Comp 310
Computer Systems and Organization

Lecture #11
Process Management
(Classic Semaphores & Deadlocks)

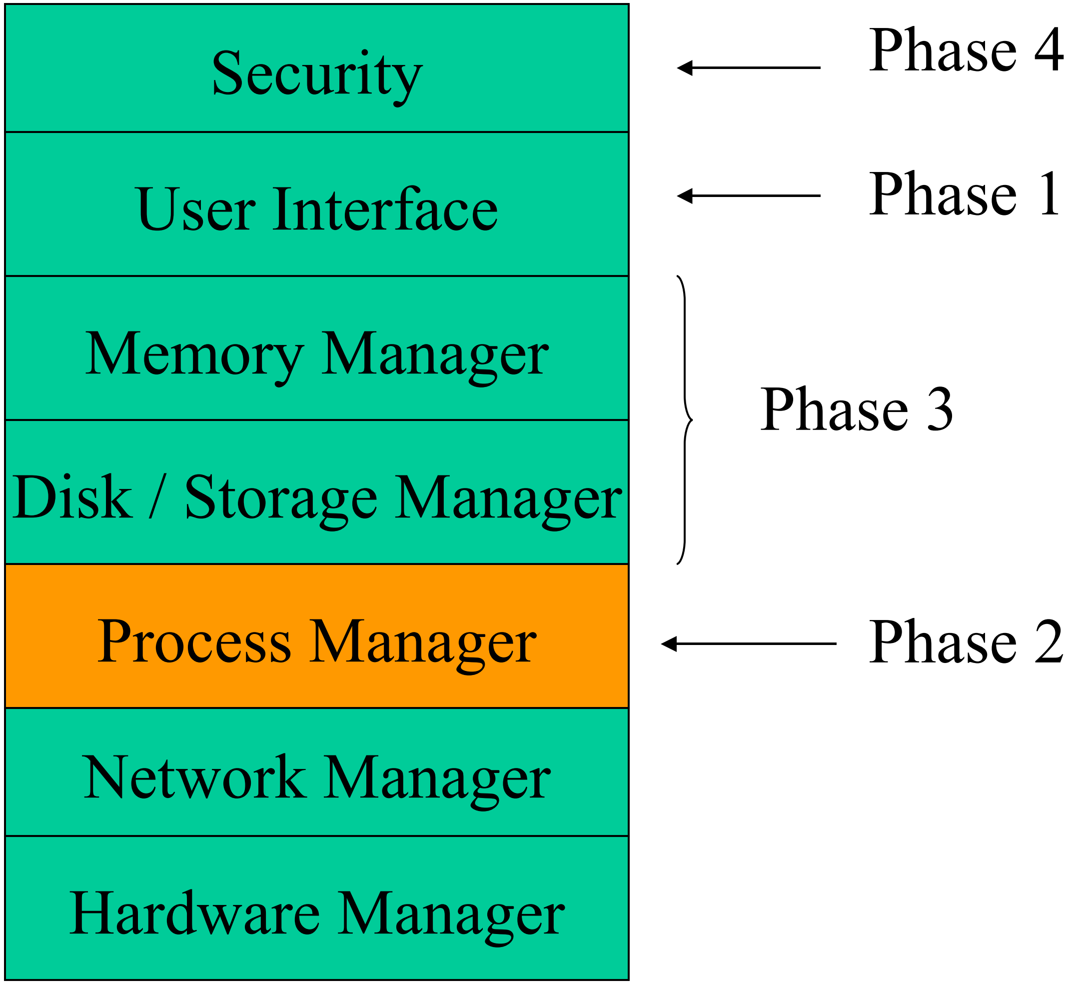
Prof. Joseph Vybihal

Announcements

- Thursday, Oct 16 Midterm exam (in class)
  - Half class review Tuesday, Oct 14
  - Overview of exam next class
- C Tutorial
  - Thursday Oct 9, 2:30-3:30 TR3060
  - Tuesday Oct 14, 10:30-11:30 TR3060
  - Arrays, pointers and malloc/free
- Midterm tutorials
  - Extra office hours

Basic OS Architecture

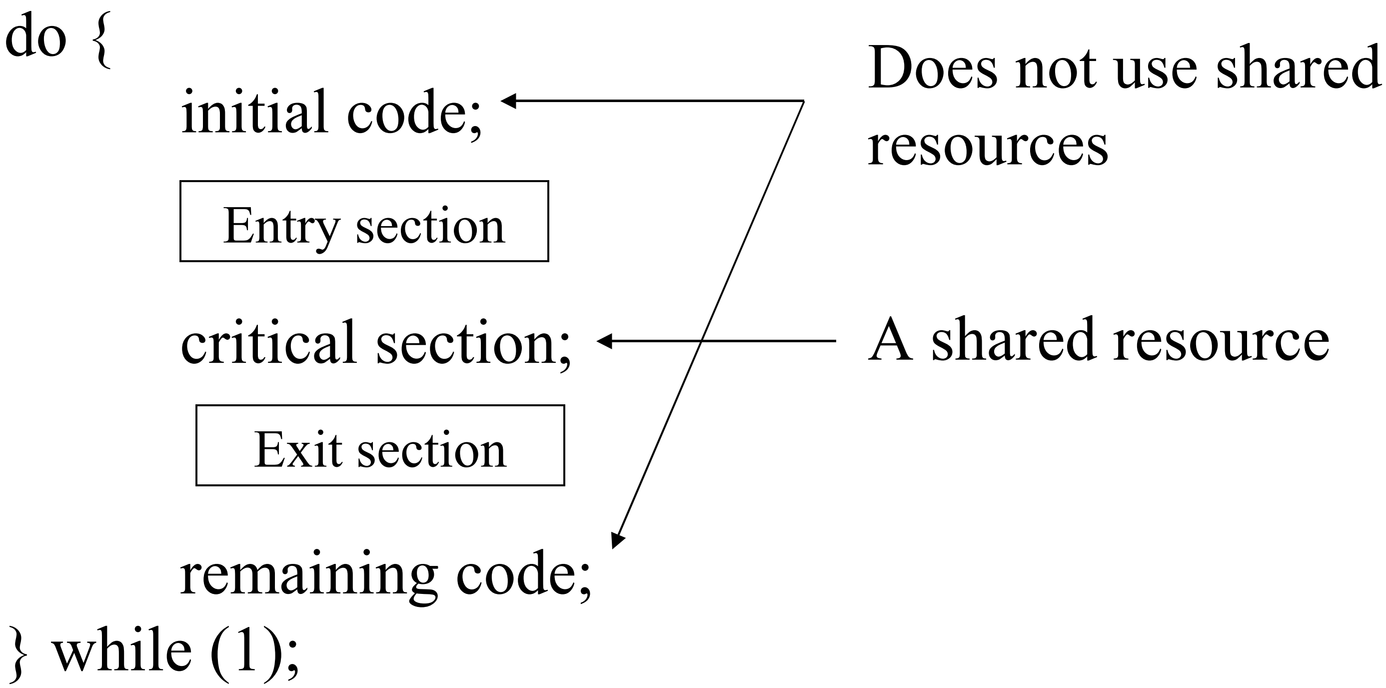
(Course Table of Contents)



Review

The Critical Section Problem

The Critical Section Problem



```
do {
        initial code;
        Entry section
        critical section;
        Exit section
        remaining code;
} while (1);
```

Does not use shared resources

A shared resource

Entry & Exit code guard the critical section:

- Mutual Exclusion: Only 1 Pi can be in the critical section (regardless of quanta)
- Progress: Entry queues requests to use critical section
- Bounded Waiting: Indefinite postponement is not permitted

The 2 Process Solution

Note: since it is a shared var, one process will overwrite the contents of the other, therefore only one will be let through.

Shared vars

```
do {
        initial section;
          flag[i] = true;                        // indicates i wants to enter
          turn = j;                              // does j want to enter?
          while (flag[j] && turn == j);          // controls who enters
        critical section;

          flag[i] = false;                       // I'm done, says Pi

        remaining section;
} while(1);
```

THIS IS PROCESS Pi

Assumed Atomic Instr.

Basic Semaphore Definition

```
wait(S) {
        while (S < 0); // spinlock
        S--;
}
```

Controls # who can get past

```
signal(S) {
        S++;
}
```

S is a shared integer variable initialized to 0.

```
do {
        initial code;
        wait(mutex);
        critical section;
        signal(mutex);
        remaining code;
} while (1);
```

Note: is a shared variable.
Limits who can get in.

Semaphore Use

- S = -1?
- S = 2?
- Spin lock vs. Sleep?
  - Implementation?

Part 1

Classic Semaphore Problems

Practical Uses

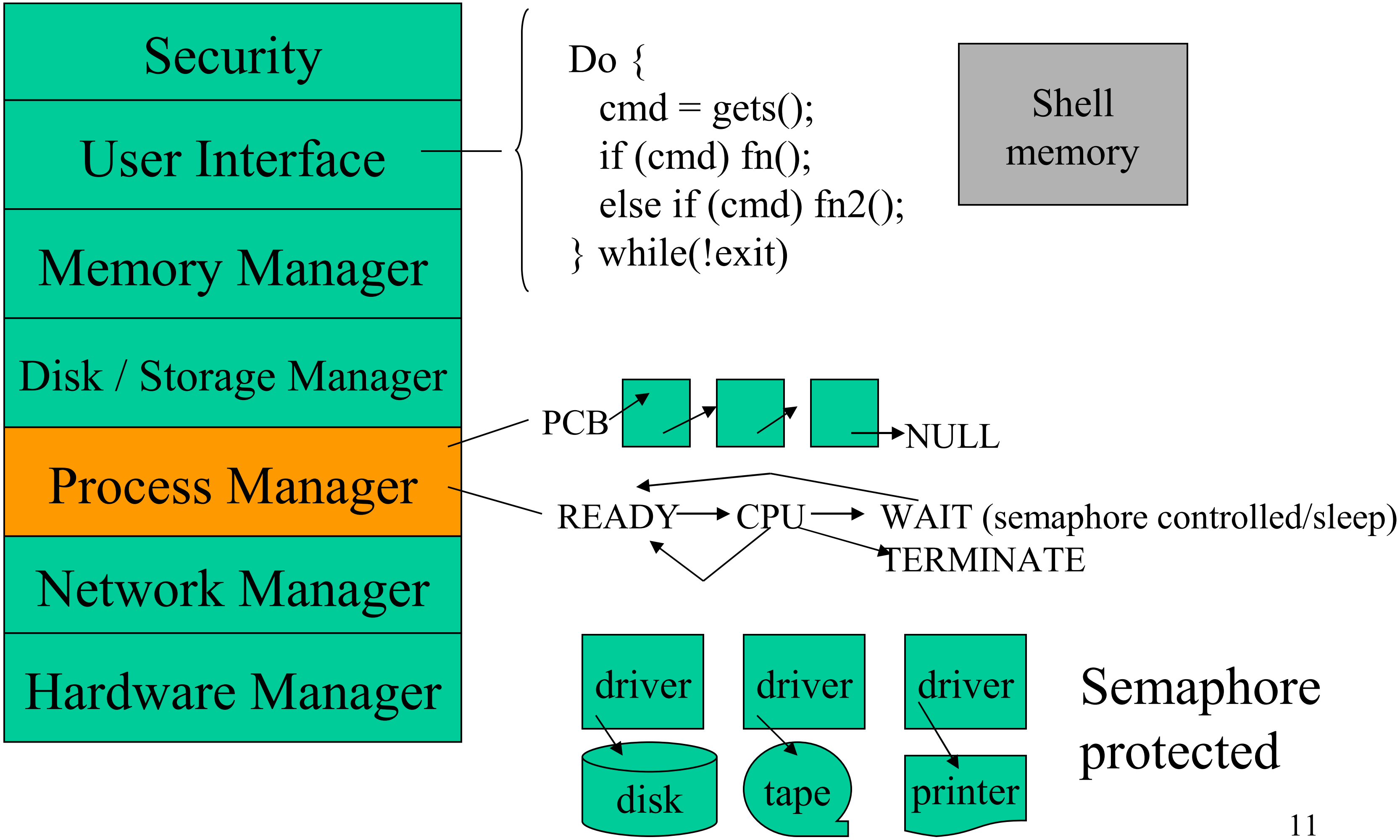
- Memory Buffers
  (Bounding Buffer problem)

- Shared Files / Variables
  (Readers & Writers Problem)

- Limited Resources with Many Processes
  (Dining Philosophers Problem)

How does this fit in?

(OS View)



Security

User Interface

Memory Manager

Disk / Storage Manager

Process Manager

Network Manager

Hardware Manager

```
Do {
   cmd = gets();
   if (cmd) fn();
   else if (cmd) fn2();
} while(!exit)
```

Shell memory

Semaphore protected

How does this fit in?

(Programmer View)

FILE *ptr;

ptr = fopen(“file.txt”,”rt”); → Semaphore to File & Buffer?

fscanf(ptr, “%d”, &d); → Semaphore access to buffer (critical section)?

Note:

- *fopen* is the true / physical critical section, since at this point we must ask for use of HDD and the creation of a buffer to read contents of file.
- *fscanf* will actually execute locally with buffer, since ptr points to buffer, until it is empty, then the semaphore kicks in and more data overwrites buffer from file.

The Bounded-Buffer Problem

- Synchronization in one direction
  - Like sending data to a printer
- Only one program should have access at any time
  - When P1 sending, no other Pi can send
  - The resource must consume until P1 is done

The Bounded-Buffer Problem

The Producer Process

Init:
- mutex = 0
- empty = n, full = -1

```
do {
      :
      // produce an item in NEXTP
      :

      wait(empty);
      wait (mutex);

      :
      // add NEXTP to buffer
      :

      signal(mutex);
      signal(full);

} while(1);
```

Access to a cell of the array (is full?)

Mutual exclusion

Access to buffer[n]

Is empty?

The Bounded-Buffer Problem

The Consumer Process

Init:
- From previous

```
do {
      wait(full);
      wait (mutex);

      :
      // remove an item from buffer to NEXTC
      :

      signal(mutex);
      signal(empty);

      :
      // consume the item in NEXTC
      :

} while(1);
```

Access to a cell of the array (is full?)

Mutual exclusion

Access to buffer[n]

Is empty?

The Readers-Writers Problem

- Sharing a resource (e.g. file)
- Many readers can read at the same time without problem
- But:
  - Writers cannot write at the same time, and
  - Readers cannot read while someone is writing

The Readers-Writers Problem

The Writer Process

```
// shared data structures

semaphore mutex=0, wrt=0;
int readcount=0;
:
:
wait(wrt);
:
// writing is performed
:
signal(wrt);
:
```

wait(wrt); → Only 1 Pi can write at any time

The Readers-Writers Problem

The Reader Process

```
// shared data structures

semaphore mutex=0, wrt=0;
int readcount=0;
:
wait(mutex);
readcount++;
if (readcount == 1) wait(wrt);
signal(mutex);
:
// reading is performed
:
wait(mutex);
readcount--;
if (readcount == 0) signal(wrt);
signal(mutex);
```

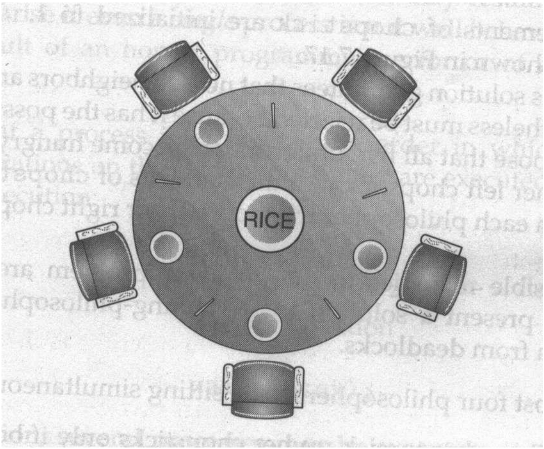
How many are reading?

If one Pi is reading then no one can write & if a Pj is writing then we halt

Only one P can access at any time

The Dining-Philosophers Problem

- Classic problem in sharing more than once resource with multiple processes. Do not want:
  - Starvation, and
  - Deadlock



- Thinking and eating
- Bowl of rice
- 5 single chop sticks
- when thinking, executing locally
- when hungry must access 2 chop sticks (wait or eat)
- when finished, drop chop sticks and think

Problematic Solution?

Deadlock when all hungry!

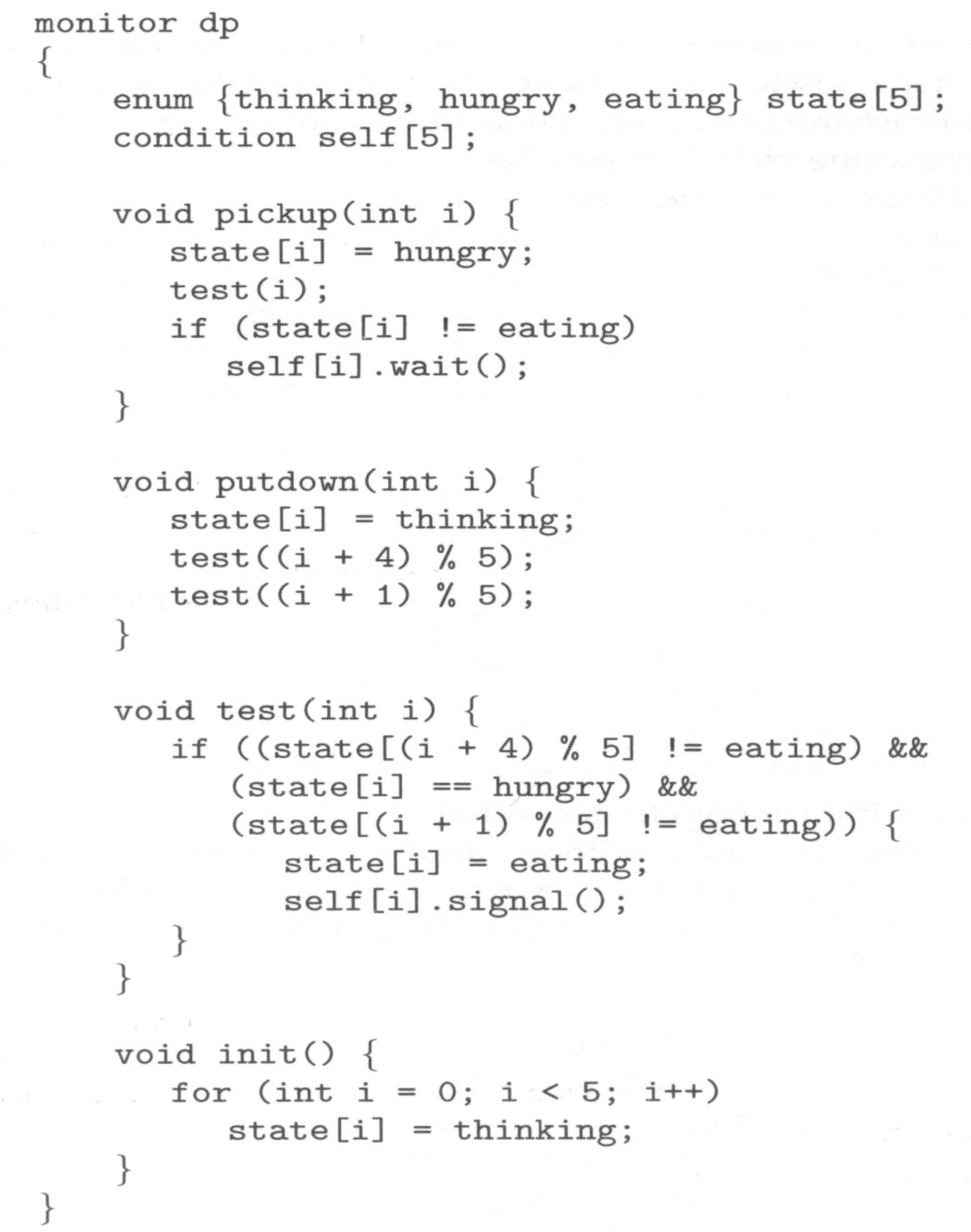
```
semaphore chopstick[5] = {0, 0, 0, 0, 0};
:
do {
      :
      wait(chopstick[i]);
      wait(chopstick[(i+1) % 5)];
      :
      // eat
      :
      signal(chopstick[i]);
      signal(chopstick[(i+1) % 5)];
      :
      // think
      :
} while(1);
```

How does this work?

Upgrades to solution…

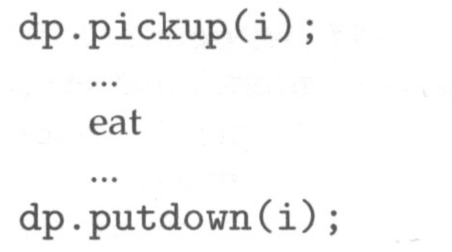
- Run with n-1 philosophers
- Pickup a chopstick iff both available (i.e. must be done in a critical section)
- Odd philosopher pick left, even pick right

Solution with Monitors



```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];

    void pickup(int i) {
       state[i] = hungry;
       test(i);
       if (state[i] != eating)
          self[i].wait();
    }

    void putdown(int i) {
       state[i] = thinking;
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

    void test(int i) {
       if ((state[(i + 4) % 5] != eating) &&
          (state[i] == hungry) &&
          (state[(i + 1) % 5] != eating)) {
             state[i] = eating;
             self[i].signal();
       }
    }

    void init() {
       for (int i = 0; i < 5; i++)
          state[i] = thinking;
    }
}
```

How does this work?



```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```

Supportive Routines

To call a monitor:

```
wait(mutex);
:
call
:
if (next_count > 0)
        signal(next);
else
        signal(mutex);
```

Init:

next=-1

mutex=0

x_sem=-1

x_count=0

next_count=0

x.wait() is implemented as:

```
x_count++;
if (next_count >0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
c_count--;
```

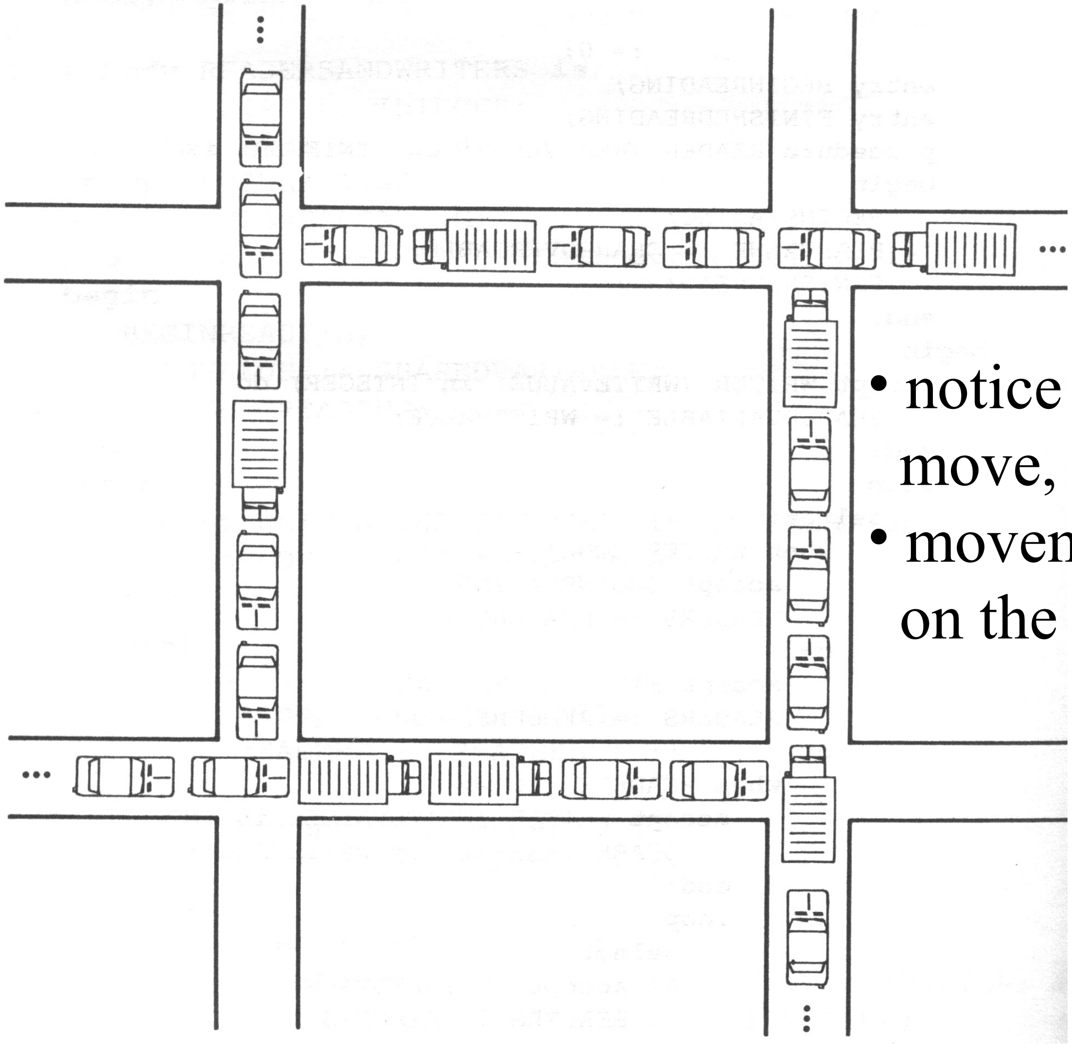
x.signal() is implemented as:

```
if (x_count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
}
```

Part 2

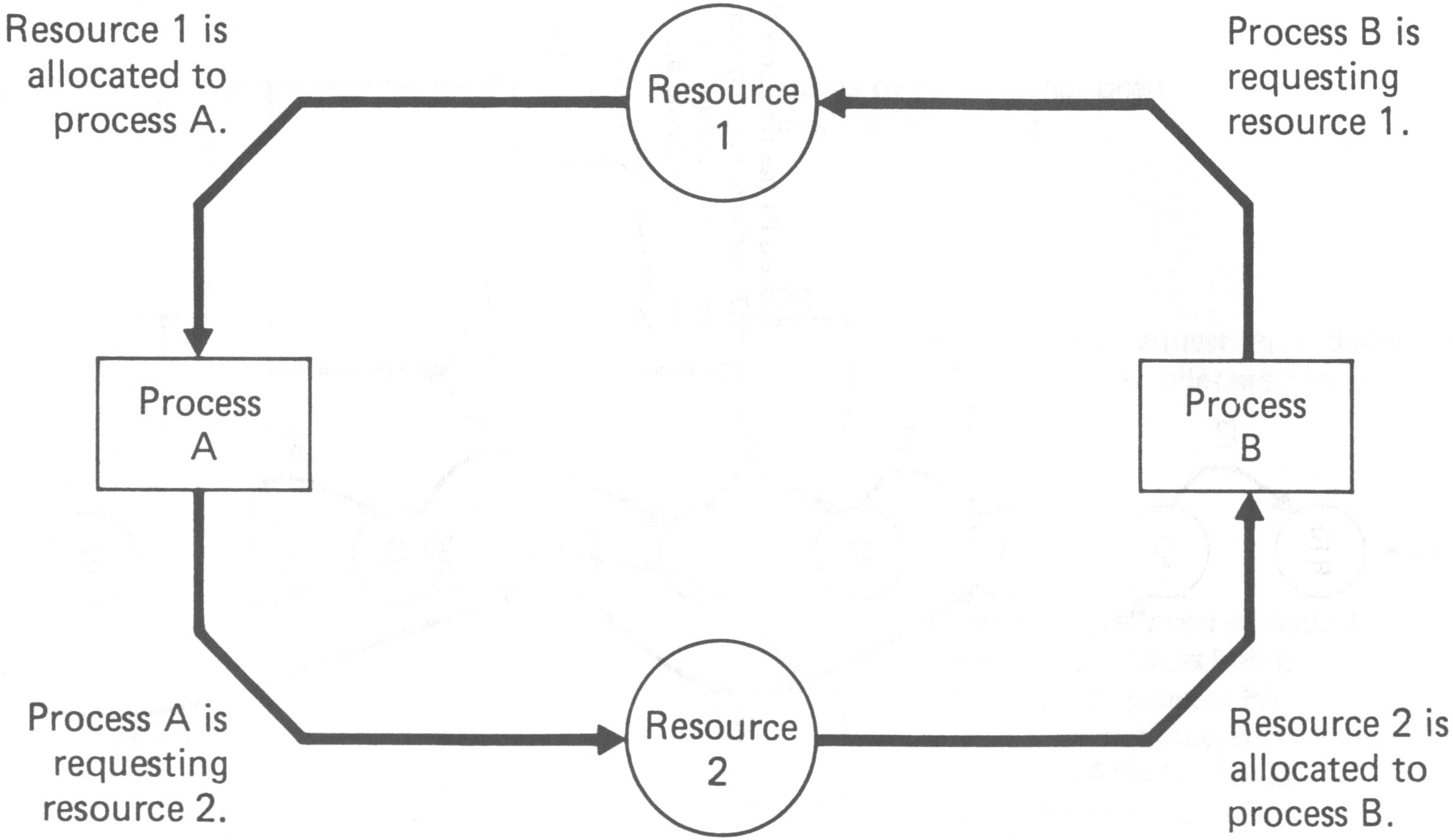
Introduction to Deadlocks

A Deadlock



- notice nothing can move, and
- movement depends on the others

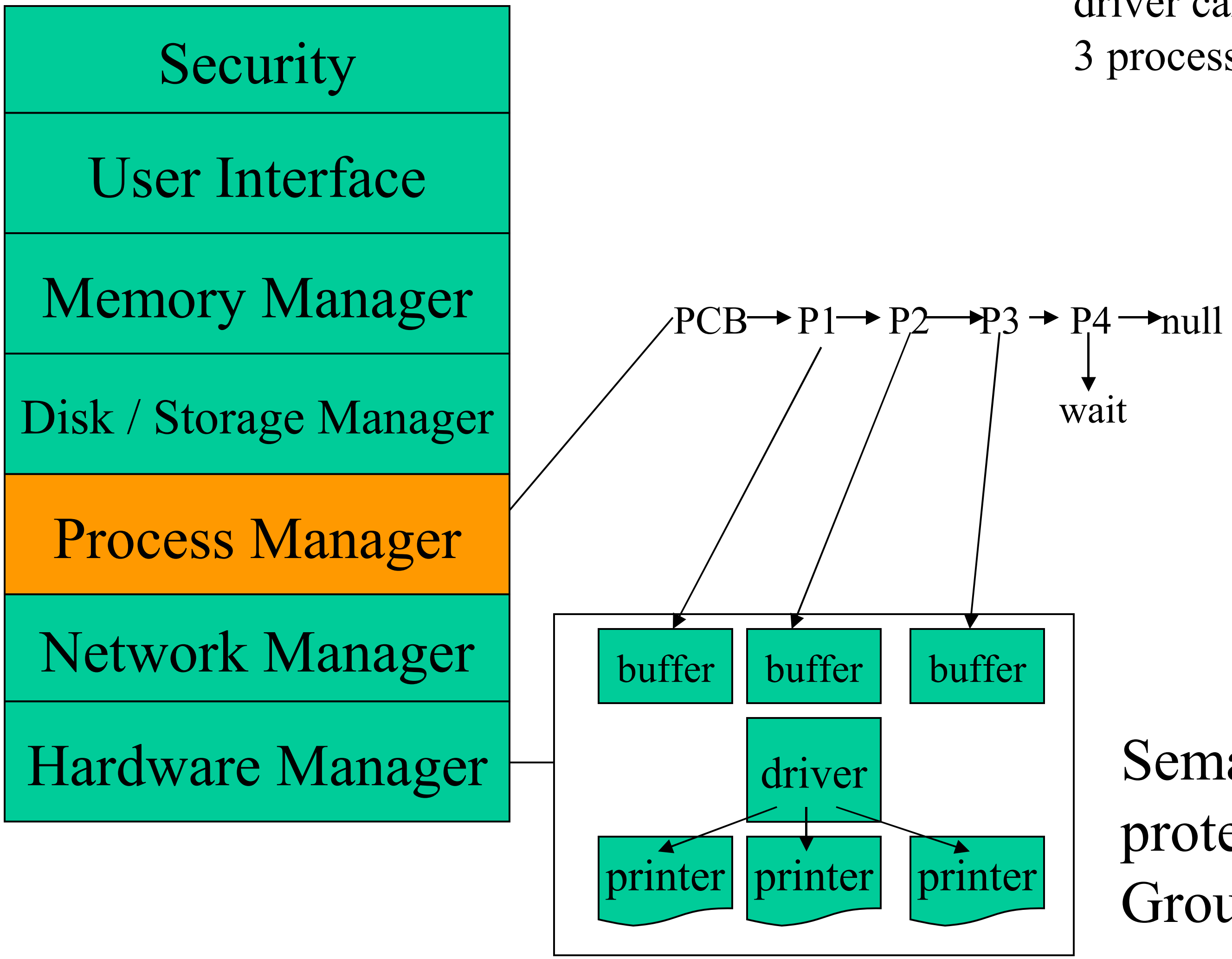
Deadlocks Based on Resources



Just like the traffic problem

Resource System Model

The common resource
driver can be spawned as
3 processes for each buffer



Security

User Interface

Memory Manager

Disk / Storage Manager

Process Manager

Network Manager

Hardware Manager

PCB → P1 → P2 → P3 → P4 → null

wait

buffer

buffer

buffer

driver

printer

printer

printer

Semaphore
protected resource
Group

Resource Sharing

- Want to share multiple resource with many processes without causing:
  - Deadlock, nor
  - Starvation
- Resource utilization:
  - Request
    - If it cannot be granted immediately the process must wait until the resource is available.
  - Use
    - The process has dedicated access to the resource until it is finished
  - Release
    - The process indicates that the resource can be used by another
- Examples of:
  - Request / Release device
  - Open / Close file
  - Malloc / free memory

Deadlock Basics

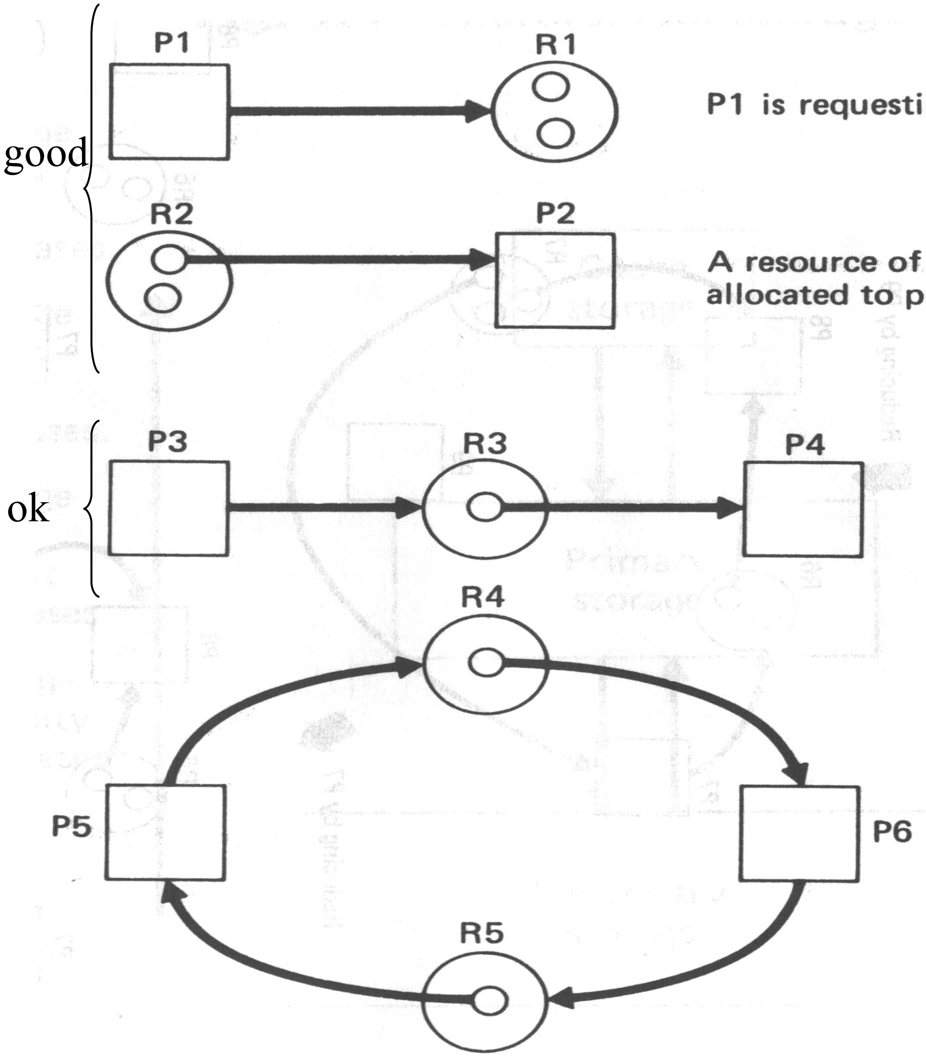
- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.
- In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting and finishing.

Necessary Conditions

- Mutual Exclusion
- Hold and Wait
  - A process must be holding at least one resource and waiting to acquire another that is being held
- No pre-emption
  - The OS does not permit pre-emption of held resources
- Circular wait
  - {P1, P2, P3} s.t. P1→ P2 → P3 → P1

Resource Diagnosis

(VIA Resource-Allocation Graphs)



P1 is requesting a resource of type R1.

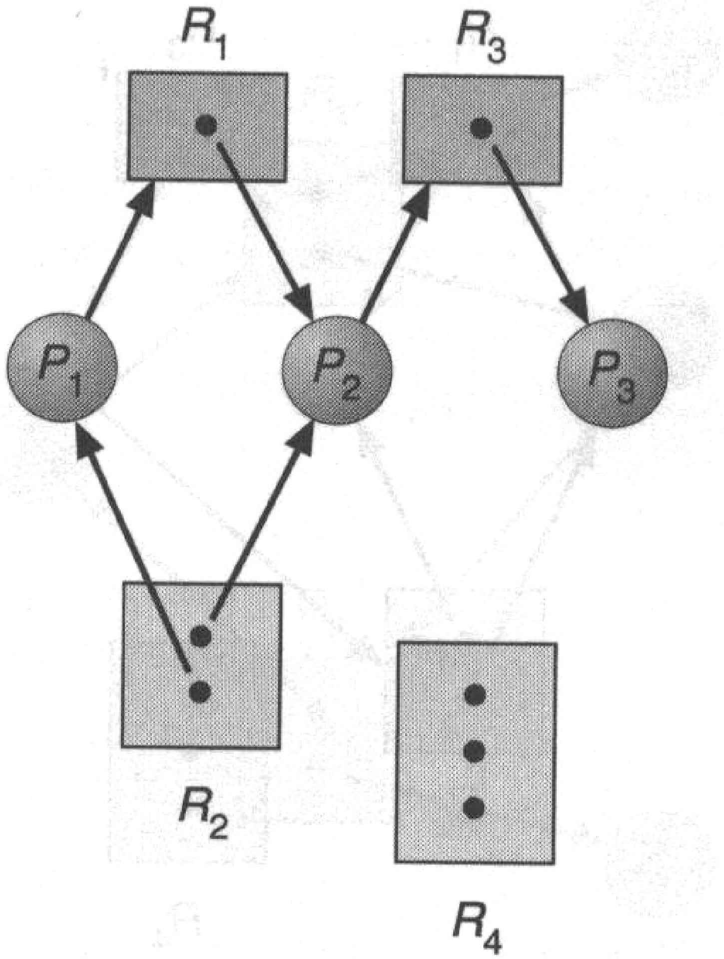
A resource of type R2 has been allocated to process P2.

Process P3 is requesting resource R3, which has been allocated to process P4.

Process P5 has been allocated resource R5 that is being requested by process P6 that has been allocated resource R4 that is being requested by process P5 (the "circular wait").

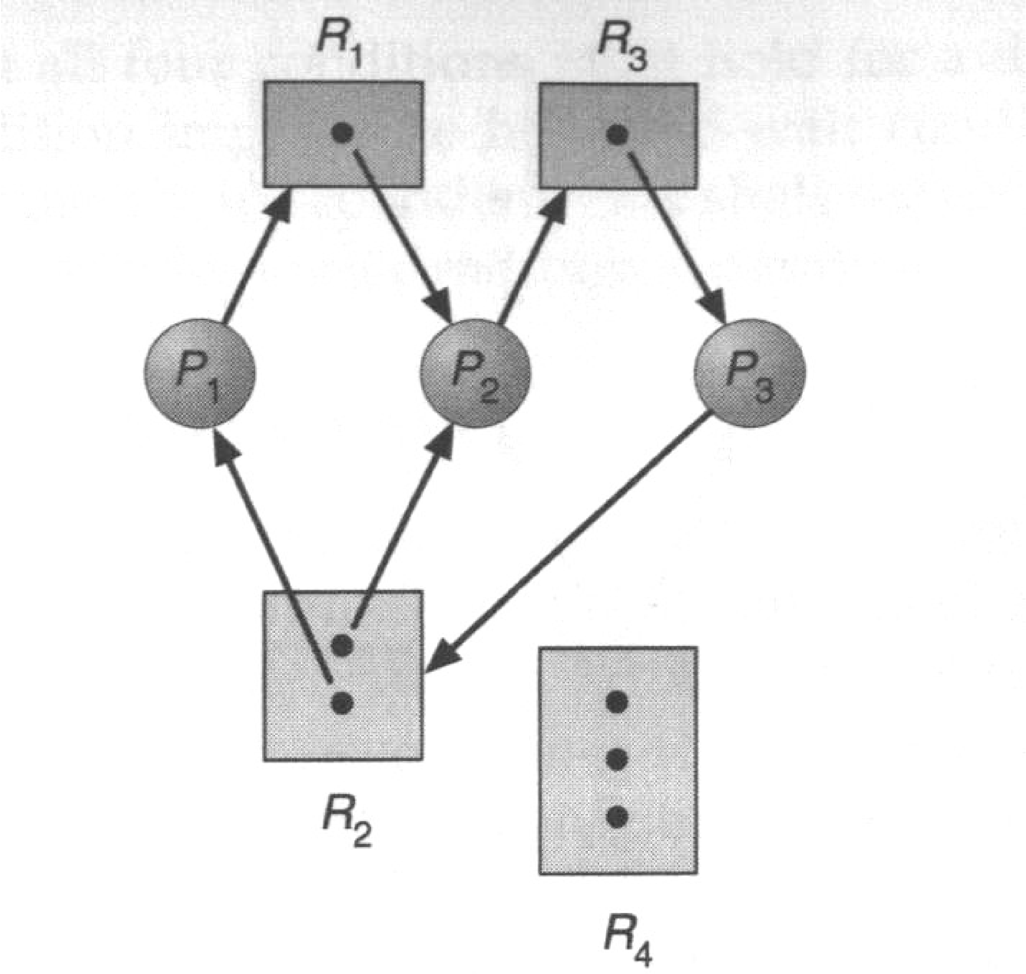
Is There a Deadlock?

1 of 3

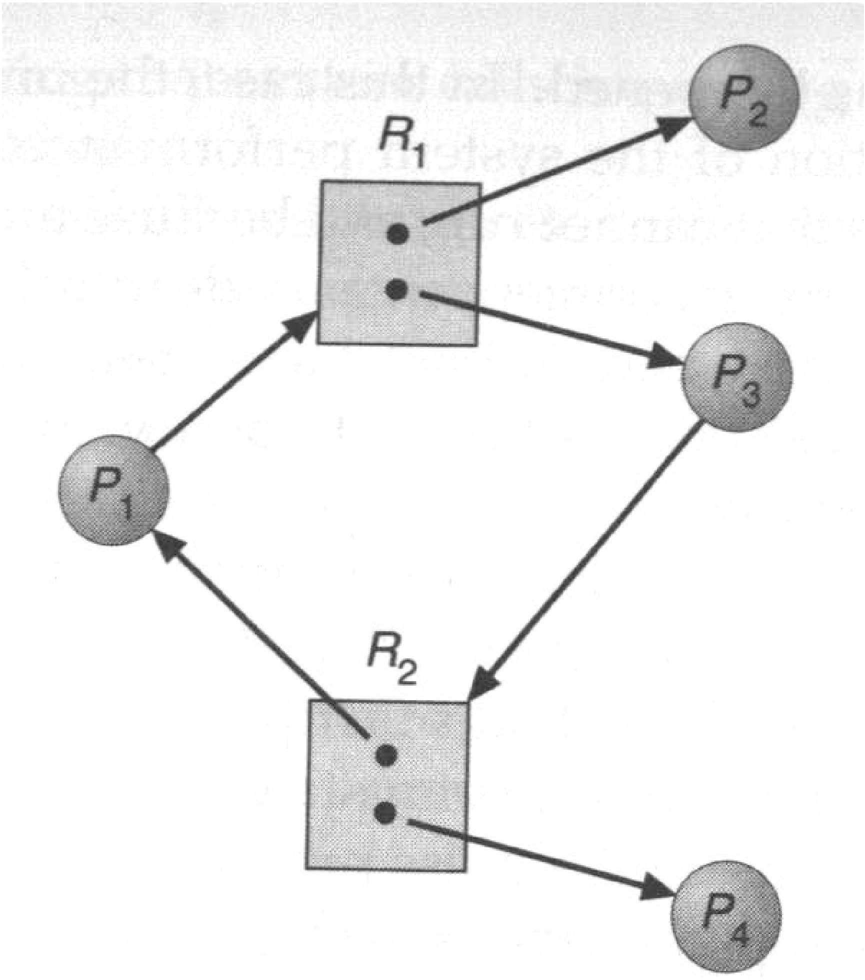


Is There a Deadlock?

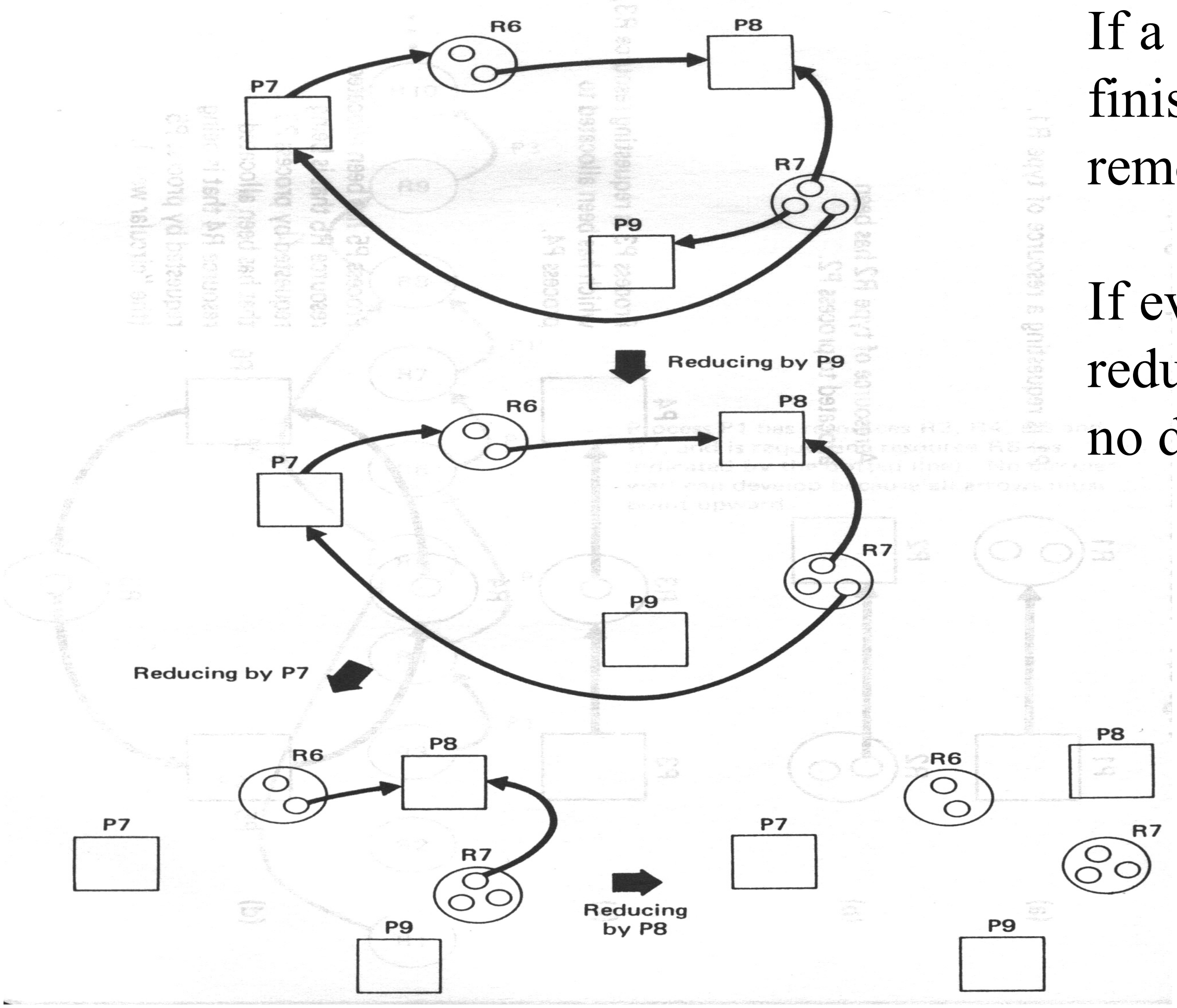
2 of 3



Is There a Deadlock? 3 of 3



Reducing Resource-Allocation Graphs



If a process can finish then remove its arcs

If everything reduces then no deadlock

Questions

To overcome the deadlock problems in an OS:

- When should the OS invoke resource allocation?
- How/where could it be implemented?

Part 3

At Home

Things to try out

1. Try to produce a resource allocation graph that has a loop in it but does not cause a deadlock.

2. Internet Resources:
    1. http://www.cs.wcupa.edu/~rkline/OS/Deadlock.html
    2. http://cgi.cse.unsw.edu.au/~cs3231/04s2/labs/threads/index.php?session=06s1