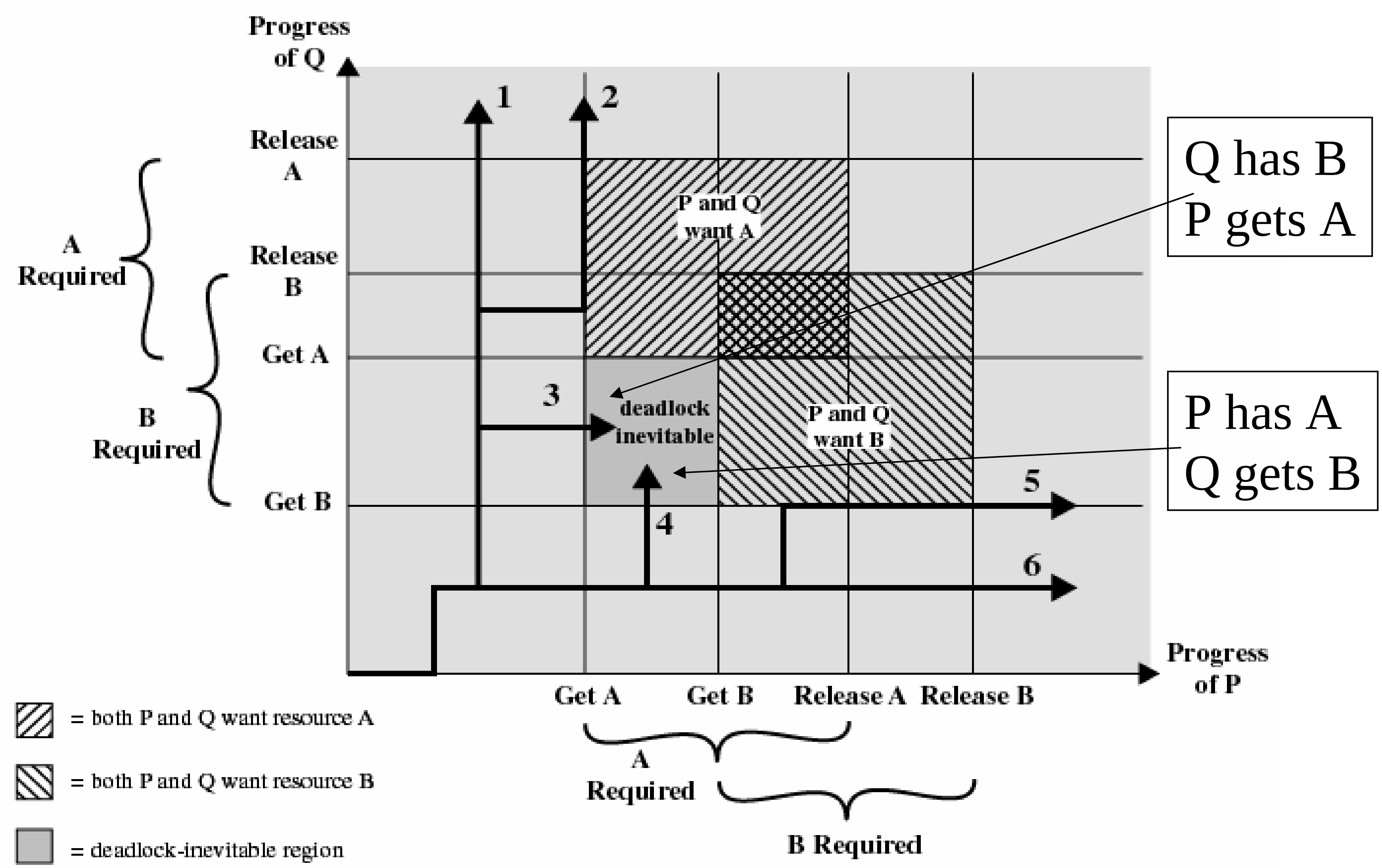
Deadlock Illustration



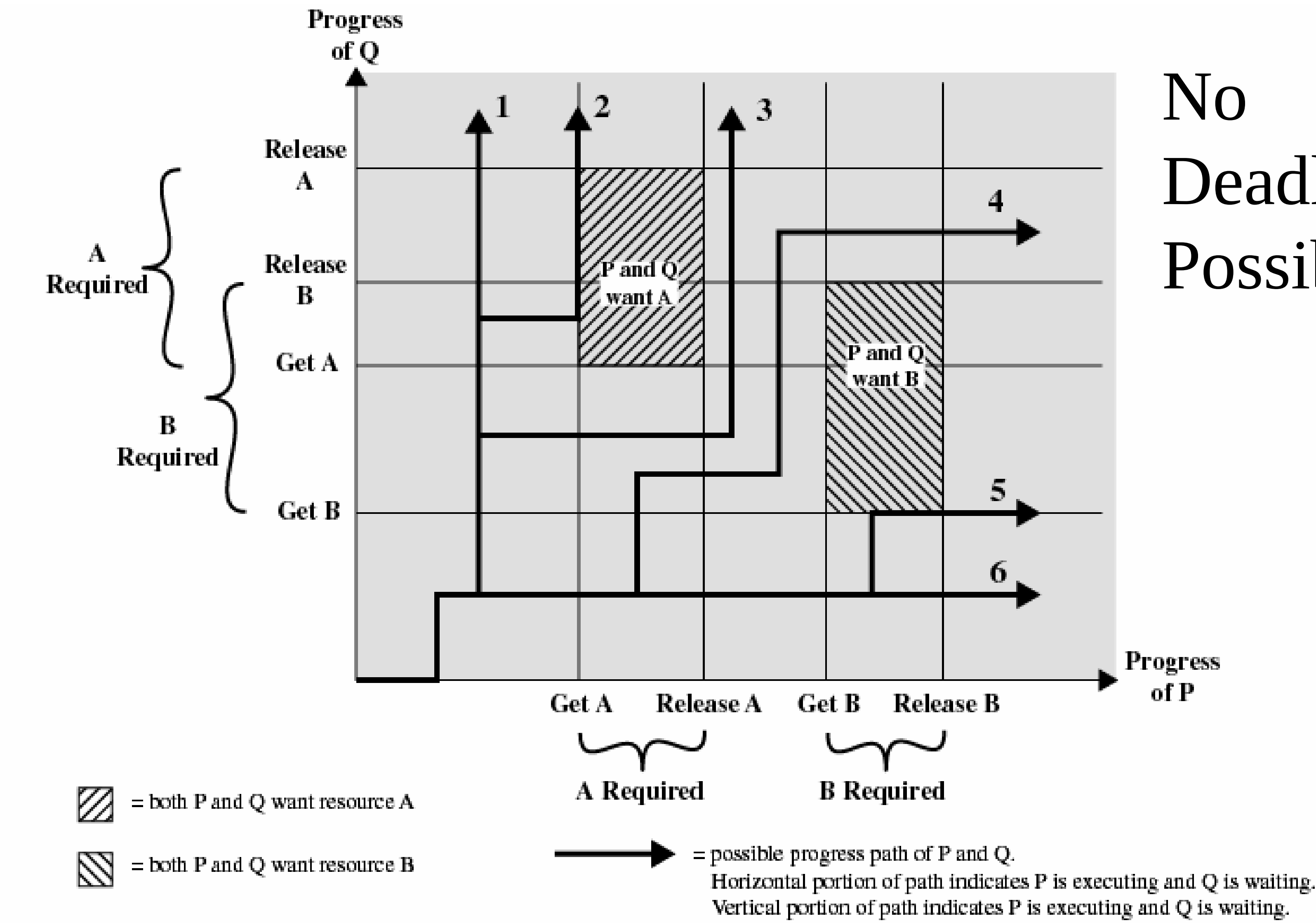
(a) Deadlock possible (b) Deadlock

Resources: Quadrants a, b, c, d

Joint Progress Diagram



Joint Progress Diagram



No Deadlock Possible

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
- Examples:
  - 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

Example of Deadlock

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

Another Example of Deadlock

- Space is available for allocation of 200Kbytes, 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

- Created (produced) and destroyed (consumed)
- Examples:
  - Interrupts, signals, messages, and info in I/O buffers
- Deadlock may occur if a Receive message is blocking

P1	P2
• • •	• • •
Receive(P2);	**Receive(P1);**
• • •	• • •
Send(P2, M1);	**Send(P1, M2);**

Resource Allocation Graphs

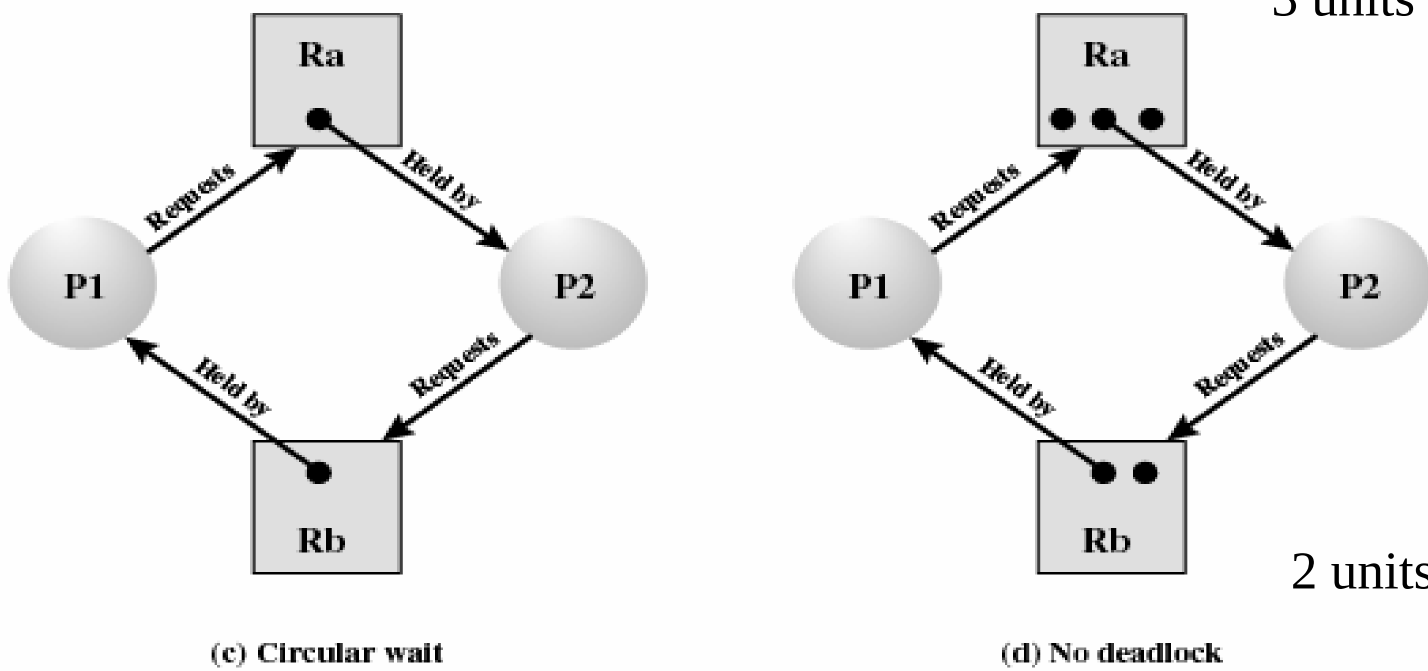
- Directed graph that depicts a state of the system of resources and processes

(a) Resouce is requested

(b) Resource is held

Resource Allocation Graphs

3 units of Ra



2 units of Rb

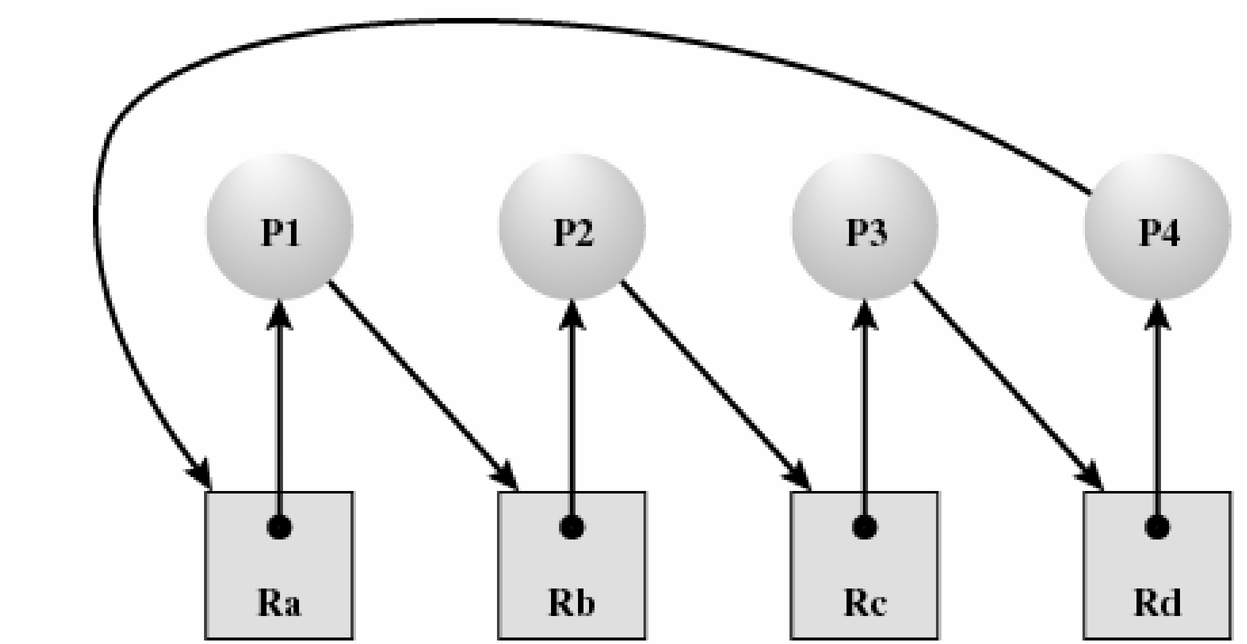
Figure 6.5 Examples of Resource Allocation Graphs

Conditions for Deadlock

1. Mutual exclusion
   – Only one process may use a resource at a time
2. Hold-and-wait
   – A process may hold allocated resources while awaiting assignment of others
3. No preemption
   – No resource can be forcibly removed form a process holding it

Conditions for Deadlock

4. 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



Possibility of Deadlock

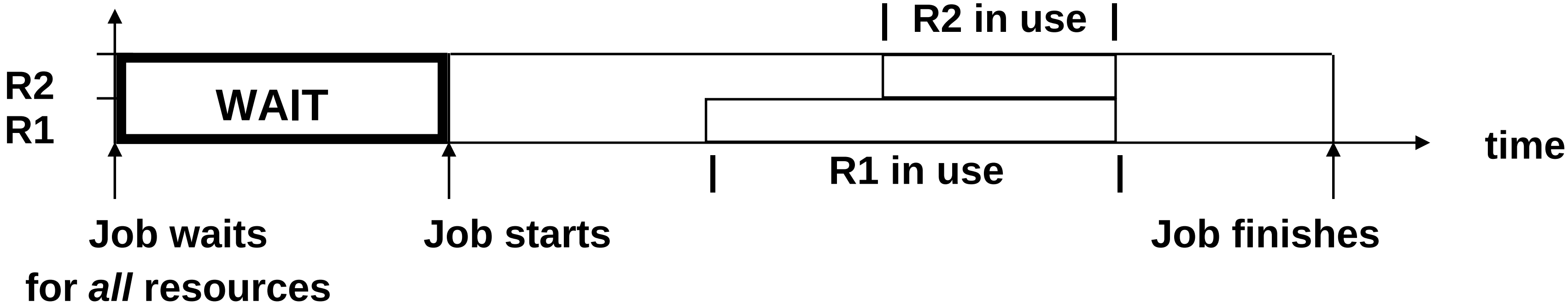
- Mutual Exclusion
- No preemption
- Hold and wait

Existence of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

Three Solutions to Deadlock

#1: Mr./Ms. Conservative ***(Prevention)***

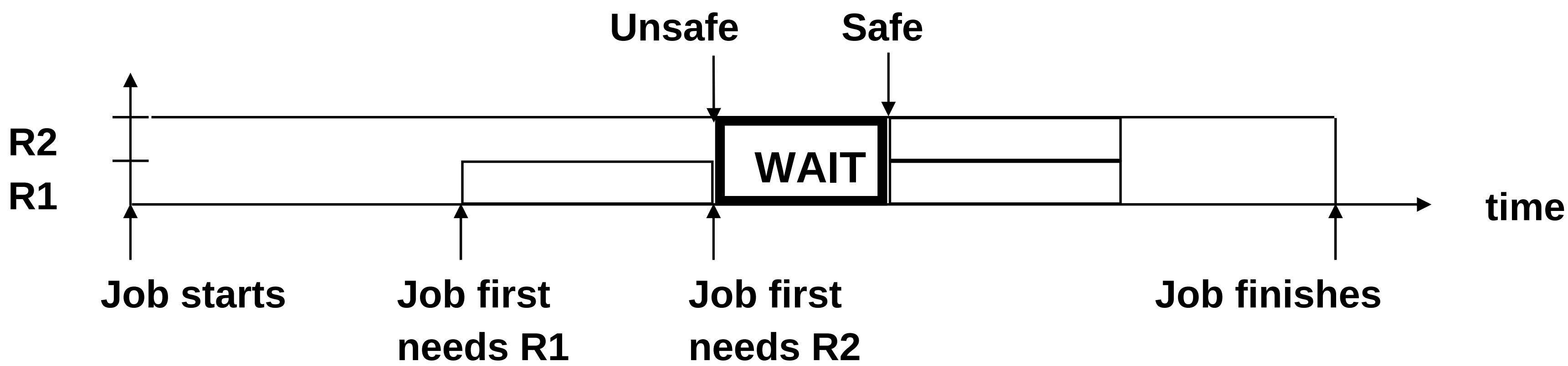


“We had better not allocate if it could ever cause deadlock”

Process **waits** until all needed resource free
Resources **underutilized**

Three Solutions to Deadlock …

<u>#2: Mr./Ms. Prudent</u> ***(Avoidance)***

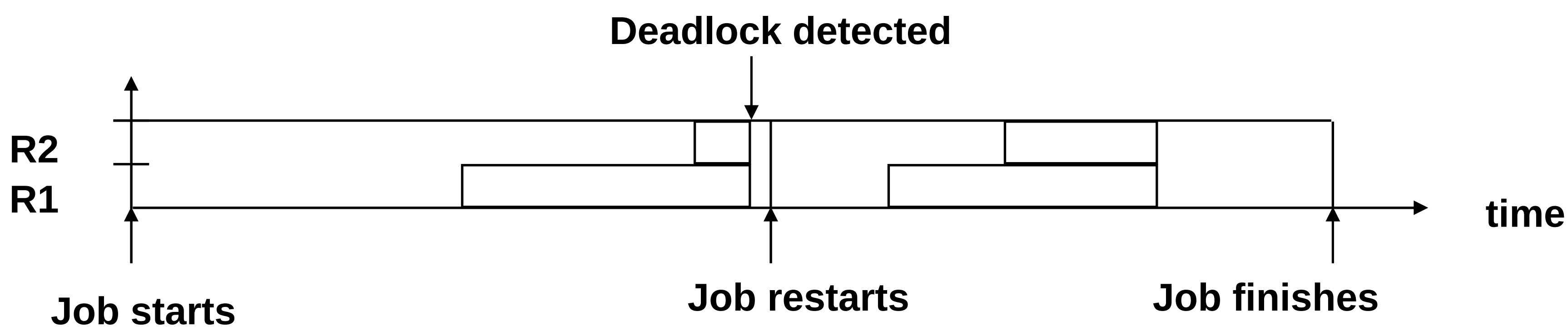


"If resource is free and with its allocation we can still guarantee that everyone will finish, **use it**."

Better resource utilization
Process still waits

Three Solutions to Deadlock…

#3: Mr./Ms. Liberal ***(Detection/Recovery)***



"If it's free, use it -- why wait?"

Good resource utilization, minimal process wait time
Until deadlock occurs....

Names for The Three Methods

1) Deadlock Prevention

– Design system so possibility of deadlock avoided *a priori*

2) Deadlock Avoidance

– Design system so that if a resource request is made that *could* lead to deadlock, then block requesting process.

– Requires knowledge of future resource requests by processes

3) Deadlock Detection and Recovery

– Algorithm to detect deadlock

– Recovery scheme

Deadlock Prevention

Deny one of the 4 necessary conditions

Mutual Exclusion

No preemption

Hold and wait

Circular wait

Deadlock Prevention

- **Do not allow "Mutual Exclusion"**
  - Use only sharable resources

    => Impossible for practical systems

Deadlock Prevention …

- **Prevent “Hold and Wait”**

  (a) Preallocation - process must request and be allocated all of its required resources before it can start execution

  (b) Process must release all of its currently held resources and re-request them along with request for new resources*

  => Very inefficient

  => Can cause "indefinite postponement": jobs needing lots of resources may never run

Deadlock Prevention …

- **Allow "Resource Preemption"**
  - Allowing one process to acquire exclusive rights to a resource currently being used by a second process

=> Some resources can not be preempted without detrimental implications (e.g., printers, tape drives)

=> May require jobs to restart

Deadlock Prevention …

- **Prevent Circular Wait**
  - Order resources and
  - Allow requests to be made only in an increasing order

Preventing Circular Wait

Impose an ordering on Resources:

1	**W**
2	**X**
3	**Y**
4	**Z**

Process:	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
Request:	**W**	**X**	**Y**	**Z**	**X**	**Y**	**Z**	**W**

After first 4 requests:

A / W

D / Z

B / X

C / Y

Process D cannot request resource W without voluntarily releasing Z first

Problems with Linear Ordering Approach

(1) Adding a new resource that upsets ordering requires <u>all</u> code ever written for system to be modified!

(2) Resource numbering affects efficiency

=> A process may have to request a resource well before it needs it, just because of the requirement that it must request resources in ascending sequence

Deadlock Avoidance

- OS never allocates resources in a way that could lead to deadlock

  => Processes must tell OS in advance how many resources they will request

- Process Initiation Denial
  - Process is started only if **maximum claim** of all current processes plus those of the new process can be met.
- **Resource Allocation Denial**
  - Do not grant request if request might lead to deadlock

Resource Allocation Denial: Banker’s Algorithm

- Banker's Algorithm runs each time:
  - a process requests resource - *Is it Safe*?
  - a process terminates - *Can I allocate released resources to a suspended process waiting for them*?
- A new state is safe if and only if every process can complete after allocation is made

=> Make allocation, then check system state and de-allocate if safe/unsafe

Definition: Safe State

- State of a system
  - An enumeration of which processes hold, are waiting for, or might request which resources
- Safe state
  - No process is deadlocked, and there exists no possible sequence of future requests in which deadlock could occur.

    or alternatively,

  - No process is deadlocked, and the current state will not lead to a deadlocked state

Deadlock Avoidance

Safe State:

	Current Loan	Max Need
Process 1	1	4
Process 2	4	6
Process 3	5	8

Available = 2

Deadlock Avoidance

Unsafe State:

	Current Loan	Max Need
Process 1	8	10
Process 2	2	5
Process 3	1	3

Available = 1

Safe to Unsafe Transition

Current state being safe does not necessarily imply future states are safe

Current Safe State:

	Current Loan	Maximum Need	
Process 1	1	4	
Process 2	4	6	
Process3	5	8	**Available = 2**

Suppose Process 3 requests and gets one more resource

	Current Loan	Maximum Need	
User1	1	4	
User2	4	6	
User3	6	8	**Available = 1**

Essence of Banker's Algorithm

- Find an allocation schedule satisfying maximum claims that allows to complete jobs

  => Schedule exists <u>iff</u> safe

- Method: "Pretend" you are the CPU.

  1. Scan table (PCB?) row by row and find a job that can finish
  2. Add finished job's resources to number available.

  Repeat 1 and 2 until

  – all jobs finish (**<u>safe</u>**), or

  – no more jobs can finish, but some are still “waiting” for their maximum claim (resource) request to satisfied **<u>(unsafe)</u>**

Banker's Algorithm

Constants

```
int   N {number of processes}
int   Total_Units
int   MaximumNeed[i]
```

Variables

```
int   i {denotes a process}
int   Available
int   CurrentLoan[i]
boolean   Cannot_Finish[i]
```

Function

```
Claim[i]  =   MaximumNeed[i] - CurrentLoan[i];
```

Banker's Algorithm

```
Begin
    Available  =  Total_Units;

    For  i  =  1  to  N  Do
        Begin
        Available  =  Available - CurrentLoan [i];
        Cannot_Finish [i]  =  TRUE;
        End;

    i = 1;
    while ( i <= N ) Do
        begin
            If ( Cannot_Finish [i]  AND  Claim [i] <= Available )
                Then Begin
                    Cannot_Finish [i]  =  False;
                    Available  =  Available  +  CurrentLoan [i];
                    i  =  1;
                    End;
                Else i = i+1;
        End;
    If  ( Available  ==  Total_Units )
        Then Return ( SAFE )
        Else Return ( UNSAFE );
End;
```

Initialize

Find schedule to complete all jobs

Banker's Example #1

Total_Units = 10 units
N = 3 processes

Process:	**1**	**2**	**3**	**1**
Request:	**2**	**3**	**4**	**1**

Can the fourth request be satisfied?

Process	Current Loan	Maximum Need	Claim	Cannot Finish
1		4		
2		4		
3		8		

Available =
i =

Banker's Example #2

Total_Units = 10 units
N = 3 processes

Process:	**1**	**2**	**3**	**1**
Request:	**4**	**1**	**1**	**2**

Can the fourth request by satisfied?

Process	Current Loan	Maximum Need	Claim	Cannot Finish
1		10		
2		6		
3		3		

Available =
i =

Determination of a Safe State
Multi-Resource Scenario

	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

Is the resulting state (above) safe?

Is C[*]-A[*] <= V[*] ?

P2 -> P1 -> P3 -> P4

Determination of an Unsafe State Multi-resource Scenario

	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

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	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 Logic

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

(a) 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 >;
}
```

(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)

Banker's Algorithm: Summary

(+) PRO's:

☺ Deadlock never occurs.

☺ More flexible & more efficient than deadlock prevention. (Why?)

(-) CON's:

☹ Must know max use of each resource when job starts.

=> No truly dynamic allocation

☹ Process might block even though deadlock would never occur

Deadlock Detection

Allow deadlock to occur, then recognize that it exists

- Run deadlock detection algorithm whenever locked resource is requested
- Could also run detector in background

Deadlock Detection

	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

Set Avail' [*] = Avail [*]

Remove process i from consideration if:

(a) Alloc [i,*] = 0, or

(b) Request [i,*] <= Avail' [*]

Add Alloc [I,*] to Avail' [*]

P1 and P2 deadlocked

Processes not removed from consideration are blocked

Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Hoping alternate request sequence (non-determinism)
  - However, original deadlock may still occur
- Successively abort deadlocked processes until deadlock no longer exists
  - Free up needed resources

Selection Criteria Aborting 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

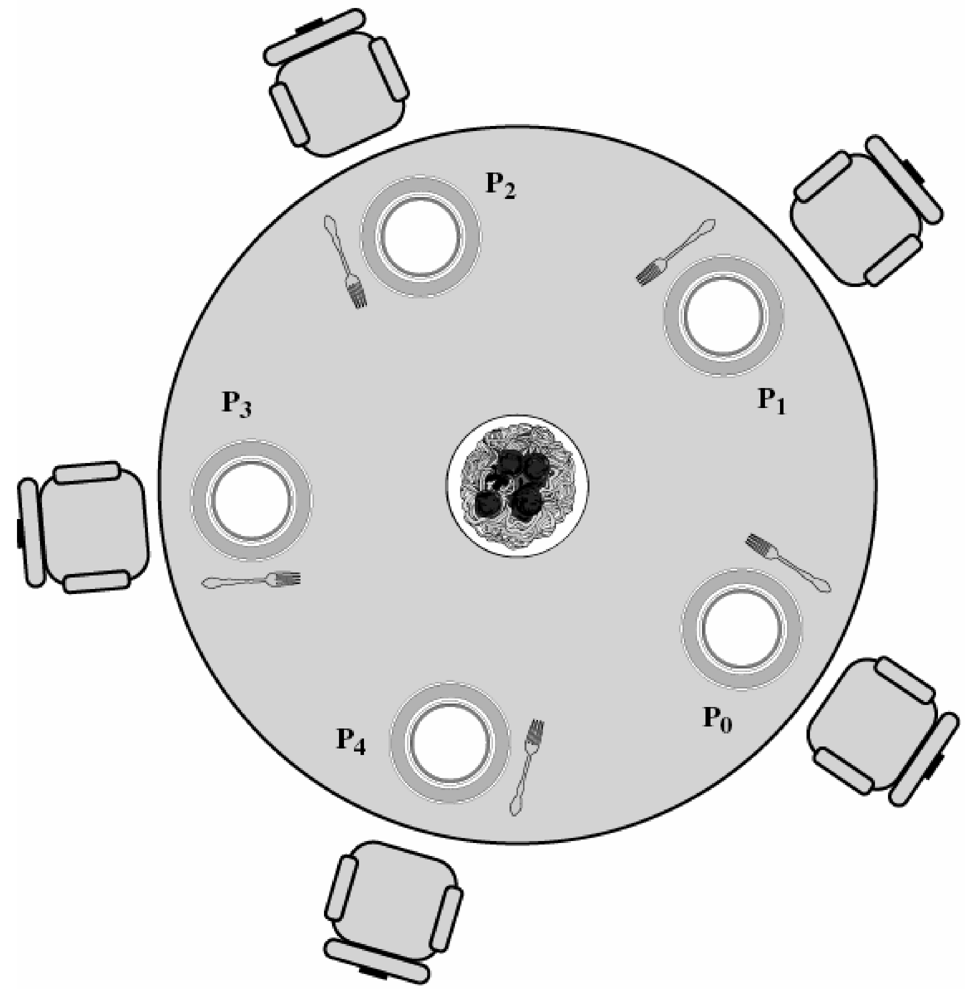


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem

```
/* 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));
     }
```

Each Philosopher request/gets fork(i)…
Deadlock

Dining Philosophers Problem

```
/* 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));
}
```

Limit number of Philosophers in dinning room… No Deadlock

Dining Philosophers: Monitor Solution

```
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]);
}
```

```
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 */
   }
}
```

First philosopher entering monitor is guaranteed to get both forks….

Appropriate waiting philosopher “woken” up