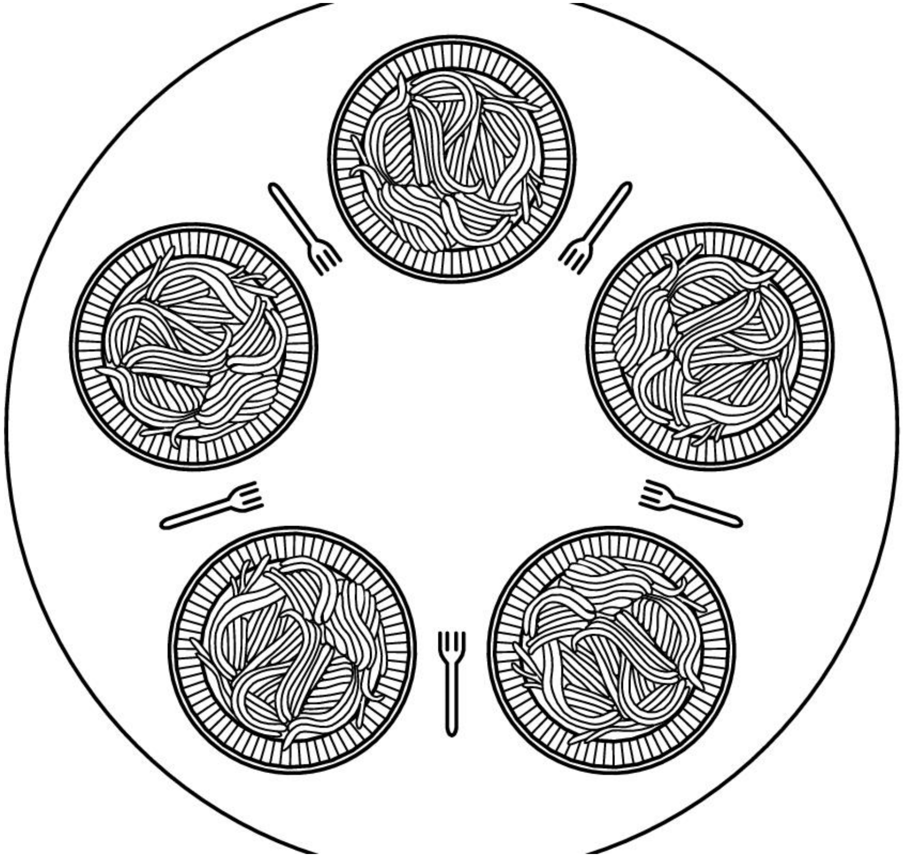
Dining Philosophers Problem



Dining Philosophers Problem
First solution

```
#define N 5                             /* number of philosophers */

void philosopher(int i)                 /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                       /* philosopher is thinking */
        take_fork(i);                   /* take left fork */
        take_fork((i+1) % N);           /* take right fork; % is modulo operator */
        eat( );                         /* yum-yum, spaghetti */
        put_fork(i);                    /* put left fork back on the table */
        put_fork((i+1) % N);            /* put right fork back on the table */
    }
}
```

- Take_fork() waits until the fork is available
- Available? Then seizes it

- **Ensure that two neighboring philosopher should not seize the same fork**

Dining Philosophers Problem

Each fork is implemented as a semaphore

- The structure of Philosopher *i*:

Semaphore fork [5] initialized to 1

```
do  {
        wait ( fork[i] );
        wait ( fork[ (i + 1) % 5] );

            //  eat

        signal ( fork[i] );
        signal (fork[ (i + 1) % 5] );

            //  think

} while (TRUE);
```

Ensures no two neighboring philosophers can eat simultaneously

- What is the problem with this algorithm?

Dining Philosophers Problem
First solution

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                  /* philosopher is thinking */
          take_fork(i);              /* take left fork */
          take_fork((i+1) % N);      /* take right fork; % is modulo operator */
          eat( );                    /* yum-yum, spaghetti */
          put_fork(i);               /* put left fork back on the table */
          put_fork((i+1) % N);       /* put right fork back on the table */
     }
}
```

- Take_fork() waits until the fork is available
- Available? Then seizes it

- Suppose all of them take the left fork simultaneously
  - None of them will get the right fork
  - Deadlock

Dining Philosophers Problem
Second solution

- After taking the left fork, philosopher checks to see if right fork is available
  - If not, puts down the left fork

Limitation

- All of them start simultaneously, pick up the left forks
- Seeing that their right forks are not available
  - Putting down their left fork
- Starvation

- Random delay (Exponential backoff) not going to help for critical systems

Dining Philosophers Problem
Third solution

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( ); Wait(mutex);       /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat( );                      /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }   signal(mutex);
}
```

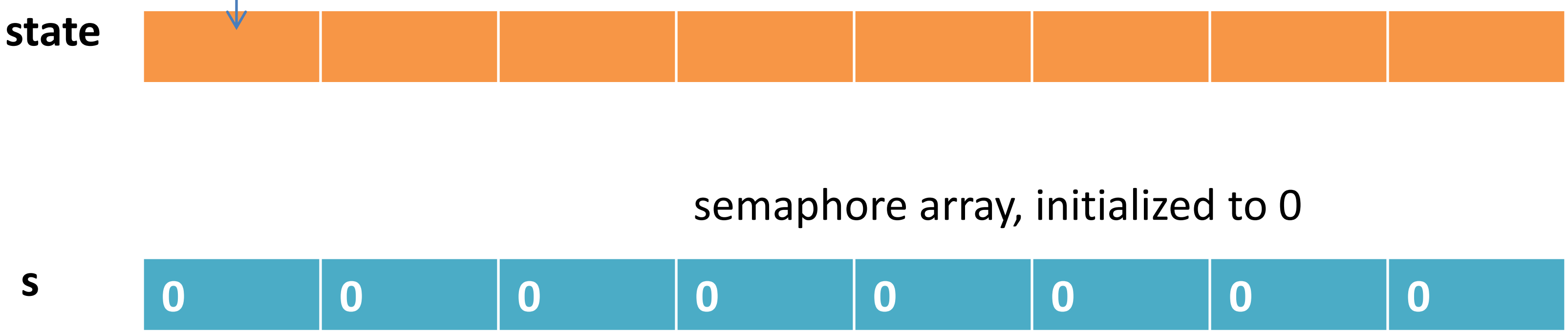
Poor resource utilization

Dining Philosophers Problem
Final solution

For each philosopher, maintain **state** and **s**

Thinking (0),
Hungry (1),
Eating (2)

Normal int array



- **State** takes care of acquiring the fork
- **s** stops a philosopher from eating when fork is not available

Dining Philosophers Problem Final solution

```
#define N          5             /* number of philosophers */
#define LEFT       (i+N−1)%N     /* number of i's left neighbor */
#define RIGHT      (i+1)%N       /* number of i's right neighbor */
#define THINKING   0             /* philosopher is thinking */
#define HUNGRY     1             /* philosopher is trying to get forks */
#define EATING     2             /* philosopher is eating */
typedef int semaphore;           /* semaphores are a special kind of int */
int state[N];                    /* array to keep track of everyone's state */
semaphore mutex = 1;             /* mutual exclusion for critical regions */
semaphore s[N];                  /* one semaphore per philosopher */

void philosopher(int i)          /* i: philosopher number, from 0 to N−1 */
{
     while (TRUE) {              /* repeat forever */
          think( );              /* philosopher is thinking */
          take_forks(i);         /* acquire two forks or block */
          eat( );                /* yum-yum, spaghetti */
          put_forks(i);          /* put both forks back on table */
..   }
}
```

Dining Philosophers Problem
Final solution

```
...

void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

...
```

Dining Philosophers Problem
Final solution

. . .

```
void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                        /* enter critical region */
      state[i] = THINKING;                 /* philosopher has finished eating */
      test(LEFT);                          /* see if left neighbor can now eat */
      test(RIGHT);                         /* see if right neighbor can now eat */
      up(&mutex);                          /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

The Sleeping Barber Problem

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting − 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair();                 /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut();              /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Barber sleeps on "**Customer**"
Customer sleeps on "**Barber**"

For Barber: Checking the waiting room and calling the customer makes the **critical section**

For customer: Checking the waiting room and informing the barber makes its **critical section**

Deadlock

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one

- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait (B);
wait(B)	wait(A)

Introduction To Deadlocks

Deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.

- Each process requests for an instance of a resource type

- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock: necessary conditions

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

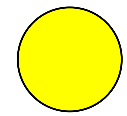
Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

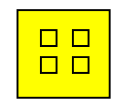
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

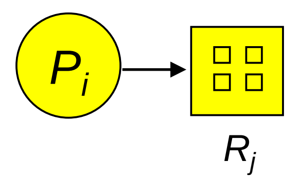
- Process



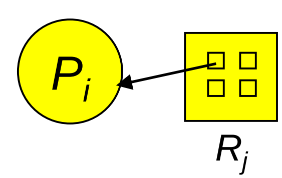
- Resource Type with 4 instances



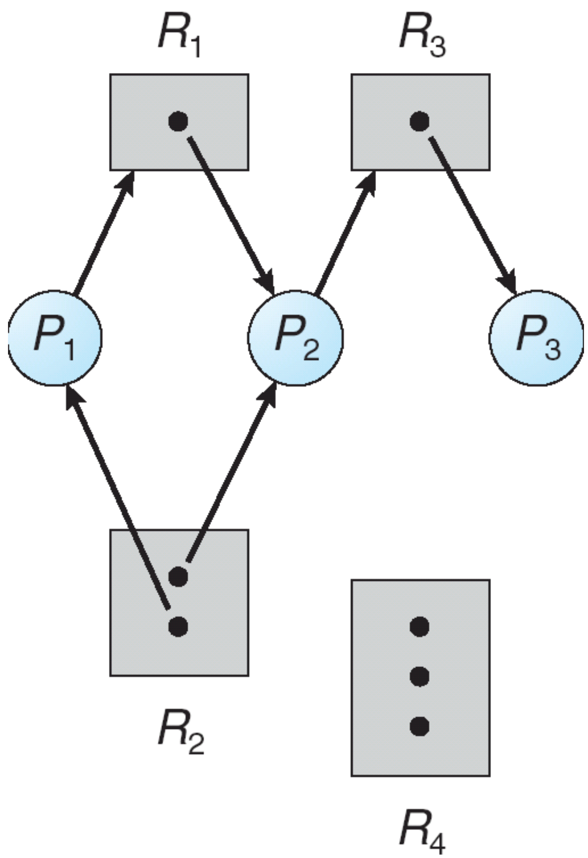
- P_i requests instance of R_j



- P_i is holding an instance of R_j

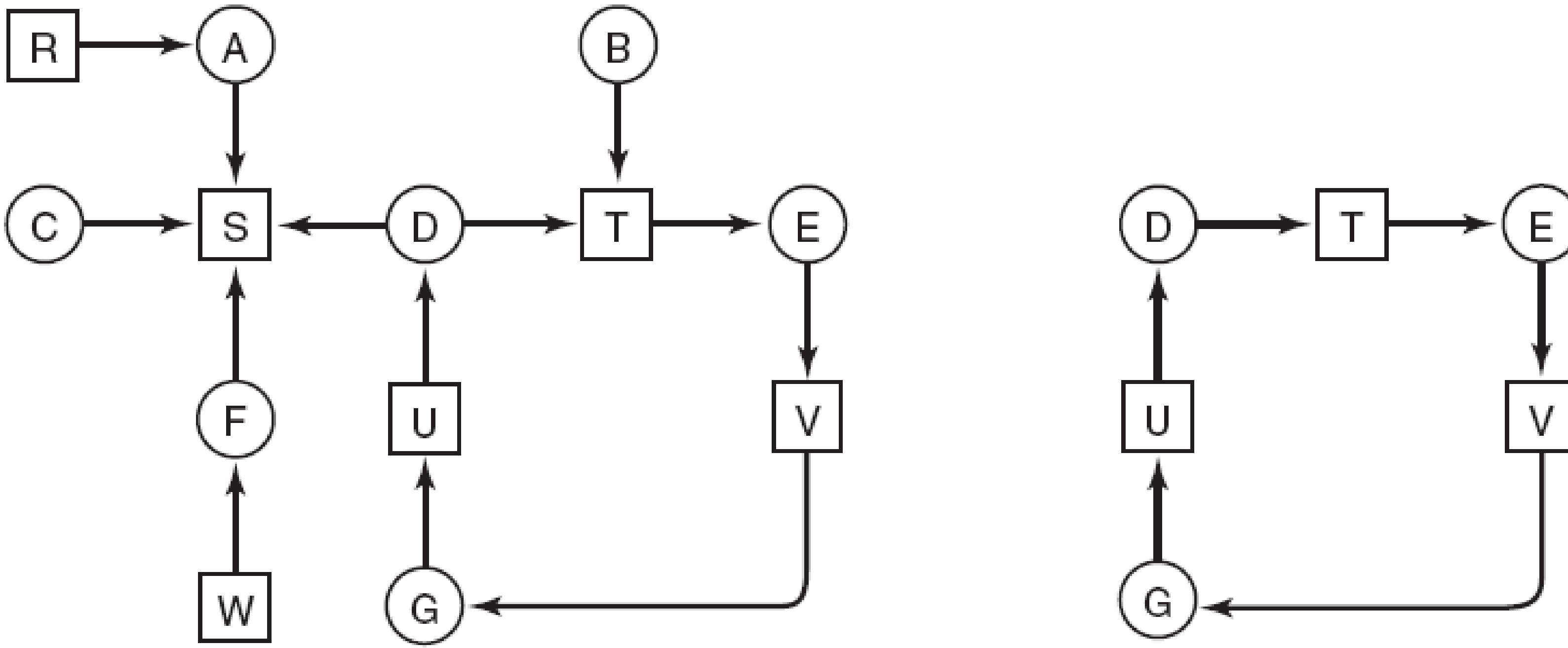


Example of a Resource Allocation Graph



No cycle; No deadlock

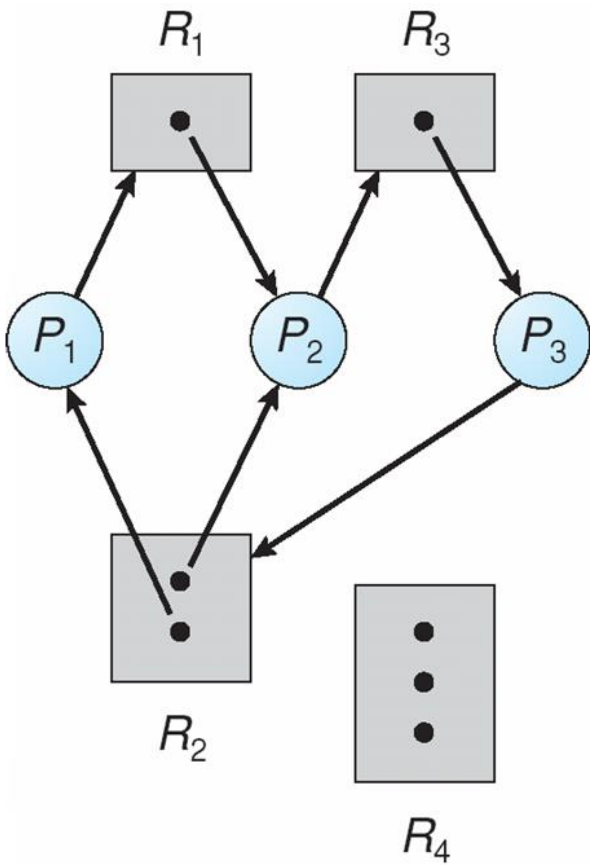
One Resource of Each Type



(a) (b)

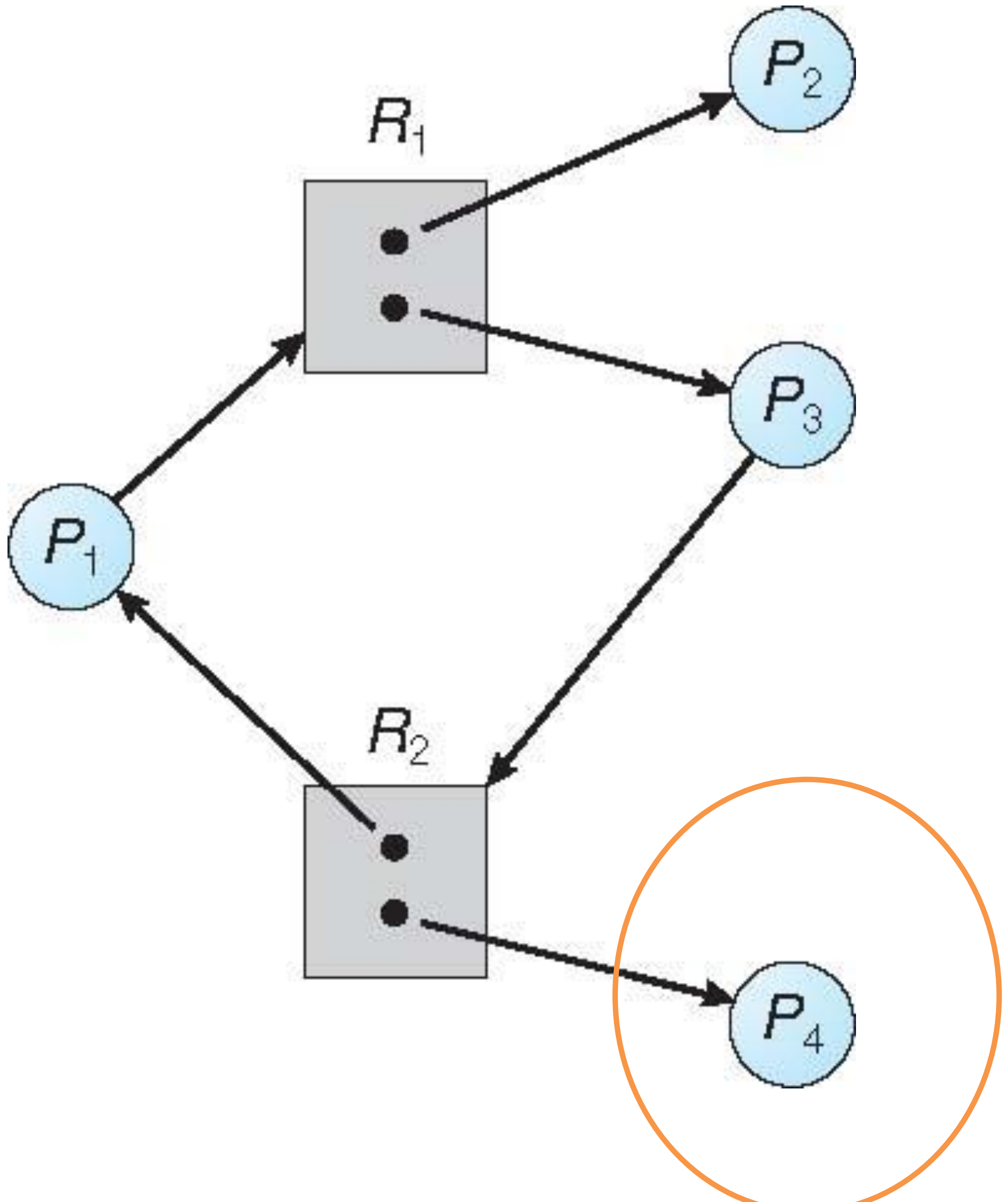
Contains Cycle; Deadlock

Resource Allocation Graph With A Deadlock



P_3 requests R_2

Graph With A Cycle But No Deadlock



P1->R1->P3->R2->P1

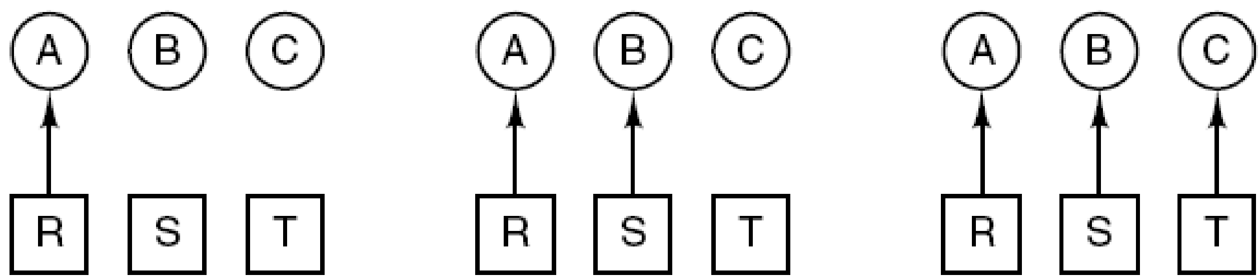
- If the resource allocation graph does not have a cycle
  - System is not in a deadlocked state
- If there is a cycle
  - May or may not be in a deadlocked state

Deadlock Modeling

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R
(a)	(b)	(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)

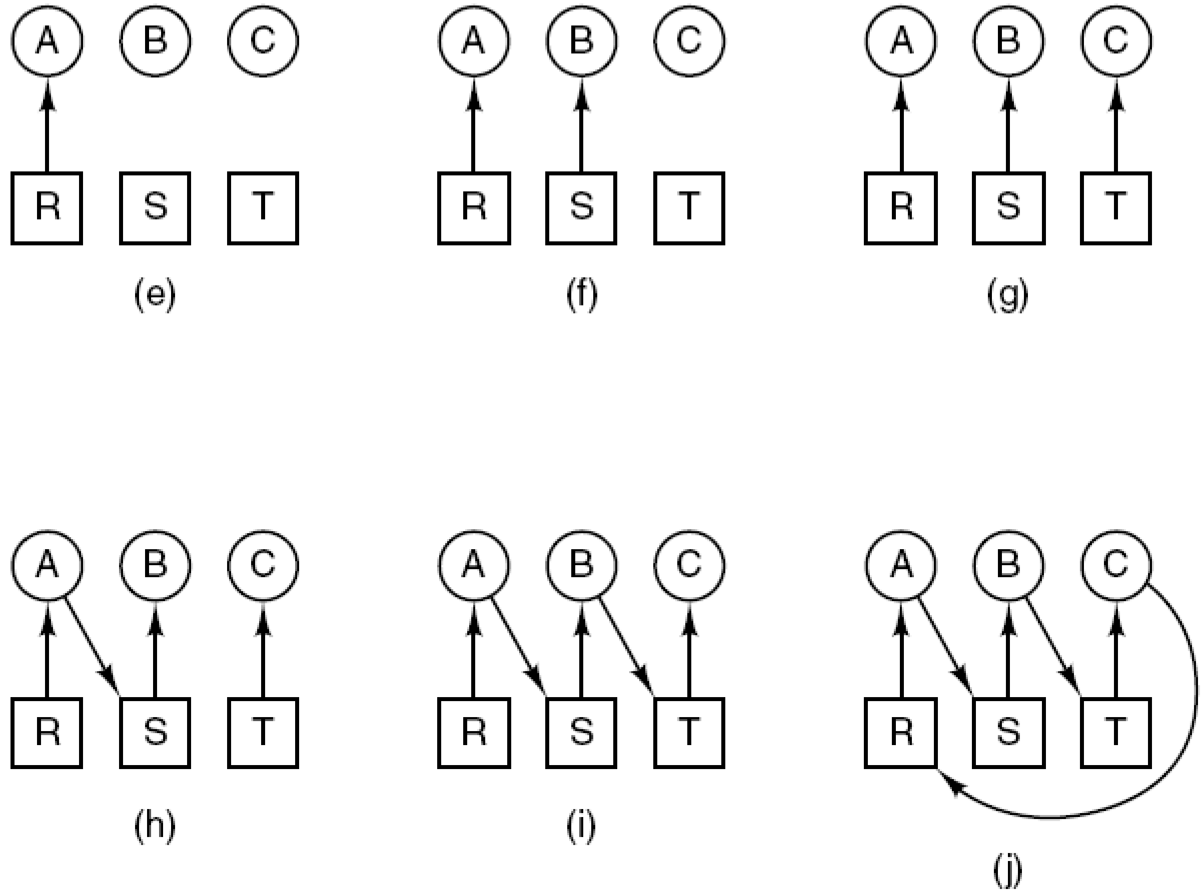


(e) (f) (g)

Deadlock Modeling

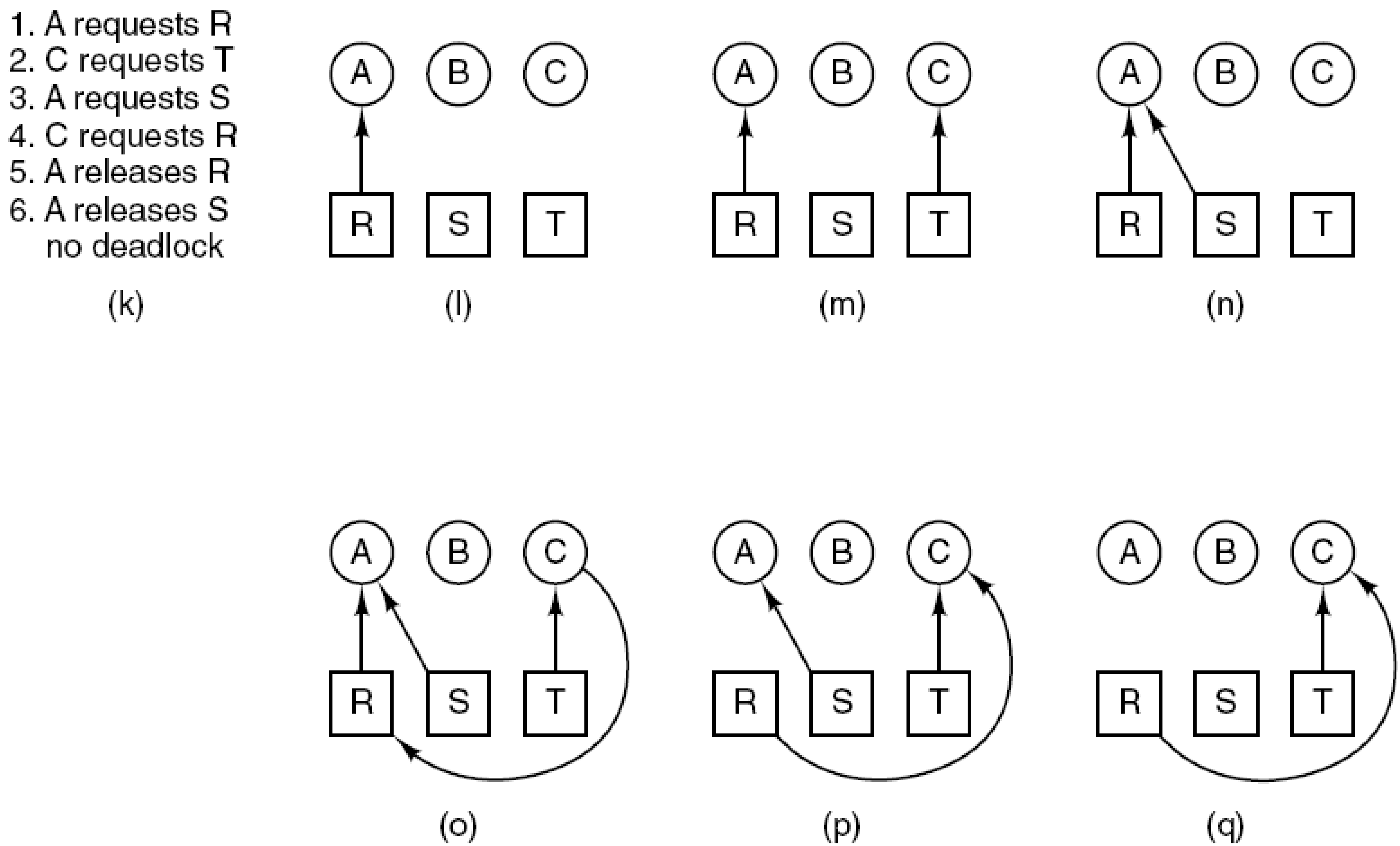
- 1. A requests R
  2. B requests S
  3. C requests T
  4. A requests S
  5. B requests T
  6. C requests R
     deadlock

(d)



Deadlock

Deadlock Modeling



Suspend process B

Deadlock Handling

Strategies for dealing with deadlocks:

1. Detection and recovery. Let deadlocks occur, detect them, take action.
2. Dynamic avoidance by careful resource allocation.
3. Prevention, by structurally negating one of the four required conditions.
4. Just ignore the problem.