Chapter 6

Deadlocks

Resources

- Examples of computer resources
  - printers
  - tape drives
  - tables
- Processes need access to resources in reasonable order
- Suppose a process holds resource A and requests resource B
  - at same time another process holds B and requests A
  - both are blocked and remain so → deadlock

Resources (1)

- Deadlocks occur when ...
  - processes are granted exclusive access to devices, files, and so forth.
  - we refer to these objects generally as <u>resources</u>
- Preemptable resources
  - can be taken away from a process with no ill effects
  - Ex) Memory
- Nonpreemptable resources
  - will cause the process to fail if taken away
  - Ex) CD-ROM
- Deadlocks generally involve nonpreemptable resources.

Resources (2)

- Sequence of events required to use a resource
  1. request the resource
  2. use the resource
  3. release the resource

- Must wait if request is denied
  - requesting process may be blocked
  - may fail with error code

Resources Acquisition

- Two resources

```
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);
}
```

```
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)
    }
```

Deadlock-free code

```
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_2);
        down(&resource_1);
        use_both_resources();
        up(&resource_1);
        up(&resource_2)
    }
```

Code with a potential deadlock

Introduction to Deadlocks

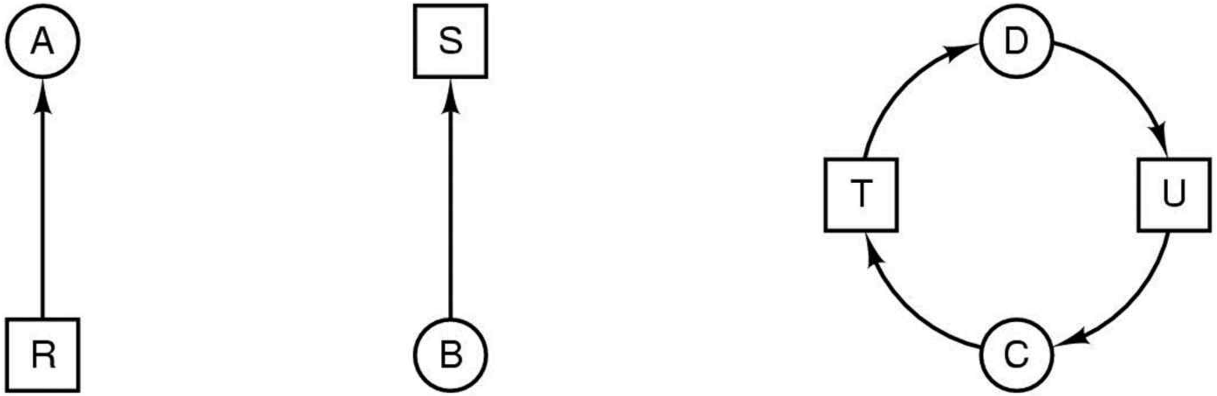
- **Formal definition :**
  A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- **Usually the event is release of a currently held resource**
- **None of the processes can …**
  - **run**
  - **release resources**
  - **be awakened**

Four Necessary Conditions for Deadlock

1. **Mutual exclusion condition**
   - each resource assigned to 1 process or is available
2. **Hold and wait condition**
   - process holding resources can request additional resources
3. **No preemption condition**
   - previously granted resources cannot forcibly be taken away
4. **Circular wait condition**
   - must be a circular chain of 2 or more processes
   - each is waiting for resource held by next member of the chain

Deadlock Modeling (2)

- Modeled with directed graphs



(a) (b) (c)

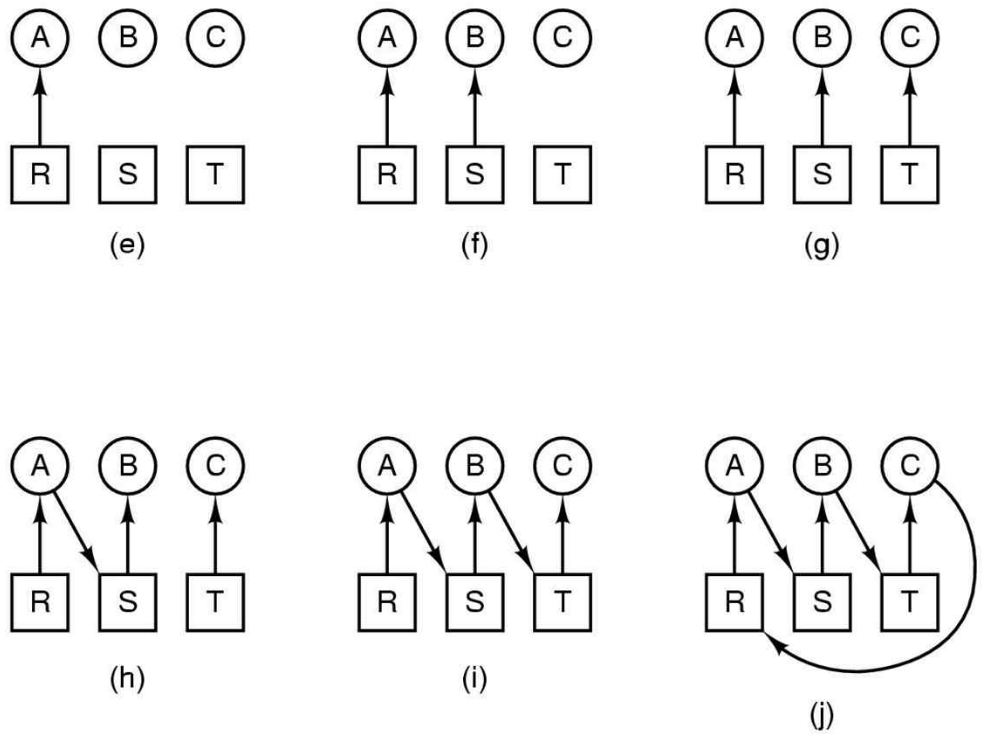
  - resource R assigned to process A
  - process B is requesting/waiting for resource S
  - process C and D are in deadlock over resources T and U

Deadlock Modeling (4)

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)

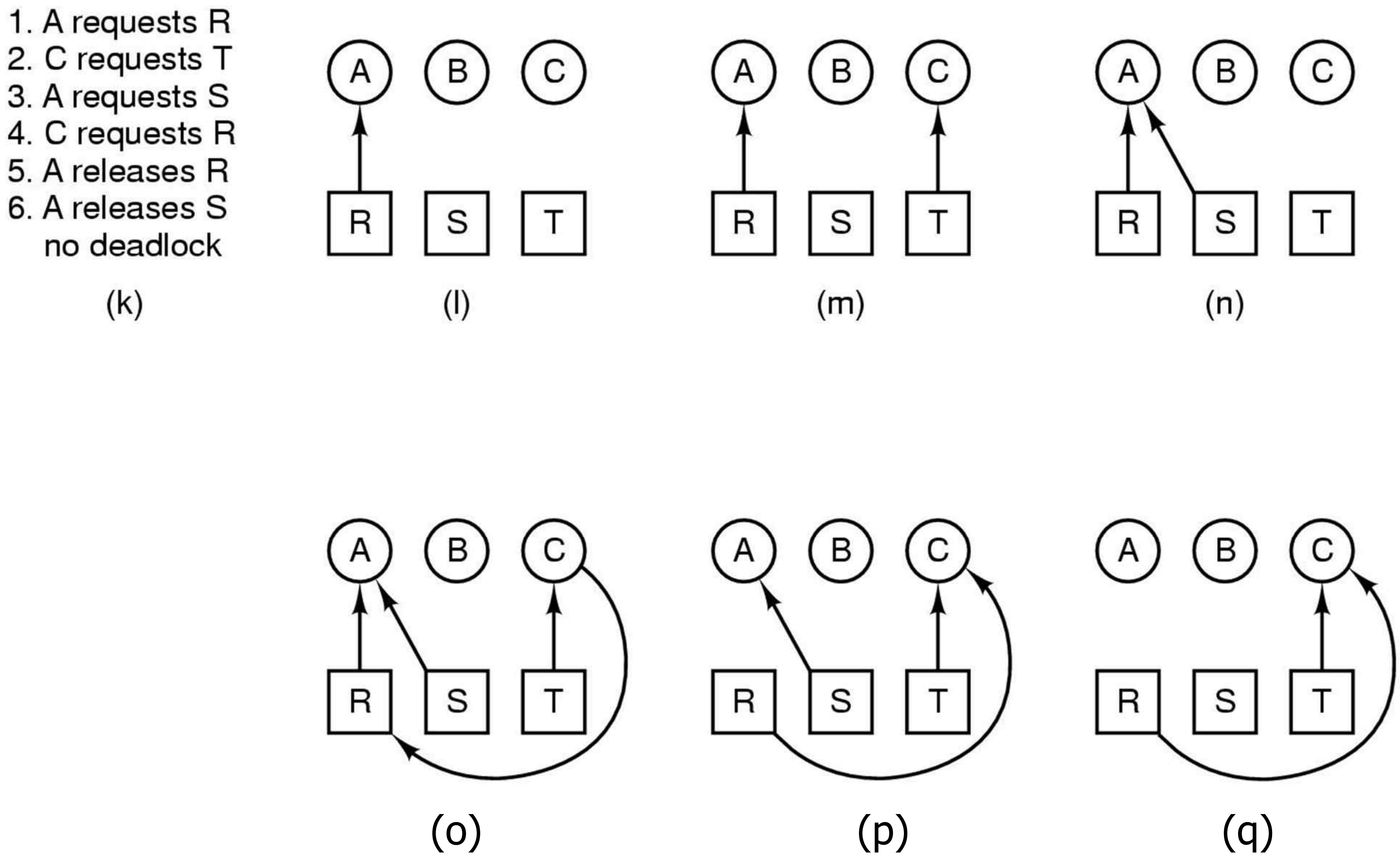
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)



How deadlock occurs

Deadlock Modeling (5)



How deadlock can be avoided

Deadlock Modeling (3)

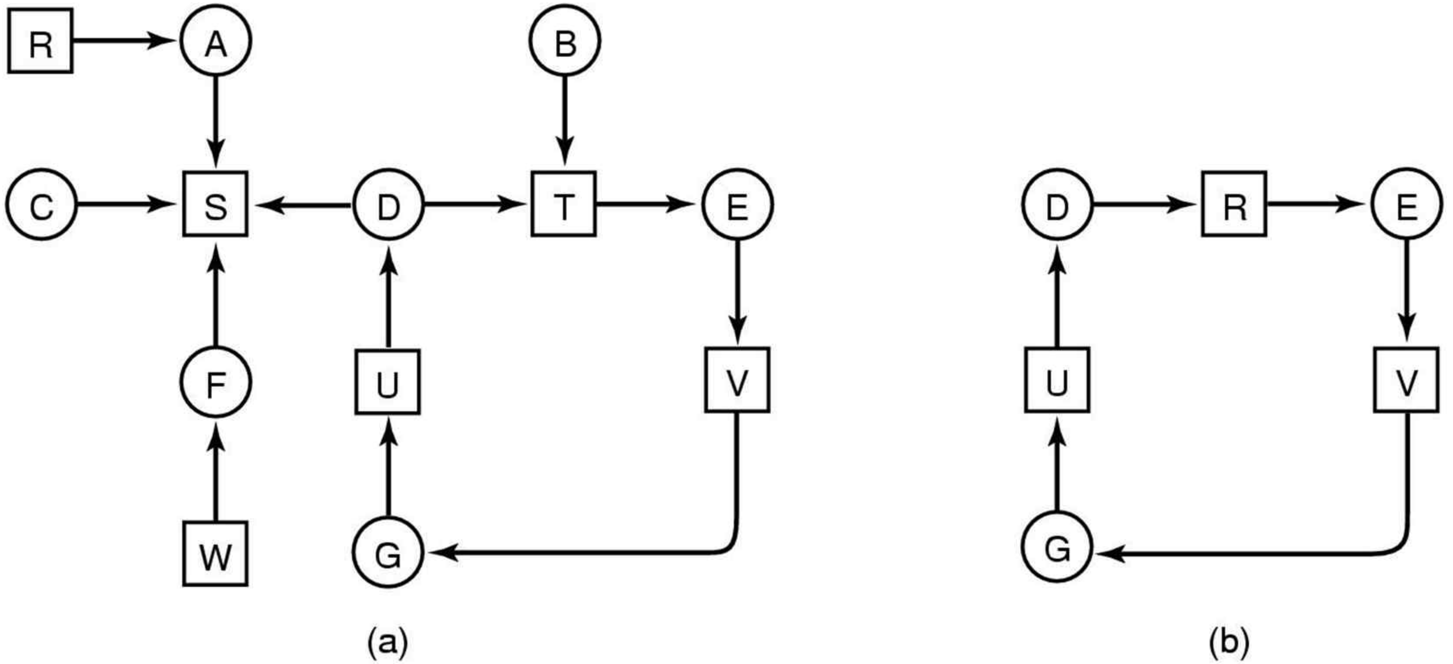
Strategies for dealing with Deadlocks

1. just ignore the problem altogether
2. detection and recovery
3. dynamic avoidance
   - careful resource allocation
4. prevention
   - negating one of the four necessary conditions

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX and Windows takes this approach
- It is a trade off between
  - convenience
  - correctness

Detection with One Resource of Each Type (1)



- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

Recovery from Deadlock (1)

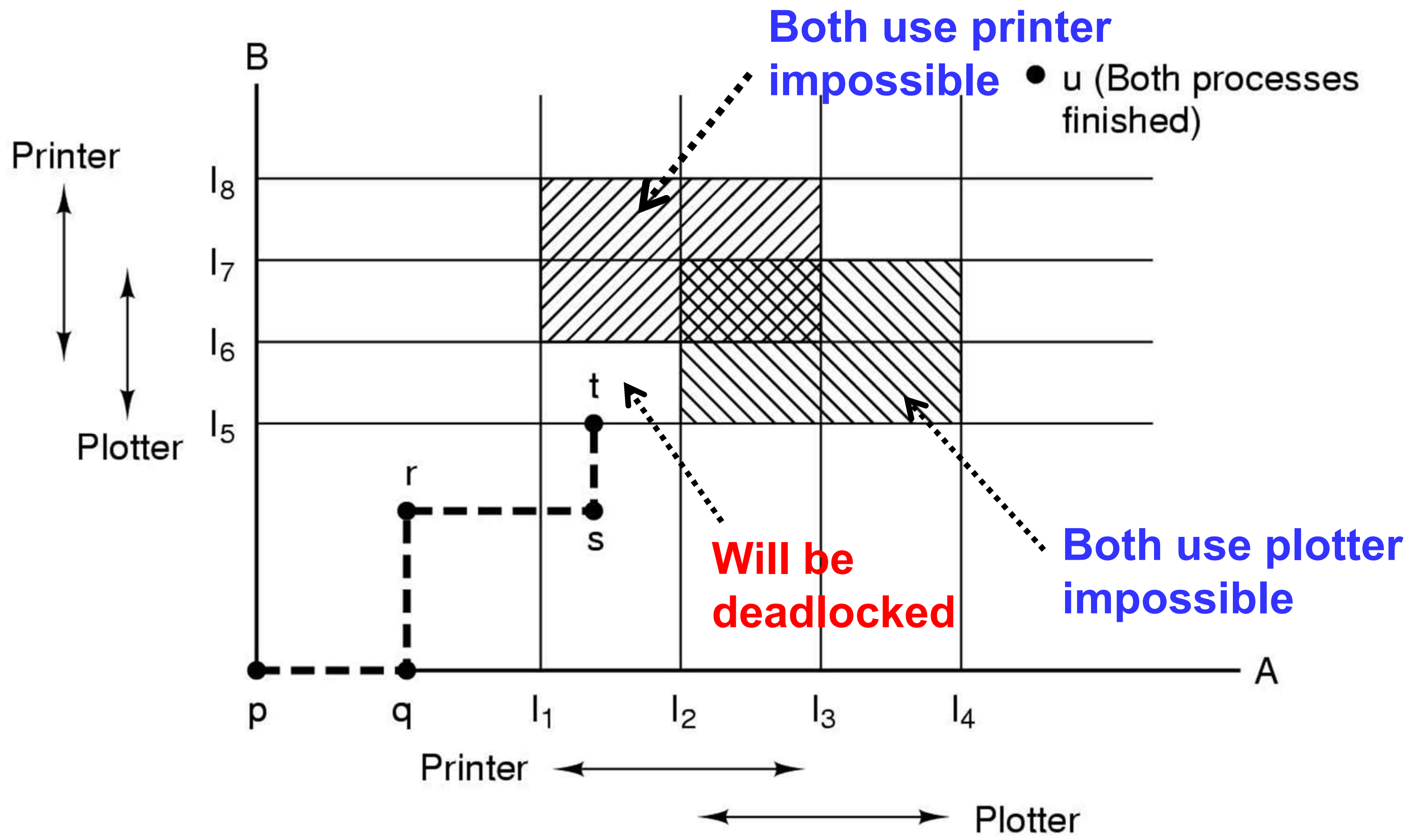
- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked
    - A process that owns a needed resource is rolled back

Recovery from Deadlock (2)

- Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun from the beginning
  - If the cycle not broken, repeat killing a process.

Deadlock Avoidance

Resource Trajectories



Two process resource trajectories

Safe and Unsafe States (0)

- **Safe State**
  - Not deadlocked and there is some scheduling order in which every process can run to completion
  - System can *guarantee* that all process will finish
- **Unsafe State**
  - State not safe

Safe and Unsafe States (1)

	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)

Demonstration that the state in (a) is safe

(total # of resources = 10)

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)

Demonstration that the sate in (b) is not safe

- ***but unsafe state (b) is not a deadlocked state yet***
  - *because B continue to complete or*
  - *because A might release a recourse before asking more*

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)

- Grants resource requests on if it leads to a safe state
- Three resource allocation states
  a. safe
  b. safe
  c. unsafe : result of granting B's request from state (b). should have been rejected by Banker's Algorithm

Problems of Deadlock Avoidance Algorithms

- In practice, it is essentially useless
  - Processes rarely known in advance their maximum resource needs
  - The number of processes is not fixed, but dynamically varying
  - Resources can be suddenly unavailable (due to some faults)
  - Etc..

Deadlock Prevention

Attacking the Mutual Exclusion Condition

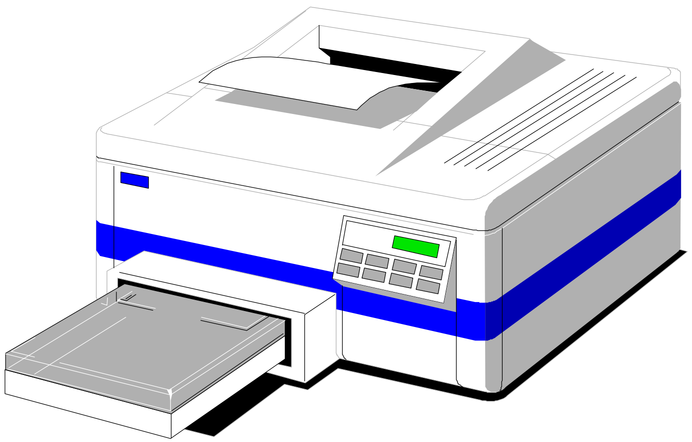
- Some devices (such as printer) can be spooled
  - only the printer daemon uses printer resource
  - thus deadlock for printer eliminated
- Not all devices can be spooled
- Principle:
  - avoid assigning resource when not absolutely necessary
  - as few processes as possible actually claim the resource

Attacking the Hold and Wait Condition

- Require processes to request all resources before starting
  - a process never has to wait for what it needs
- Problems
  - may not know required resources at start of run
  - also ties up resources other processes could be using
- Variation:
  - process must give up all resources then request all immediately needed

Attacking the No Preemption Condition

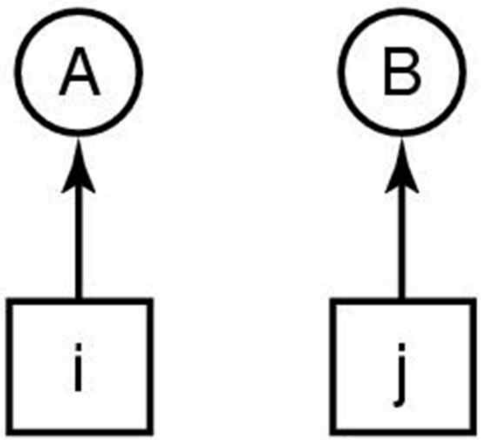
- This is not a viable option
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??



Attacking the Circular Wait Condition (1)

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

(a)



(b)

- Normally ordered resources
- Processes request resources in the order
- A resource graph can never have cycles

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

Other Issues
Two-Phase Locking

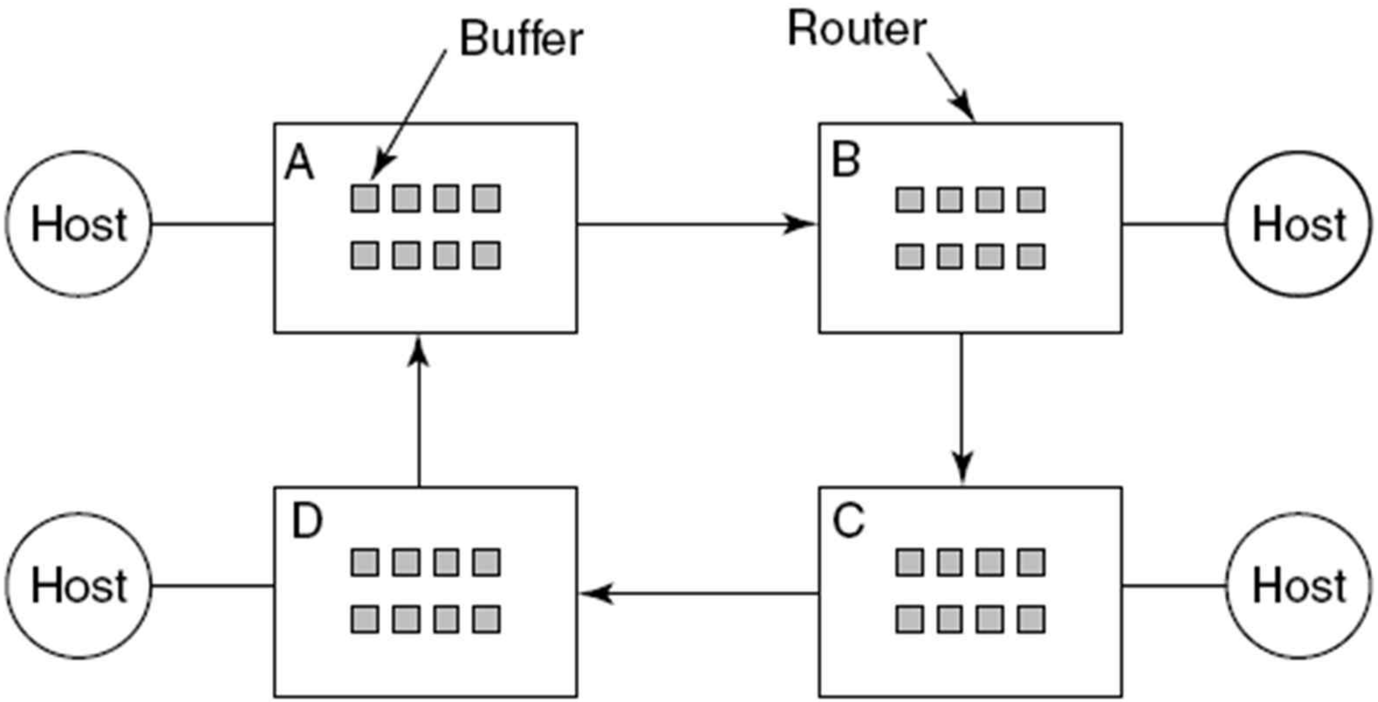
- Phase One
  - process tries to lock all records it needs, one at a time
  - if needed record found locked, start over
  - (no real work done in phase one)
- If phase one succeeds, it starts second phase,
  - performing updates
  - releasing locks
- Note similarity to requesting all resources at once
- Algorithm works where programmer can arrange
  - program can be stopped, restarted

Nonresource Deadlocks

- **Possible for two processes to deadlock**
  - **each is waiting for the other to do some task**
- **Can happen with semaphores**
  - **each process required to do a *down()* on two semaphores (*mutex* and another)**
  - **if done in wrong order, deadlock results**

Communication Deadlocks

Figu



Livelock

```
void process_A(void) {
        enter_region(&resource_1);
        enter_region(&resource_2);
        use_both_resources( );
        leave_region(&resource_2);
        leave_region(&resource_1);
}
```

```
void process_B(void) {
        enter_region(&resource_2);
        enter_region(&resource_1);
        use_both_resources( );
        leave_region(&resource_1);
        leave_region(&resource_2);
}
```

Figure 6-16. Busy waiting that can lead to livelock.

Starvation

- Algorithm to allocate a resource
  – may be to give to shortest job first
- Works great for multiple short jobs in a system
- May cause long job to be postponed indefinitely
  – even though not blocked
- Solution:
  – First-come, first-serve policy