MODERN OPERATING SYSTEMS

Third Edition
ANDREW S. TANENBAUM

Chapter 6
Deadlocks

Preemptable and Nonpreemptable Resources

- Non-sharable resource: the resource can be used by only one process at a time
- A process may use a resource in only the following sequence:

1. Request: If the resource cannot be granted, the requesting process must wait until it can acquire the resource.
2. Use: The process can operate on the resource.
3. Release: The process releases the resource.

Resource Acquisition (1)

```
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)

```
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)

Figure 6-2. (a) Deadlock-free code.

Resource Acquisition (3)

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

```
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)

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.

Conditions for Resource Deadlocks

Necessary conditions for a deadlock to occur

1. Mutual exclusion: The resource is non-sharable.
2. Hold and wait: A process that is holding resources can request new resources.
3. No preemption: A resource can be released only by the process holding it.
4. Circular wait: There is a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.
5. All four conditions must simultaneously hold.

Deadlock Modeling (1)

Resource graph: a directed graph with two types of nodes:
Processes (circles) and resources (squares)

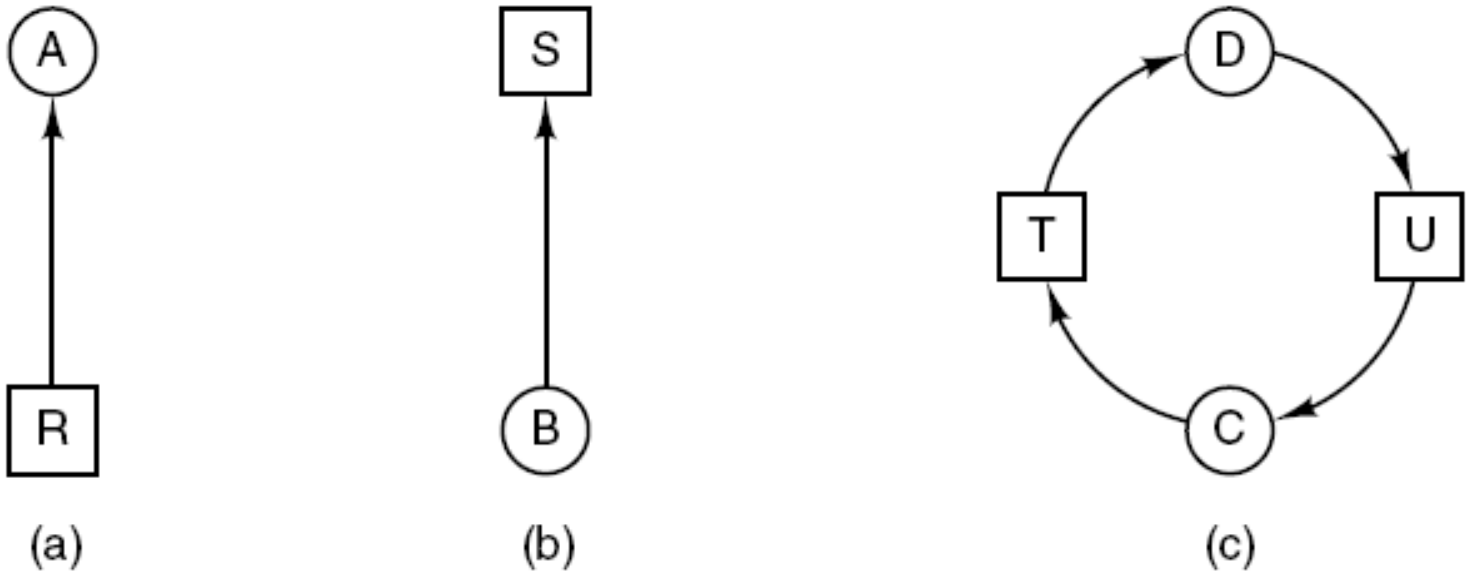


Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Deadlock Modeling

Use resource graph to detect deadlocks

An example:

_ Three processes A, B, and C

_ Three resources R, S and T

_ Round robin scheduling

Using resource graph, we can see if a given request/release sequence leads to deadlock:

Carry out the request and release step by step, check if there is any circle after each step.

Deadlock Modeling (2)

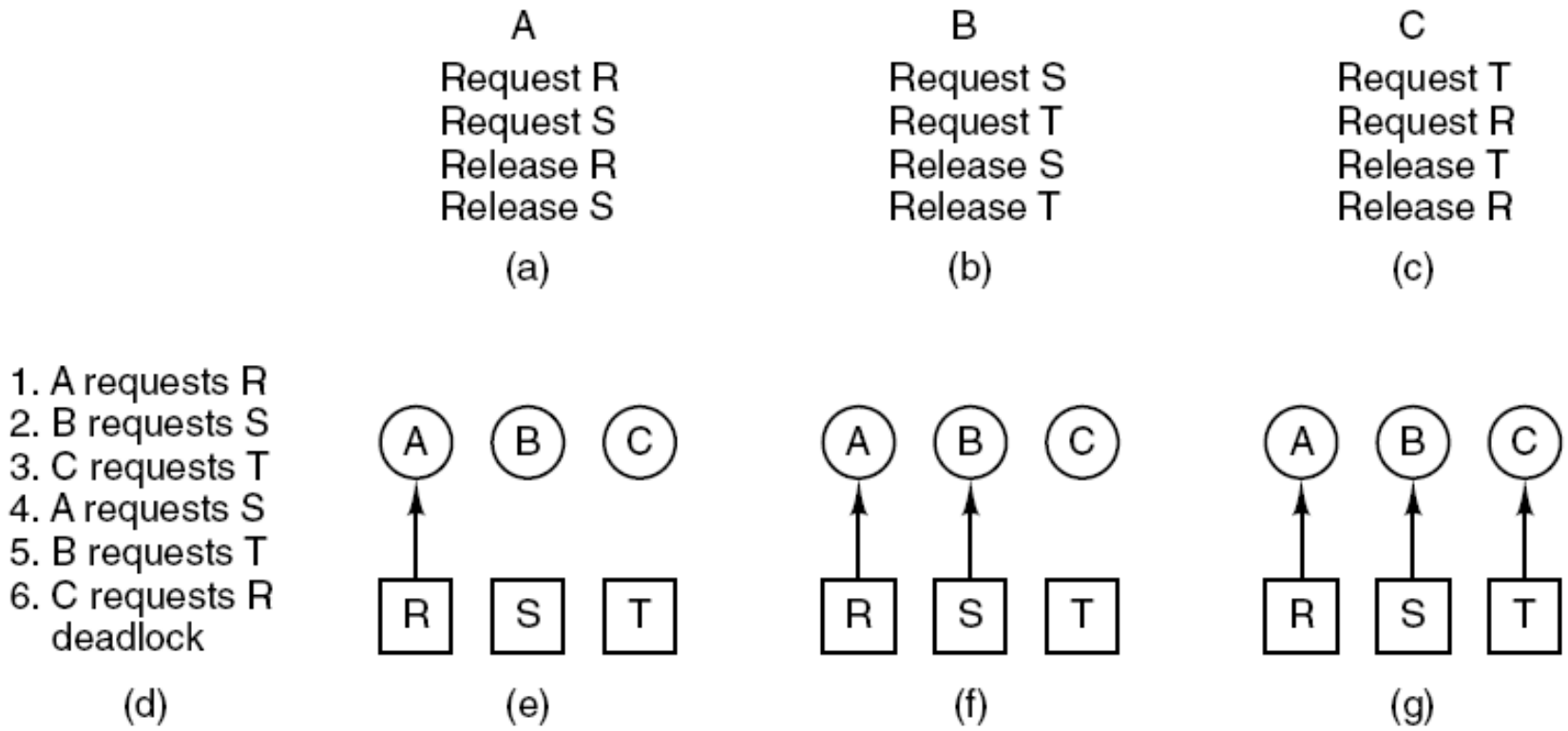


Figure 6-4. An example of how deadlock occurs and how it can be avoided.

Deadlock Modeling (3)

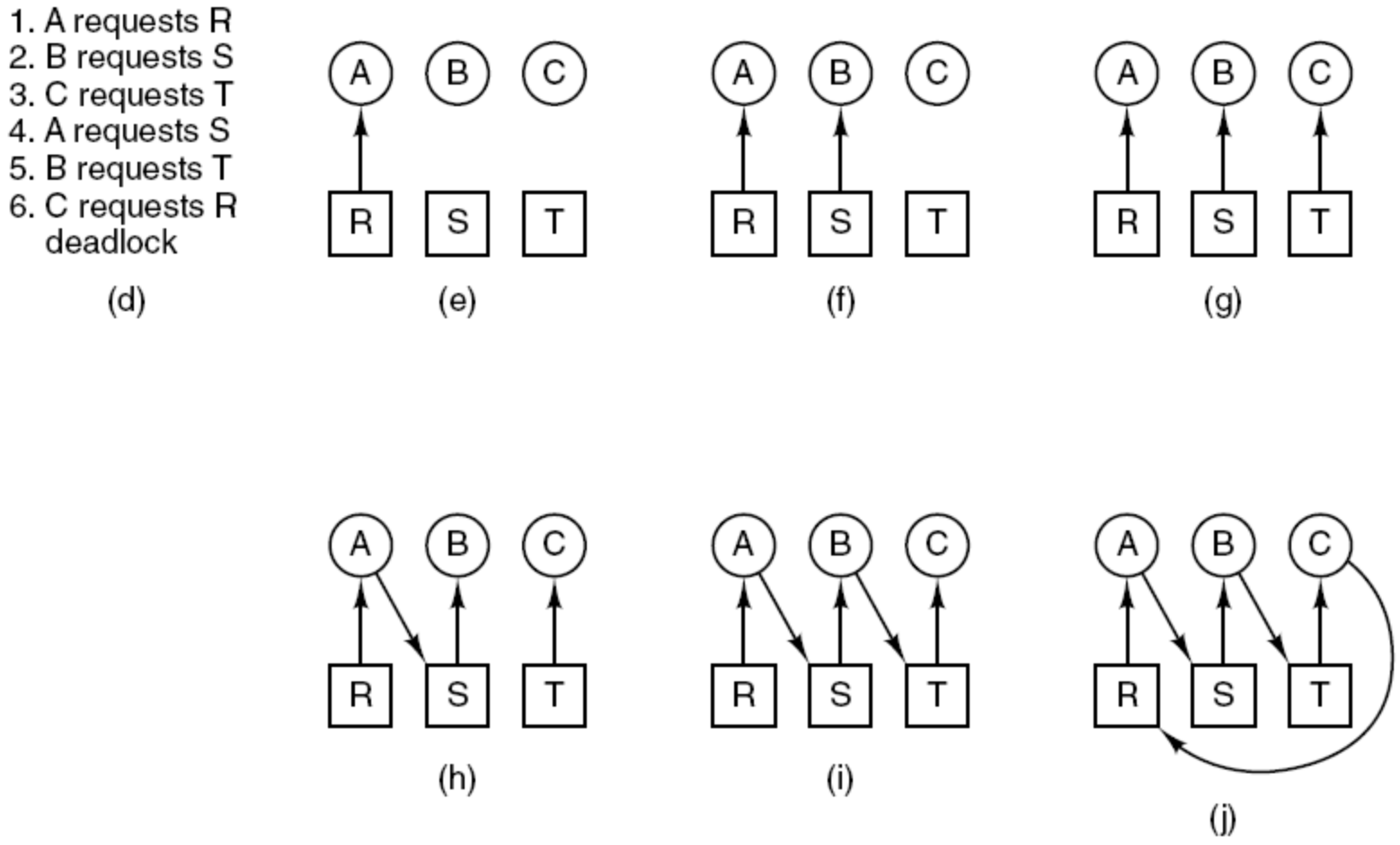


Figure 6-4. An example of how deadlock occurs and how it can be avoided.

Deadlock Modeling (4)

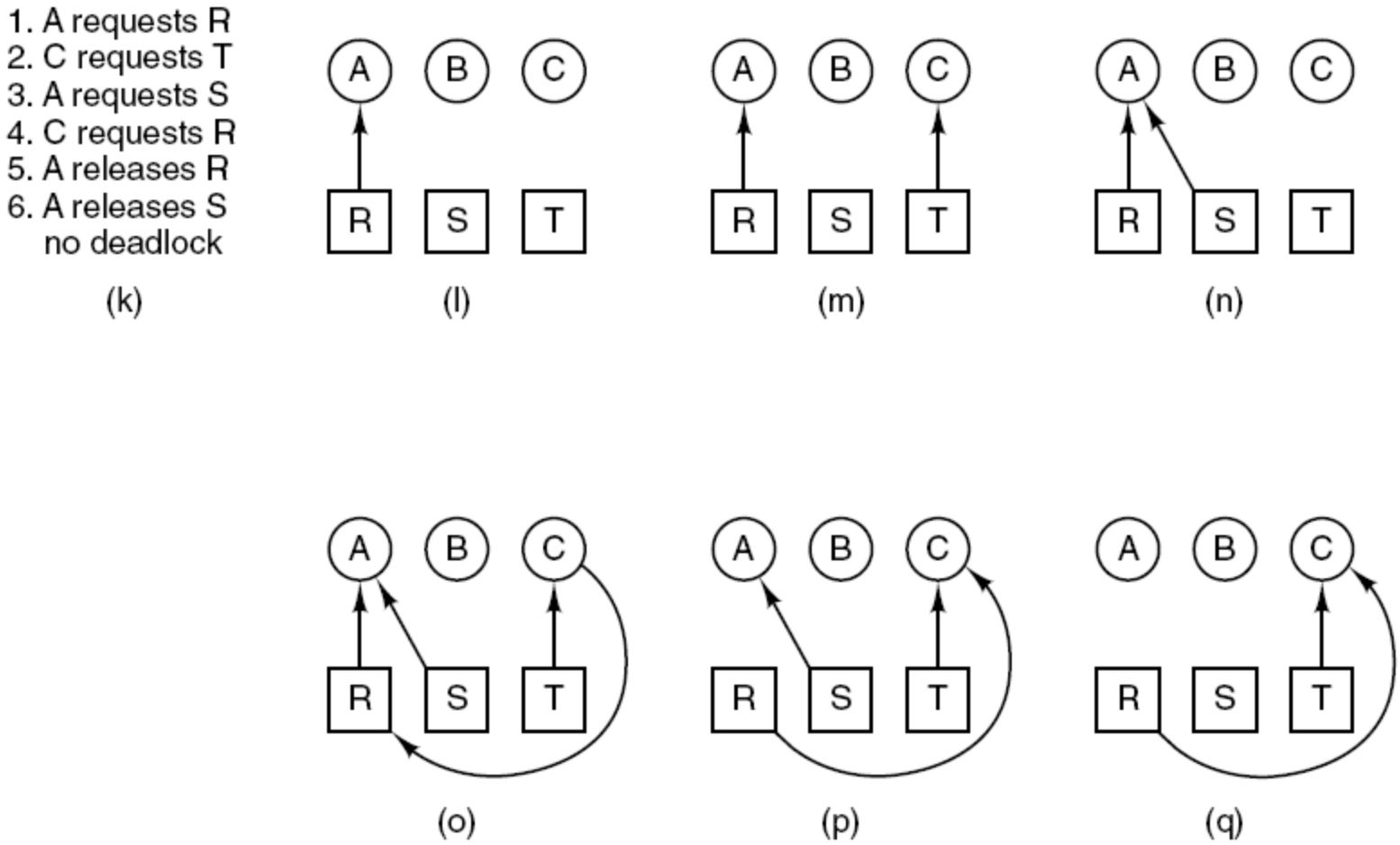


Figure 6-4. An example of how deadlock occurs and how it can be avoided.

Deadlock Modeling (5)

Strategies for dealing with deadlocks:

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

Ignoring Deadlocks

The Ostrich algorithm:

- Stick your head in the sand and pretend that deadlocks never occur.
- Used by most operating systems, including UNIX.
- Tradeoff between convenience and correctness

An Example in Unix

An example of deadlock in UNIX:

- Process table has 100 slots
- 10 processes are running
- Each process needs to fork 12 subprocesses
- After each forks 9 subprocesses, the table is full
- Each original process sits in the endless loop: fork and fail

Deadlock Detection

- In a system where a deadlock may occur, the system must provide:
- An algorithm than exams the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Detection

- Every time a resource is requested or released, check resource graph to see if any cycles exist.
- How to detect cycles in a directed graph?
- Depth-first search from each node. See if any repeated node. O(N) algorithm.

Deadlock Detection with One Resource of Each Type (1)

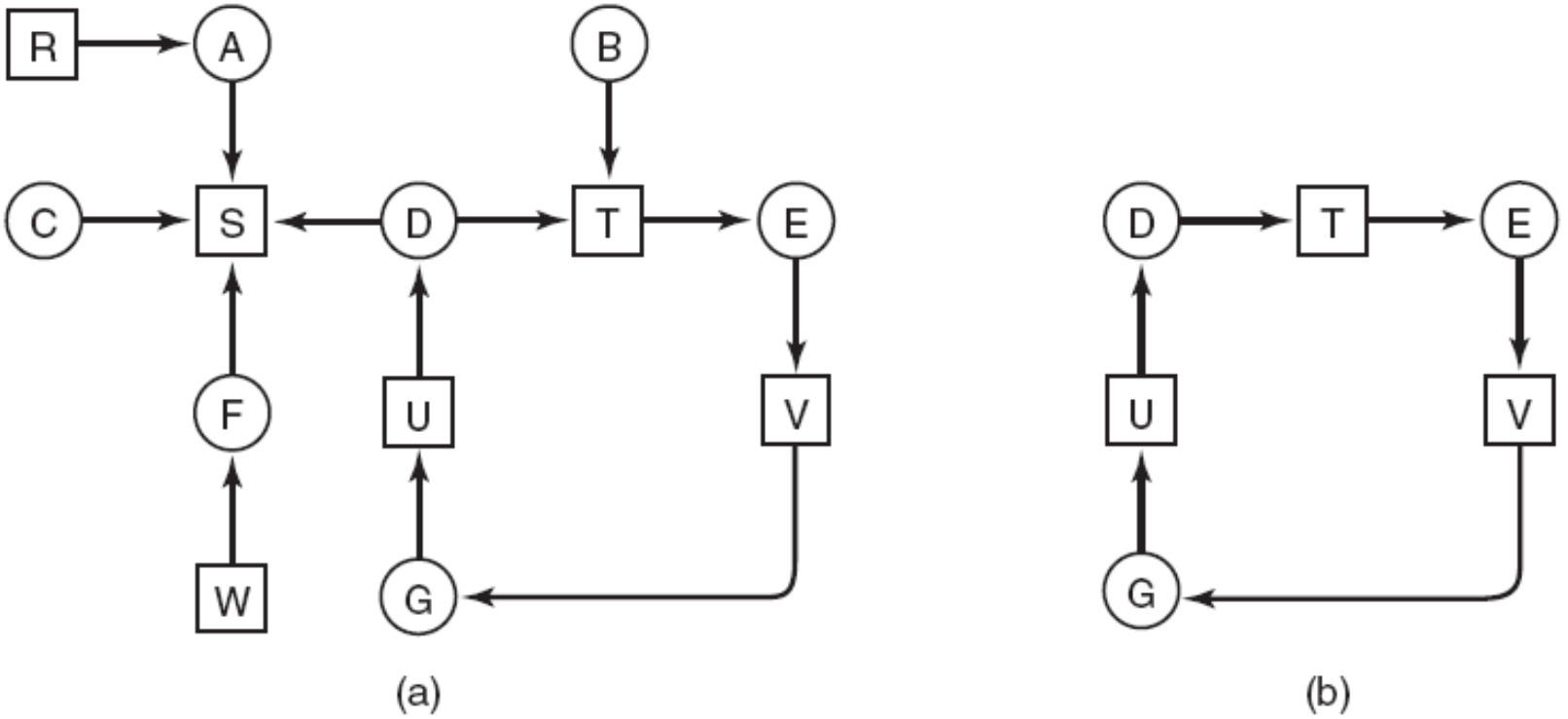


Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

Deadlock Detection with One Resource of Each Type (2)

Algorithm for detecting deadlock:

1. For each node, N in the graph, perform the following five steps with N as the starting node.
2. Initialize L to the empty list, 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 it does, graph contains a cycle (listed in L), algorithm terminates.

   …

Deadlock Detection with One Resource of Each Type (3)

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

Deadlock Detection with Multiple Resources of Each Type (1)

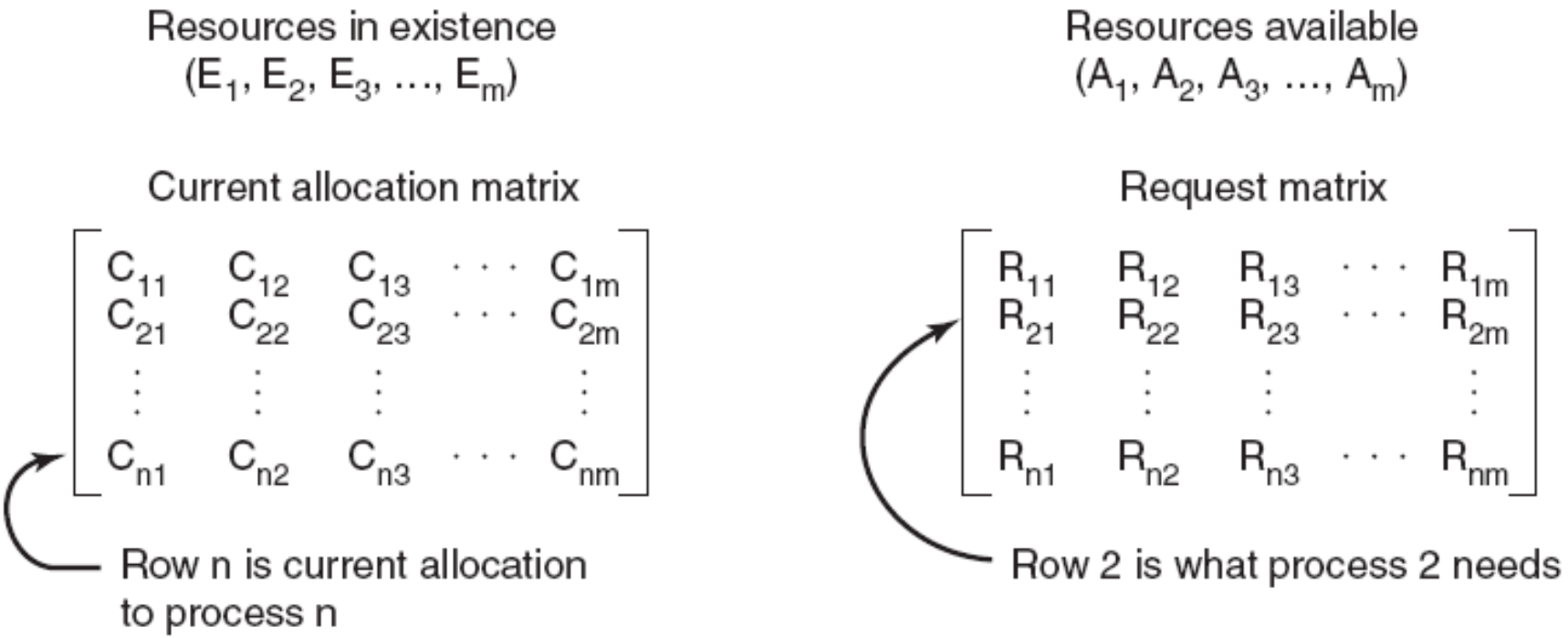


Figure 6-6. The four data structures needed by the deadlock detection algorithm.

Deadlock Detection with Multiple Resources of Each Type (2)

Deadlock detection algorithm:

1. Look for an unmarked process, P_i , 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, and go back to step 1.
3. If no such process exists, the algorithm terminates.

Deadlock Detection with Multiple Resources of Each Type (3)

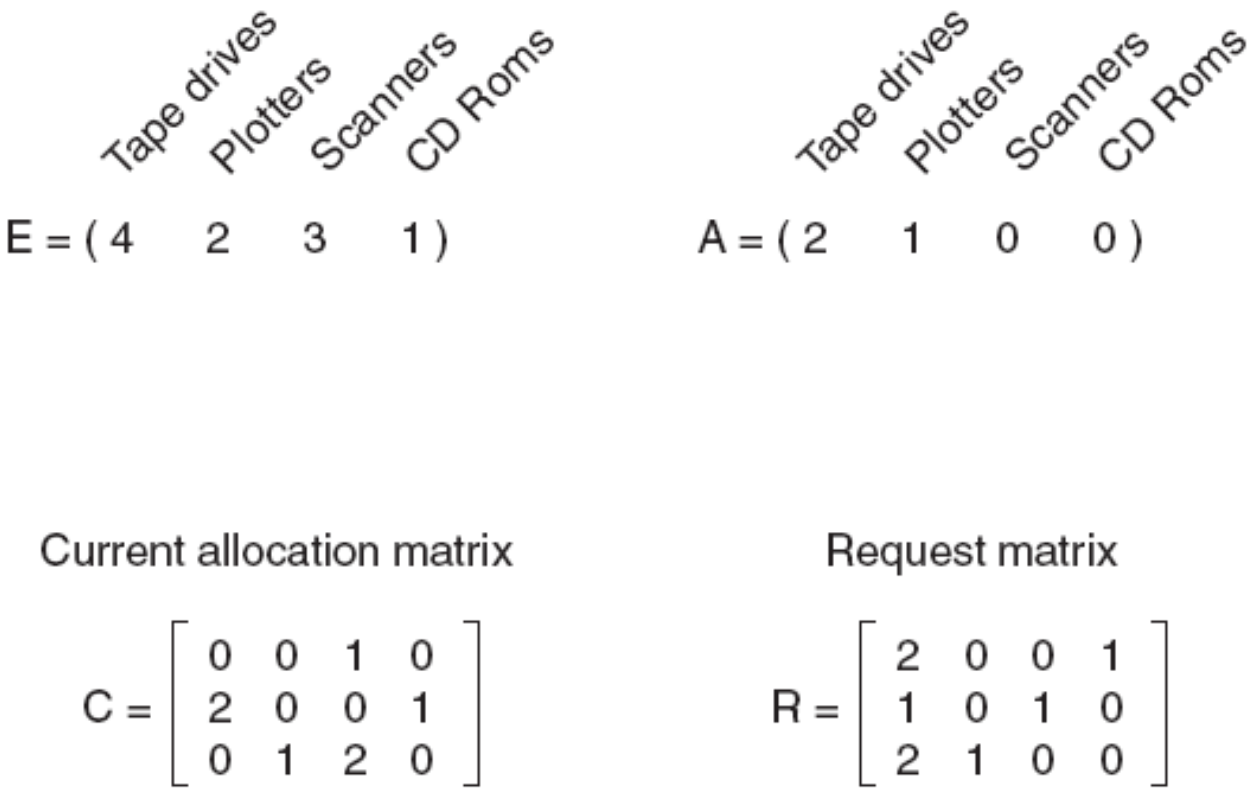


Figure 6-7. An example for the deadlock detection algorithm.

Recovery from Deadlock

Recovery

- Abort one process at a time until the deadlock cycle is eliminated.
- A simpler way (used in large main frame computers):
- Do not maintain a resource graph. Only periodically check to see if there are any processes that have been blocked for a certain amount of time, say, 1 hour. Then kill such processes.
- To recover the killed processes, need to restore any modified files. Keep different versions of the file.

Recovery from Deadlock

- Recovery through preemption
- Recovery through rollback
- Recovery through killing processes

Deadlock Avoidance

- Analyzing each resource request to see if it can be safely granted.
- Resource trajectories: A model for two processes and two resources
- An example:
- Process A and B
- Resources: printer and plotter
- A needs printer from I1 to I3
- A needs plotter from I2 to I4
- B needs plotter from I5 to I7
- B needs printer from I6 to I8
- Each point in the diagram is a joint state of A & B
- Can only go vertical or horizontal (one CPU)
- Start at point p, run A to point q, run B to point r, run A to point s, granted printer, run B to point t, request plotter, can only run A to completion.

Deadlock Avoidance

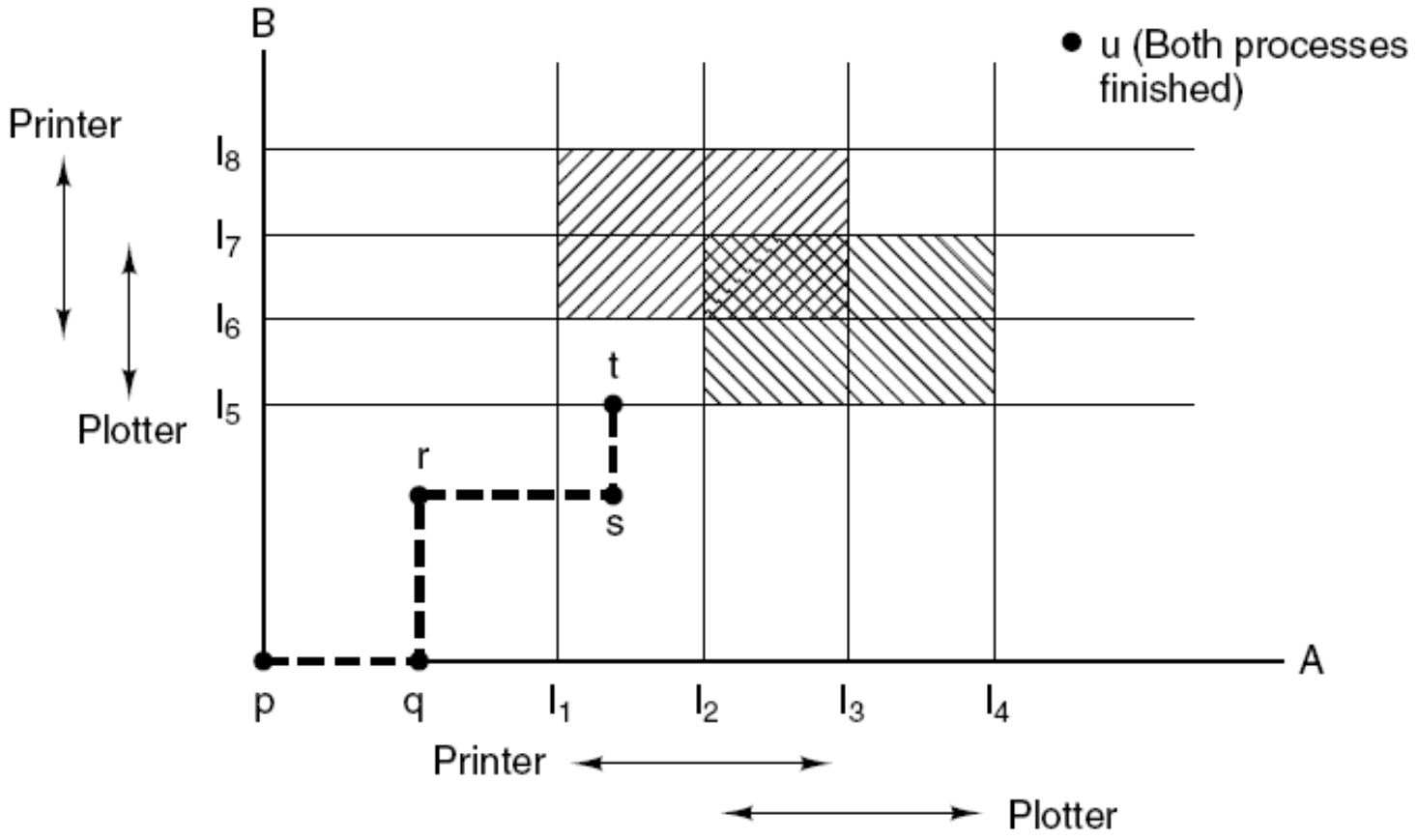


Figure 6-8. Two process resource trajectories.

Deadlock Avoidance

- Find a general algorithm that can always avoid deadlock by making right decisions.
- Banker's algorithm for a single resource:
- A small town banker deals with a group of customers with granted credit lines.
- The analogy:
- Customers: processes
- Units: copies of the resource
- Banker: O.S.
- State of the system: showing the money loaned and the maximum credit available

Deadlock Avoidance

- Safe state:
- There exists a sequence of other states that lead to all customers getting loans up to their credit lines.

- Algorithm:
- For each request, see if granting it leads to a safe state. If it does, the request is granted. Otherwise, it is postponed until later.
- Check a safe state:
- (1) See if available resources can satisfy the customer closest to his maximum. If so, these loans are assumed to be repaid.
- (2) Then check the customer now closet to his maximum, and so on.
- (3) If all loans can be eventually paid, the current state is safe.

Safe and Unsafe States (1)

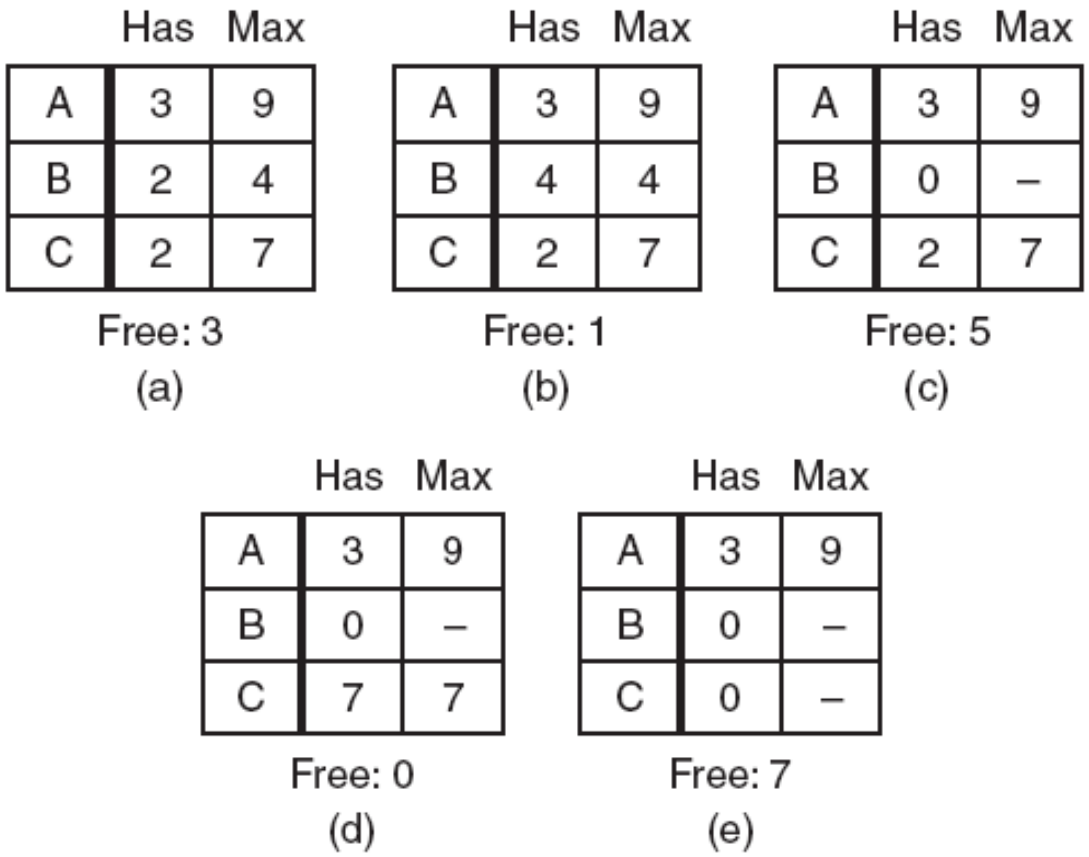


Figure 6-9. Demonstration that the state in (a) is safe.

Safe and Unsafe States (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 6-10. Demonstration that the state in (b) is not safe.

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 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

The Banker's Algorithm for Multiple Resources

Processes must state their total resource needs before executing
n processes and m types of resources
Two matrices:
Current allocation matrix
Request matrix
Three vectors:
Existing resource: E = (E1, E2, …, Em)
Possessed resource: P = (P1, P2, …, Pm)
Available resource: A = (A1, A2, …, Am)
A = E - P

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 6-12. The banker's algorithm with multiple resources.

The Banker's Algorithm for Multiple Resources

Algorithm for checking to see if a state is safe:

1. Look for row, R, whose unmet resource needs all ≤ A. If no such row exists, system will eventually deadlock since no process can run to completion
2. Assume process of row chosen requests all resources it needs and finishes. Mark process as terminated, add all its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes marked terminated (initial state was safe) or no process left whose resource needs can be met (there is a deadlock).

The Banker's Algorithm for Multiple Resources

- An example:

- Row D <= A, then A = A + (1101) = (2121)
- Row A <= A, then A = A + (3011) = (5132)
- Row B <= A, then A = A + (0100) = (5232)
- Row C <= A, then A = A + (1110) = (6342)
- Row E <= A, then A = A+(0000) = (6342) = E
- So, the current state is safe.

The Banker's Algorithm for Multiple Resources

- Suppose process B requests a printer
- Now A = (1010)
- Row D <= A, then A = A + (1101) = (2111)
- Row A <= A, then A = A + (3011) = (5122)
- Row B <= A, then A = A + (0110) = (5232)
- Row C <= A, then A = A + (1110) = (6342)
- Row E <= A, then A = A+(0000) = (6342) = E
- So, the request is still safe.
- If E requests the last printer.
- A = (1000)
- No row <= A, will lead to a deadlock.
- So E's request should be deferred.

Deadlock Prevention

- Use a protocol to ensure that the system will never enter a deadlock state.
- Negating one of the four necessary conditions.
- 1. Mutual exclusion
- 2. Hold and wait
- 3. No preemption
- 4. Circular wait

Attacking Mutual Exclusion Condiiton

- Mutual exclusion
- Ensure that no resource is assigned exclusively to a single process. Spooling everything.
- Drawback: not all resources can be spooled (such as process table)
- Competition for disk space for spooling itself may lead to deadlock.

Attacking Hold and Wait Condition

- Process requires all its resources before starting
- Problem: processes may not know how many resources needed in advance; not an optimal approach using resources (low utilization)

- A variant: a process requesting a resource first temporarily releases all the resources it holds. Once the request is granted, it gets all resource back.

Attacking No Preemption Condition

- Forcibly take away the resource. Not realistic.

Attacking the Circular Wait Condition

- Solution 1: A process is entitled only a single resource at any time.

- Solution 2: Global numbering all resources:
- Give a unique number to each resource. All requests must be made in a numerical order

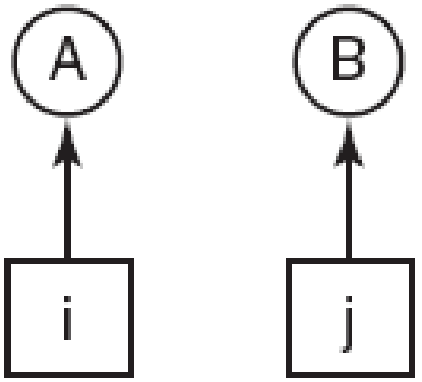
Attacking the Circular Wait Condition

- An example: two processes and five devices. Number the resources as follows:
- (a) Card reader
- (b) Printer
- (c) Plotter
- (d) Tape drive
- (e) Card punch
- Assume process A holds i and process B holds j (i \= j).
- If i > j, A is not allowed to request j.
- If i < j, B is not allowed to request i.
- Suitable to multiple processes. At any time, there must be a assigned resource with the highest number. This process will not request other assigned resources, only requests higher numbered resource and finishes. Then releases all resources.

Attacking the Circular Wait Condition

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

(a)



(b)

Figure 6-13. (a) Numerically ordered resources. (b) A resource graph.

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

Figure 6-14. Summary of approaches to deadlock prevention.

Approaches to Deadlock Prevention

Problems:

(1) Process don't know the maximum resources they need in advance
(2) The number of processes is not fixed
(3) Available resources may suddenly break

In summary,
Prevention: too overly restrictive
Avoidance: required information may not be available
Still no good general solution yet.

Communication Deadlocks

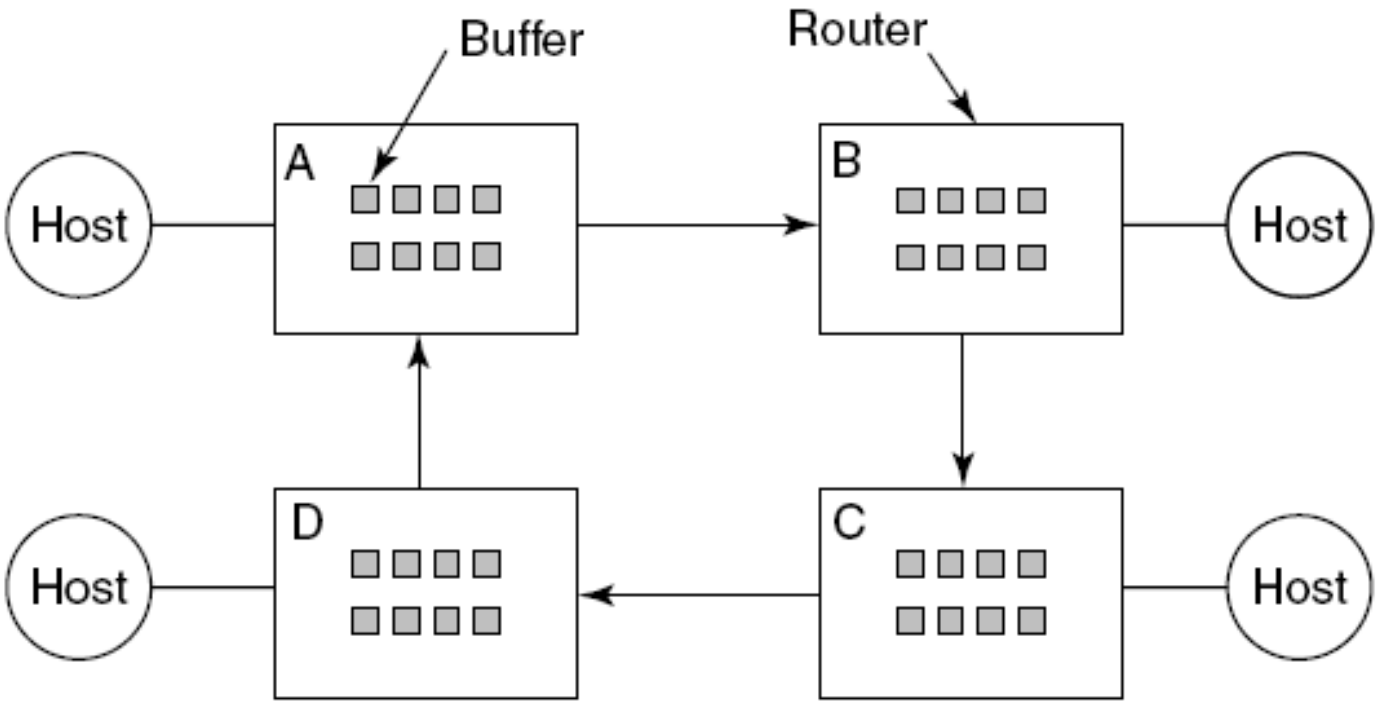


Figure 6-15. A resource deadlock in a network.

Security

Some common security problems

Abuse of valid privileges
e.g. on Unix, a super user can do anything.

Trojan Horse
Modify a normal program to do nasty things in addition to its normal function.
e.g. leave a program lying around that looks like the login process when people type passwords, remember them.

Spoiler
Use up all resources and make system crash.
e.g. grab all disk space or create thousands of processes.

Worm or virus
A Trojan Horse that is also capable of spreading itself from machine to machine.

Security

Famous historical security flaws

Unix utility lpr
Print a file with an option to remove the file after printing.
May delete password file this way.

TENEX O.S. (Used on DEC-10 computers)
Can call a user function on each page fault
_ To access a file, a program had to present a password.
_ Can find a password by putting password crossing the page boundary.
For password length n and 128 characters, at most 128n times are needed to determine the password instead of 128^{n} times.

Security

Sendmail/finger worm (Internet worm)

_ Disabled thousands of computers in Nov. 1988.

_ Used two bugs in Berkeley Unix system.

_ Sendmail attack:

Worm can mail a copy of program, get it executed and set up a Trojan Horse on machine.

_ Finger attack:

Give a carefully designed long name to finger which overflows buffer, modifies stack, causing /bin/sh to be executed.

Security

How to test a system's security

Make sure the system can withstand the following attacks

Request memory pages, disk space or tapes and just read them (to see if the system erases the information before allocating it).

Try illegal system calls, or legal system calls with illegal parameters to confuse the system.

Start login in and then hit DEL, or BREAK halfway through the login sequence. May kill password checking program and login successfully without a password.

Try modifying complex operating system structures (security related) in user space. Look for manual that says ``Do not do X." Try as many variations of X as possible.

Fool the user by writing a program that types ``login:" on the screen and go away (record password).

Security

Password:
a secret piece of information used to establish the identity of a user.

Password file: a series of ASCII lines (encrypted password), one line per user. When login, user types password. O.S. encrypts the password and compares it with that in password file.

In theory, for 7 char password length, randomly chosen from 95 printable ASCII char, there will be 95^{7} = 7x10^{13} possibilities.

At 1000 encryptions per second, requires 2000 years to build the list to check the password file.

Security

In practice, people use first names, last names, street names, city names, some common words, car license plate numbers,...

Research shows 86% matched.

Solution: Concatenate an n-bit random number with each password. Encrypted together and stored in password file.
For one password, has to encrypt 2^{n} string to match the password file. Unix uses n = 12.

This way can only prevent someone guess your password off-line, but cannot prevent someone try to login into your account on-line.

Tip: Choose password as random as possible.

Protection

The objects need to be protected:
CPU, memory segments, terminals, files, semaphores,...

Each object has a name and a set of operations.

Example:
file: name, read/write; semaphore: name, up/down

Protection domain: a collection of (object, right) pairs.
Right: permission to perform one of the operations

At any time, each process runs in some protection domain.

Protection

Keep track of which object belongs to which domain

1. Protection matrix:

Rows are the domains and the columns are the objects. Each matrix entry lists the rights.

Problem: The matrix is large and sparse.

Protection

2. Access control lists: store the matrix by columns.

Associated with each file. Indicate which users are allowed to perform which operations.

General form: each file has a list of (user, privilege) pairs.

Protection

Example:
Four users A, B, C, and D, belong to groups: system, staff, and student.
File0: (A, *, RWX)
File1: (A, system, RWX)
File2: (A, *, RW-), (B, staff, R--), (D, *, R--)
File3: (*, student, R--)
File4: (C, *, ---), (*, student, R--)

File0 can be RWX by any process with uid=A, gid=any.
File1 can be RWX by uid=A, gid =system.
File2 can be RW by uid=A, gid=any.
File2 can be R by uid=B, gid=staff.
File2 can be R by uid=D, gid=any.
File3 can be R by uid=any, gid=student.
Processes with uid=C, gid=any have no access to file4, but all other processes with uid=any, gid=student can read file4.

Protection

Compressed form: users are grouped into classes.

e.g. in Unix, each file has 9 bits RWXRWXRWX for self, group and other three classes.

Default: every one can access
Simple, used in most file systems

Protection

3. Capabilities: store the matrix by rows

Associated with each user. Indicate which file may be accessed and in what ways.

Store a list of (object, privilege) pairs with each user. Called capability list.

Default: no one can access.
Used in systems that need to be very secure.
Difficult to share information.