Operating Systems

Lecture 12
Inter-Process Communication and Deadlock

Department of Computer Science & Technology
Tsinghua University

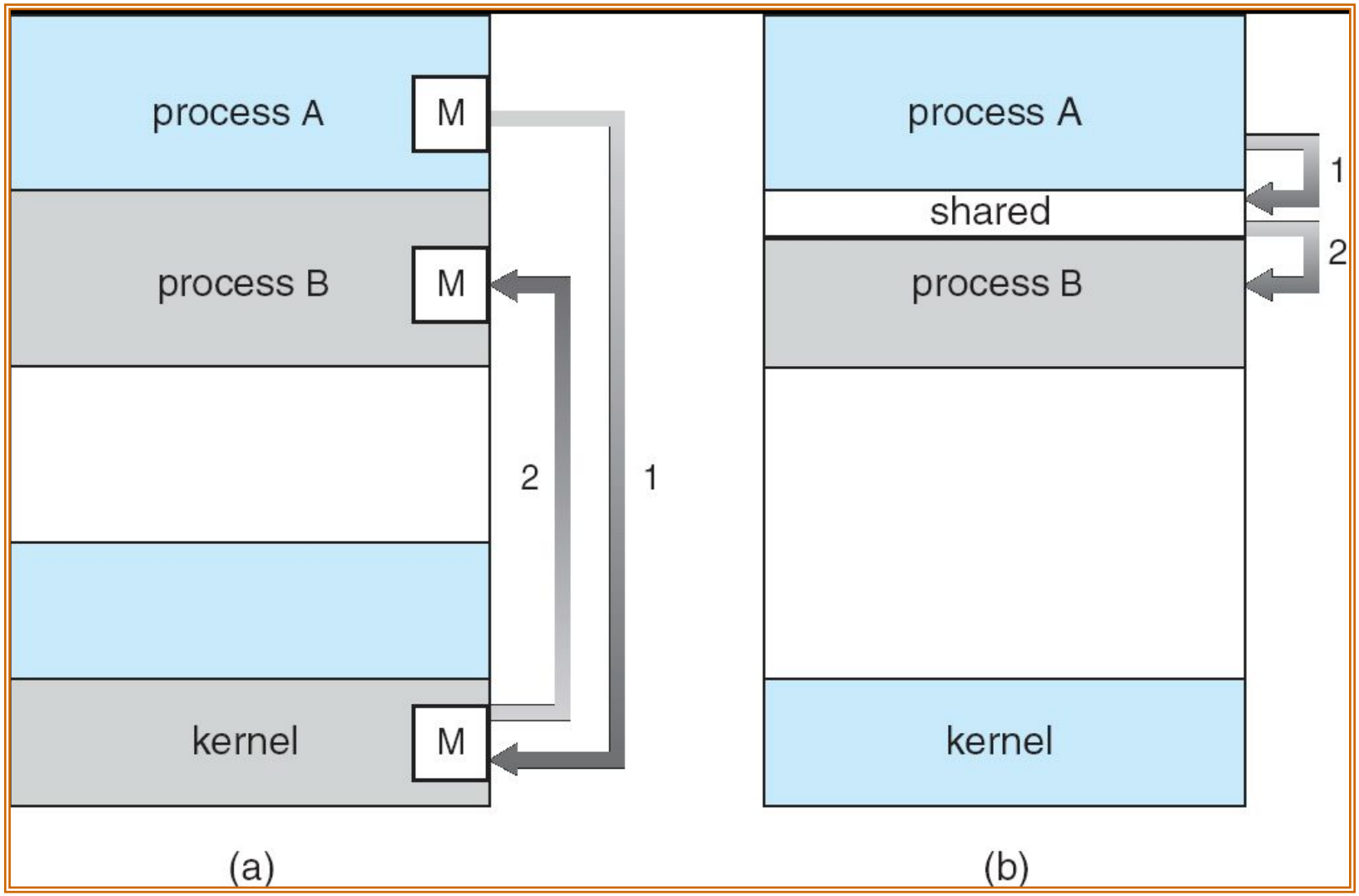
Outline

- IPC
  - Overview
    - Communications Models
    - Direct&Indirect Communication
    - Blocking and Non-blocking
    - Buffer of Communication Link
  - Signal
  - Pipe
  - Message Queue
  - Shared Memory
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

IPC Overview

- Mechanism for processes to communicate and to synchronize their actions
- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Communications Models



Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from message queues
  - Each message queue has a unique id
  - Processes can communicate only if they share a message queue
- Properties of communication link
  - Link established only if processes share a common message queue
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new message queue
  - send and receive messages through message queue
  - destroy a message queue
- Primitives are defined as:

send(*A, message*) – send a message to Queue A

receive(*A, message*) – receive a message from Queue A

Indirect Communication

- Message queue sharing
  - P_1, P_2, and P_3 share message queue A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Blocking and Non-blocking

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Outline

- IPC
  - Overview
  - Signal
    - Signal Generation
    - Delivery
    - Signal API
    - Signal Example
  - Pipe
  - Message Queue
  - Shared Memory
  - Solaris Doors (opt)
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

Signals

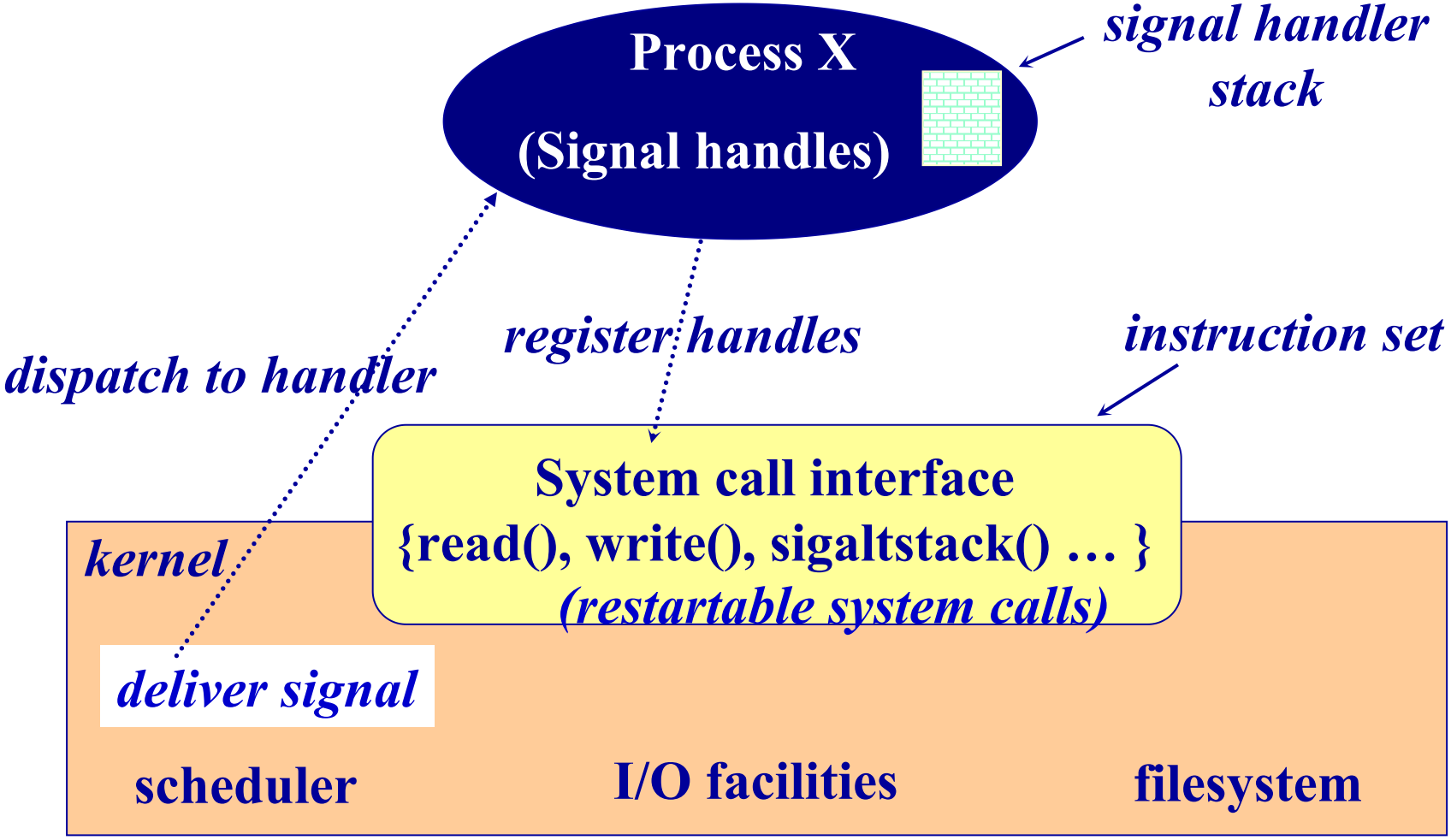
- Signal
  - Software interrupt that notifies a process of an event
  - Examples: SIGFPE, SIGKILL, SIGUSR1, SIGSTOP, SIGCONT
- What happens when a signal is received?
  - Catch: Specify signal handler to be called
  - Ignore: Rely on OS default action
    - Example: Abort, memory dump, suspend or resume process
  - Mask: Block signal so it is not delivered
    - May be temporary (while handling signal of same type)
- Disadvantage
  - Does not specify any data to be exchanged

Signals

- Divided into *asynchronous* (CTL-C) and *synchronous* (illegal address)
- Three phases to processing signals:
  - *generation*: event occurs requiring process notification
  - *delivery*: process recognizes and takes appropriate action
  - *pending*: between generation and delivery

Signals

- Asynchronous signal:
  - ctrl-C
  - child process completes
  - alarm scheduled by the process expires
    - Unix: SIGALRM from `alarm()` or `setitimer()`
  - resource limit exceeded (disk quota, CPU time...)
- Synchronous signal: programming errors, such as invalid data, divide by zero
  - SIGTRAP: a condition arises that a debugger has requested to be informed of.
  - SIGBUS: a bus error
  - SIGSEGV: an invalid memory reference, or segmentation fault.
  - SIGFPE: an erroneous arithmetic operation


Process X
(Signal handles)
signal handler stack
register handles
instruction set
dispatch to handler
System call interface
{read(), write(), sigaltstack() … }
(restartable system calls)
kernel
deliver signal
scheduler
I/O facilities
filesystem

Signal Generation

- Exceptions - kernel notifies process with signal
- Other Process - using *kill* or *sigsend*.
- Terminal interrupts - *stty* allows binding of signals to specific keys, sent to *foreground process*
- Job control - background processes attempt to read/write to terminal. Process terminate or suspends, kernels sends signal to parent
- Quotas - exceeding limits
- Notifications - event notification (device ready)
- Alarms - process notified of alarm via signal reception

Delivery

- Default actions
  - Abort: terminate process, generate core dump
  - Exit: terminate without generating core dump
  - Ignore: ignore signal
  - Stop: suspend process
  - Continue: resume process
- User specified actions
  - Default action,
  - Ignore signal,
  - Catch signal - invoke user specified signal handler
- User may not ignore, catch or block SIGKILL and SIGSTOP
- User may change action at any time
- User may block signal
  - signal remains pending until unblocked

Signals API

- Sending signals
  - `kill( pid, signal )` – system call to send *signal* to *pid*
  - `raise( signal )` – call to send *signal* to executing program/current process
- Signal handling
  - a signal handler can be invoked when a specific signal is received
  - a process can deal with a signal in one of the following ways:
    - default action
    - block the signal (some signals cannot be ignored)
    - catch the signal with a handler: `signal( signal, void (*func)() )`
      - e.g., to ignore a signal (not SIGKILL), use `signal( sig_nr, SIG_IGN )`
      - write a function yourself - `void func() {}`

Signal Example

```
#include <stdio.h>
#include <signal.h>
void sigproc()
{
signal(SIGINT, sigproc); /* NOTE some versions of UNIX will reset
                          * signal to default after each call. So for
                          * portability reset signal each time */


   printf("you have pressed ctrl-c - disabled \n");
}

void quitproc()
{
   printf("ctrl-\\ pressed to quit\n");   /* this is "ctrl" & "\" */
   exit(0); /* normal exit status */
}

main()
{
signal(SIGINT, sigproc);    /* DEFAULT ACTION: term */
signal(SIGQUIT, quitproc);  /* DEFAULT ACTION: term */
   printf("ctrl-c disabled use ctrl-\\ to quit\n");

   for(;;);
}
```

Outline

- IPC
  - Overview
  - Signal
  - Pipe
    - Pipe Size
    - Pipe Creation
    - Pipe Examples
  - Message Queue
  - Shared Memory
  - Solaris Doors (opt)
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

Pipes

- Process inherits file descriptors from parent
  - file descriptor 0 stdin, 1 stdout, 2 stderr
- Process doesn't know (or care!) when reading from keyboard, file, or process or writing to terminal, file, or process
- System calls:
  - read(fd, buffer, nbytes) (`scanf()` built on top)
  - write(fd, buffer, nbytes) (`printf()` built on top)
  - pipe(rgfd) creates a pipe
    - rgfd array of 2 fd. Read from rgfd[0], write to rgfd[1]

Pipes

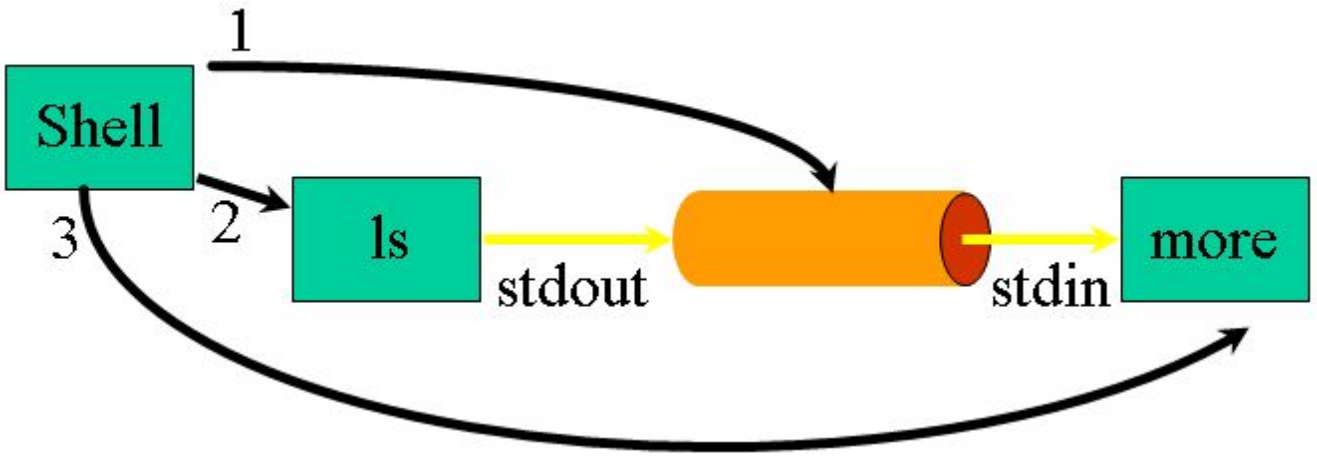
- Classic IPC method under UNIX:
  `> ls -l | more`
  - shell runs two processes `ls` and `more` which are linked via a pipe
  - the first process (`ls`) writes data (e.g., using `write`) to the pipe and the second (`more`) reads data (e.g., using `read`) from the pipe
- the system call `pipe( fd[2] )`
  creates one file descriptor for reading (`fd[0]`) and one for writing (`fd[1]`)
  - allocates a temporary inode and a memory page to hold data

```
struct pipe_inode_info {
   wait_queue_head_t wait;
   char *base;
   unsigned int len;
   unsigned int start;
   unsigned int readers, writers;
   unsigned int waiting_readers,
   unsigned int r_counter, w_cour
}
```

Pipes

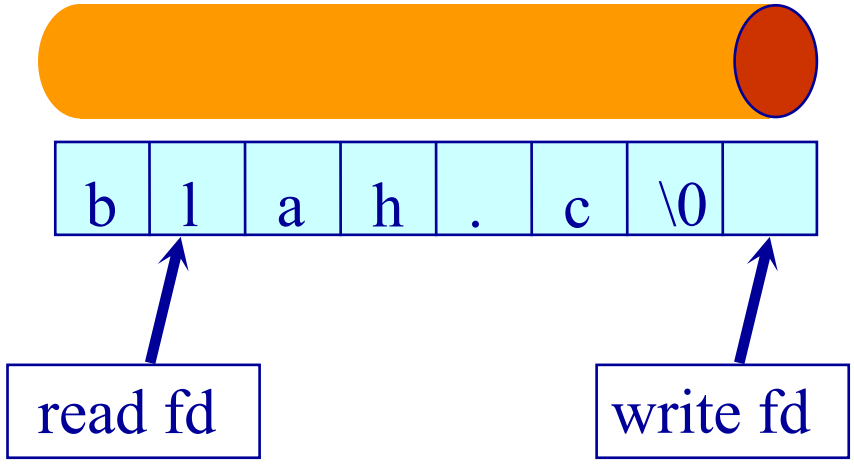
- One process writes, 2nd process reads

```
% ls | more
```



- shell:
  1. create a pipe
  2. create a process for `ls` command, setting `stdout` to write side of pipe
  3. create a process for `more` command, setting `stdin` to read side of pipe

The Pipe



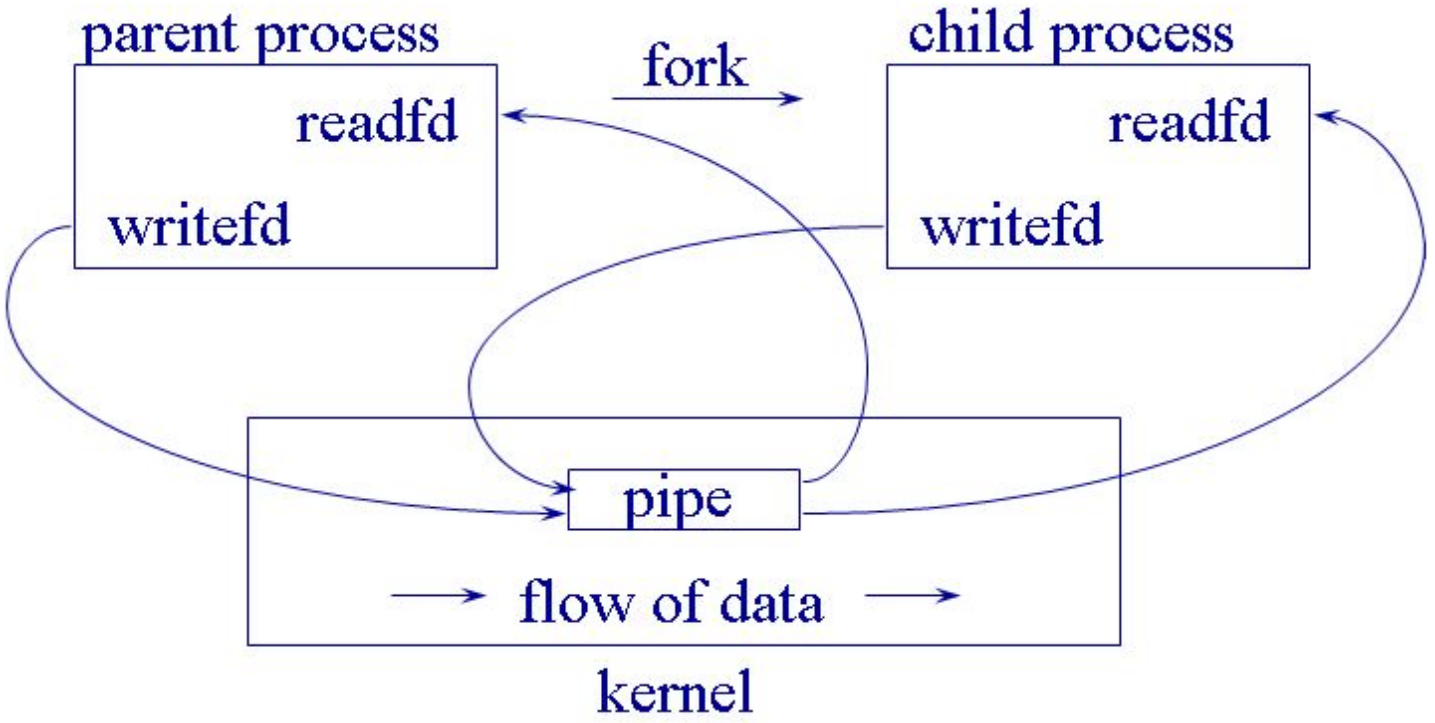
- Bounded Buffer
  - shared buffer (Unix 4096K)
  - block writes to full pipe
  - block reads to empty pipe

Pipe Size

- The size of a pipe is finite, i.e., only a certain amount of bytes can remain in the pipe without being read
- If a write is made on a pipe and there is enough space, then the data is sent down the pipe and the call returns immediately.
- If, however, a write is made that will overfill the pipe, process execution is suspended until room is made by another process reading from the pipe.

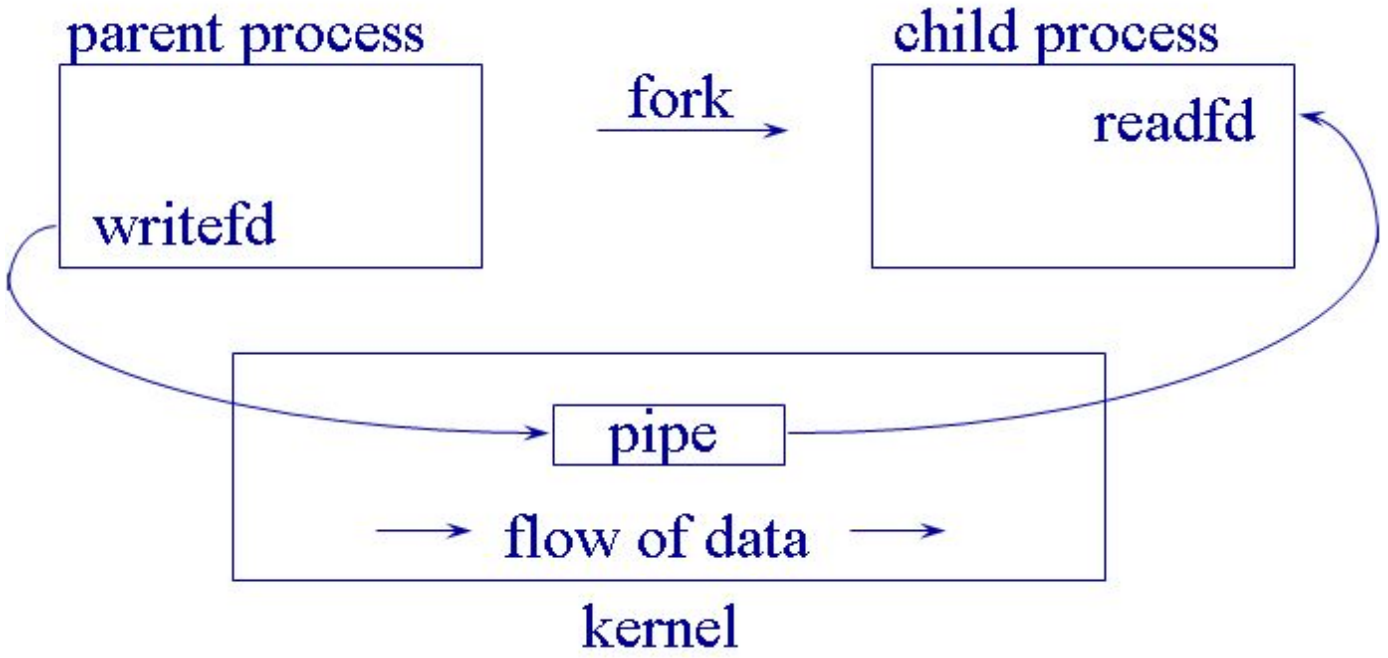
Pipe Creation

- First, a process creates a pipe, and then forks to create a copy of itself.



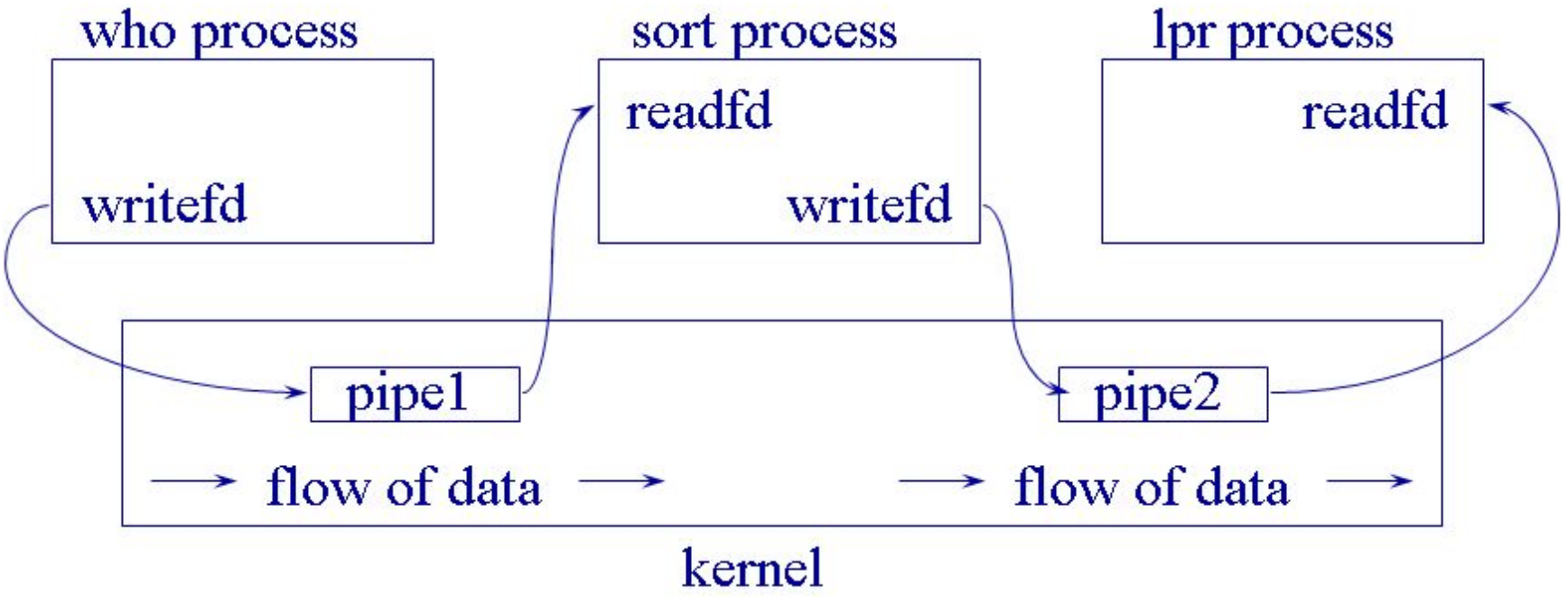
Pipe Examples

- Parent opens file, child reads file
  - parent closes read end of pipe
  - child closes write end of pipe



who | sort | lpr

- who process writes to pipe1
- sort process reads from pipe1, writes to pipe2
- lpr process reads from pipe2



Pipe Example

```
#include <unistd.h>
#include <stdio.h>
char *msg = "hello";
main()
{
   char inbuf[MSGSIZE];
   int p[2];
   pid_t pid;
   /* open pipe */
   if (pipe(p) == -1) { perror("pipe call error"); exit(1); }

   switch( pid = fork() ) {
   case -1: perror("error: fork call");
            exit(2);

   case 0:  close(p[0]);  /* close the read end of the pipe */
                          write(p[1], msg, MSGSIZE);
                          printf("Child: %s\n", msg);
            break;
   default: close(p[1]);  /* close the write end of the pipe */
                          read(p[0], inbuf, MSGSIZE);
                          printf("Parent: %s\n", inbuf);
            wait(0);
   }
```

Outline

- IPC
  - Overview
  - Signal
  - Pipe
  - Message Queue
  - Shared Memory
  - Solaris Doors (opt)
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

Message Queues

- Message queues sorting messages according to FIFO
  - messages are stored as a sequence of bytes
  - system V IPC messages also have a type
  - get a message queue identifier: `msgget ( key, flags )`
  - sending messages: `msgsnd ( Qid, buf, size, flags )`
  - receiving messages: `msgrcv ( Qid, buf, size, type, flags )`
  - control a shared segment: `msgctl( ... )`

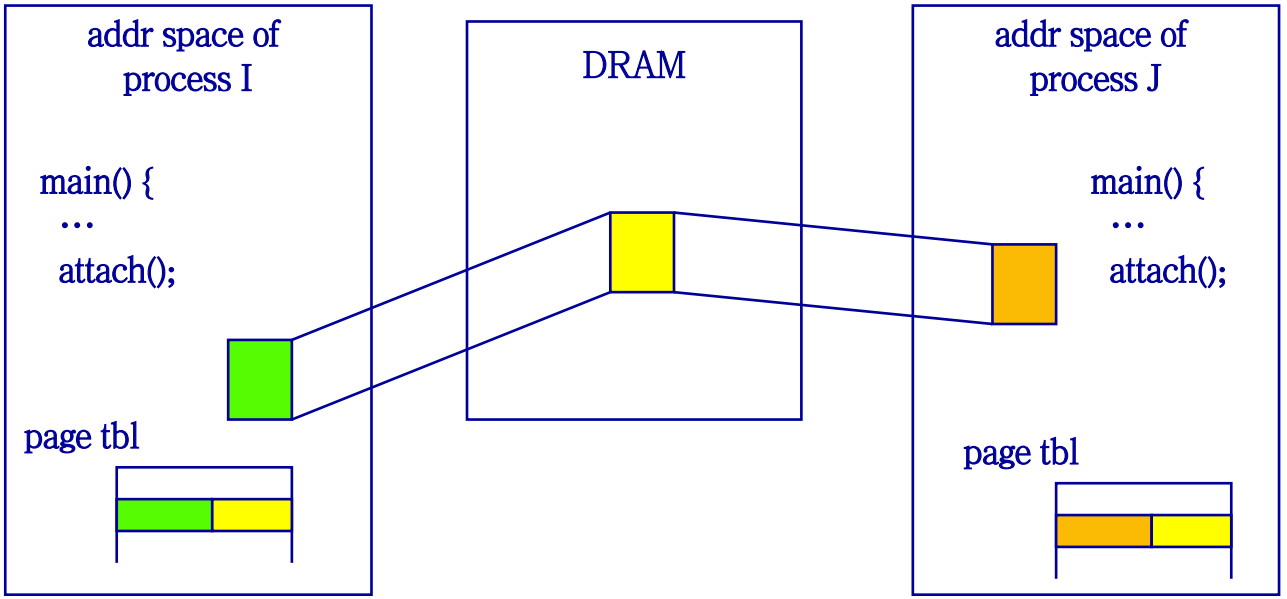
Outline

- IPC
  - Overview
  - Signal
  - Pipe
  - Message Queue
  - Shared Memory
  - Solaris Doors (opt)
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

Shared Memory

- Processes
  - Each process has private address space
  - Explicitly set up shared memory segment within each address space
- Threads
  - Always share address space (use heap for shared data)
- Advantages
  - Fast and easy to share data
- Disadvantages
  - Must **synchronize** data accesses;

Shared Memory



- the fastest method
- write by a process can be seen by another process instantly
- no syscall intervention
- no data copying
- no synchronization is provided
  - it is up to programmers' responsibility

Shared Memory

- Shared memory is an efficient and fast way for processes to communicate
  - multiple processes can attach a segment of physical memory to their virtual address space
  - create a shared segment: `shmget( key, size, flags )`
  - attach a shared segment: `shmat( shmid, *shmaddr, flags )`
  - detach a shared segment: `shmdt( *shmaddr )`
  - control a shared segment: `shmctl( … )`
  - if more than one process can access segment, an outside protocol or mechanism (like semaphores) should enforce consistency/avoid collisions

Programming Shared Memory

- segment_id = shmget (key, size, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
  - a brand new segement is created if
    - key is IPC_PRIVATE or
    - IPC_CREAT is asserted
    - otherwise, segment_id of existing segment is returned
      - you have to know the "key"
  - size is rounded up to multiple of the page size
  - S_I*xxx
    - * for read/write
    - xxx for USR(owner) OTH(others) GRP(group)
  - IPC_EXCL
    - for segment creation(IPC_CREAT option), it guarantees the key is unique (not existing one)

Program Example

```
#define KEY  ((key_t)(1234))
#define SEGSIZE  sizeof(struct some_data_structure))

struct some_data_structure *ap;

int id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
if (id < 0) error_rtn(id);

ap = (struct some_data_structure *) shmat(id, 0, 0);
```

let the system choose the location of the segment

```
#define KEY  ((key_t)(1234))
#define SEGSIZE  sizeof(struct some_data_structure))

struct some_data_structure *ap2;

int id = shmget(KEY, SEGSIZE, 0);
if (id < 0) error_rtn(id);

ap2 = (struct some_data_structure *) shmat(id, 0, 0);
```

0 for R/W
SHM_RDONLY

Mapped File

```
ptr = mmap(       /* map the file to ptr */
    addr, /* suggest ptr valur. 0 means let  system choose */
    length,       /* size of mapped region */
    prot,  /* access: PROT_READ | PROT_WRITE  */
    flags,/* mapping type: MAP_SHARED */
    fd,           /* file */
    offset);      /* location within the file */
```

- fd can be a device
  - a device can be accessed without full blown device driver
- operations are more familiar than shared memory
  - open, close, chmod, unlink, ….

Outline

- IPC
  - Overview
  - Signal
  - Pipe
  - Message Queue
  - Shared Memory
  - Solaris Doors (opt)
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

Objectives

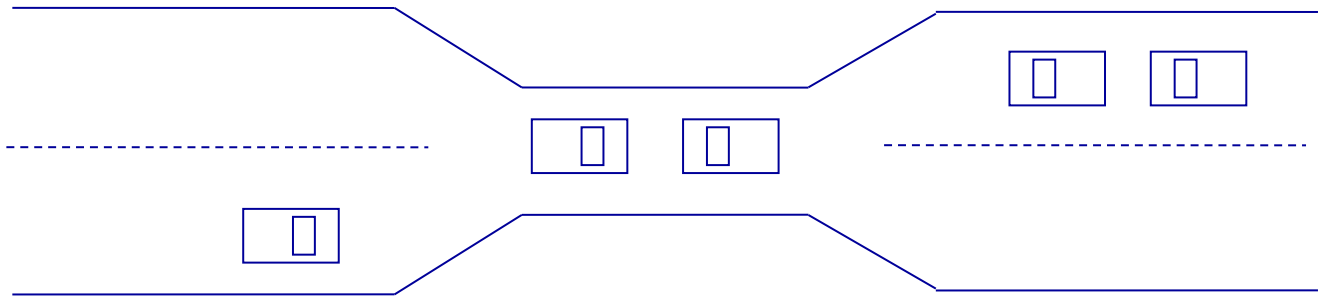
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

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 tape drives.
  - P_1 and P_2 each hold one tape 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)*

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

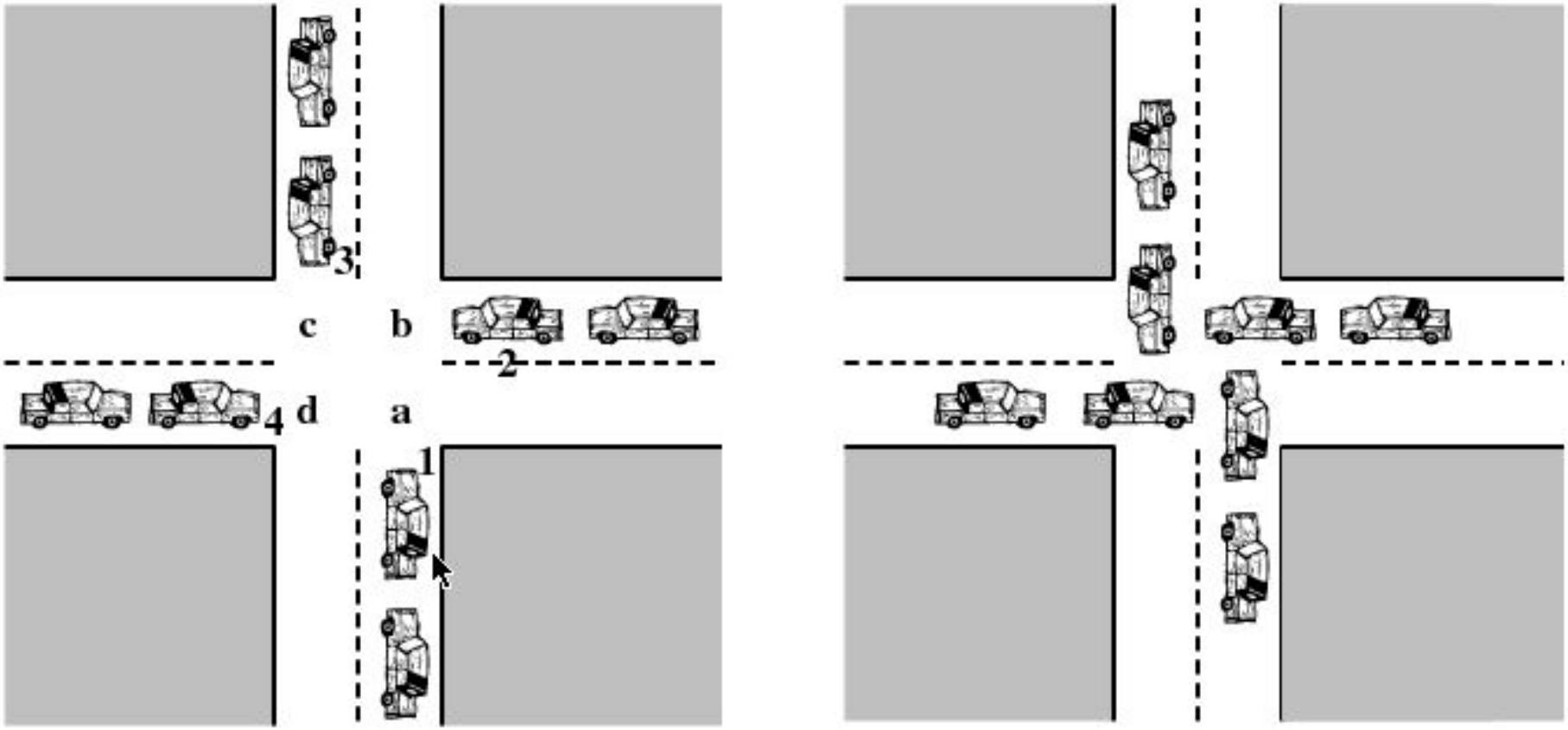


Figure 6.1 Illustration of Deadlock

Outline

- IPC
  - Overview
  - Signal
  - Pipe
  - Message Queue
  - Shared Memory
  - Solaris Doors (opt)
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

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 utilizes a resource as follows:

  Πrequest

  Πuse

  Πrelease

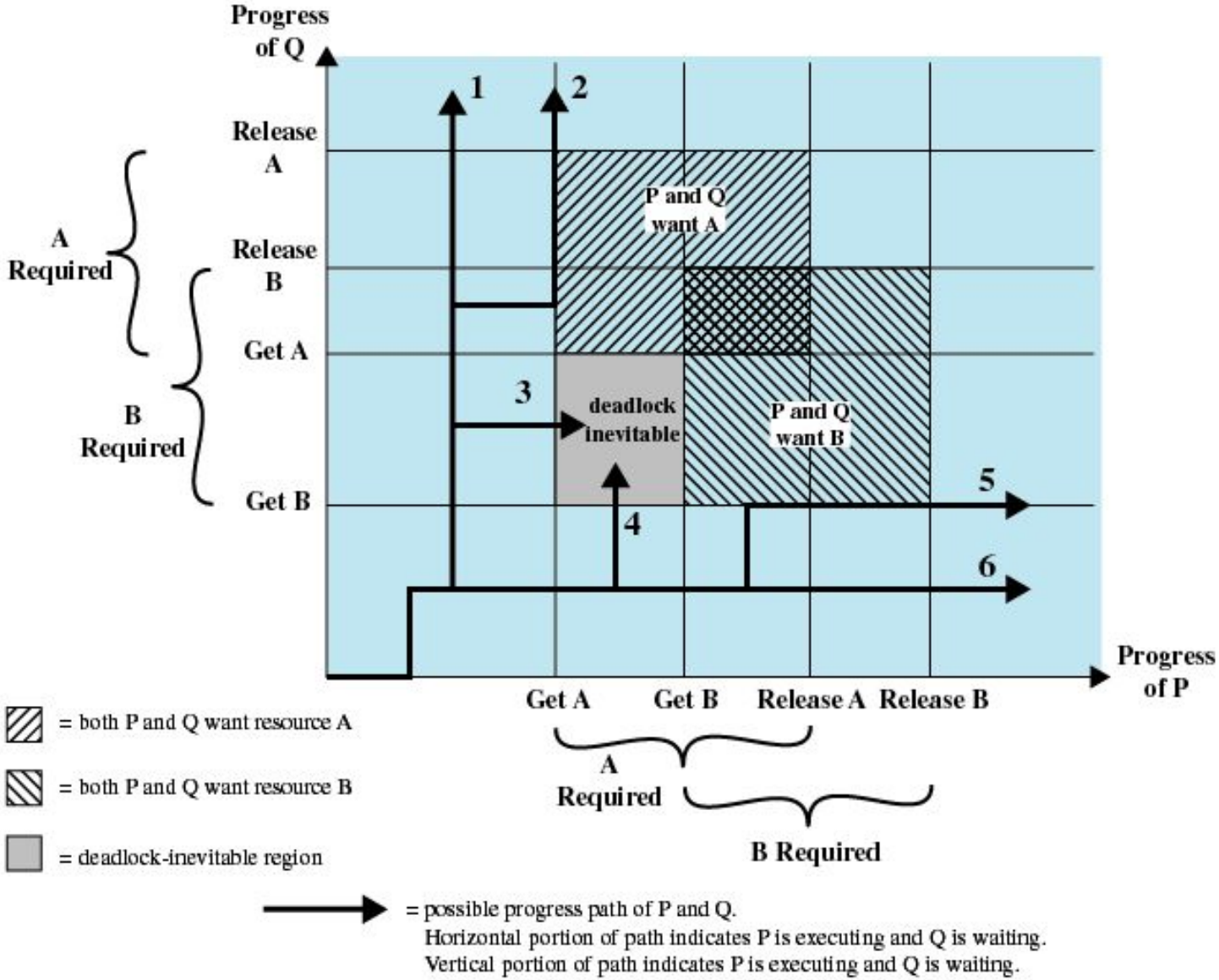


Figure 6.2 Example of Deadlock

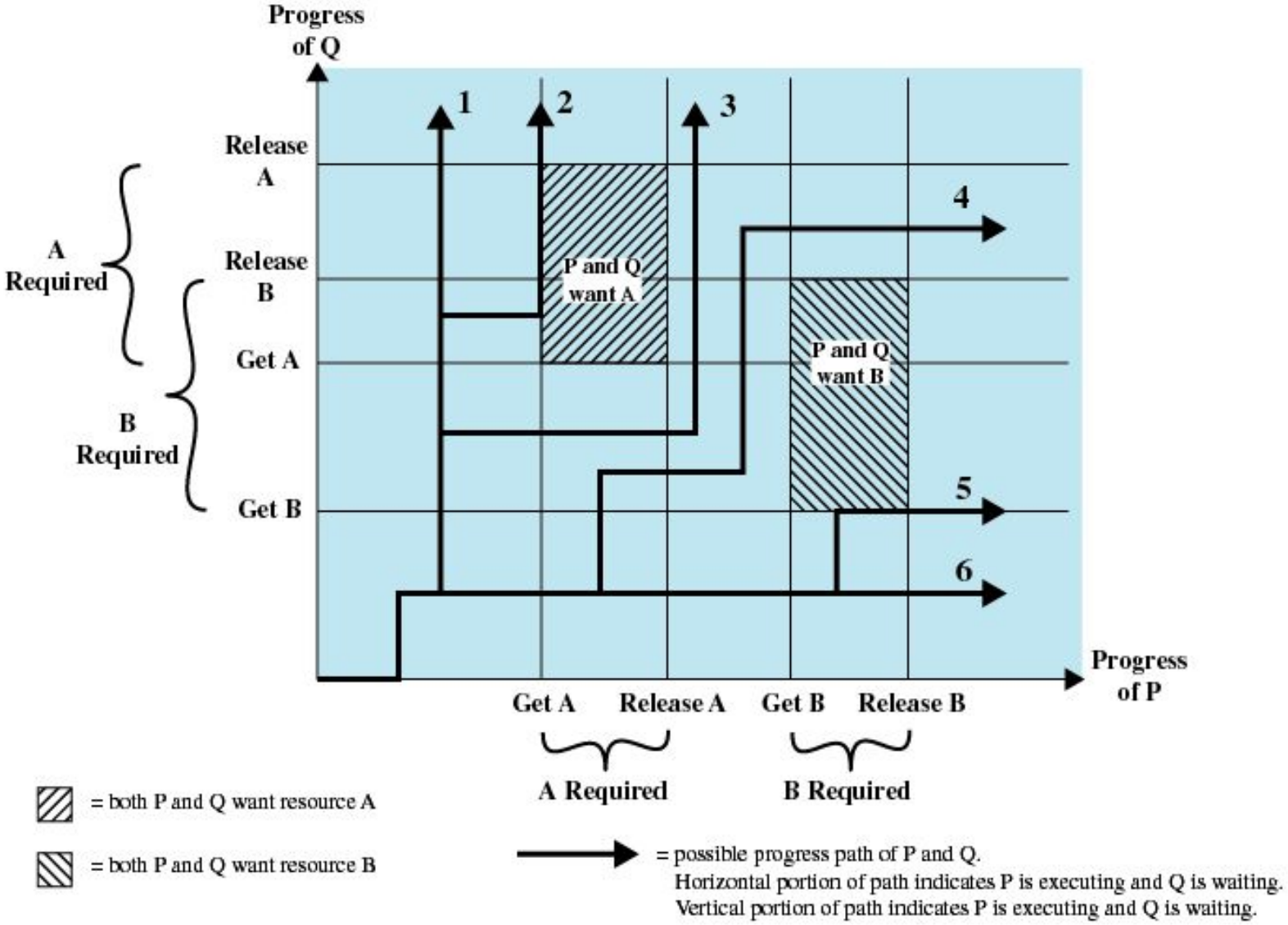


Figure 6.3 Example of No Deadlock [BACO03]

Outline

- IPC
  - Overview
  - Signal
  - Pipe
  - Message Queue
  - Shared Memory
  - Solaris Doors (opt)
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
    - Reusable Resources
    - Consumable Resources
    - Resource-Allocation Graph
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

Deadlock Characterization

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 $\{P_0, P_1, \cdots, P_n\}$ 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, $\cdots$, 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.

Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

Example of Deadlock

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

P1	P2
. . .	. . .
Request 80 Kbytes;	Request 70 Kbytes;
. . .	. . .
Request 60 Kbytes;	Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Example of Deadlock

- Deadlock occurs if receive is blocking

P1

```
. . .
Receive(P2);
. . .
Send(P2, M1);
```

P2

```
. . .
Receive(P1);
. . .
Send(P1, M2);
```

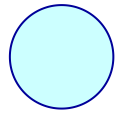
Resource-Allocation Graph

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

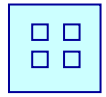
- V is partitioned into two types:
  - $P = \{P_1, P_2, \cdots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \cdots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph (Cont.)

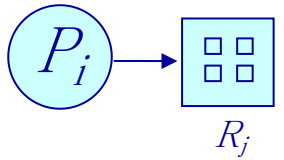
- Process



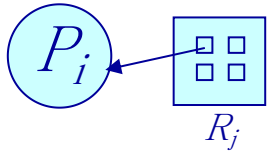
- Resource Type with 4 instances



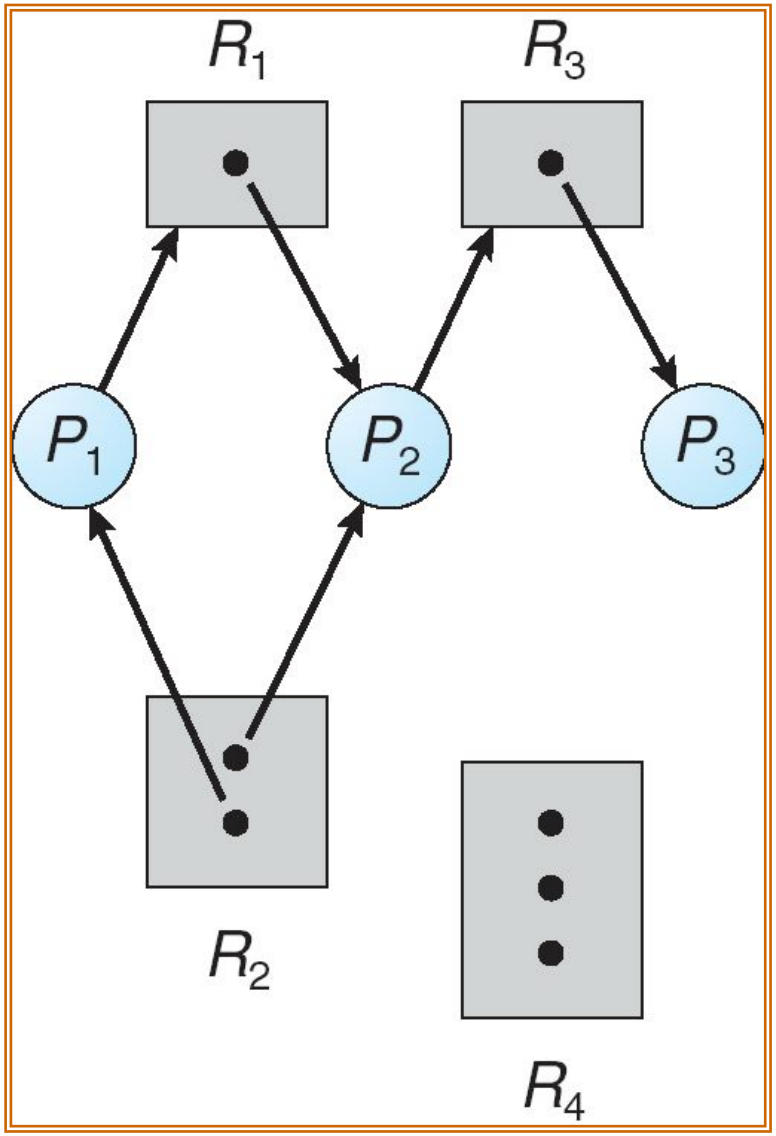
- P_i requests instance of R_j



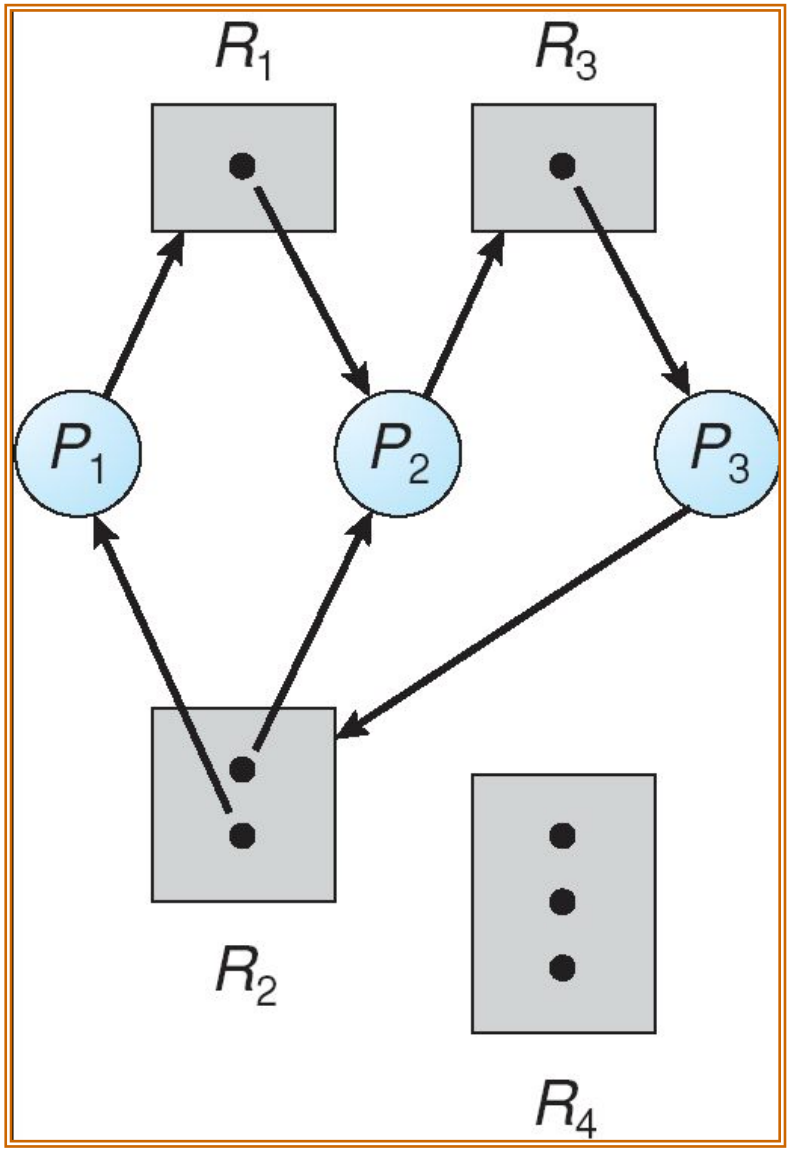
- P_i is holding an instance of R_j



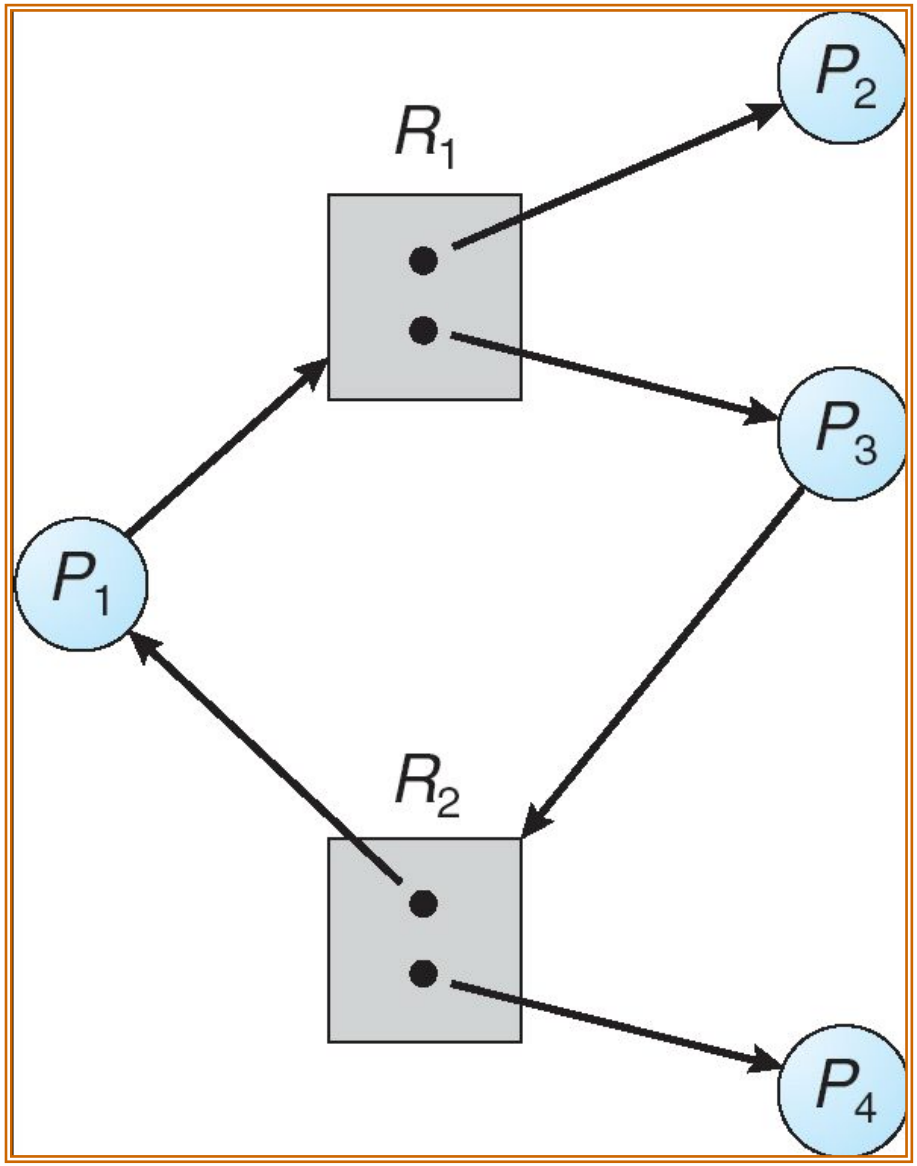
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Outline

- IPC
  - Overview
  - Signal
  - Pipe
  - Message Queue
  - Shared Memory
  - Solaris Doors (opt)
- Deadlocks
  - Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Recovery from Deadlock

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

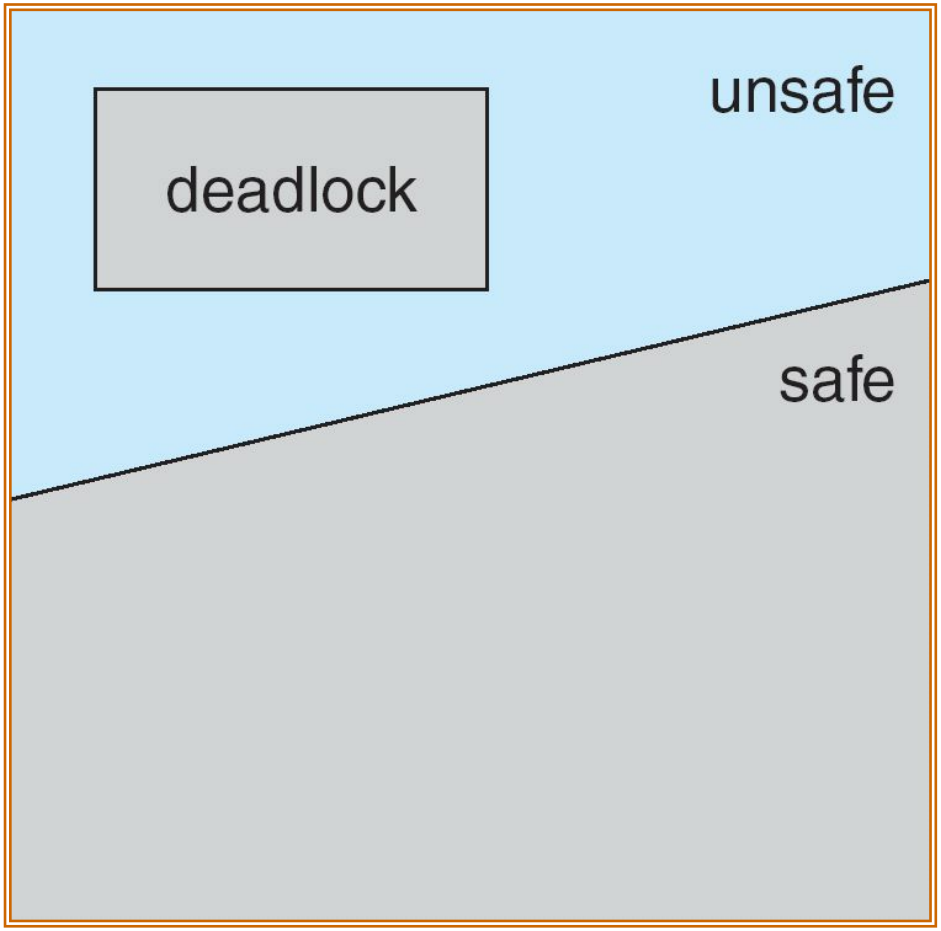
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $<P_1, P_2, \cdots, P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with j<i.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

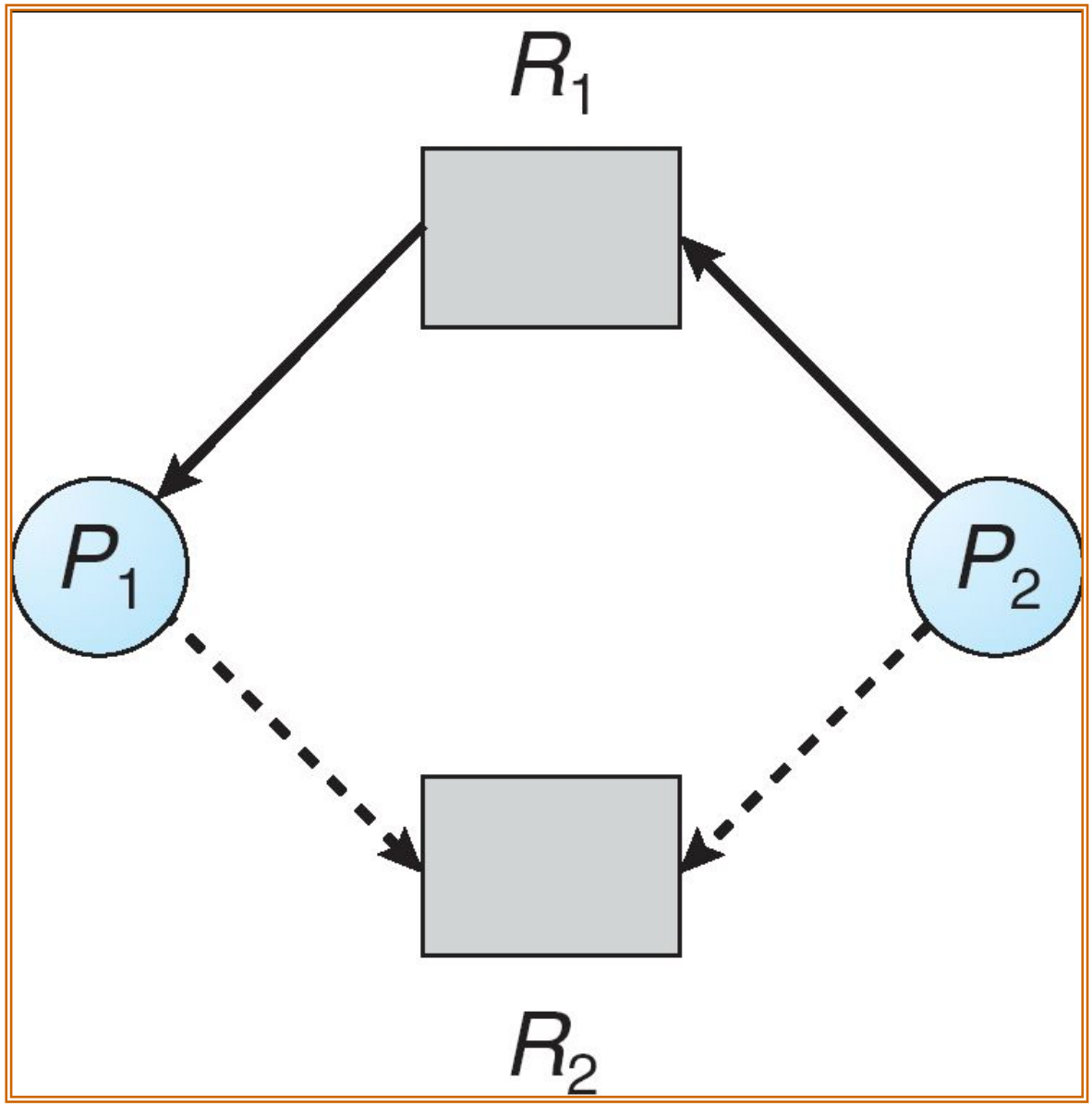
Safe, Unsafe , Deadlock State



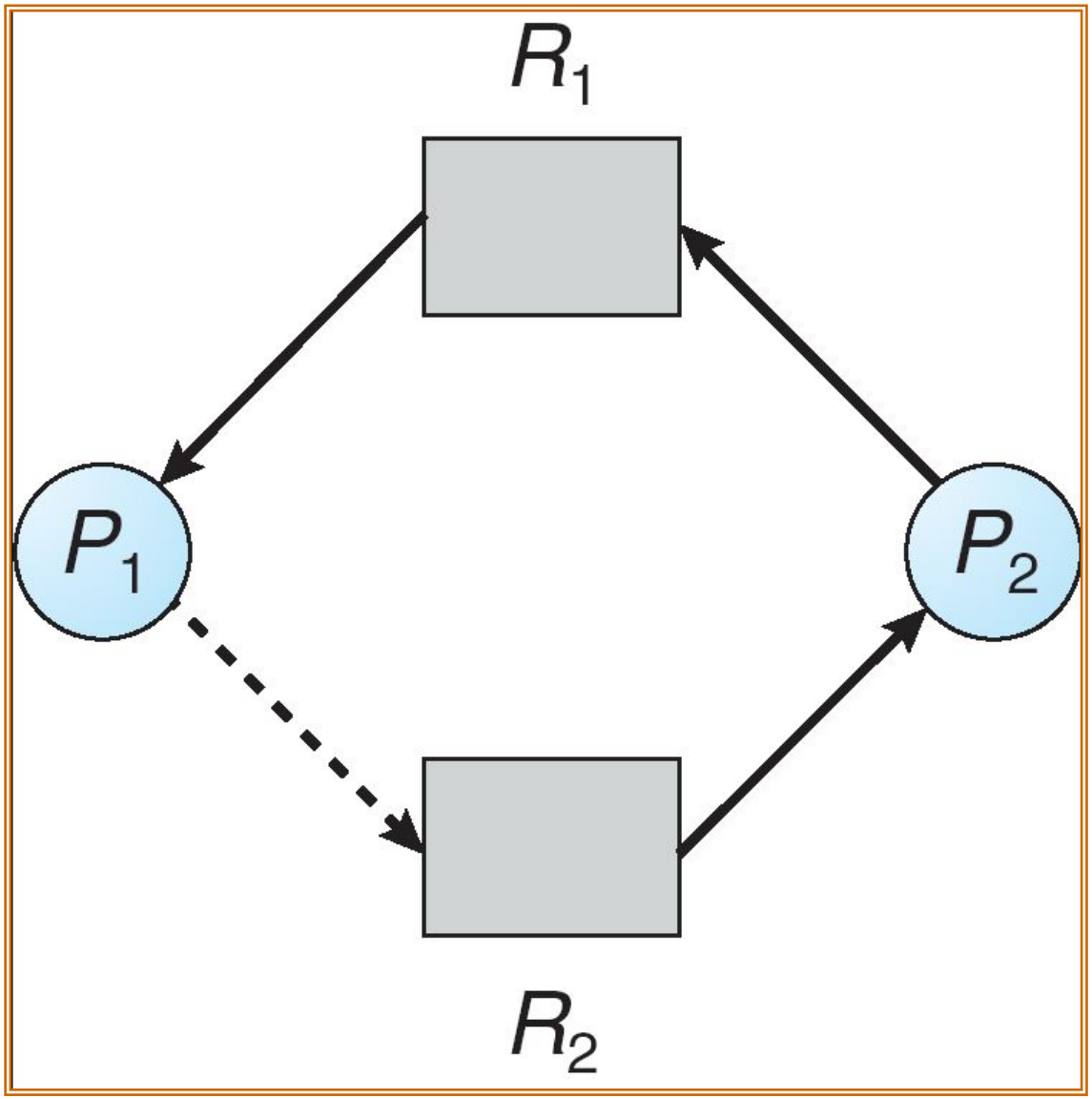
Resource-Allocation Graph Algorithm

- Claim edge Pi → Rj indicated that process Pj may request resource Rj; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed a priori in the system.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- *Available:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
- *Max:* n x m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- *Need:* n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task.

Need [i,j] = Max[i,j] − Allocation [i,j].

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:

   $Work = Available$

   $Finish\ [i] = false$ for $i - 1,2, \cdots, n$.

2. Find an i such that both:

   (a) $Finish\ [i] = false$

   (b) $Need_i \leq Work$

   If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$

   $Finish[i] = true$

   go to step 2.

4. If $Finish\ [i] ==$ true for all i, then the system is in a safe state.

Resource-Request Algorithm for Process Pi

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available = Available - Request_i$;

   $Allocation_i = Allocation_i + Request_i$;

   $Need_i = Need_i - Request_i$;

   - *If **safe** , the resources are allocated to Pi.*
   - *If **unsafe,** Pi must wait, and the old resource-allocation state is restored*

Determination of a Safe State
Initial State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Determination of a Safe State
P2 Runs to Completion

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Determination of a Safe State
P1 Runs to Completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Determination of a Safe State
P3 Runs to Completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Determination of an Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

Determination of an Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

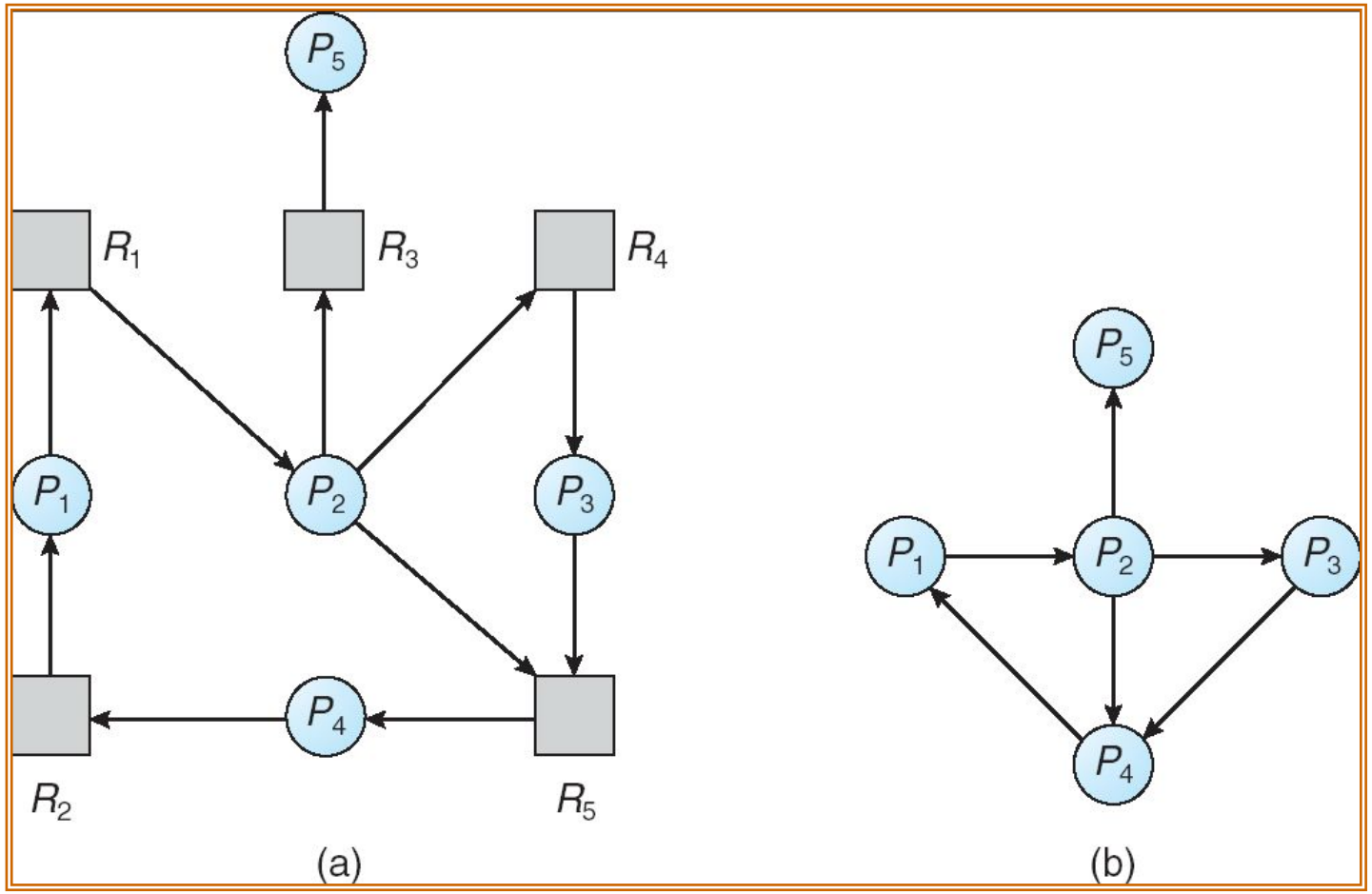
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource–Allocation Graph Corresponding wait–for graph

Several Instances of a Resource Type

- Available: A vector of length m indicates the number of available resources of each type.
- Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.
- Request: An n x m matrix indicates the current request of each process. If Request [ij] = k, then process Pi is requesting k more instances of resource type. Rj.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, ⋯, *n*, if $Allocation_i$鑳 0, then
   Finish[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i$ 《= *Work*

   If no such *i* exists, go to step 4.

Detection Algorithm (Cont.)

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.

4. If $Finish[i] ==$ false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked.

Algorithm requires an order of $O(m \times n^{2})$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4;three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[i] = true for all i.

Example (Cont.)

- P_2 requests an additional instance of type C.

Request

	A	B	C
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.