Betriebssysteme und Systemnahe Programmierung

Kapitel 4 • Koordination

Winter 2016/17

Marcel Waldvogel

Race Conditions

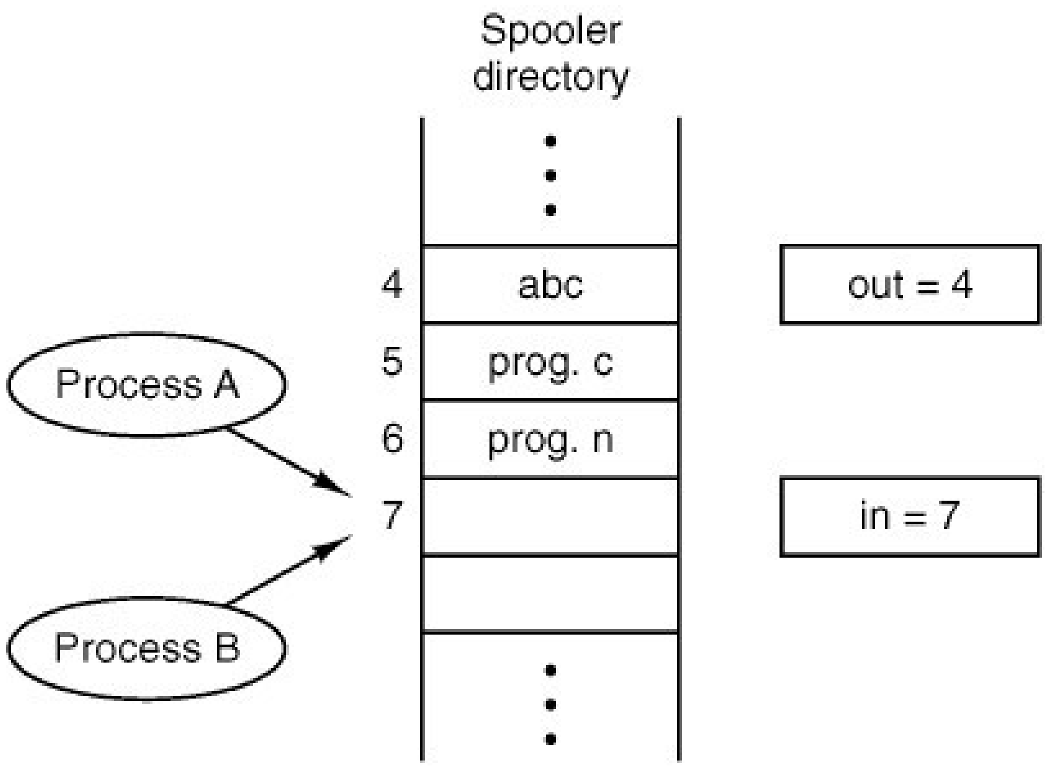


Figure 2-8 Two processes want to access shared memory at the same time.

Critical Sections

Necessary to avoid race conditions:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Mutual Exclusion with Busy Waiting

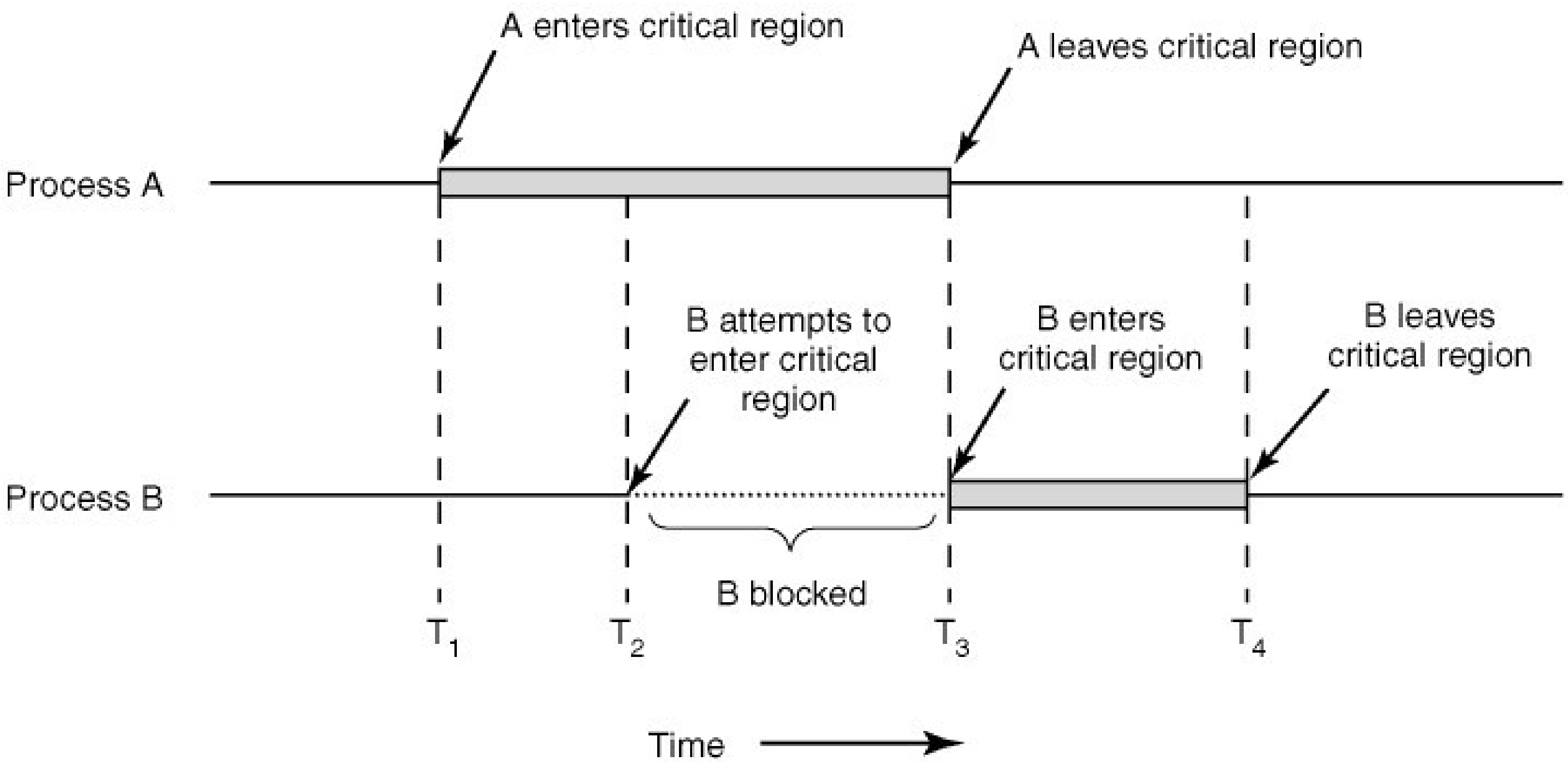


Figure 2-9 Mutual exclusion using critical regions.

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-10. A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Peterson's Solution (1)

```
#define FALSE  0
#define TRUE   1
#define N      2                       /* number of processes */

int turn;                              /* whose turn is it? */
int interested[N];                     /* all values initially 0 (FALSE) */

void enter_region(int process)         /* process is 0 or 1 */
{ . . .
```

Figure 2-11 Peterson's solution for achieving mutual exclusion.

Peterson's Solution (2)

```
void enter_region(int process)          /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```

Figure 2-11 Peterson's solution for achieving mutual exclusion.

The TSL Instruction

```
enter_region:
        tsl register,lock       | copy lock to register and set lock to 1
        cmp register,#0         | was lock zero?
        jne enter_region        | if it was non zero, lock was set, so loop
        ret                     | return to caller; critical region entered

leave_region:
        move lock,#0            | store a 0 in lock
        ret                     | return to caller
```

Figure 2-12. Entering and leaving a critical region using the TSL instruction.

The Producer-Consumer Problem (1)

```
#define N 100                               /* number of slots in the buffer */
int count = 0;                              /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                         /* repeat forever */
          item = produce_item( );           /* generate next item */
          if (count == N) sleep( );         /* if buffer is full, go to sleep */
          insert_item(item);                /* put item in buffer */
          count = count + 1;                /* increment count of items in buffer */
          if (count == 1) wakeup(consumer); /* was buffer empty? */
     }
}
   . . .
```

Figure 2-13. The producer-consumer problem with a fatal race condition.

The Producer-Consumer Problem (2)

```
  . . .
void consumer(void)
{
     int item;

     while (TRUE) {                          /* repeat forever */
          if (count == 0) sleep( );          /* if buffer is empty, got to sleep */
          item = remove_item( );             /* take item out of buffer */
          count = count – 1;                 /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                /* print item */
     }
}
```

Figure 2-13. The producer-consumer problem with a fatal race condition

The Producer-Consumer Problem (3)

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                    /* TRUE is the constant 1 */
          item = produce_item( );      /* generate something to put in buffer */
          down(&empty);                /* decrement empty count */
          down(&mutex);                /* enter critical region */
          insert_item(item);           /* put new item in buffer */
          up(&mutex);                  /* leave critical region */
          up(&full);                   /* increment count of full slots */
     }
}
 . . .
```

Figure 2-14. The producer-consumer problem using semaphores.

The Producer-Consumer Problem (4)

```
. . .

void consumer(void)
{
     int item;

     while (TRUE) {                      /* infinite loop */
          down(&full);                   /* decrement full count */
          down(&mutex);                  /* enter critical region */
          item = remove_item( );         /* take item from buffer */
          up(&mutex);                    /* leave critical region */
          up(&empty);                    /* increment count of empty slots */
          consume_item(item);            /* do something with the item */
     }
}
```

Figure 2-14. The producer-consumer problem using semaphores.

Monitors (1)

```
monitor example
  integer i;
  condition c;

  procedure producer(x);
  .
  .
  .
  end;

  procedure consumer(x);
  .
  .
  .
  end;
end monitor;
```

Figure 2-15. A monitor.

Monitors (2)

```
monitor ProducerConsumer
      condition full, empty;
      integer count;

      procedure insert(item: integer);
      begin
            if count = N then wait(full);
            insert_item(item);
            count := count + 1;
            if count = 1 then signal(empty)
      end;

      function remove: integer;
      begin
            if count = 0 then wait(empty);
            remove = remove_item;
            count := count − 1;
            if count = N − 1 then signal(full)
      end;

      count := 0;
end monitor;
```

Figure 2-16. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots

Monitors (3)

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Figure 2-16. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots

Message Passing (1)

```
#define N 100                              /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                            /* message buffer */

     while (TRUE) {
          item = produce_item( );          /* generate something to put in buffer */
          receive(consumer, &m);           /* wait for an empty to arrive */
          build_message(&m, item);         /* construct a message to send */
          send(consumer, &m);              /* send item to consumer */
     }
}
          . . .
```

Figure 2-17. The producer-consumer problem with N messages.

Message Passing (2)

```
. . .

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);              /* get message containing item */
          item = extract_item(&m);            /* extract item from message */
          send(producer, &m);                 /* send back empty reply */
          consume_item(item);                 /* do something with the item */
     }
}
```

Figure 2-17. The producer-consumer problem with N messages.

The Dining Philosophers Problem (1)

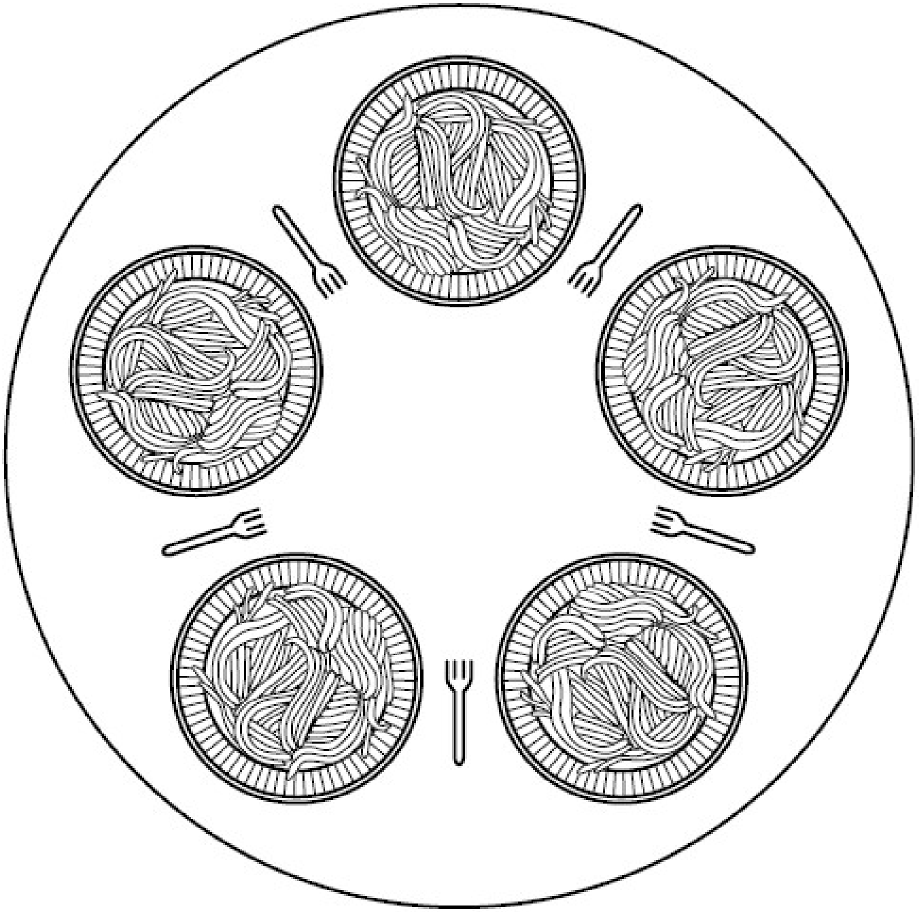


Figure 2-18. Lunch time in the Philosophy Department.

The Dining Philosophers Problem (2)

```
#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 */
     }
}
```

Figure 2-19. A nonsolution to the dining philosophers problem.

The Dining Philosophers Problem (3)

```
#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 */

    . . .
```

Figure 2-20. A solution to the dining philosophers problem.

The Dining Philosophers Problem (4)

```
. . .
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 */
      }
}

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 */
}
      . . .
```

Figure 2-20. A solution to the dining philosophers problem.

The Dining Philosophers Problem (5)

```
. . .
    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]);
         }
    }
```

Figure 2-20. A solution to the dining philosophers problem.

The Readers and Writers Problem (1)

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                     /* repeat forever */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc + 1;                  /* one reader more now */
          if (rc == 1) down(&db);       /* if this is the first reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          read_data_base();             /* access the data */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc – 1;                  /* one reader fewer now */
          if (rc == 0) up(&db);         /* if this is the last reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          use_data_read();              /* noncritical region */
     }
}
     . . .
```

Figure 2-21. A solution to the readers and writers problem.

The Readers and Writers Problem (2)

```
. . .

void writer(void)
{
     while (TRUE) {                  /* repeat forever */
          think_up_data( );          /* noncritical region */
          down(&db);                 /* get exclusive access */
          write_data_base( );        /* update the data */
          up(&db);                   /* release exclusive access */
     }
}
```

Figure 2-21. A solution to the readers and writers problem.

Definition of Deadlock

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.

Conditions for Deadlock

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock Modeling

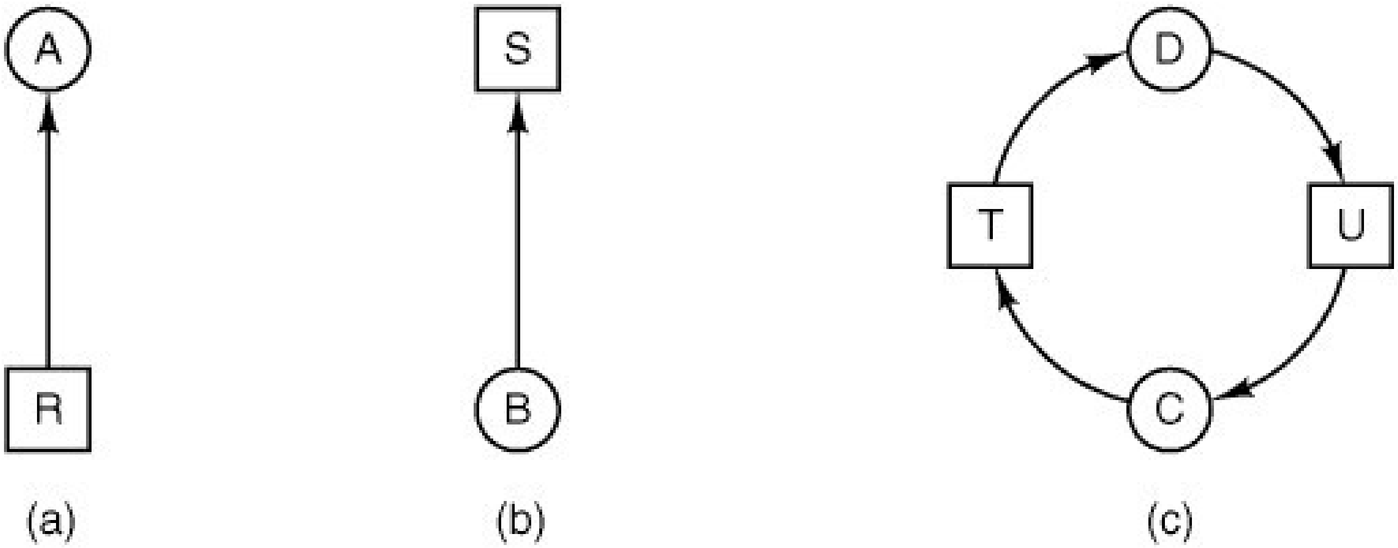


Figure 3-9. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Deadlock Handling Strategies

1. Ignore the problem altogether
2. Detection and recovery
3. Avoidance by careful resource allocation
4. Prevention by negating one of the four necessary conditions

Deadlock Avoidance (1)

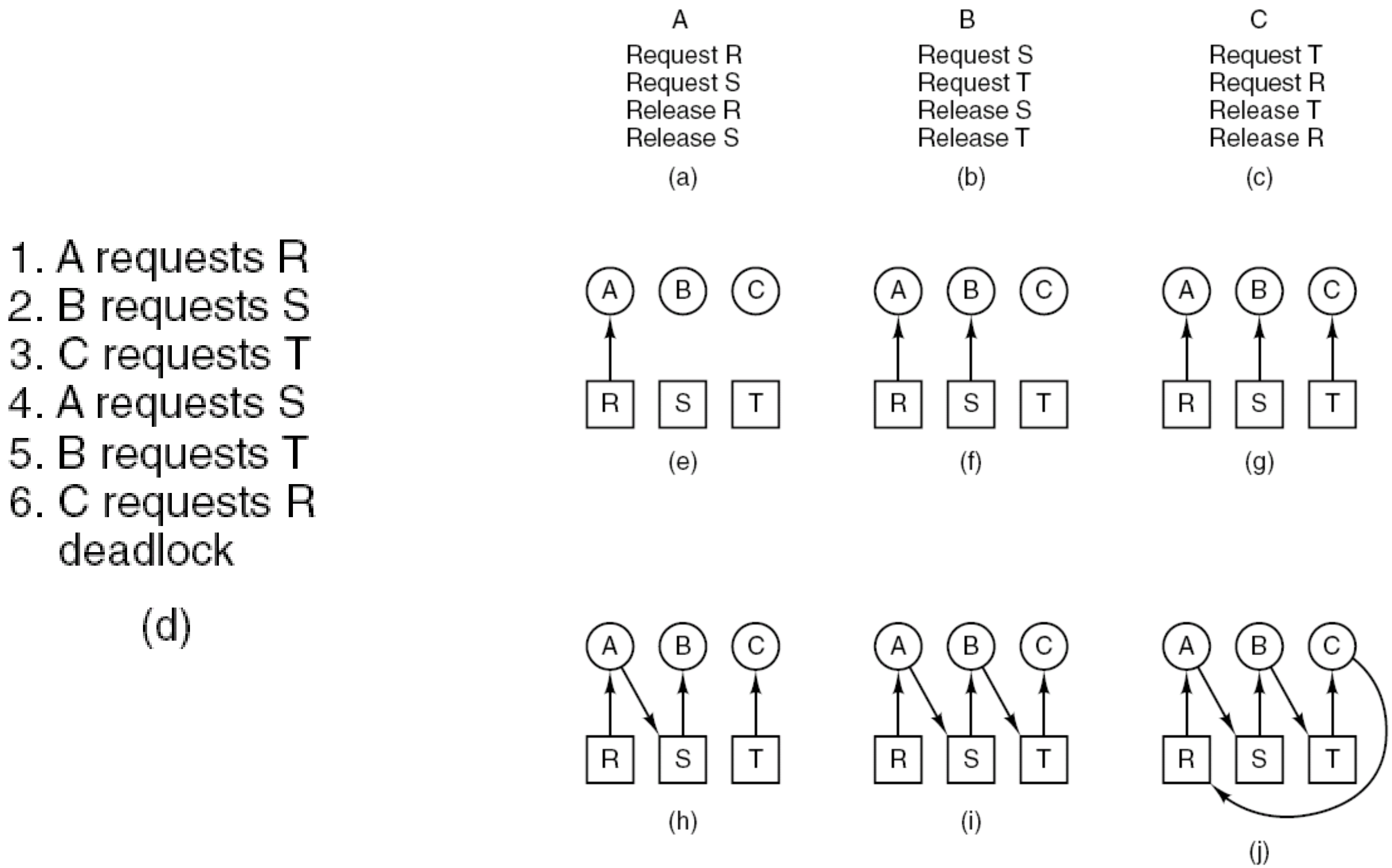


Figure 3-10. An example of how deadlock occurs and how it can be avoided.

Deadlock Avoidance (2)

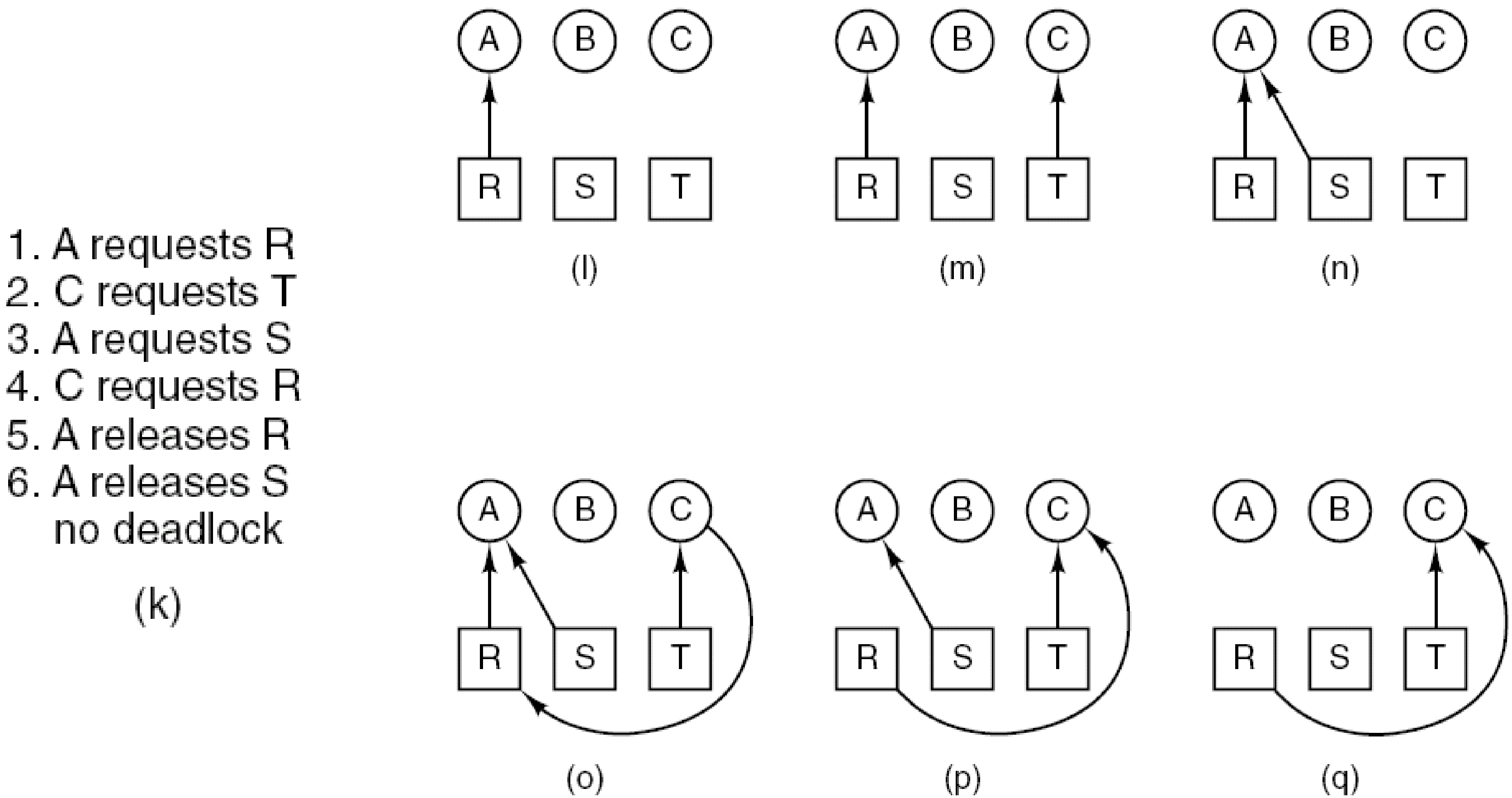
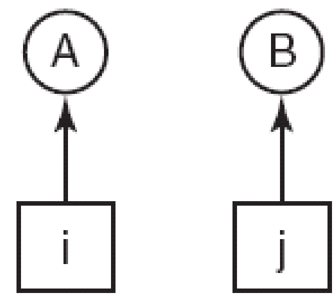


Figure 3-10. An example of how deadlock occurs and how it can be avoided.

Deadlock Prevention (1)

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a)



(b)

Figure 3-11. (a) Numerically ordered resources.
(b) A resource graph.

Deadlock Prevention (2)

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 3-12. Summary of approaches to deadlock prevention.

The Banker's Algorithm for a Single Resource

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Figure 3-13. Three resource allocation states:
(a) Safe. (b) Safe. (c) Unsafe.

Resource Trajectories

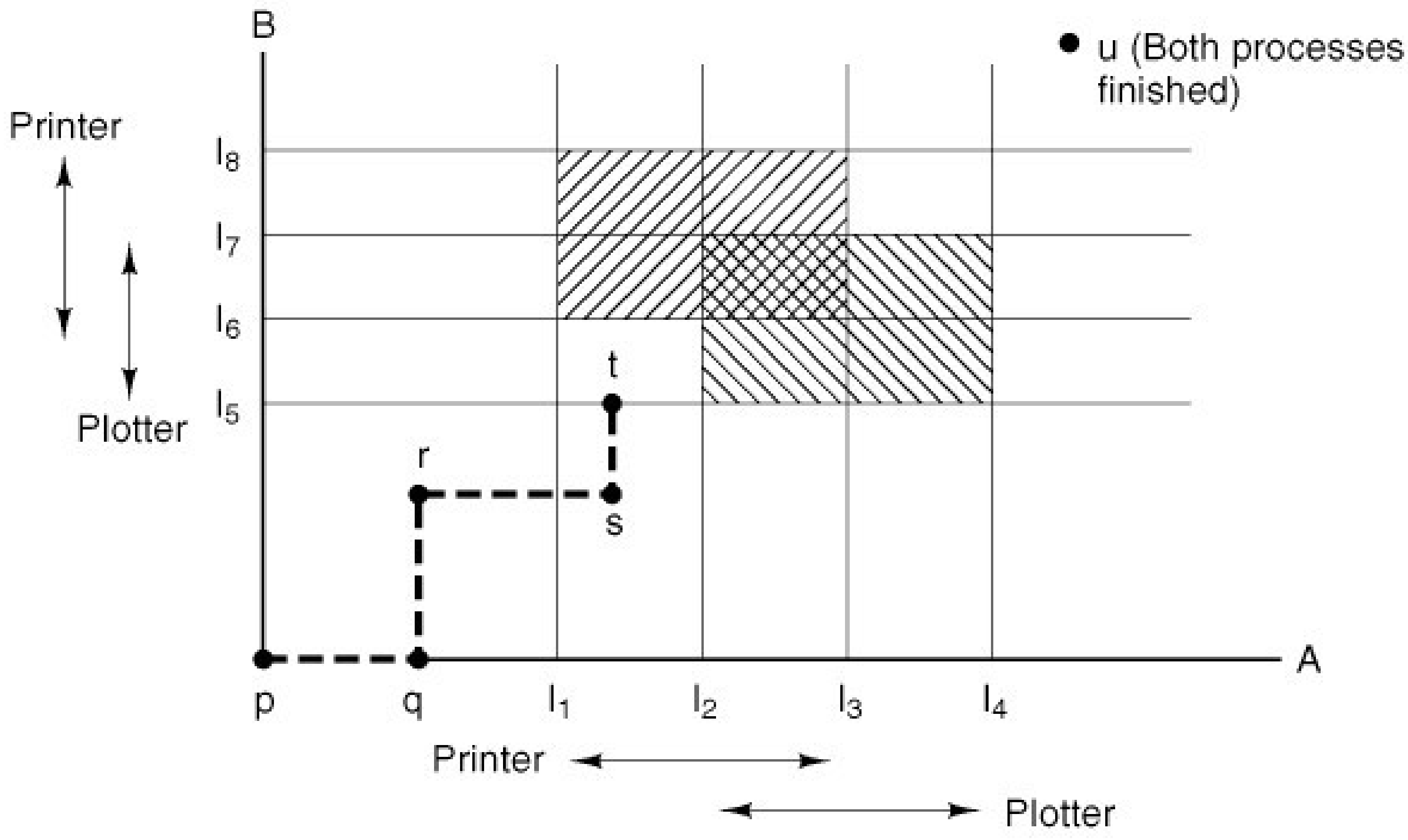


Figure 3-14. Two process resource trajectories.

The Banker's Algorithm for Multiple Resources

Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Figure 3-15. The banker's algorithm with multiple resources.

Safe State Checking Algorithm

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion.
2. Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.