CSE 210 Operating Systems

Module3 - Process Synchronization and Deadlocks

The Critical-Section Problem- Peterson's Solution- Semaphores, Classic Problems of Synchronization, Monitors.

Introduction to Deadlocks, Necessary conditions for deadlock, deadlock Characterization, Deadlock Prevention- Deadlock Avoidance- Deadlock detection & recovery from Deadlock.

Process Synchronization

- Background, Race Condition
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples

Process Sync. Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers.

- Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */


        while (counter == BUFFER_SIZE); /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0); /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** may be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** may be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Race Condition

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 – 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

- The value of **count** may be either 4 or 6, where the correct result should be 5.

Race Condition

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

Race Condition

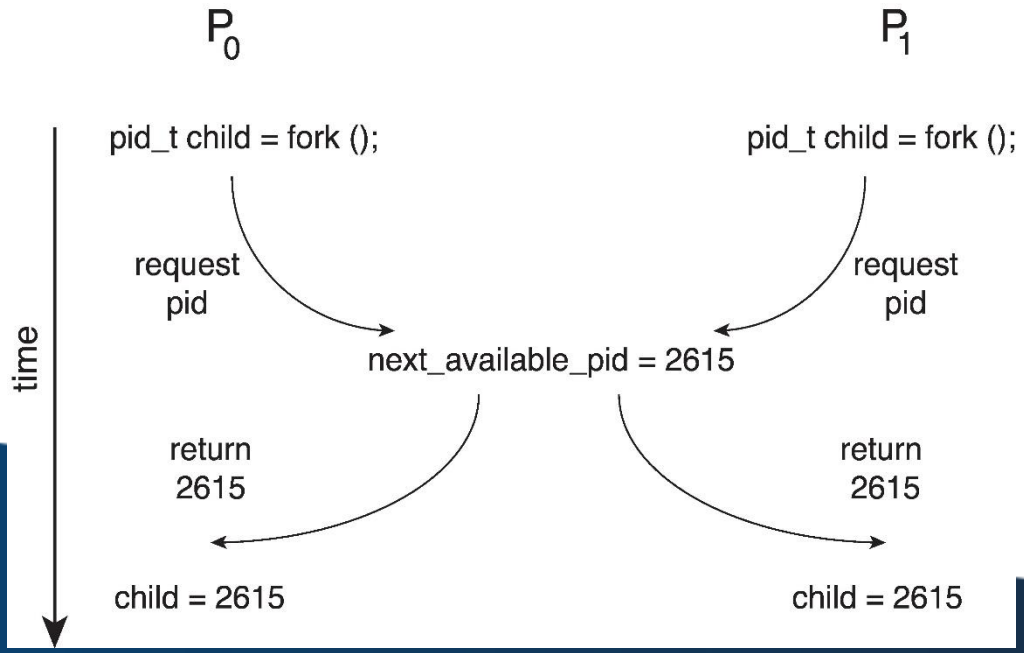
- The statements
  - **counter++;**
  - **counter--;**
  - must be performed *atomically*.
- Atomic operation means an operation that completes in its entirety without interruption.

Race Condition

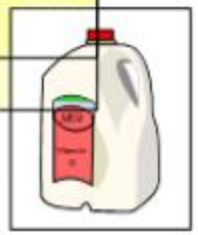
- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

Race Condition

- Processes P_0 and P_1 are creating child processs using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!



Example of Race: The ATM "Withdraw at the same time from a joint account" Problem

Balance = $300

ATM -1	**ATM-2**
$300 ←Check Balance	
	Check Balance → $300
??? ←Withdraw $300	**Withdraw $300 → ???**

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Initial Attempt to Solve the problem

- Only 2 processes, *P*0 and *P*1
- General structure of process *Pi* (other process *Pj*)

do {

entry section

critical section

exit section

reminder section

} **while (1)**;

- Processes may share some common variables to synchronize their actions.

Algorithm 1

Shared variables:

int turn;
initially **turn = 0**

turn - i $\Rightarrow$ *Pi* can enter its critical section

Process *Pi*

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        reminder section
} while (1);
```

Process Pj

```
do {
    while (turn != j) ;
        critical section
    turn = i;
        reminder section
} while (1);
```

Satisfies mutual exclusion, but not progress

Pi, Pj, Pi, Pj… strict alternation of processes

Pi leaves, Pj busy with long I/O, Pi comes back to CS-entry;

No one in the CS, but Pi has to wait until Pj to come to the CS

What if Pj never comes back to CS ????

Algorithm 2

Shared variables

boolean flag[2];
initially **flag [0] = flag [1] = false.**

flag [i] = true $\Rightarrow$ *Pi* ready to enter its critical section

Process *Pi*

```
do {
    flag[i] := true;
    while (flag[j]) ;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

Process *Pj*

```
do {
    flag[j] := true;
while (flag[i]) ;
    critical section
    flag [j] = false;
    remainder section
} while (1);
```

Satisfies mutual exclusion and progress, but not bounded waiting requirement.

Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted

Two process solution

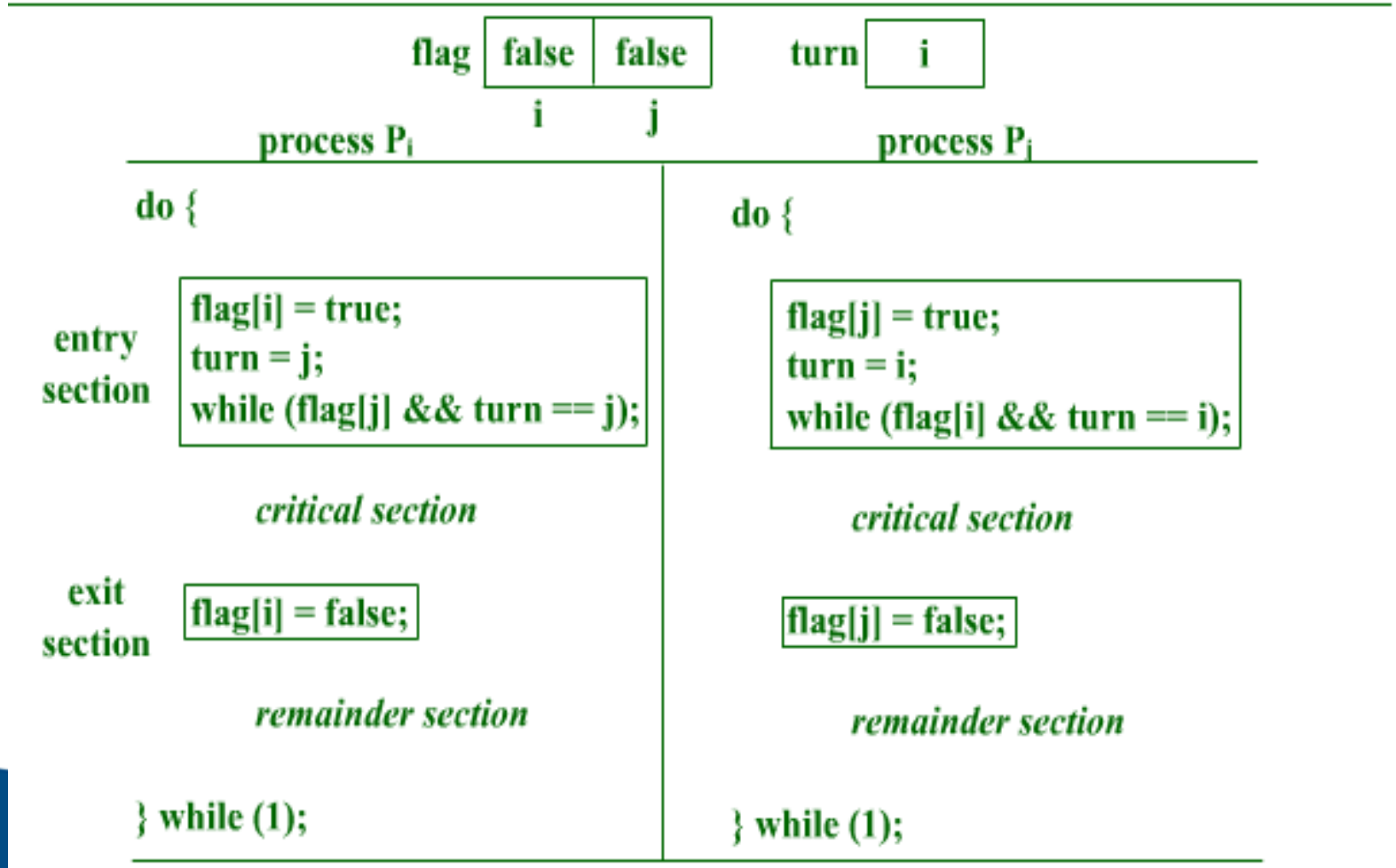
- Assume that the LOAD and STORE instructions are atomic

- The two processes share two variables:
  - int turn;
  - Boolean flag[2] Initially: flag[0]=flag[1]= false

- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i, P_j,

The Critical Section Problem:



Algorithm 3 - cont.

- Claim: Mutual exclusion is preserved
- Proof:
  - Assume Pi and Pj in CS at the same time
  - flag[i] & flag[j] == True
  - (I) flag[j] & turn == j → False → turn = i
  - (II) flag[i] & turn == i → False → turn = j
  - turn cannot be i and j at the same time

Algorithm 3- cont.

- Claim: Progress is met.
- Proof:
  - Pi stuck at "flag[j] == true & turn == j" while loop
  - (case 1) Pj is not ready to enter
    - flag[j] == false → Pi can enter
  - (case 2) Pj is ready to enter
    - Pj : set flag[j] to true and at its while
    - Observation: turn == i XOR turn == j
    - (case 2.1) turn == i → Pi will enter
    - (case 2.2) turn == j → Pj will enter
      - (case 2.2.1) Pj leaves CS; sets flag[j] to false → Pi enters
      - (case 2.2.2) Pj leaves CS; sets flag[j] to true ;
        - then sets turn to i → Pi enters
        - (Pi at while cannot change turn)
- Pi (Pj) will enter the CS (progress) after at most one entry by Pj (Pi) (bounded waiting)

Peterson's Algorithm

- Less complicated than Dekker's Algorithm
  - Still uses busy waiting.
  - Requires fewer steps to perform mutual exclusion primitives
  - Easier to demonstrate its correctness

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */

 } while (true);
```

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
 while (compare_and_swap(&lock, 0, 1) != 0)
   ; /* do nothing */
 /* critical section */
lock = 0;
 /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

Boolean waiting[n]; and Boolean lock are set to false.

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)

      key = test_and_set(&lock);

   waiting[i] = false;

   /* critical section */

   j = (i + 1) % n;

   while ((j != i) && !waiting[j])

      j = (j + 1) % n;

   if (j == i)

      lock = false;

   else

      waiting[j] = false;

   /* remainder section */

} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

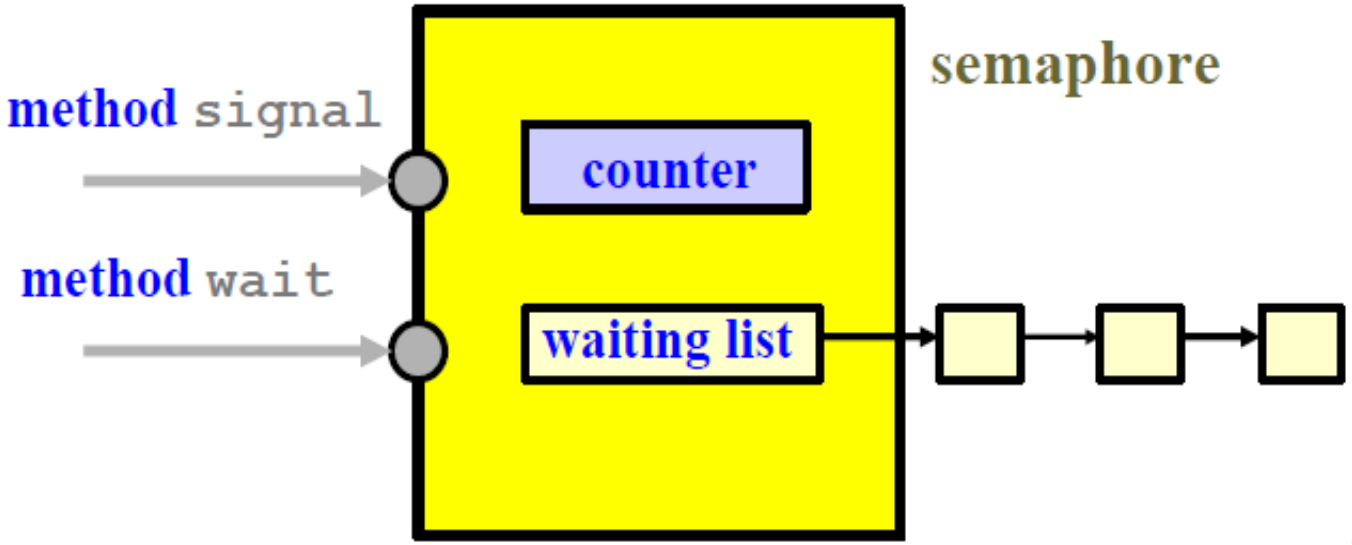
acquire() and release()

- ```
  acquire() {
      while (!available)
          ; /* busy wait */
      available = false;
  }
  ```
- ```
  release() {
      available = true;
  }
  ```
- ```
  do {
      acquire lock
      critical section
      release lock
      remainder section
  } while (true);
  ```
  ```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

❑ A *semaphore* is an object that consists of a counter, a waiting list of processes and two methods (*e.g.*, functions): `signal` and `wait`.



Semaphore Method: wait

```
void wait(sem S)
{
    S.count--;
    if (S.count < 0) {
        add the caller to the waiting list;
        block();
    }
}
```

- After decreasing the counter by 1, if the counter value becomes negative, then
  - add the caller to the waiting list, and then
  - block itself.

Semaphore Method: signal

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume(P);
    }
}
```

- After increasing the counter by 1, if the new counter value is not positive, then
  - remove a process P from the waiting list,
  - resume the execution of process P, and return

Important Note: 1/4

```
S.count--;                    S.count++;
if (S.count<0) {              if (S.count<=0) {
    add to list;                  remove P;
    block();                      resume(P);
}                             }
```

- If `S.count < 0`, `abs(S.count)` is the number of waiting processes.
- This is because processes are added to (*resp.*, removed from) the waiting list only if the counter value is `< 0` (*resp.*, `<= 0`).

Important Note: 2/4

```
S.count--;                    S.count++;
if (S.count<0) {              if (S.count<=0) {
    add to list;                  remove P;
    block();                      resume(P);
}                             }
```

- The waiting list can be implemented with a queue if FIFO order is desired.
- However, the correctness of a program should not depend on a particular implementation of the waiting list.
- Your program should not make any assumption about the ordering of the waiting list.

Important Note: 3/4

```
S.count--;
if (S.count<0) {
    add to list;
    block();
}
```

```
S.count++;
if (S.count<=0) {
    remove P;
    resume(P);
}
```

- The caller may be blocked in the call to `wait()`.
- The caller never blocks in the call to `signal()`. If `S.count > 0`, `signal()` returns and the caller continues. Otherwise, a waiting process is released and the caller continues. In this case, *two* processes continue.

The Most Important Note: 4/4

```
S.count--;                      S.count++;
if (S.count<0) {                if (S.count<=0) {
    add to list;                    remove P;
    block();                        resume(P);
}                               }
```

- `wait()` and `signal()` must be executed *atomically* (*i.e.*, as one uninterruptible unit).
- Otherwise, *race conditions* may occur.
- Homework: use execution sequences to show race conditions if `wait()` and/or `signal()` is not executed atomically.

Three Typical Uses of Semaphores

- There are three typical uses of semaphores:
  - **mutual exclusion**:

    Mutex (*i.e.*, *Mut*ual *Ex*clusion) locks
  - **count-down lock**:

    Keep in mind that semaphores have a counter.
  - **notification**:

    Indicate an event has occurred.

Use 1: Mutual Exclusion (Lock)

initialization is important

```
semaphore S = 1;
int       count = 0;
```

Process 1

```
while (1) {
   // do something
   S.wait();
      count++;
   S.signal();
   // do something
}
```

Process 2

```
while (1) {
   // do something
   S.wait();
      count--;
   S.signal();
   // do something
}
```

entry

critical sections

exit

- What if the initial value of S is zero?
- S is a *binary semaphore* (similar to a *lock*).

Use 2: Count-Down Counter

```
semaphore  S = 3
```

Process 1

```
while (1) {
    // do something
    S.wait();
    // do something
    S.signal();
    // do something
}
```

Process 2

```
while (1) {
    // do something
    S.wait();
    // do something
    S.signal();
    // do something
}
```

at most 3 processes can be here!!!

- After three processes pass through `wait()`, this section is locked until a process calls `signal()`.

Use 3: Notification

```
semaphore S1 = 1, S2 = 0;
```

process 1

```
while (1) {
    // do something
    S1.wait();
    cout << "1";
    S2.signal();
    // do something
}
```

process 2

```
while (1) {
    // do something
    S2.wait();
    cout << "2";
    S1.signal();
    // do something
}
```

notify

notify

- Process 1 uses `S2.signal()` to notify process 2, indicating "I am done. Please go ahead."
- The output is `1 2 1 2 1 2 .....`
- What if both `S1` and `S2` are both 0's or both 1's?

CSE 210 Operating Systems

Module3 - Process Synchronization and Deadlocks

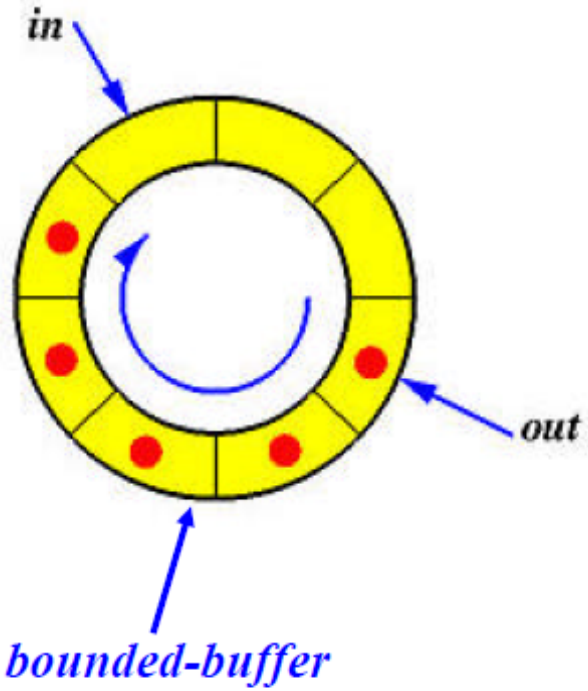
The Critical-Section Problem- Peterson's Solution- Semaphores, Classic Problems of Synchronization, Monitors.

Introduction to Deadlocks, Necessary conditions for deadlock, deadlock Characterization, Deadlock Prevention- Deadlock Avoidance- Deadlock detection & recovery from Deadlock.

Classical Problems of Synchronization

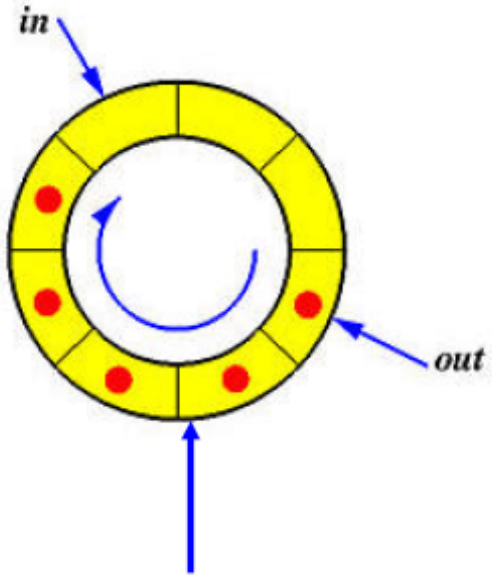
- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem

The Producer/Consumer Problem

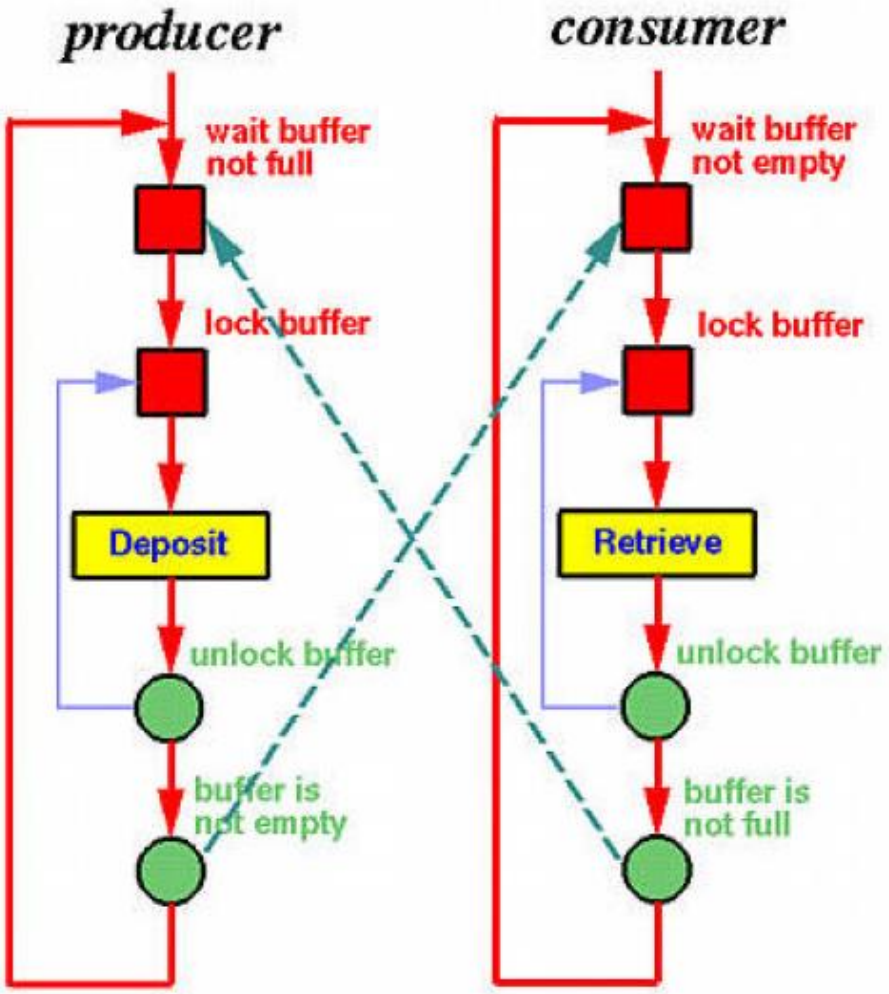


- Suppose we have a circular buffer of *n* slots.
- Pointers *in* (*resp.*, *out*) points to the first empty (*resp.*, filled) slot.
- Producer processes keep adding info into the buffer
- Consumer processes keep retrieving info from the buffer.

Problem Analysis



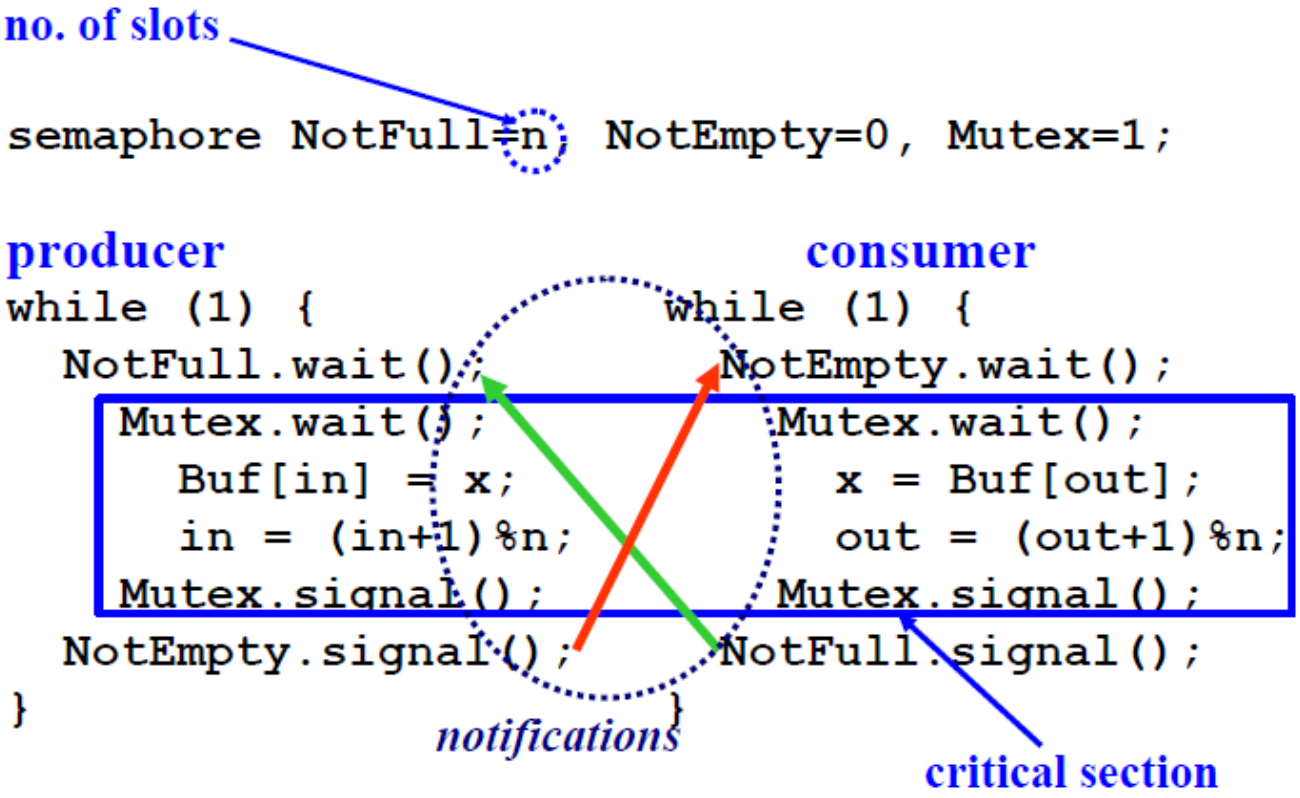
- A producer deposits info into `Buf[in]` and a consumer retrieves info from `Buf[out]`.
- `in` and `out` must be advanced.
- `in` is shared among producers.
- `out` is shared among consumers.
- If `Buf` is full, producers should be blocked.
- If `Buf` is empty, consumers should be blocked.



- We need a sem. to protect the buffer.
- A second sem. to block producers if the buffer is full.
- A third sem. to block consumers if the buffer is empty.

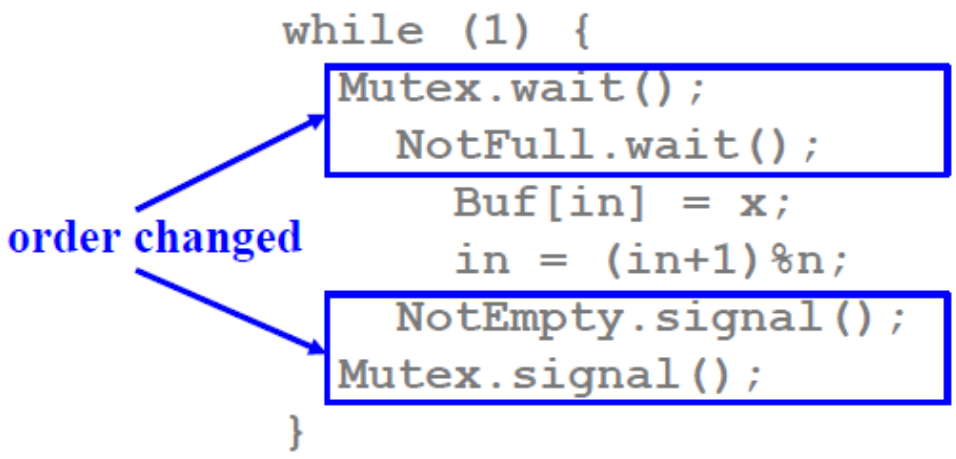
22

Solution


no. of slots
semaphore NotFull=n, NotEmpty=0, Mutex=1;
producer
consumer
while (1) {
NotFull.wait();
Mutex.wait();
Buf[in] = x;
in = (in+1)%n;
Mutex.signal();
NotEmpty.signal();
}
while (1) {
NotEmpty.wait();
Mutex.wait();
x = Buf[out];
out = (out+1)%n;
Mutex.signal();
NotFull.signal();
}
notifications
critical section

Question

- What if the producer code is modified as follows?
- Answer: a deadlock may occur. Why?

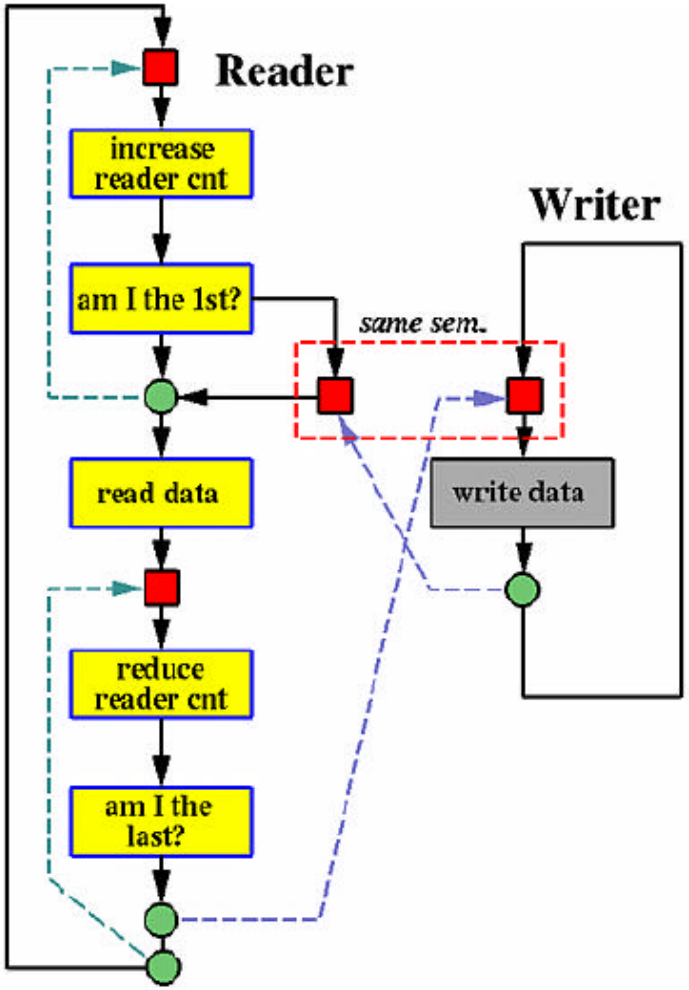


The Readers/Writers Problem

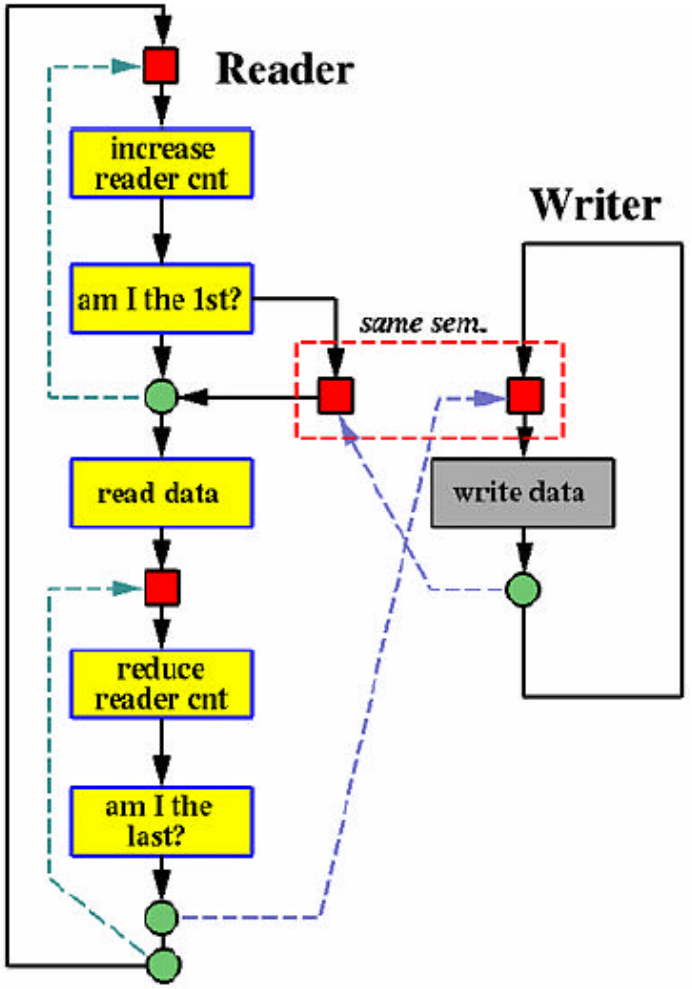
- Two groups of processes, readers and writers, are accessing a shared resource by the following rules:
  - Readers can read simultaneously.
  - Only one writer can write at any time.
  - When a writer is writing, no reader can read.
  - If there is any reader reading, all incoming writers must wait. Thus, readers have higher priority.

Problem Analysis

- We need a semaphore to block readers if a writer is writing.
- When a writer arrives, it must be able to know if there are readers reading. So, a reader count is required which must be protected by a lock.
- This reader-priority version has a problem: bounded waiting condition may be violated if readers keep coming, causing the waiting writers no chance to write.



- When a reader comes in, it increase the count.
- If it is the 1st reader, waits until no writer is writing,
- Reads data.
- Decreases the counter.
- Notifies the writer that no reader is reading if it is the last.



- When a writer comes in, it waits until no reader is reading and no writer is writing.
- Then, it writes data.
- Finally, notifies readers and writers that no writer is in.

28

Solution

```
semaphore Mutex = 1, WrtMutex = 1;
int       RdrCount;
```

reader

```
while (1) {
  Mutex.wait();
    RdrCount++;
    if (RdrCount == 1)
      WrtMutex.wait();
  Mutex.signal();
  // read data
  Mutex.wait();
    RdrCount--;
    if (RdrCount == 0)
      WrtMutex.signal();
  Mutex.signal();
}
```

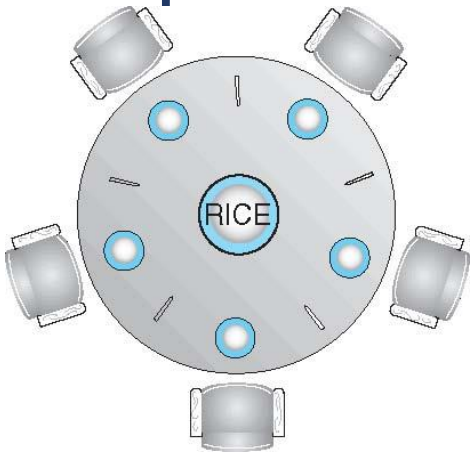
writer

```
while (1) {
    WrtMutex.wait();
    // write data
    WrtMutex.signal();
}
```

blocks both readers and writers

29

Classical Problems of Synchronization Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Solution - 1 Take left fork first and then take second fork.

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1)%5];

// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];

think
}while(true);
```

- Two neighbours cant eat at same time
- If all philosopher's are hungry then they take

Left chopstick lead to deadlock

- If 2 philosophers are faster in taking chopstick ?

Solution - 2

Lefty

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1)%5];

// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];

think
}while(true);
```

Righty

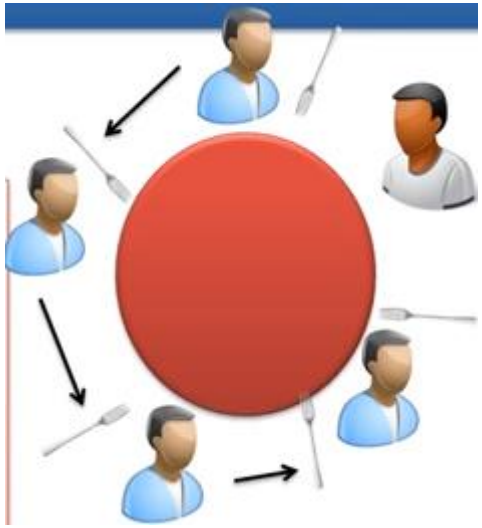
```
do {
wait(chopstick[(i+1)%5];
wait(chopstick[i]);

// Critical Section
eat

signal(chopstick[(i+1)%5];
signal(chopstick[i]);

think
}while(true);
```

- Problem - Starvation



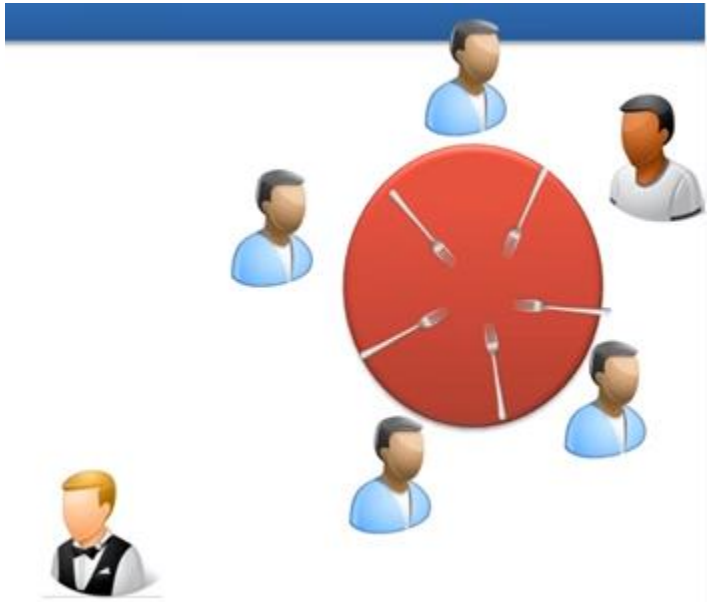
Solution - 3

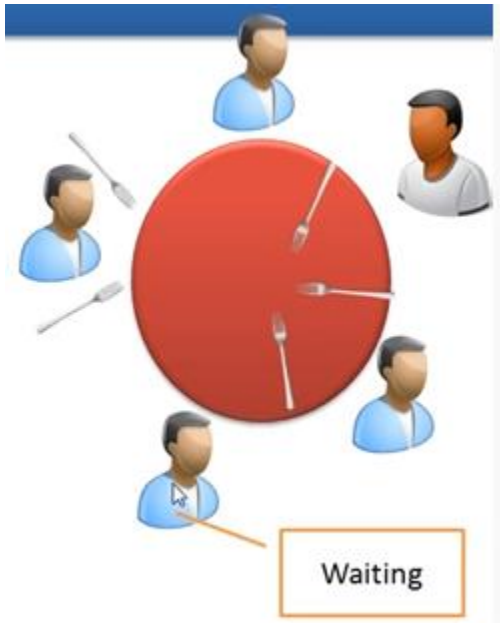
Use of Arbitrator

```
do {
wait(mutex)

wait(chopstick[i]);
wait(chopstick[(i+1)%5];
signal(mutex)
// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];
think
}while(true);
```





- This is proposed by Dijsktra

Tannenbaum solution

Solution with Semaphores

Uses N semaphores (s[1], s[2], ..., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
        state[i] = EATING;
        up(s[i]);
    }
}
```

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	T	T	T
semaphore	0	0	0	0	0

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	H	T	T
semaphore	0	0	0	0	0

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	1	0	0

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

s[i] is 1, so down will not block.
The value of s[i] decrements by 1.

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	1	0	0

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

s[i] is 1, so down will not block.
The value of s[i] decrements by 1.

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	0	0	0

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

blocked

	P1	P2	P3	P4	P5
state	T	T	E	H	T
semaphore	0	0	0	0	0

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

blocked

	P1	P2	P3	P4	P5
state	T	T	T	H	T
semaphore	0	0	0	0	0

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

blocked

	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	0	0

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

blocked

	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	1	0

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

wakeup

P1	P2	P3	P4	P5
T	T	T	E	T
0	0	0	0	0

Monitors

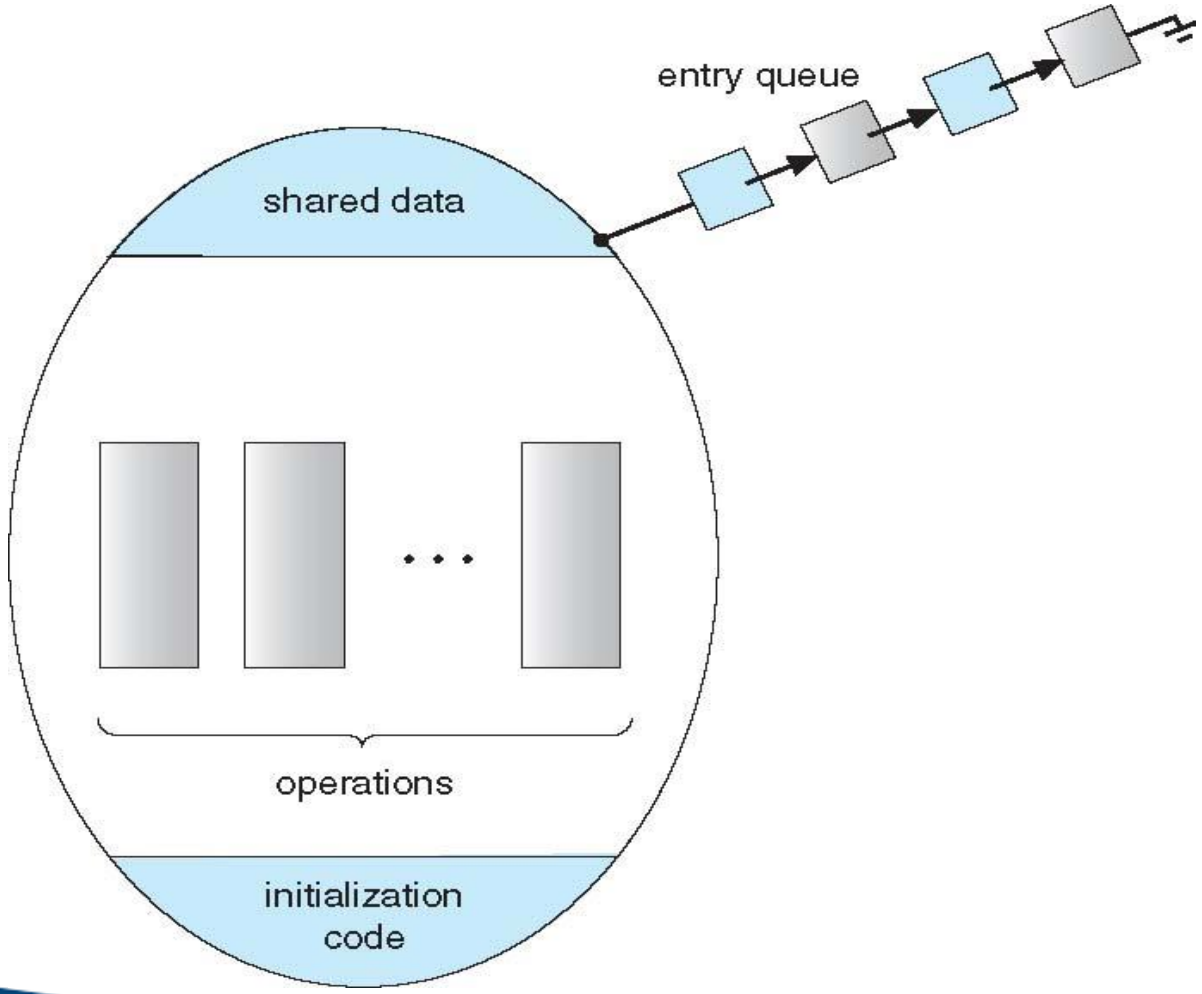
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

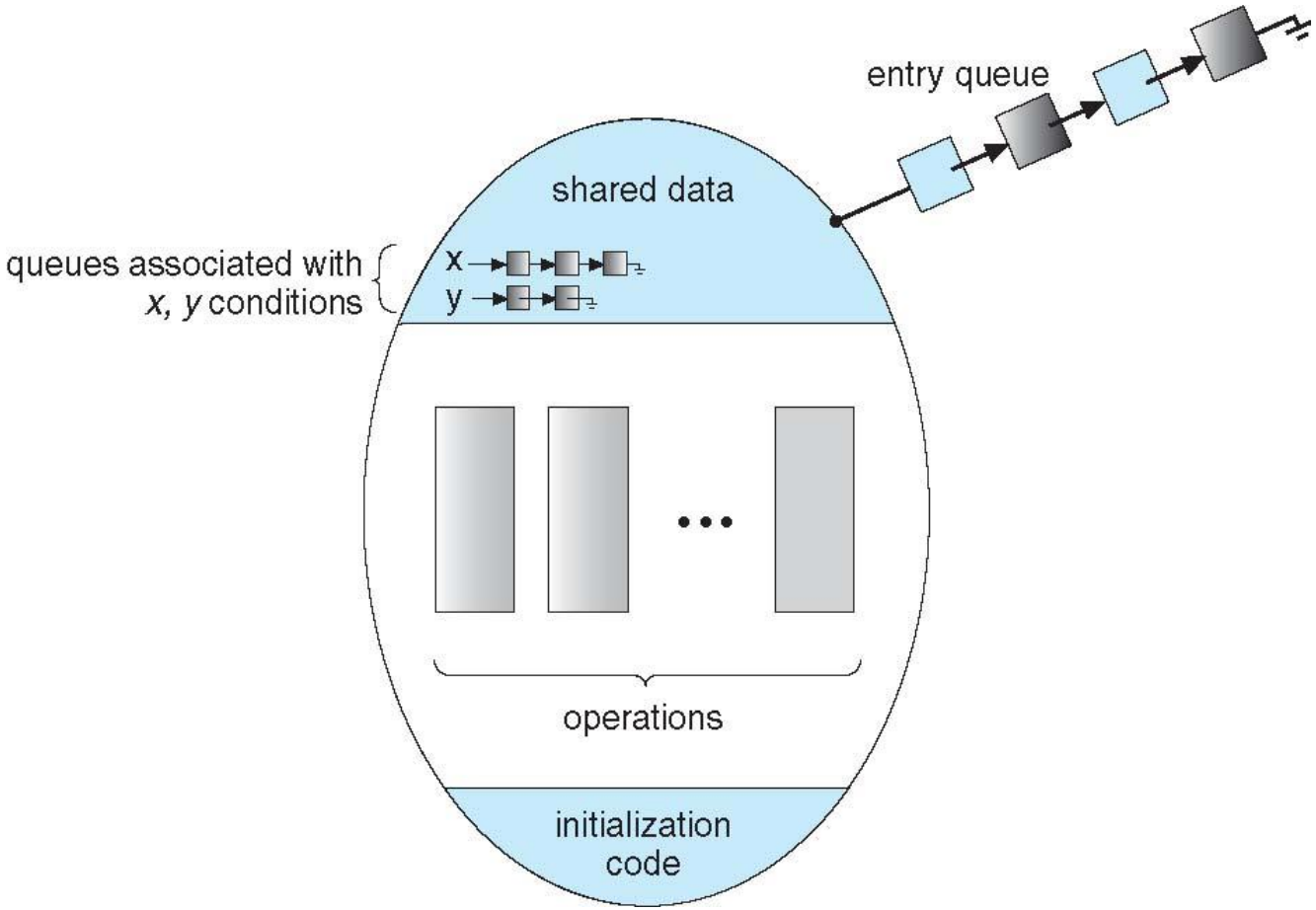
Schematic view of a Monitor



Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
  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 left and right neighbors
         test((i + 4) % 5);
         test((i + 1) % 5);
}
```

Solution to Dining Philosophers (Cont.

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

      initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
      }
}
```

Solution to Dining Philosophers (Cont.)

- Each philosopher *i* invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);

        EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Deadlocks

Contents

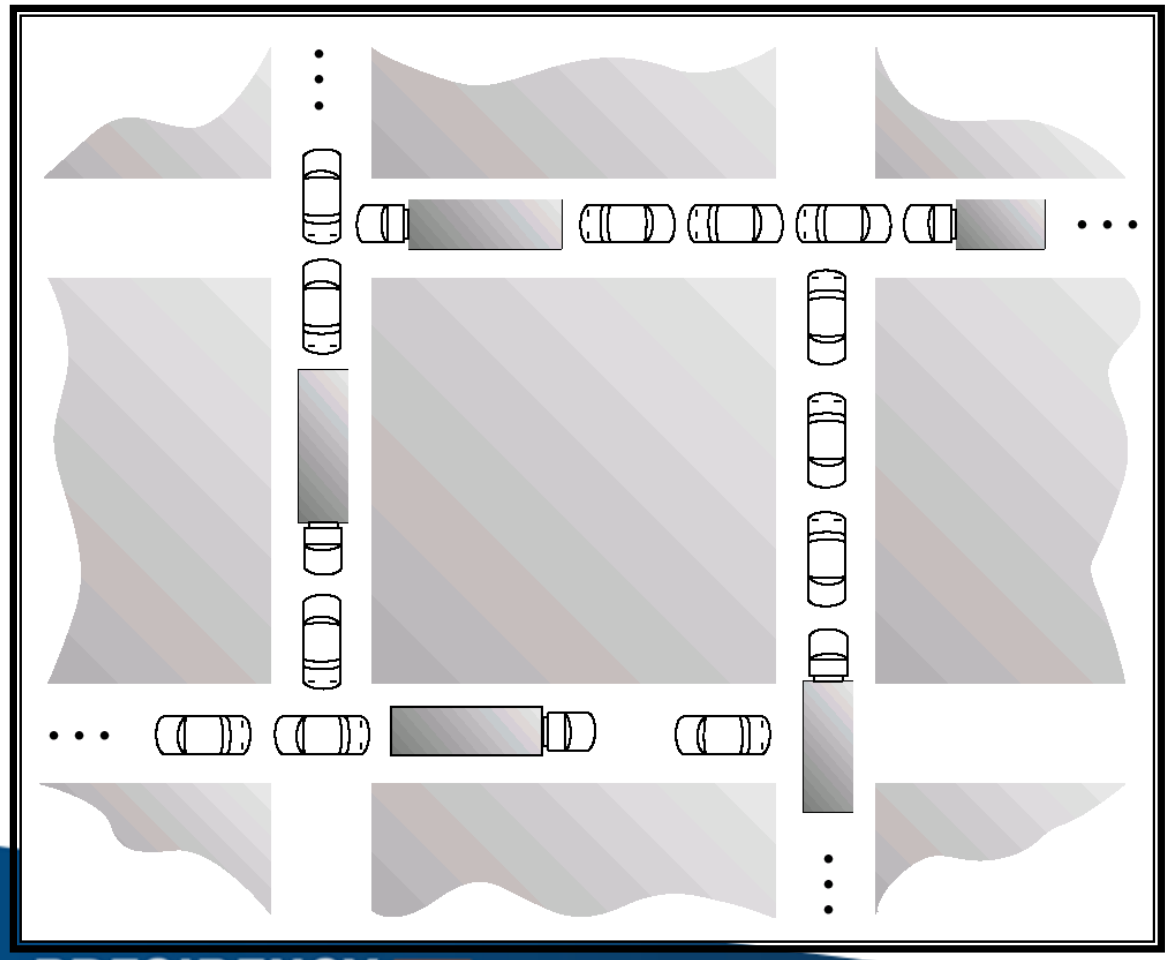
- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance – Banker's algorithm
  - Deadlock Detection
  - Recovery from Deadlock

The Deadlock Problem

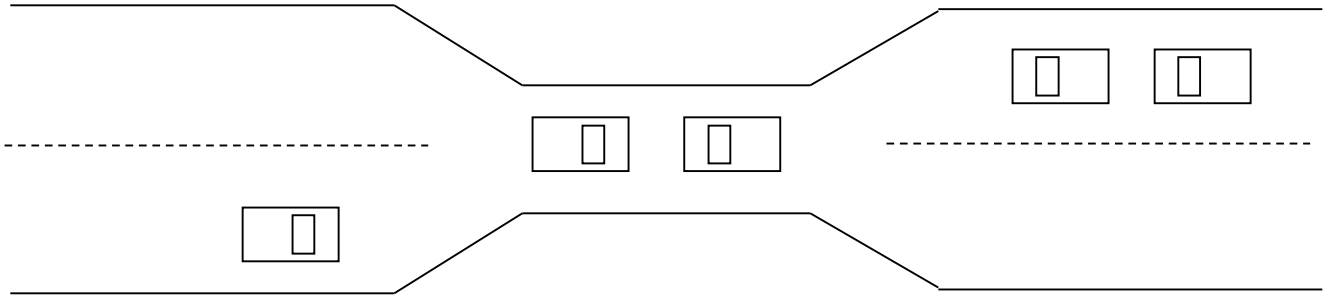
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example 1
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example 2
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B);
wait (B);	wait(A);

Traffic Deadlock Example



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.

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:
  - request
  - use
  - release

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 $\{P_0, P_1, \ldots, P_n\}$ 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, …, 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.

Resource-Allocation Graph

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

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

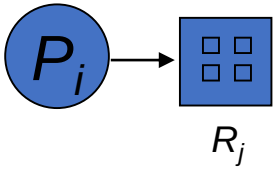
- E is partitioned into two types:
  - request edge – directed edge $P_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

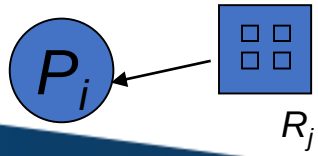
- Process

- Resource type with 4 instances

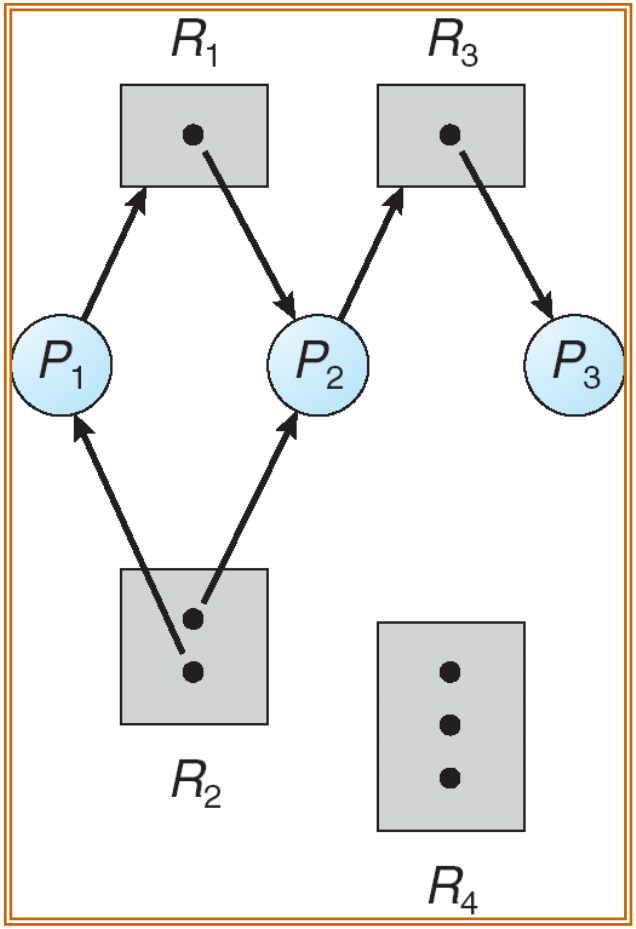
- P_i requests an instance of R_j



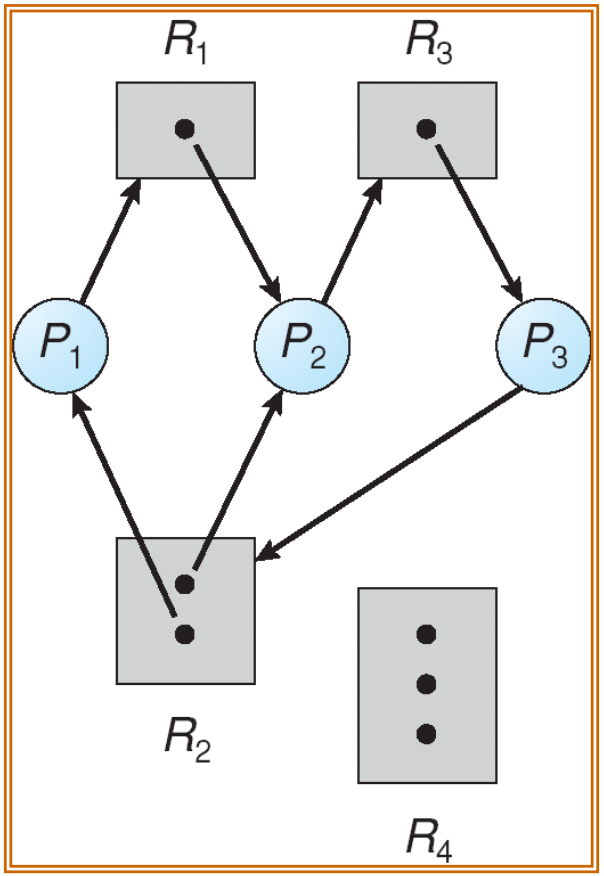
- P_i is holding an instance of R_j



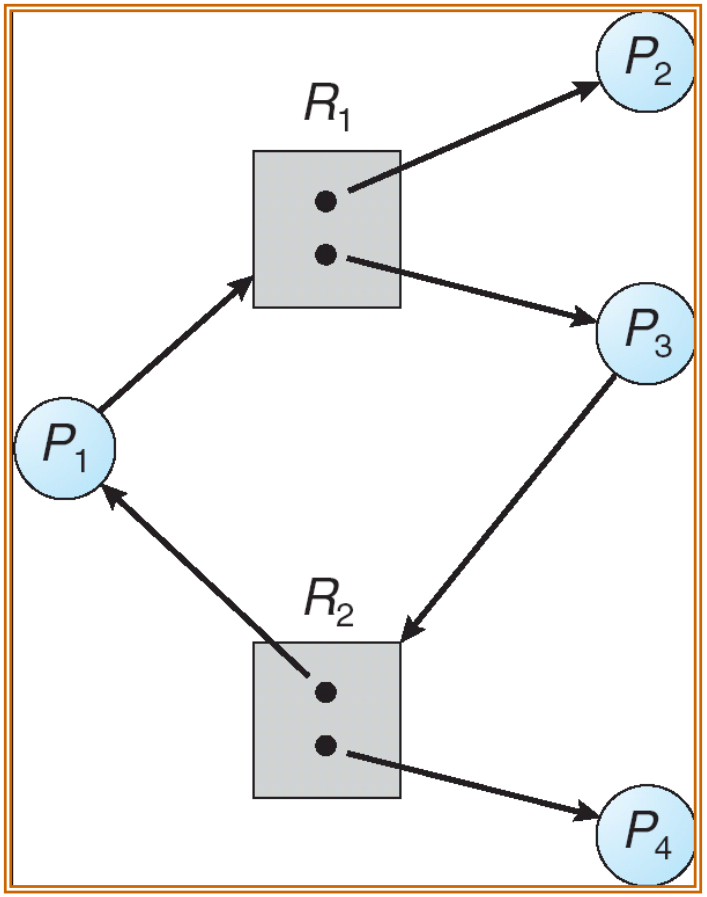
Example of a Resource Allocation Graph



Resource Allocation Graph with a Deadlock



Graph with a Cycle But No Deadlock



Basic Facts

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

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
  - deadlock *prevention*
  - deadlock *avoidance*

- Allow the system to enter a deadlock state and then recover.
  - deadlock *detection* and *recovery*

- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX and Linux.

Deadlock Prevention

Restrain the ways request can be made.

- Mutual Exclusion
  - not required for sharable resources e.g. read-only files
  - must hold for nonsharable resources e.g. printers

Deadlock Prevention

- 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.

  - side effects
    - low resource utilization
    - starvation possible

Deadlock Prevention

- No Preemption
  - Implict 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
  - suitable for resources whose state can be easily saved and restored, such as CPU registers and memory space, but generally cannot apply to resources like printers and tape drives

Deadlock Prevention

- Circular Wait
  - impose a total ordering of all resource types
  - require that each process requests resources in an increasing order of enumeration, or
    - whenever a process requests a resource, it has released any resources of higher order

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.
- The system is in safe state if there exists a safe sequence of *all* processes in the systems.
- A sequence <P_1, P_2, …, 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.

Example

- A system with 12 tape drives
- Current state

	Max Need	Current Hold
P_0	10	5
P_1	4	2
P_2	9	2

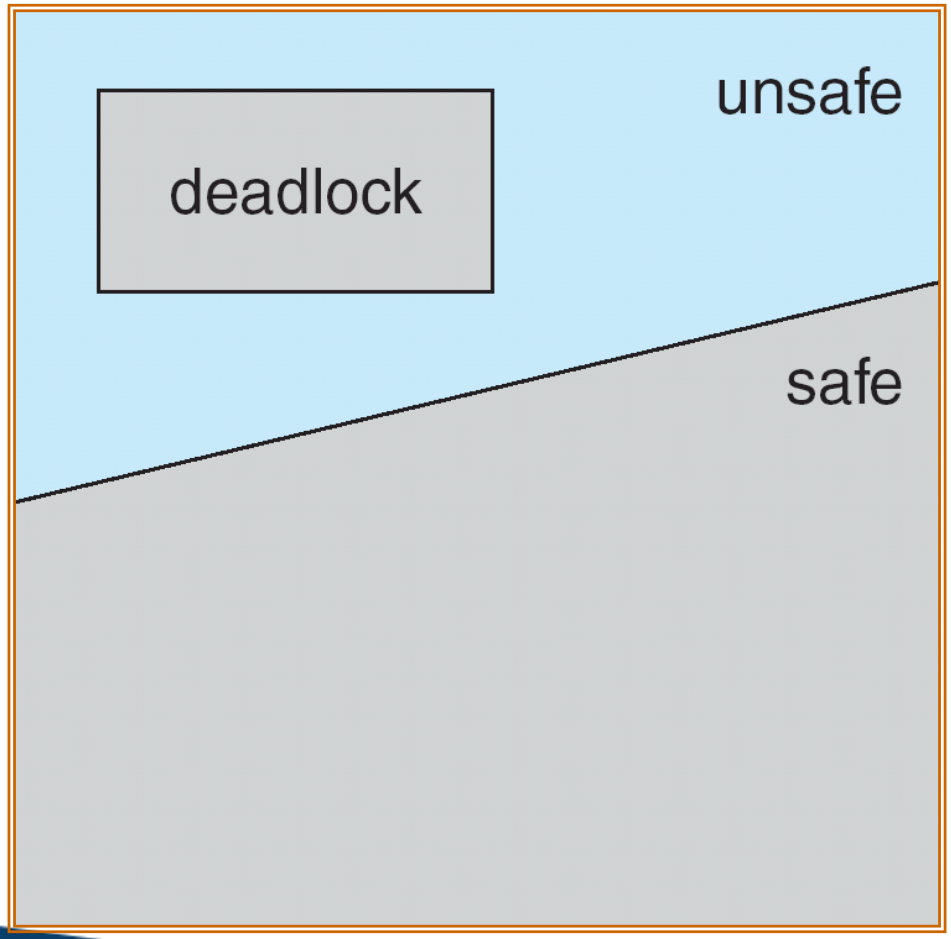
Safe sequence: <P1, P0, P2>

It is a safe state.

Basic Facts

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

Safe, Unsafe , Deadlock State



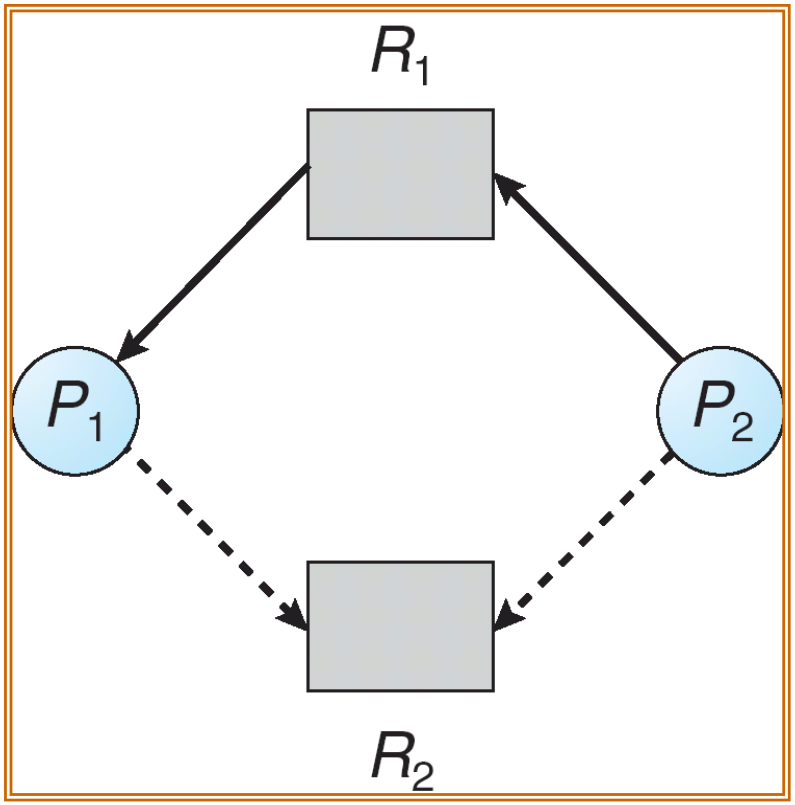
Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process.
- 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

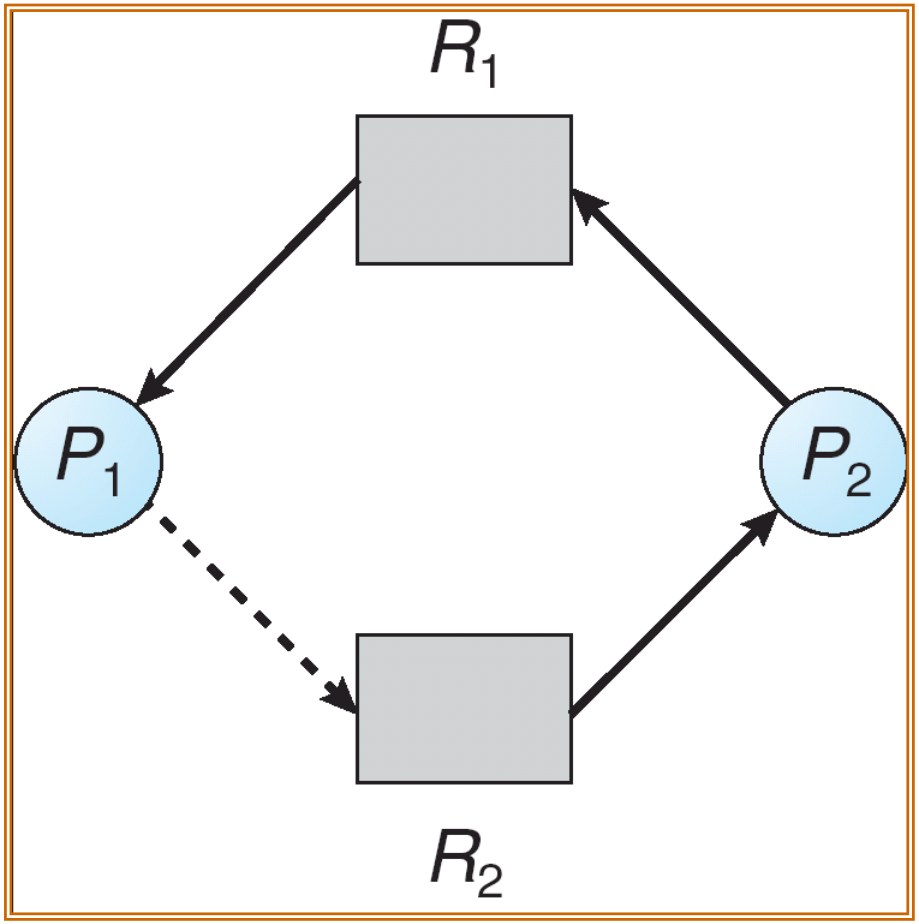


if P_2 request R_2,

and R_2 is currently free,

but …

Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Suitable for 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.
- Basic idea: keep the system in safe state

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 = 0, 1, …, n- 1.

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 P_i

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:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

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

Example of Banker's Algorithm

- 5 processes P_0 through P_4
- 3 resource types:

  A (10 instances), B (5 instances), and C (7 instances).
- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max* – *Allocation*.

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0$ > Work (7,4,3 > 3,3,2)

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $Need_1$ < Work (1,2,2 < 3,3,2)

So P_1 must be kept in safe sequence

Step 3

Work = Work + $Allocation_1$ (3, 3, 2 + 2, 0, 0)

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ > Work (6, 0, 0 > 5,3, 2)

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ < Work (0, 1, 1 < 5, 3, 2)

So P_3 must be kept in safe sequence

Step 3

Work = Work + $Allocation_3$ (5, 3, 2 + 2, 1, 1)

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4$ < Work (4, 3, 1 < 7, 4, 3)

So P_4 must be kept in safe sequence

Step 3

Work = Work + $Allocation_4$ (7, 4, 3 + 0, 0, 2)

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need < Work (7, 4, 3 < 7, 4, 5)

So P_0 must be kept in safe sequence

Step 3

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1 , 0)

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ < Work (6, 0, 0 < 7, 5, 5)

So P_2 must be kept in safe sequence

Step 3

Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2)

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for 0 ≤ i ≤ n

Hence the system is in Safe state

The safe sequence is P_1, P_3 , P_4 , P_0, P_2

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available
  - that is, (1,0,2) ≤ (3,3,2) ⇒ true.
- Pretend the request has been granted

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

Example (cont.)

- Can request for (3,3,0) by P_4 be granted?
  - No, because the resources are not available.
- Can request for (0,2,0) by P_0 be granted?
  - No. Even though the resources are available, the resulting state is unsafe

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Deadlock Detection

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

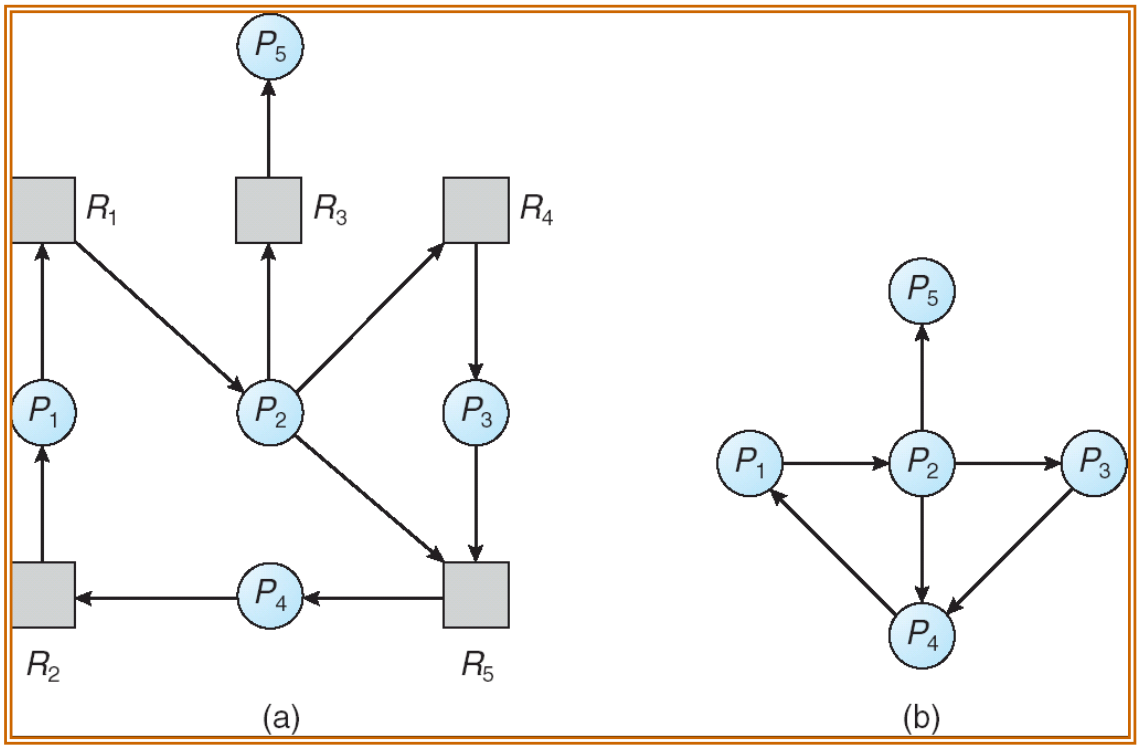
Single Instance of Each Resource Type

- 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. If there is a cycle, there exists a deadlock.

- 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* $[i_j] = k$, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively
   Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then
   Finish[i] = false; otherwise, *Finish*[i] = *true*.

2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   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.

The 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.)

- Suppose P_2 requests an additional instance of type C , make the *Request* matrix to be

	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

- called at specific interval, say every an hour
- called whenever a request cannot be granted immediately
  - can locate the specific process "caused" the deadlock
- called when CPU utilization drops, say, below 40%
  - If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph, and we cannot tell which process "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?

Resource Preemption

- Selecting a victim – minimize cost
  - factors similar as above mentioned
- Rollback – return to some safe state, restart process for that state
  - OS should keep track of states of all processes
- Starvation – same process may always be picked as victim
  - solution: include number of rollback in cost factor

Thank You