Chapter 5: Process Synchronization

concept of process synchronization

critical-section problem

solutions of the critical-section problem

classical process-synchronization problems

Contents

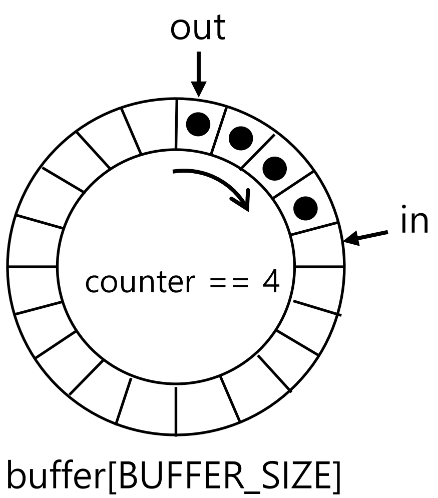
- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors

5.1 Background

- Processes can execute concurrently
- May be interrupted at any time, partially completing executio n
- Concurrent access to shared data may result in data inconsist ency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Producer-consumer 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.




```
while (true) {    /* PRODUCER */
    /* produce an item in next produced */
    while (counter == BUFFER_SIZE) ;
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

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

Race Condition

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

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

- **`counter--`** could be implemented as

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

```
register1 = counter
register1 = register1 + 1
counter = register1
register2 = counter
register2 = register2 - 1
counter = register2
```

OR

```
register2 = counter
register2 = register2 - 1
counter = register2
register1 = counter
register1 = register1 + 1
counter = register1
```

- Consider this execution interleaving with "counter = 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}

5.2 Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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 s ection
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **e ntry section**, may follow critical section with **exit section**, the n **remainder section**

Solution of Critical Section Problem

- General structure of process P_i

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

- An example

```
do {

    while (turn == j);

        critical section

    turn = j;

        remainder section

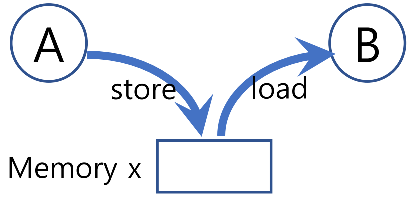
} while (true);
```

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

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the n processes

5.3 Peterson’s Solution



- 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
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- 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_0 and P_1

```
/* P0 */
do {
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn == 1);
                critical section
        flag[0] = false;
                remainder section
  } while (true);
```

```
/* P1 */
do {
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn == 0);
                critical section
        flag[1] = false;
                remainder section
  } while (true);
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved
   P_i enters CS only if:
   either **flag[j] = false** or **turn = i**

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met

```
/* P0 */
do {
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn == 1);
                critical section
        flag[0] = false;
                remainder section
 } while (true);
```

5.4 Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- 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

`test_and_set` Instruction

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

- Executed atomically

Solution using test_and_set

- Shared Boolean variable `lock`, initialized to `FALSE`

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

compare_and_swap Instruction

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

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

- Executed atomically

Solution using compare_and_swap

- Shared integer `lock`, initialized to `0`;

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
       ;
     /* critical section */
  lock = 0;
     /* remainder section */
 } while (true);
```

Bounded-waiting Mutual Exclusion with test _and_set

```
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);
```

```
boolean lock
boolean waiting[n];
```

5.5 Mutex Locks

- Previous solutions are complicated and generally inaccessible to a pplication 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);
  ```
  ```

5.6 Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**

- Definition of the **`wait()`** operation

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

- Definition of the **`signal()`** operation

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems. Consider P1 and P2 that require S1 to happen before S2

```
Create a semaphore "synch" initialized to 0
P1:
   S1;
   signal(synch);
P2:
   wait(synch);
   S2;
```

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy w aiting

- With each semaphore there is an associated waiting queue

```
typedef struct{
  int value;
  struct process *list;
  } semaphore;
```

- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place i t in the ready queue

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}


signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

5.7 Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronizati on schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- Producer process

```
do {
      . . .
     /* produce an item
        in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
     /* add next produced
        to the buffer */
      . . .
   signal(mutex);
   signal(full);
} while (true);
```

- Consumer process

```
do {
     wait(full);
     wait(mutex);
        . . .
       /* remove an item from
          buffer to next_consumed */
        . . .
     signal(mutex);
     signal(empty);
        . . .
       /* consume the item
          in next consumed */
        . . .
   } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Integer **`read_count`** initialized to 0
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1

Readers-Writers Problem (Cont.)

- Writer process

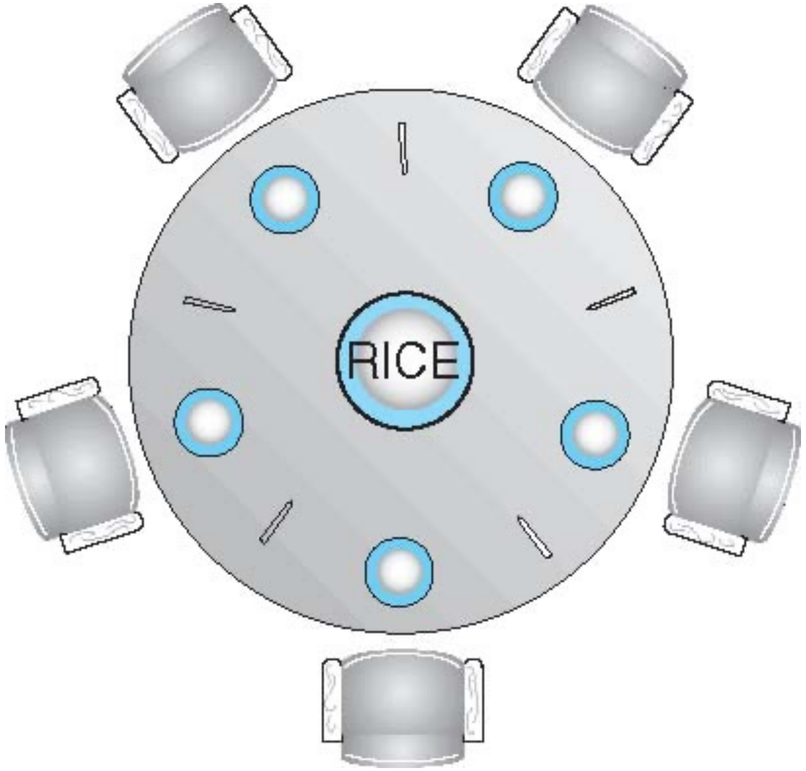
```
do {
    wait(rw_mutex);
        ...
    /* writing is performed */
        ...
   signal(rw_mutex);
} while (true);
```

- Reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
         wait(rw_mutex);
      signal(mutex);
          ...
        /* reading is performed */
          ...
      wait(mutex);
        read count--;
        if (read_count == 0)
      signal(rw_mutex);
      signal(mutex);
} while (true);
```

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?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the ta ble.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher pick s up first the left chopstick and then the right chopstick. Even-num bered philosopher picks up first the right chopstick and then the le ft chopstick.

5.8 Monitors

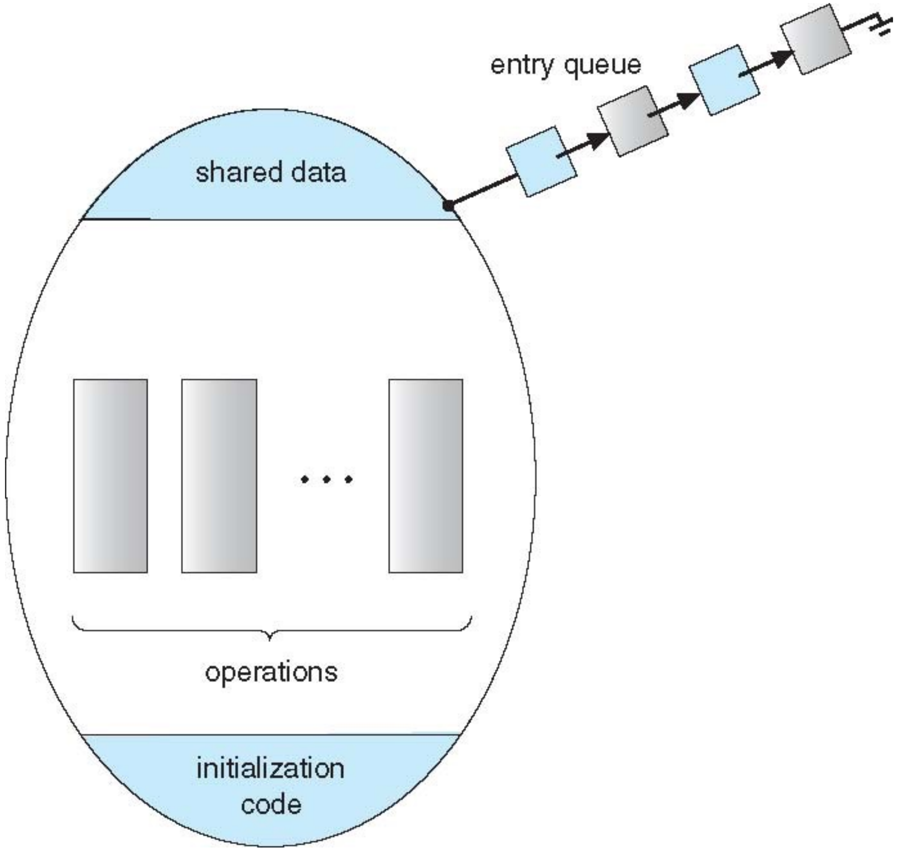
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure

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

  procedure Pn (…) {……}

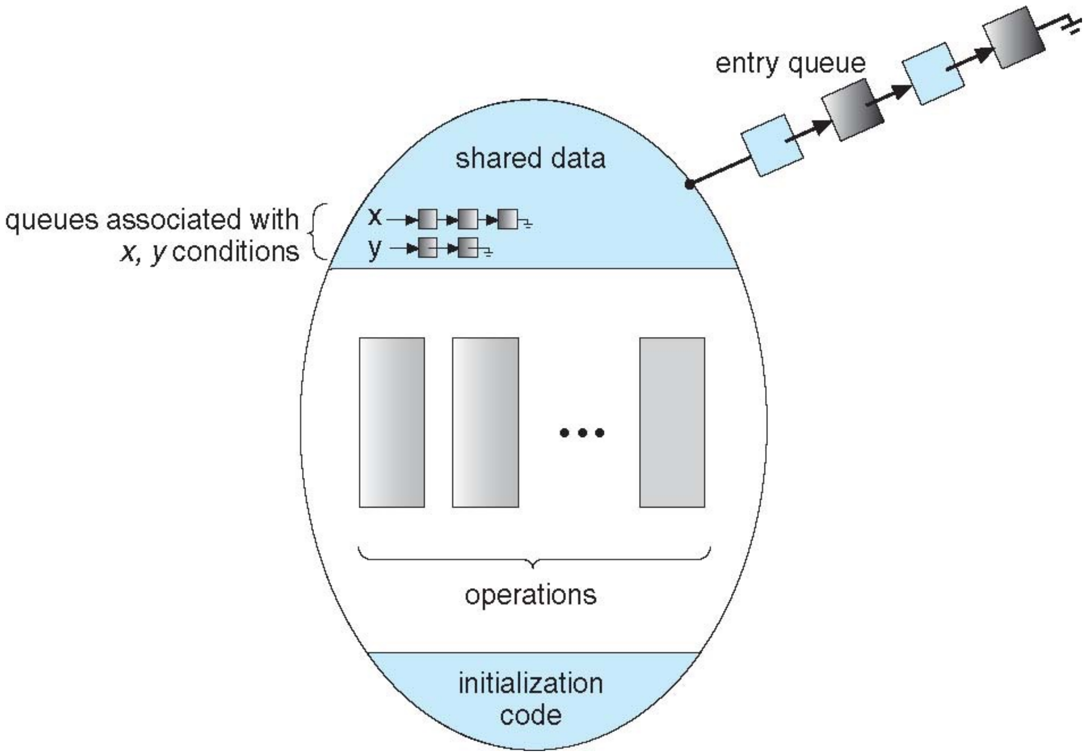
    Initialization code (…) { … }
}
```

- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes



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



5.11 The Deadlock Problem

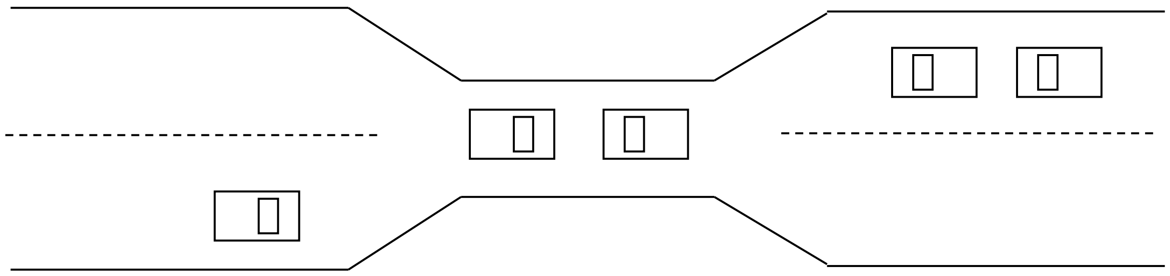
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one

- Example: semaphores *A* and *B*, initialized to 1

P_1	P_2
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

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

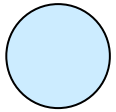
- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by th e process holding it, after that 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 waitin g 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 P0.

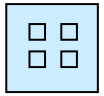
Resource-Allocation Graph

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

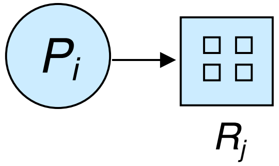
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

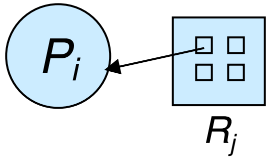
- Process 

- Resource Type with 4 instances 

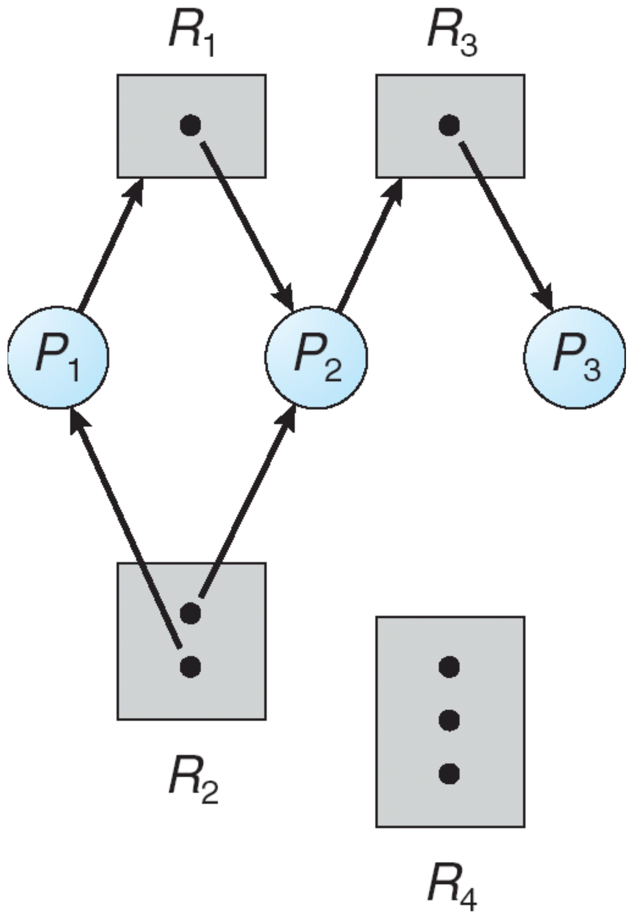
- P_i requests instance of R_j



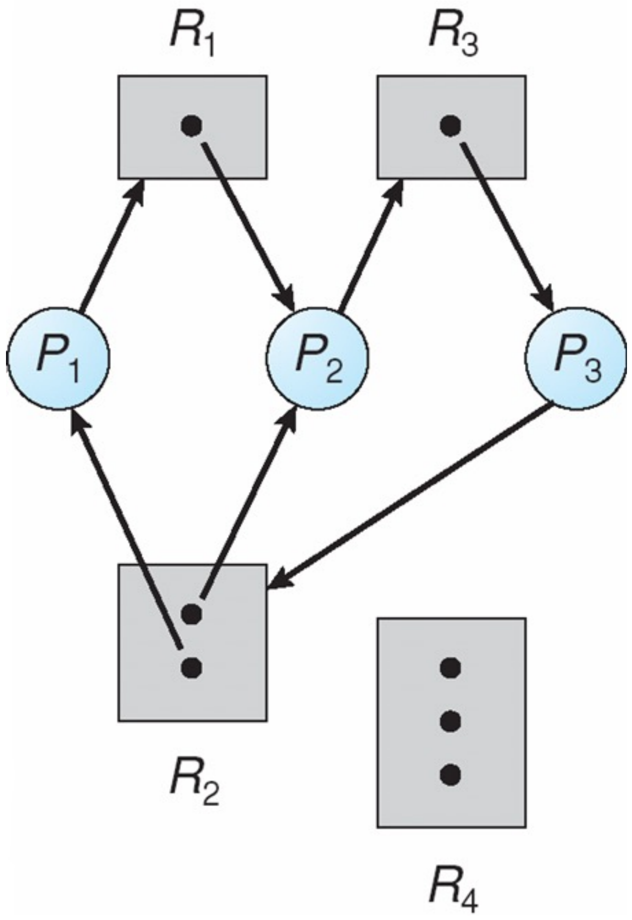
- P_i is holding an instance of R_j



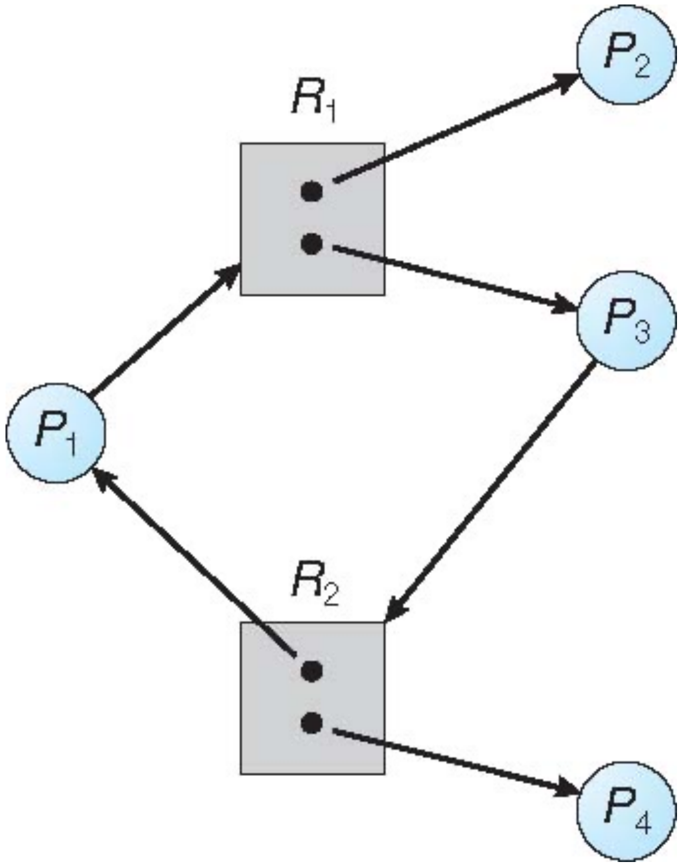
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



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