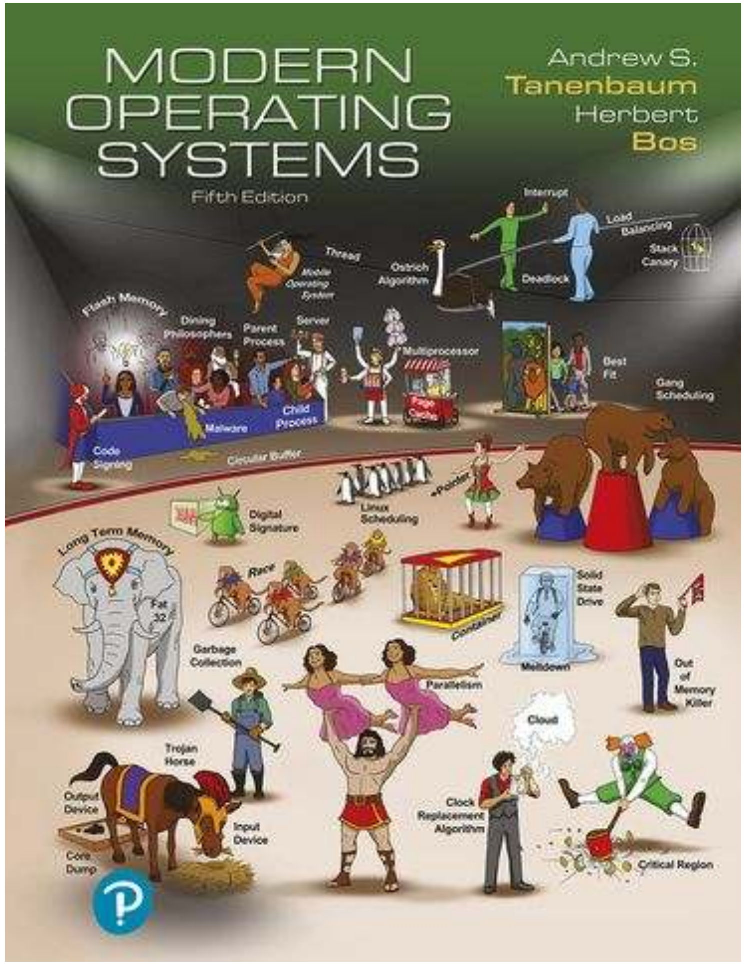
Modern Operating Systems

Fifth Edition



Chapter 6

Deadlocks

Pearson

Overview

- Deadlock definition and modeling
- Deadlock detection
- Deadlock avoidance
- Deadlock prevention
- Deadlock handling in practice

Overview

- **Deadlock definition and modeling**
- Deadlock detection
- Deadlock avoidance
- Deadlock prevention
- Deadlock handling in practice

Using Resources

Sequence of events required to use a resource (e.g., disk)

1. Request the resource
2. Use the resource
3. Release the resource

Resource Acquisition (1 of 2)

```
typedef int semaphore;
semaphore resource_1;


void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}
```

(b)

Figure 6-1. Using a semaphore to protect resources. (a) One resource. (b) Two resources.

Resource Acquisition (2 of 2)

```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```

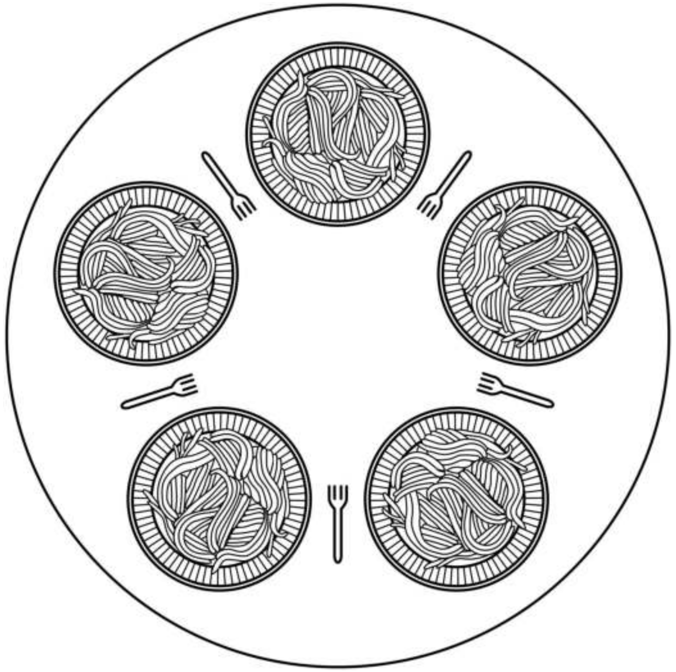
(b)

Figure 6-2. (a) Deadlock-free code. (b) Code with a potential deadlock.

The Dining Philosophers Problem (1 of 5)

- There are 5 philosophers, each seated at a round table in front of a plate of spaghetti. A philosopher needs 2 forks to eat the spaghetti, located on his left and his right.
- Obvious (non)solution:

```
void philosopher(int i) {
    while(TRUE) {
        think();
        take_fork(LEFT(i));
        take_fork(RIGHT(i));
        eat();
        put_fork(LEFT(i));
        put_fork(RIGHT(i));
    }
}
```



The Dining Philosophers Problem (2 of 5)

- There are 5 philosophers, each seated at a round table in front of a plate of spaghetti. A philosopher needs 2 forks to eat the spaghetti, located on his left and his right.
- Obvious (non)solution:

```
void philosopher(int i) {
    while(TRUE) {
        think();
        take_fork(LEFT(i));
        take_fork(RIGHT(i));
        eat();
        put_fork(LEFT(i));
        put_fork(RIGHT(i));
    }
}
```

All the left forks taken at the same time?
→ *DEADLOCK*

Pearson

The Dining Philosophers Problem (3 of 5)

```
#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 */
     }
}
```

Figure 6-5. A solution to the dining philosophers problem.

The Dining Philosophers Problem (4 of 5)

```
        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 */
}

void put_forks(i)                           /* i: philosopher number, from 0 to N−1 */
```

Figure 6-5. A solution to the dining philosophers problem.

Pearson

The Dining Philosophers Problem (5 of 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 6-5. A solution to the dining philosophers problem.

Resource Deadlocks: Examples

- Dining Philosophers: Everybody starts by taking the left fork
- Four cars arrive simultaneously at a junction and each yields to the car on the right
- Process A opens file #1 and tries to open file #2. File #2 is currently opened by Process B and Process B waits for file #1

Deadlock Definition

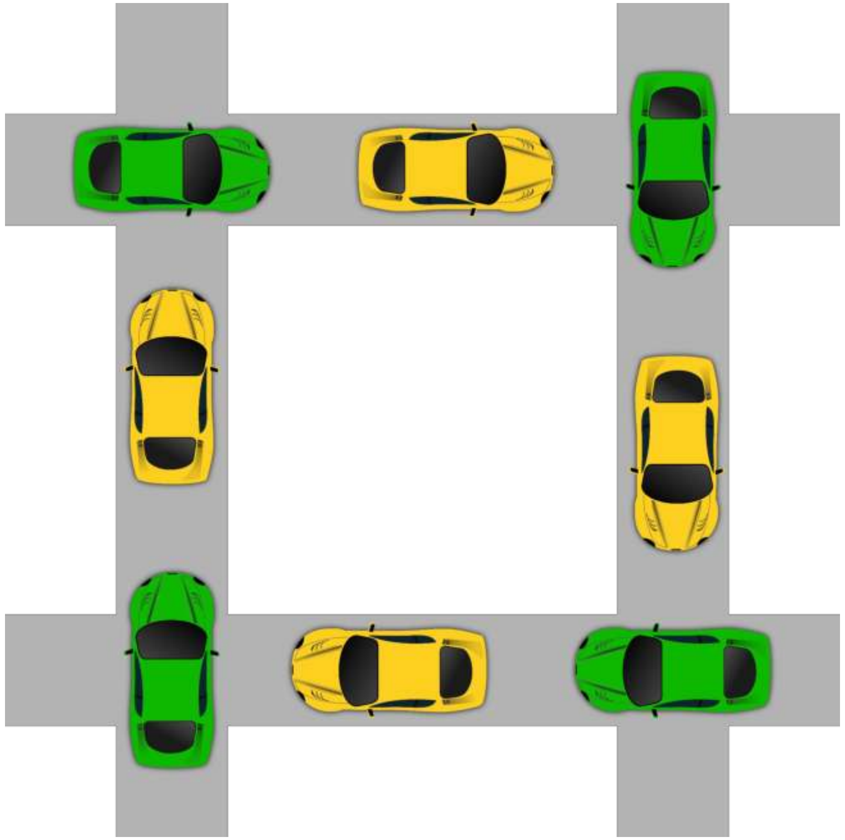
A set of processes is deadlocked if …

- Each process in the set waiting for an event
- That event can be caused only by another process

Other issues:

- Starvation?
- Livelock?

Deadlock



Nothing flows, everything hangs

Communication Deadlock

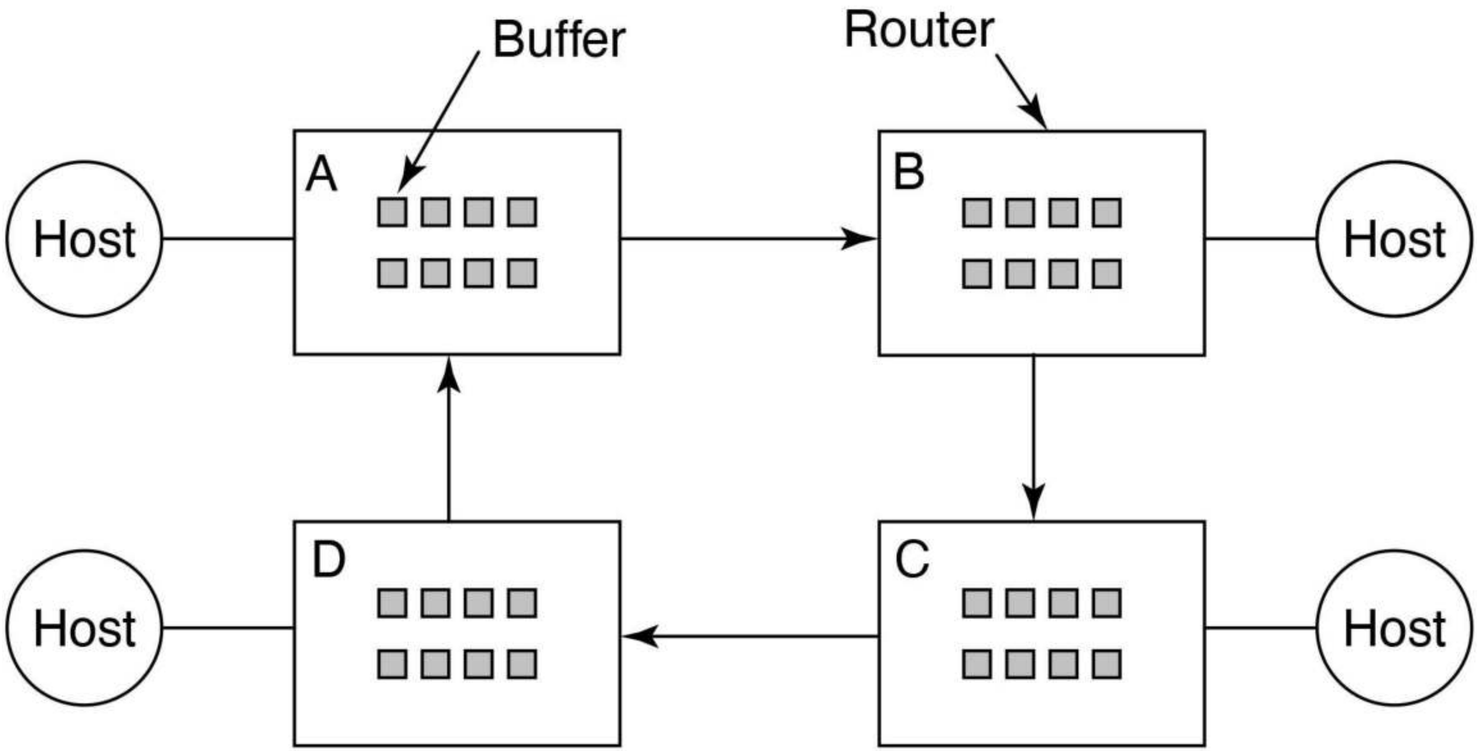
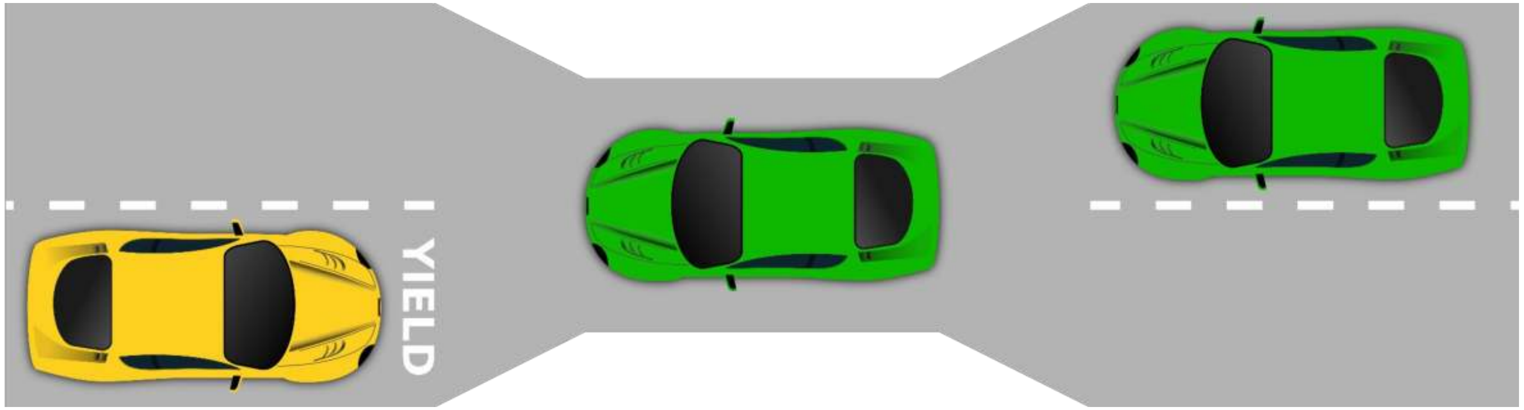


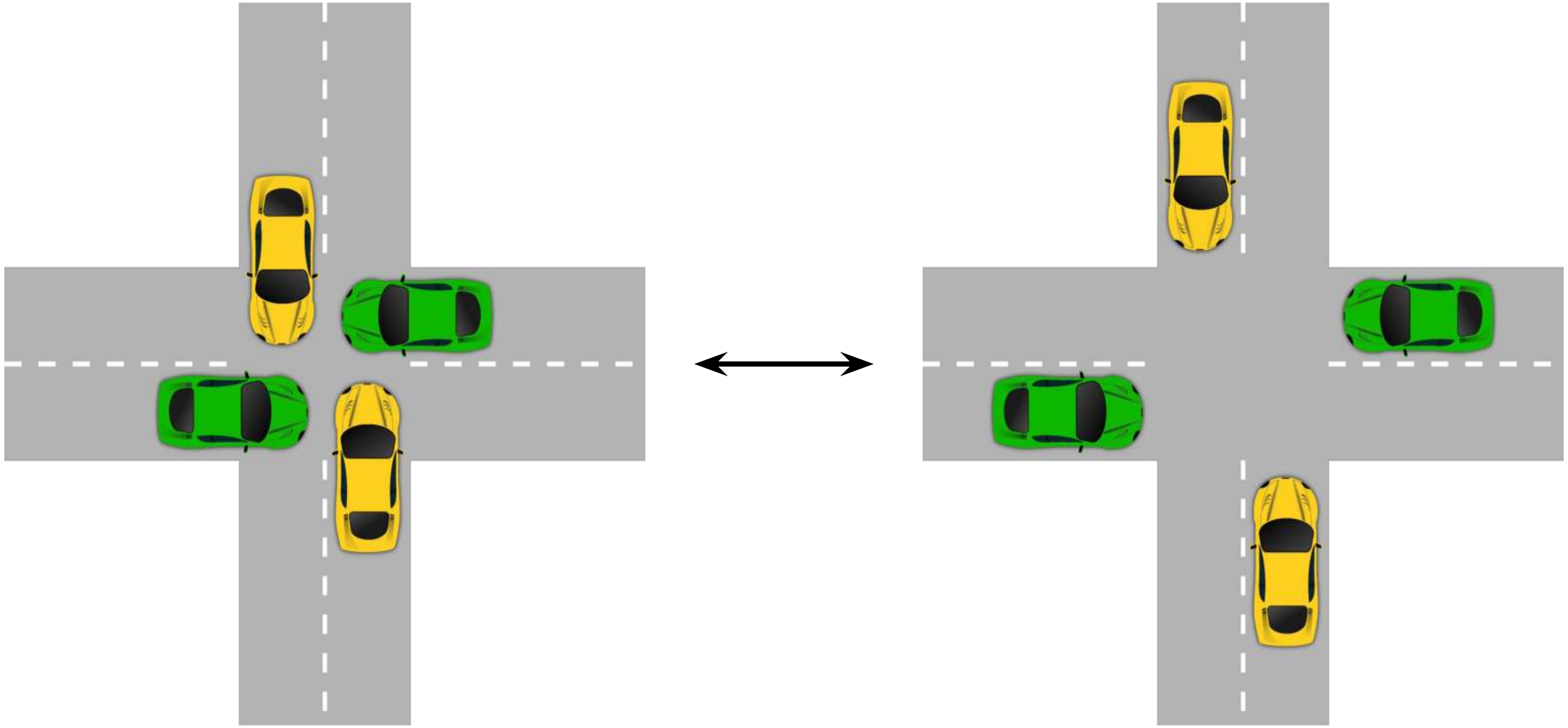
Figure 6-18. A resource deadlock in a network.

Starvation



- **Resource**: The narrow bridge
- **Access**: Traffic from the right has right of way
- **Deadlock resolution**: Cars back up (preemption & rollback)
- **Starvation** is possible

Livelock



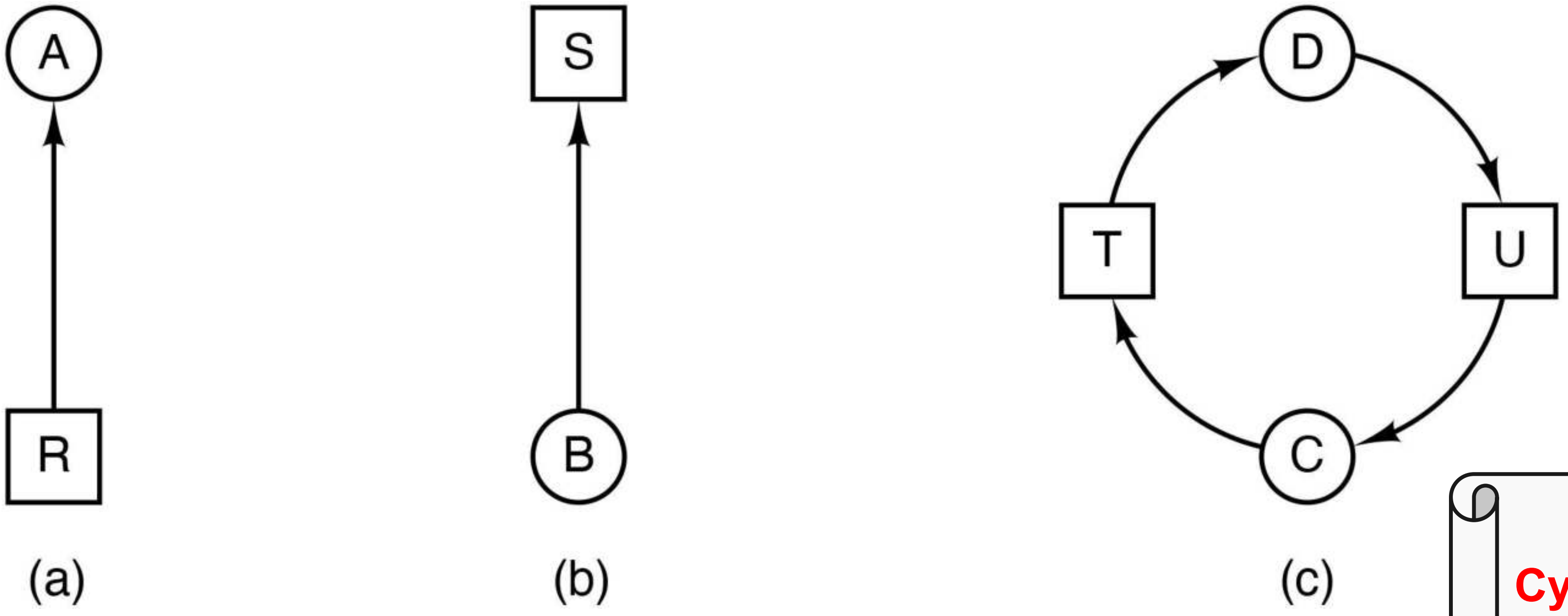
Many operations executed, but no progress

Conditions for Resource Deadlocks

Four conditions that must hold:

1. **Mutual exclusion**
   - Each resource is assigned to at most one process
2. **Hold and wait**
   - Processes can request a resource when holding another one
3. **No preemption**
   - Resources cannot be taken away from a process
4. **Circular wait condition**
   - Chain of two or more processes must be waiting for a resource held by next process in the chain

Deadlock Modeling (1 of 4)



Cycle: C→T→D→U
→ ***DEADLOCK***

Figure 6-6. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Deadlock Modeling (2 of 4)

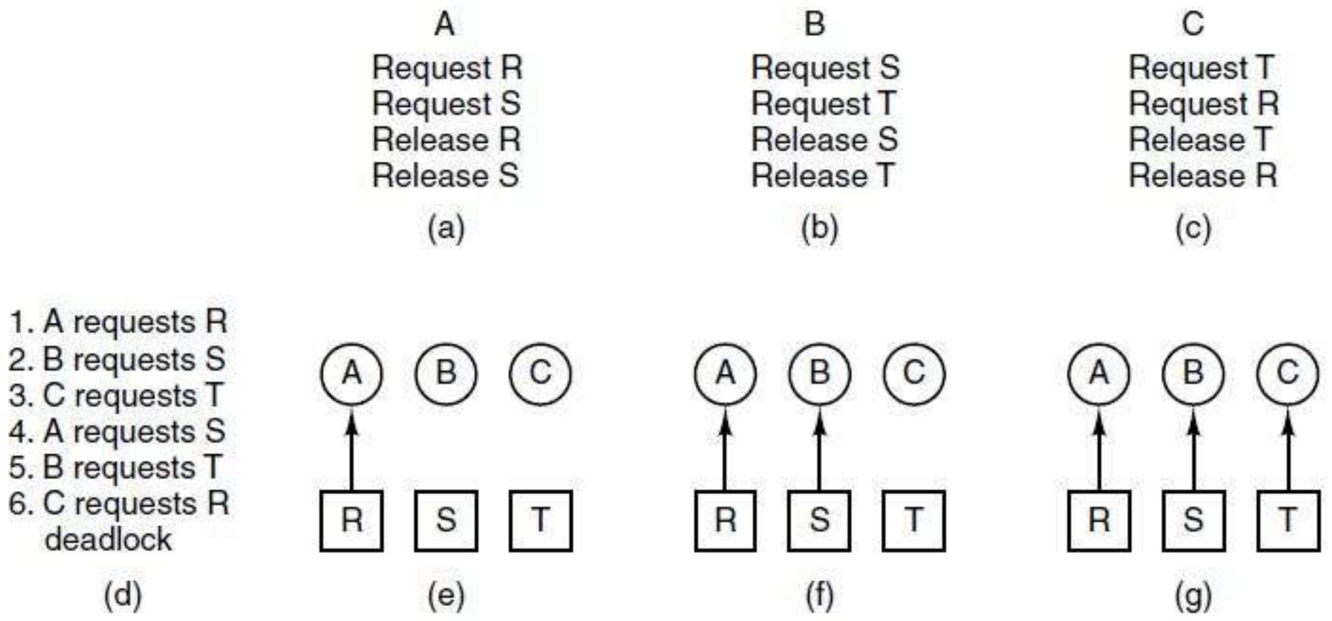


Figure 6-7. An example of how deadlock occurs and how it can be avoided.

Deadlock Modeling (3 of 4)

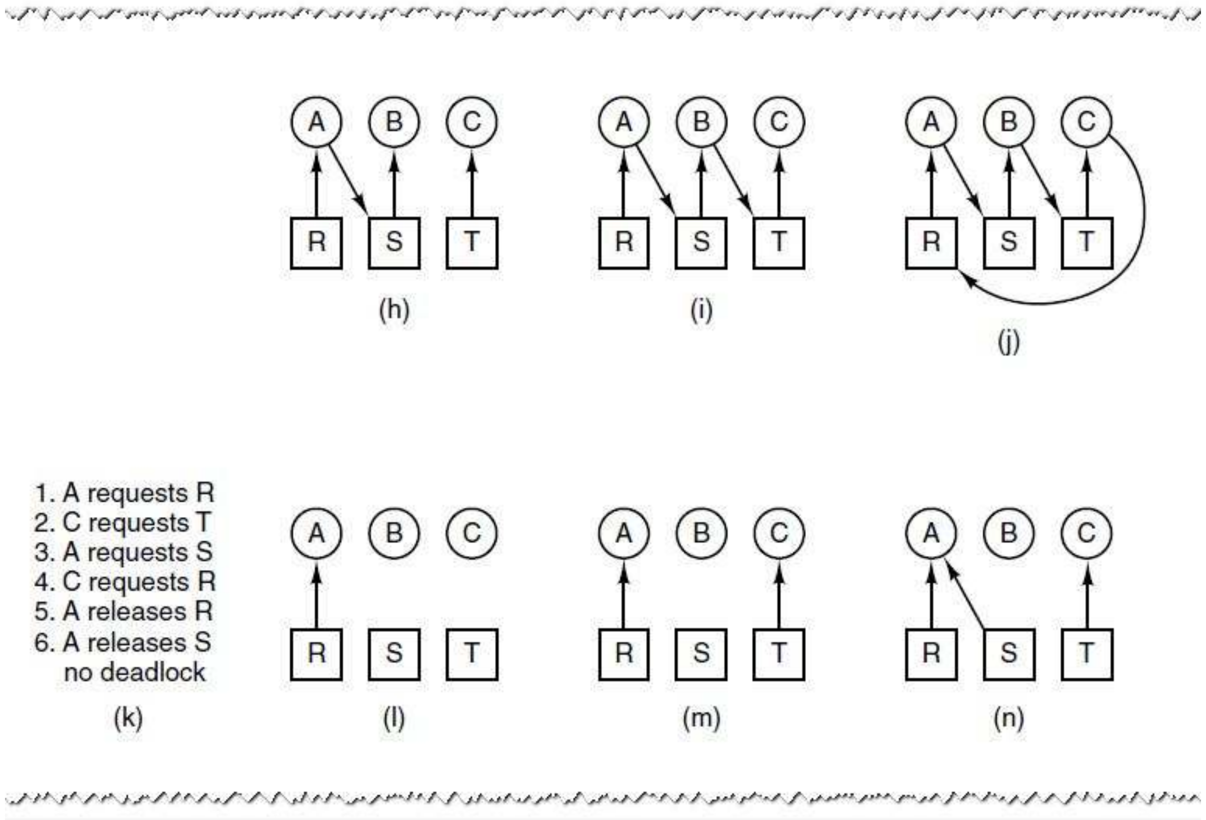


Figure 6-7. An example of how deadlock occurs and how it can be avoided.

Deadlock Modeling (4 of 4)

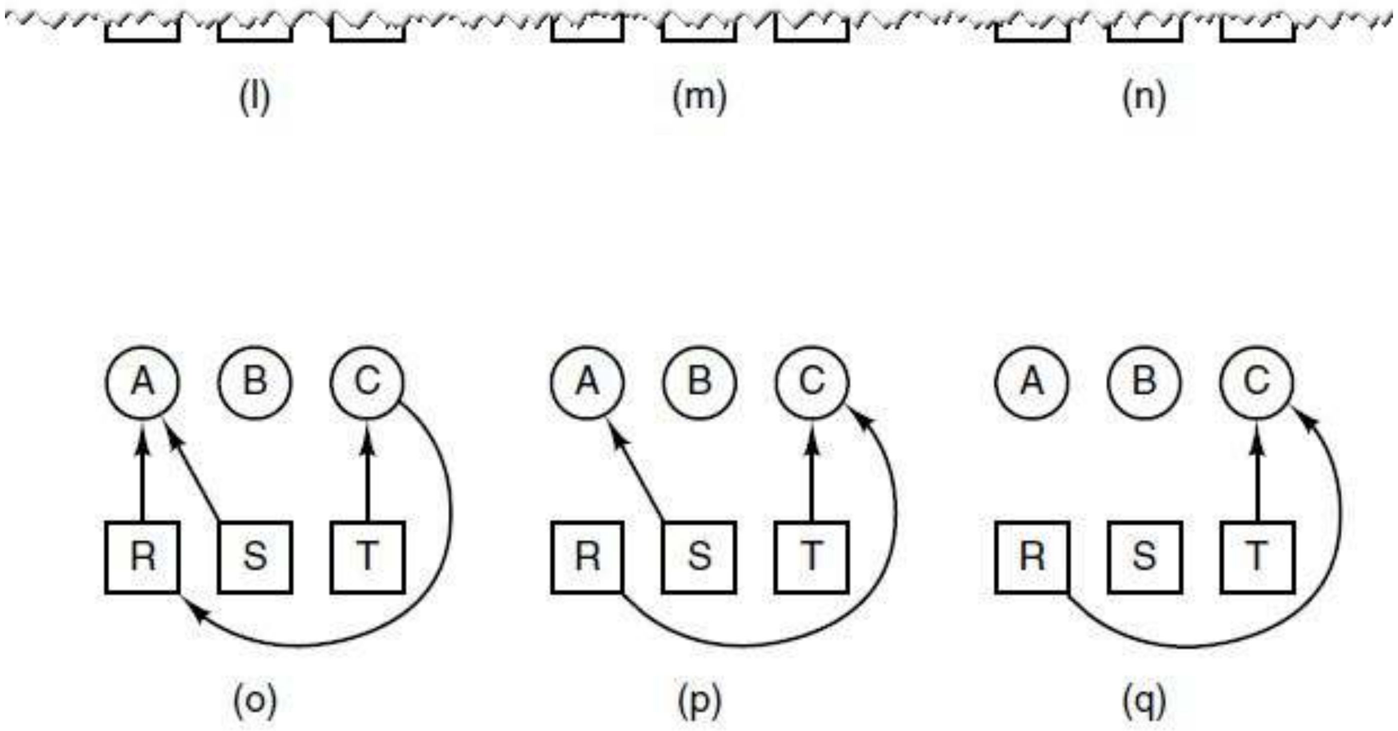


Figure 6-7. An example of how deadlock occurs and how it can be avoided.

Deadlock Handling

Strategies are used for dealing with deadlocks:

1. **Ignore the problem**
   - No action taken
2. **Deadlock detection**
   - Detect deadlock and perform recovery actions
3. **Deadlock avoidance**
   - Carefully allocate resources to avoid deadlocks
4. **Deadlock prevention**
   - Structurally prevent any of the deadlock conditions

Ignore the Problem

- Also known as the *ostrich algorithm*
- Cost-effective solution to deadlocks
- Assumes deadlocks are rare
- Assumes cost of handling deadlocks is high
- Assumes effects of deadlocks are tolerable
- Simplest solution to manage system resources, i.e., process table, inode table, swap space, etc.

Overview

- Deadlock definition and modeling
- **Deadlock detection**
- Deadlock avoidance
- Deadlock prevention
- Deadlock handling in practice

Deadlock Detection

- Detection
  - Check for cycles in the resource allocation graph
  - Track progress and time out
  - Explicit detection (e.g., OOM)

- Useful in practice when simple and efficient detection mechanisms (e.g., progress tracking or explicit detection) along with recovery actions are available

Detecting Cycles with One Resource of Each Type (1 of 2)

Example of a system – is it deadlocked?

1. Process A holds R, wants S
2. Process B holds nothing, wants T
3. Process C holds nothing, wants S
4. Process D holds U, wants S and T
5. Process E holds T, wants V
6. Process F holds W, wants S
7. Process G holds V, wants U

Detecting Cycles with One Resource of Each Type (2 of 2)

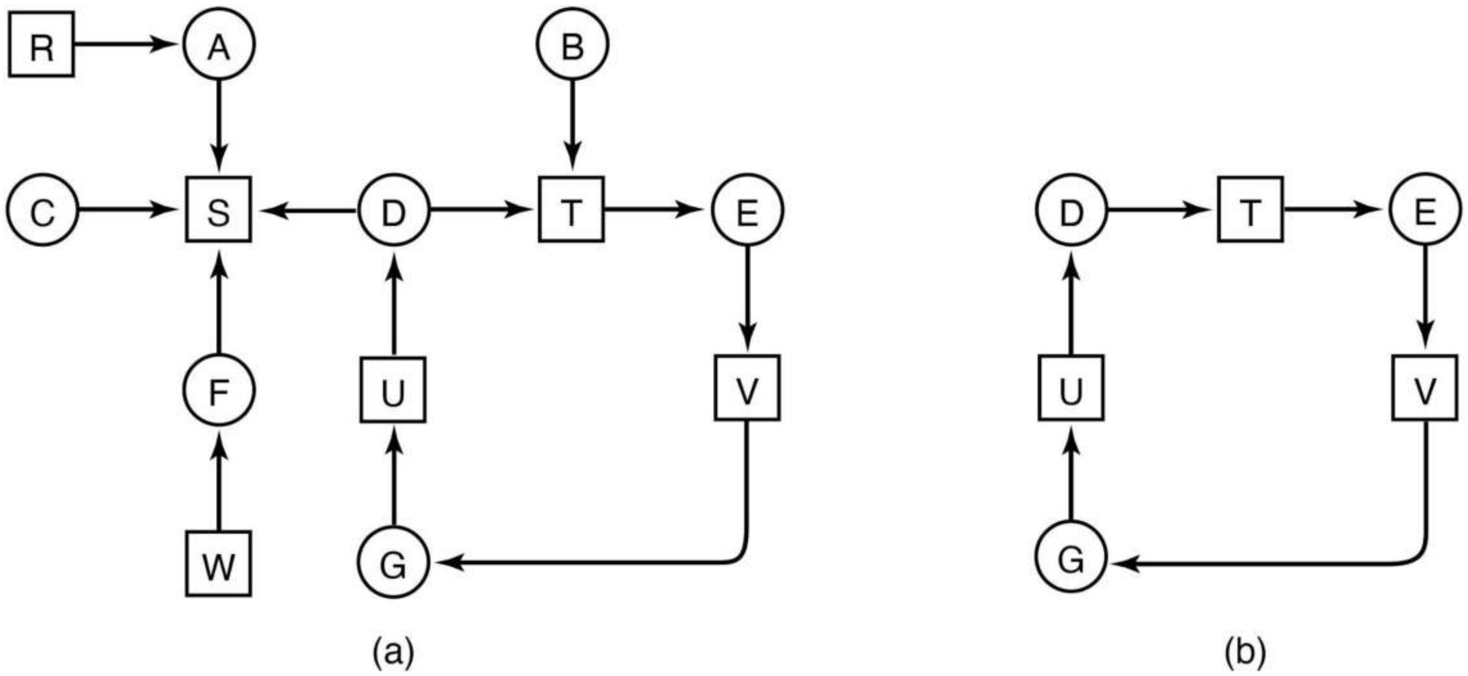


Figure 6-8. (a) A resource graph. (b) A cycle extracted from (a).

Algorithm to Detect Cycles (1 of 2)

1. For each node, **N** in the graph, perform following five steps with **N** as starting node.
2. Initialize **L** to empty list, and designate all arcs as unmarked.
3. Add current node to end of L, check to see if node now appears in L two times. If so, graph contains a cycle (listed in L) and algorithm terminates

Algorithm to Detect Cycles (2 of 2)

4. From given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick unmarked outgoing arc at random, mark it. Then follow to new current node and go to step 3.
6. If this is initial node, graph does not contain cycles, algorithm terminates. Otherwise, dead end. Remove it and go back to the previous node.

Detecting Cycles with Multiple Resources of Each Type (1 of 3)

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Figure 6-9. The four data structures needed by the deadlock detection algorithm.

Detecting Cycles with Multiple Resources of Each Type (2 of 3)

Deadlock detection algorithm:

1. Look for unmarked process, Pi, for which the i-th row of R is less than or equal to A.
2. If such a process is found, add the i-th row of C to A, mark the process, go back to step 1.
3. If no such process exists, the algorithm terminates.

Detecting Cycles with Multiple Resources of Each Type (3 of 3)

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{Cameras} \\ (4 & 2 & 3 & 1) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{Cameras} \\ (2 & 1 & 0 & 0) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Figure 6-10. An example for the deadlock detection algorithm.

Recovery from Deadlock

Possible Methods of recovery (though none are “attractive”):

1. Force preemption
2. Checkpoint-rollback
3. Killing the offending processes

Quiz

Given a deadlock-prone multithreaded program, you implement deadlock detection and recovery, where recovery rolls back a random thread by N instructions.

Will this fix your problem?

Livelock

```
void process_A(void) {
        acquire_lock(&resource_1);
        while (try_lock(&resource_2) == FAIL) {
              release_lock(&resource_1);
              wait_fixed_time();
              acquire_lock(&resource_1);
        }
        use_both_resources( );
        release_lock(&resource_2);
        release_lock(&resource_1);
}

void process_B(void) {
        acquire_lock(&resource_2);
        while (try_lock(&resource_1) == FAIL) {
              release_lock(&resource_2);
              wait_fixed_time();
              acquire_lock(&resource_2);
        }
        use_both_resources( );
        release_lock(&resource_1);
        release_lock(&resource_2);
}
```

Figure 6-19. Busy waiting that can lead to livelock.

Overview

- Deadlock definition and modeling
- Deadlock detection
- **Deadlock avoidance**
- Deadlock prevention
- Deadlock handling in practice

Deadlock Avoidance: 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)

- **Banker's algorithm (Dijkstra):**
  - Customers (**processes**) request credits (**resources**)
  - Banker (**OS**) only satisfies requests resulting in **safe** states
  - A state is safe iff there exists a sequence of other states that allows all the customers to complete
  - Maximum credit demands are known in advance

Deadlock Avoidance: 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)

- **Banker's algorithm (Dijkstra):**
  - Customers (**processes**) request credits (**resources**)
  - Banker (**OS**) only satisfies requests resulting i
  - A state is safe iff there exists a sequence of o allows all the customers to complete
  - Maximum credit demands are known in advar e

(a): Safe state
→ Anybody can obtain new credits and complete

Deadlock Avoidance: 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)

- **Banker's algorithm (Dijkstra):**
  - Customers (**processes**) request credits (**resources**)
  - Banker (**OS**) only satisfies requests resulting i
  - A state is safe iff there exists a sequence of o allows all the customers to complete
  - Maximum credit demands are known in advar e

(b): Safe state
→ C (then others) can obtain credits and complete

Deadlock Avoidance: 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)

- **Banker’s algorithm (Dijkstra):**
  - Customers (**processes**) request credits (**resources**)
  - Banker (**OS**) only satisfies requests resulting i
  - A state is safe iff there exists a sequence of o allows all the customers to complete
  - Maximum credit demands are known in advar e

(c): Unsafe state
→ Nobody can obtain new credits and complete

Deadlock Avoidance Resource Trajectories

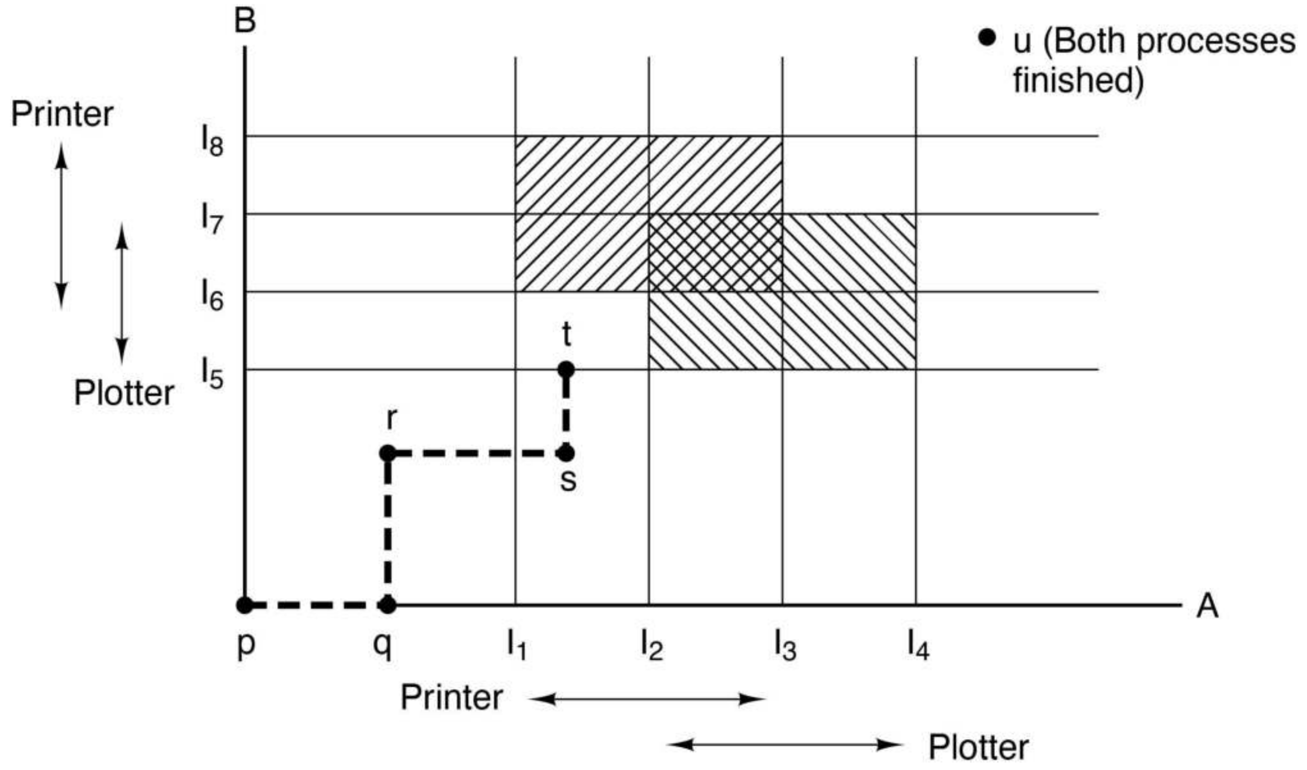


Figure 6-11. Two process resource trajectories.

Safe and Unsafe States (1 of 2)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

Figure 6-12. Demonstration that the state in (a) is safe.

Safe and Unsafe States (2 of 2)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Figure 6-13. Demonstration that the state in (b) is not safe.

Banker's Algorithm for Multiple Resources (1 of 2)

Process	Tape drives	Plotters	Printers	Cameras
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	Cameras
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 assigned

E= (6342) [*Existing*]
P= (5322) [*Possessed*]
A= (1020) [*Available*]
N= (6432) [*Needed*]

Figure 6-15. The banker's algorithm with multiple resources.

Banker's Algorithm for Multiple Resources (2 of 2)

Resources assigned

Process	Tape drives	Plotters	Printers	Cameras
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 still assigned

Process	Tape drives	Plotters	Printers	Cameras
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

E= (6342) [*Existing*]
P= (5322) [*Possessed*]
A= (1020) [*Available*]
N= (6432) [*Needed*]

Generalized safe state detection:

1. Select row R whose unmet resource needs N are all <= A.
2. Mark R as terminated and add its resources to the A vector.
3. Repeat until completion (**safe**) or deadlock (**unsafe**).

Overview

- Deadlock definition and modeling
- Deadlock detection
- Deadlock avoidance
- **Deadlock prevention**
- Deadlock handling in practice

Deadlock Prevention

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

Figure 6-17. Summary of approaches to deadlock prevention (negating any of the four deadlock conditions)

Deadlock Prevention

1. **Mutual exclusion**
   - Spool everything
     - Typically shifts the problem somewhere else
2. **Hold and wait**
   - Request all resources initially (or reacquire them)
     - Poor parallelism and resource utilization
3. **No preemption**
   - Take resources away
     - N/A in many cases (e.g., printer vs memory)
4. **Circular wait**
   - Order resources numerically
     - Hard to consistently enforce in practice

Attacking Circular Wait Condition (1 of 2)

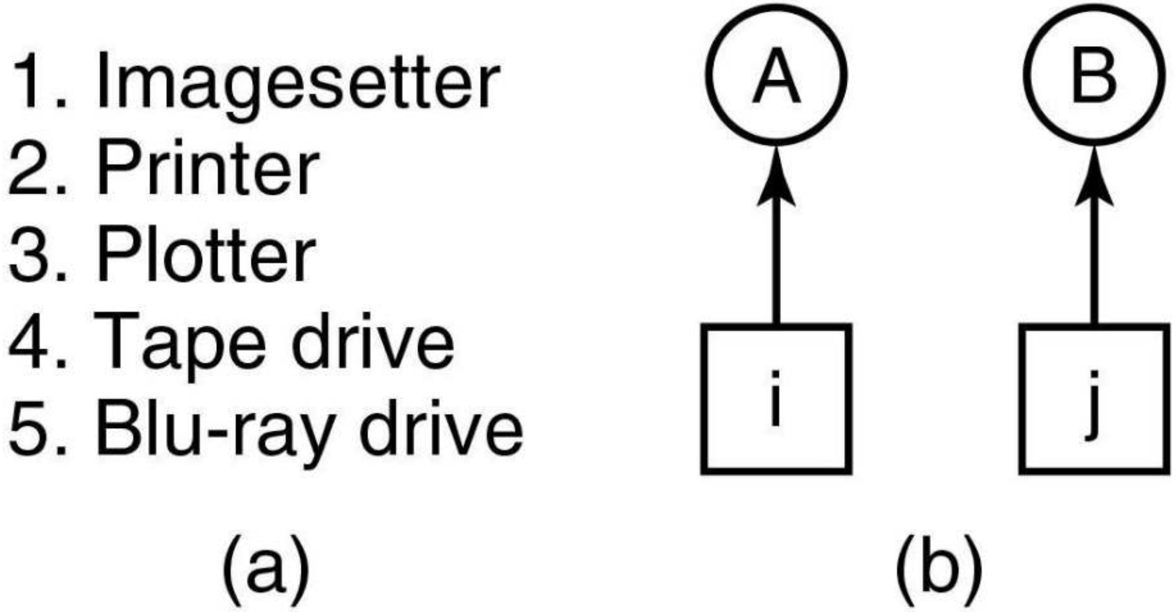


Figure 6-16. (a) Numerically ordered resources. (b) A resource graph

Attacking Circular Wait Condition (2 of 2)

Figure 6-17. Summary of approaches to deadlock prevention.

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

Overview

- Deadlock definition and modeling
- Deadlock detection
- Deadlock avoidance
- Deadlock prevention
- **Deadlock handling in practice**

Dealing with Deadlocks in the Real World

- **Deadlock avoidance**
  - Rarely an option (hard to know resource needs a priori)
- **Deadlock prevention**
  - Adopted in particular domains (e.g., lock ordering or two-phase locking in transaction-processing systems)
- **Ignore the problem**
  - Last resort when nothing else available
- **Deadlock detection**
  - Solution of choice when adequate detection (and recovery) mechanisms are available

Deadlock (Circular Wait) Prevention on Linux

From `linux/mm/rmap.c`:

```
/*
 * Lock ordering in mm:
 *
 * inode->i_rwsem  (while writing or truncating, not reading or faulting)
 *   mm->mmap_lock
 *     mapping->invalidate_lock (in filemap_fault)
 *       page->flags PG_locked (lock_page)   * (see hugetlbfs below)
 *         hugetlbfs_i_mmap_rwsem_key (in huge_pmd_share)
 *           mapping->i_mmap_rwsem
 *             hugetlb_fault_mutex (hugetlbfs specific page fault mutex)
 ...
 */
```

Deadlock Detection on Linux

- **Global OOM killer**
  - Resource: memory
  - Mechanism: explicit detection
- **Soft lockup detection**
  - Resource: locks
  - Mechanism: progress tracking
- **Locking validator**
  - Resource: locks
  - Mechanism: cycles in the resource allocation graph

Global OOM Killer on Linux

```
INFO: memcached invoked oom-killer
CPU: 1 PID: 2859
Call Trace:
 [<c10e1c15>] dump_header.isra.7+0x85/0xc0
 [<c10e1e6c>] oom_kill_process+0x5c/0x80
 [<c10e225f>] out_of_memory+0xbf/0x1d0
 [<c10fec2c>] handle_pte_fault+0xec/0x220
 [<c10fee68>] handle_mm_fault+0x108/0x210
 [<c152fb5b>] do_page_fault+0x15b/0x4a0
 [<c152cfcf>] error_code+0x67/0x6c
Out of memory: Kill process 2603 score 761 or sacrifice child
Killed: process 2603 vm:1498MB, anon-rss:721MB, file-rss:4MB
```

Soft Lockup Detection on Linux

```
BUG: soft lockup - CPU#1 stuck for 23s!
CPU: 1 PID: 954
RIP: 0010:[<ffffffff8104de62>]  __ticket_spin_lock+0x22/0x30
Call Trace:
 [<ffffffff816ee5fe>] _raw_spin_lock+0xe/0x20
 [<ffffffff811671c7>] handle_pte_fault+0x827/0xab0
 [<ffffffff811681f9>] handle_mm_fault+0x299/0x670
 [<ffffffff816f2a6c>] do_page_fault+0x2c/0x50
 [<ffffffff8120950d>] proc_reg_read+0x3d/0x80
 [<ffffffff811a6f9e>] vfs_read+0x9e/0x170
 [<ffffffff811a7ac9>] SyS_read+0x49/0xa0
 [<ffffffff816f725d>] system_call_fastpath+0x1a/0x1f
```

Pearson

Locking Validator (lockdep) on Linux

```
[ INFO: possible circular locking dependency detected ]
sshd/2280 is trying to acquire lock: (cpu_add_remove_lock)
    but task is already holding lock: (console_lock)
    which lock already depends on the new lock.
Chain exists of:
    cpu_add_remove_lock --> cpu_hotplug.lock --> console_lock
Possible unsafe locking scenario:
    lock(console_lock);
    lock(cpu_hotplug.lock);
    lock(console_lock);
    lock(cpu_add_remove_lock);
    *** DEADLOCK ***
```

Quiz

In a real-world system with some deadlock prevention mechanisms, does it make sense to consider deadlock avoidance and/or detection mechanisms?

Copyright