18CSC205J OPERATING SYSTEMS UNIT – II

Course Learning Rationale (CLR):

CLR-2 : Insist the Process Management functions of an Operating system

Course Learning Outcomes (CLO):

CLO-2 : Know the Process management functions of an Operating system

TOPICS COVERED

- PROCESS SYNCHRONIZATION :

  Peterson's solution, Synchronization Hardware, Semaphores, usage, implementation, Classical Problems of synchronization – Readers writers problem, Bounded Buffer Problem, Dining Philosophers problem (Monitor)
- CPU SCHEDULING :

  FCFS,SJF, Priority scheduling, Round robin, Multilevel queue Scheduling, Multilevel feedback Scheduling.
- REAL TIME SCHEDULING:

  Rate Monotonic Scheduling and Deadline Scheduling
- DEADLOCKS:

  Necessary conditions, Resource allocation graph, Deadlock prevention methods, Deadlock Avoidance, Detection and Recovery

Process synchronization

What is Process Synchronization ?

- **Process Synchronization** is the process of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

What is Critical section ?

- The portion in any program which accesses a shared resource (such as a shared variable in the memory) is called as critical section or Critical region.

Types of solutions to CS problem

- Software solution
  - Peterson’s solution
- Hardware solutions
  - Synchronization Hardware – TSL Instruction
  - Compare and swap Instruction
- MUTEX Locks (Spin lock- Software Tool)
- Programming language construct
  - Semaphores

Software Solution to CS problem

Peterson’ s Solution (overview)

Helps to solve the critical section problem – applicable for 2 processes. (Scenarios - Classical problems of synchronization – Bounded Buffer, Producer-consumer, Dining philosophers)

Peterson’ s solution

- The algorithm deals with 2 variables
- Turn and flag

Using these 2 variables , the critical section problem is addressed by Peterson.

Contd..

Two variables are used:

- i nt t ur n
- bool ean f l ag[2]

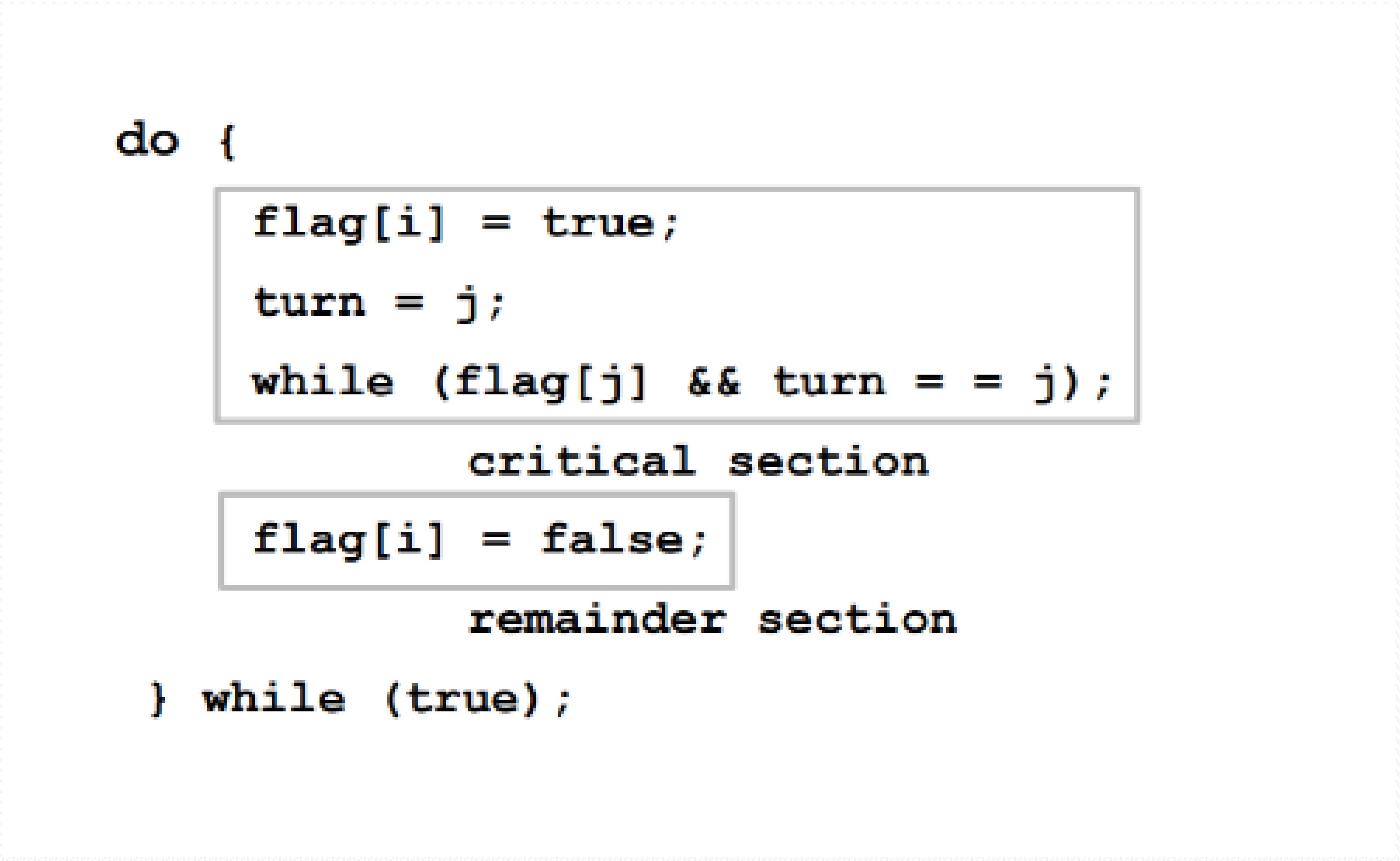
Turn whose turn to enter critical section

f l ag[i] ==

Flag to indicate if the process is ready to enter critical section

critical section

Peterson'   s Algorithm



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

Explanation :

Peterson's Algorithm is used to synchronize two processes.

- In this algorithm , the variable i can be the (Process i) Producer and j can be Consumer (Process j).
- Initially the flags are false.
- When a particular process wants to enter its critical section, **it sets it's flag to true and turn as the index of the other process.** This means that the process wants to execute but it will allow the other process to run first.
- **The process performs busy waiting** until the other process has finished it's own critical section.

Peterson’s sol. Contd..

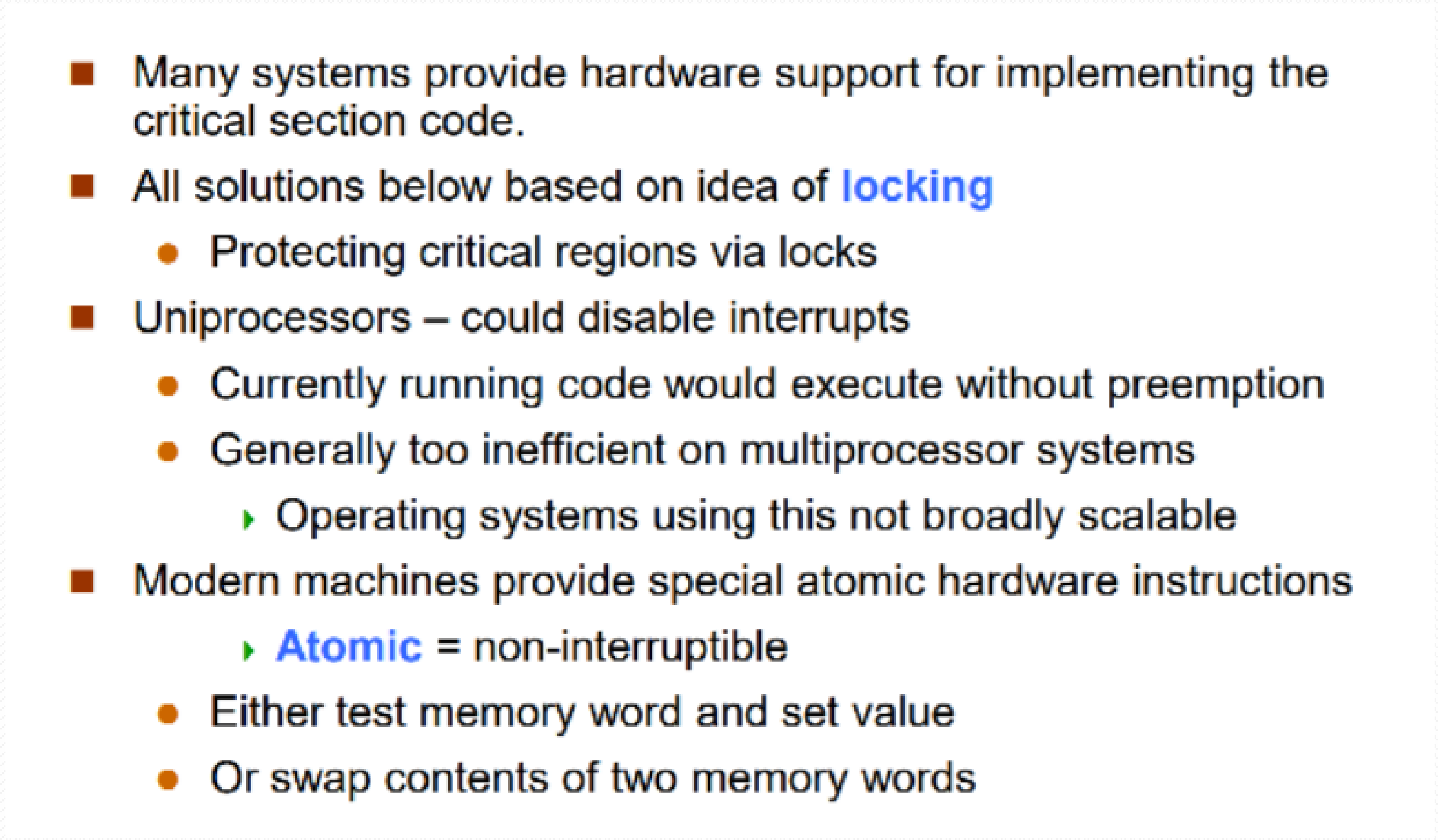
The solution addresses all the 3 conditions required for solving a Critical section problem.

(i) Mutual exclusion
(ii) Progress
(iii) Bounded waiting

Limitations of Peterson’ s sol

- Applicable only between processes
- Busy waiting.

Hardware solution - Overview



- 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

Hardware sol..

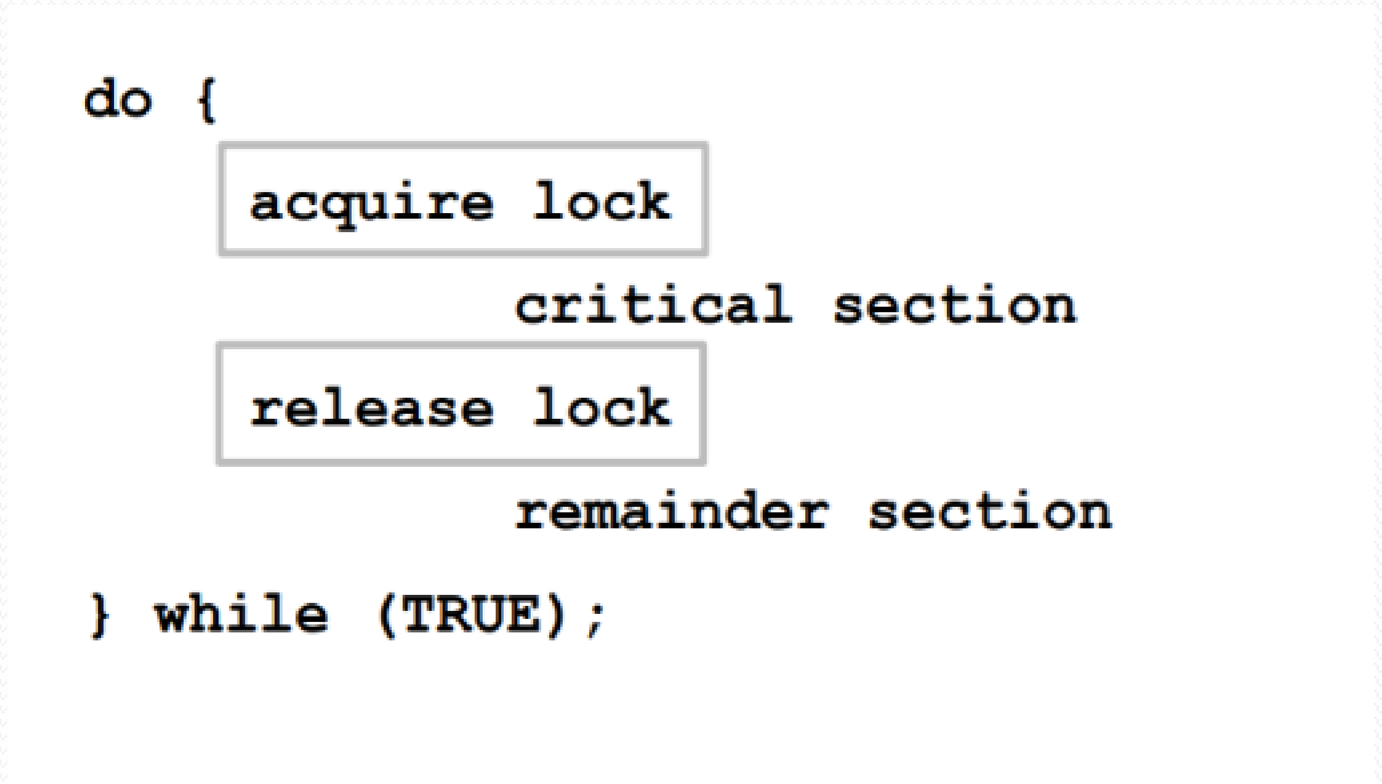
- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically— **that is, as one uninterruptible unit (TSL/Compare and Swap Instruction)**.

Hardware solutions to CS problem

(i) TSL

(ii) Compare and swap

General Structure of a process using locks :



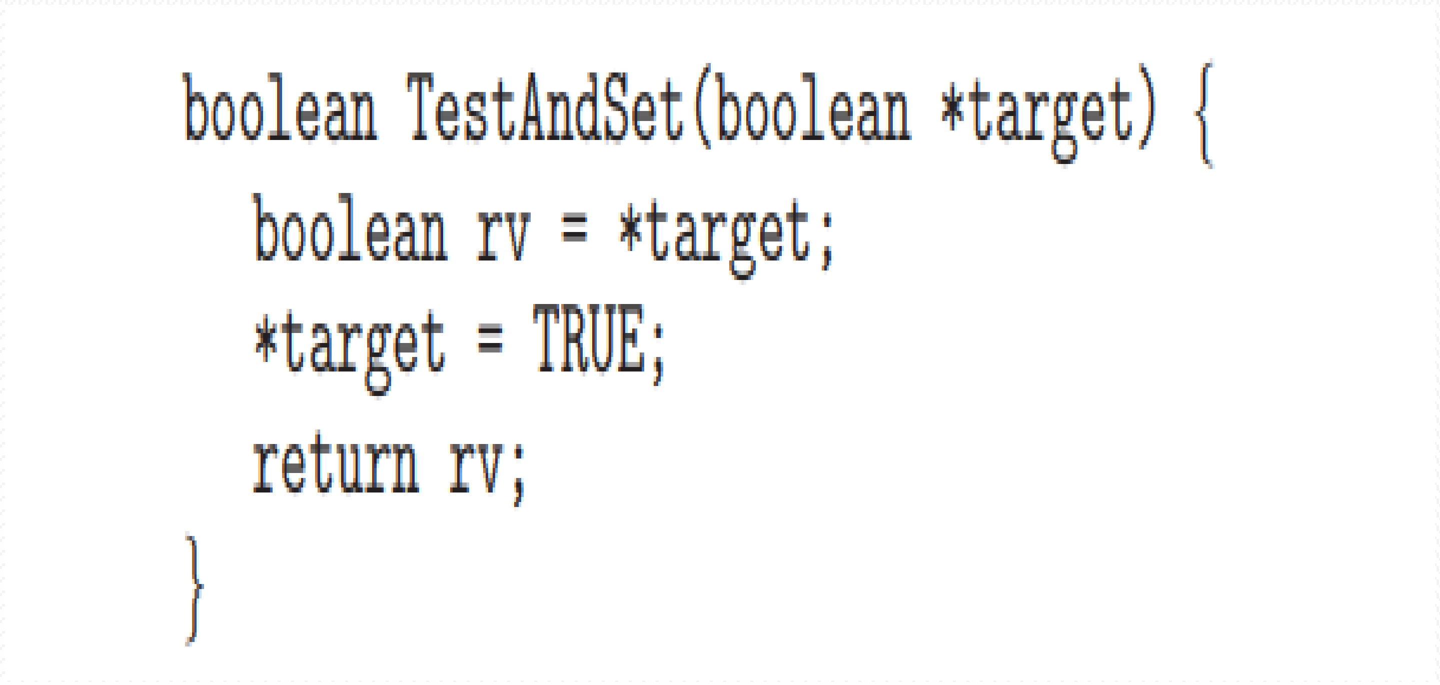
```
do {
      acquire lock
                critical section
      release lock
                remainder section
} while (TRUE);
```

(i) Test and set lock (TSL)

- This instruction reads the contents of a memory location, stores it in a register and then stores a non-zero value at the address.
- This operation is guaranteed to be indivisible (TSL- atomic).
- That is, no other process can access that memory location until the TSL instruction has finished.

(i) TSL Instruction (Test and Set Lock)

- Defining the Test and Set Lock:

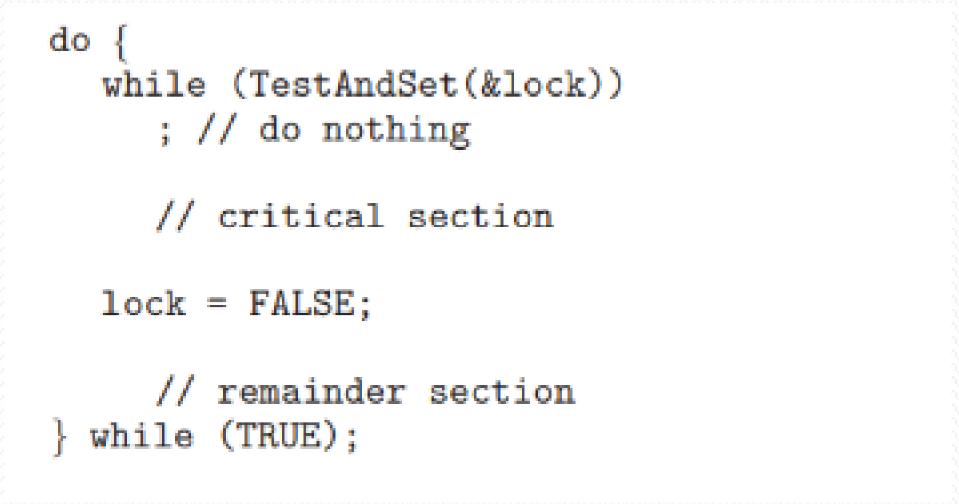


```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Test and Set Lock Instruction

- If two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
- If the machine supports the TestAndSet() instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of a Process using TSL is presented as follows:

General structure of a process using Test and set lock (TSL)



```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

(ii) Compare and swap

- The Swap() instruction, in contrast to the TestAndSet() instruction, operates on the contents of two words.
- Like the TestAndSet() instruction, it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows:
- A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key.
- The structure of process Pi is given in the next slide.

(ii) Compare and swap Instruction

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

Note:

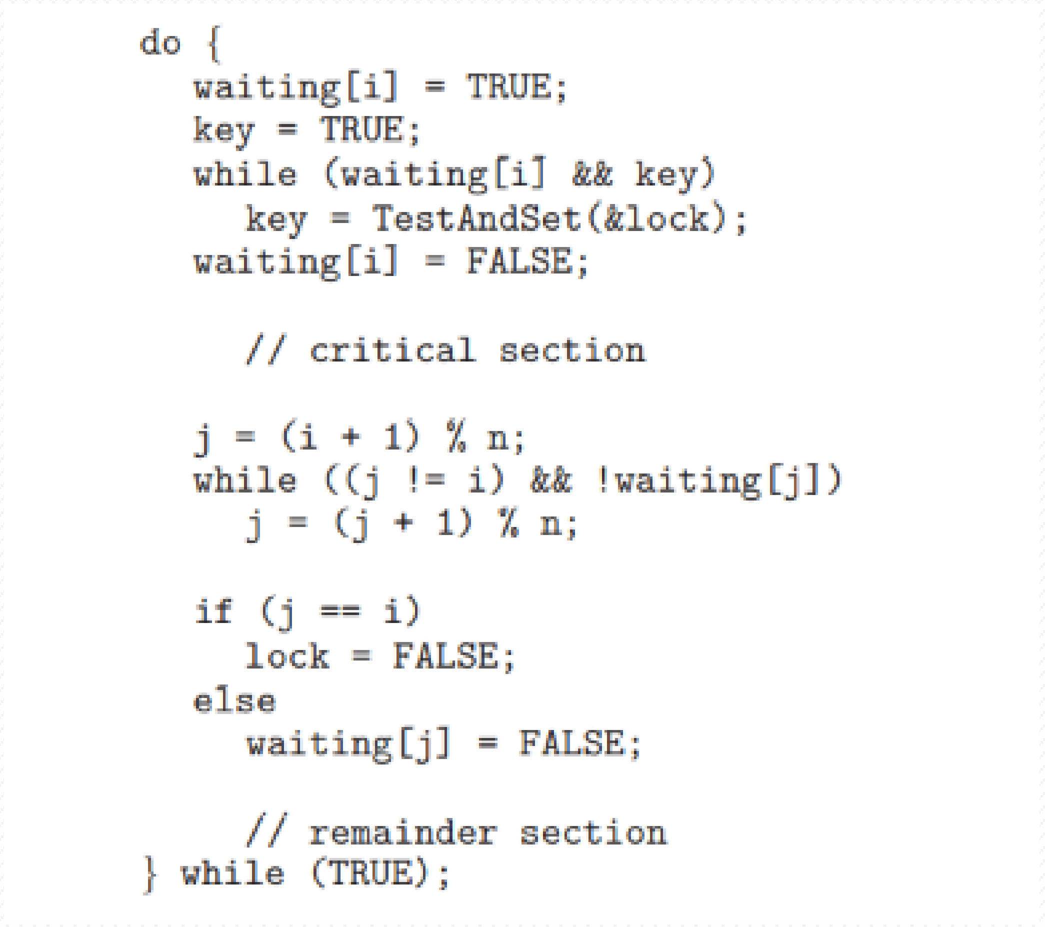
- Although mutual exclusion exclusion is guarenteed. Bounded waiting is not met in both Test and set Lock & Compare and swap methods. **Hence, to fulfill all 3 conditions for the Critical section problem following improved version of TSL is introduced**.

Data structures used.. (Improved version) TSL

- boolean waiting[n];
- boolean lock;

- Initially all the variables are initialized to False.

Bounded waiting Mutual exclusion with Test and set Lock



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

Explanation

- These data structures are initialized to false.
- To prove that the mutualexclusion requirement is met, we note that process Pi can enter its critical section only if either waiting[i] == false or key == false.
- The value of key can become false only if the TestAndSet() is executed.
- The first process to execute the TestAndSet() will find key == false; all others must wait.
- The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement

Contd..

- To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false.

- Both allow a process that is waiting to enter its critical section to proceed.

Contd..

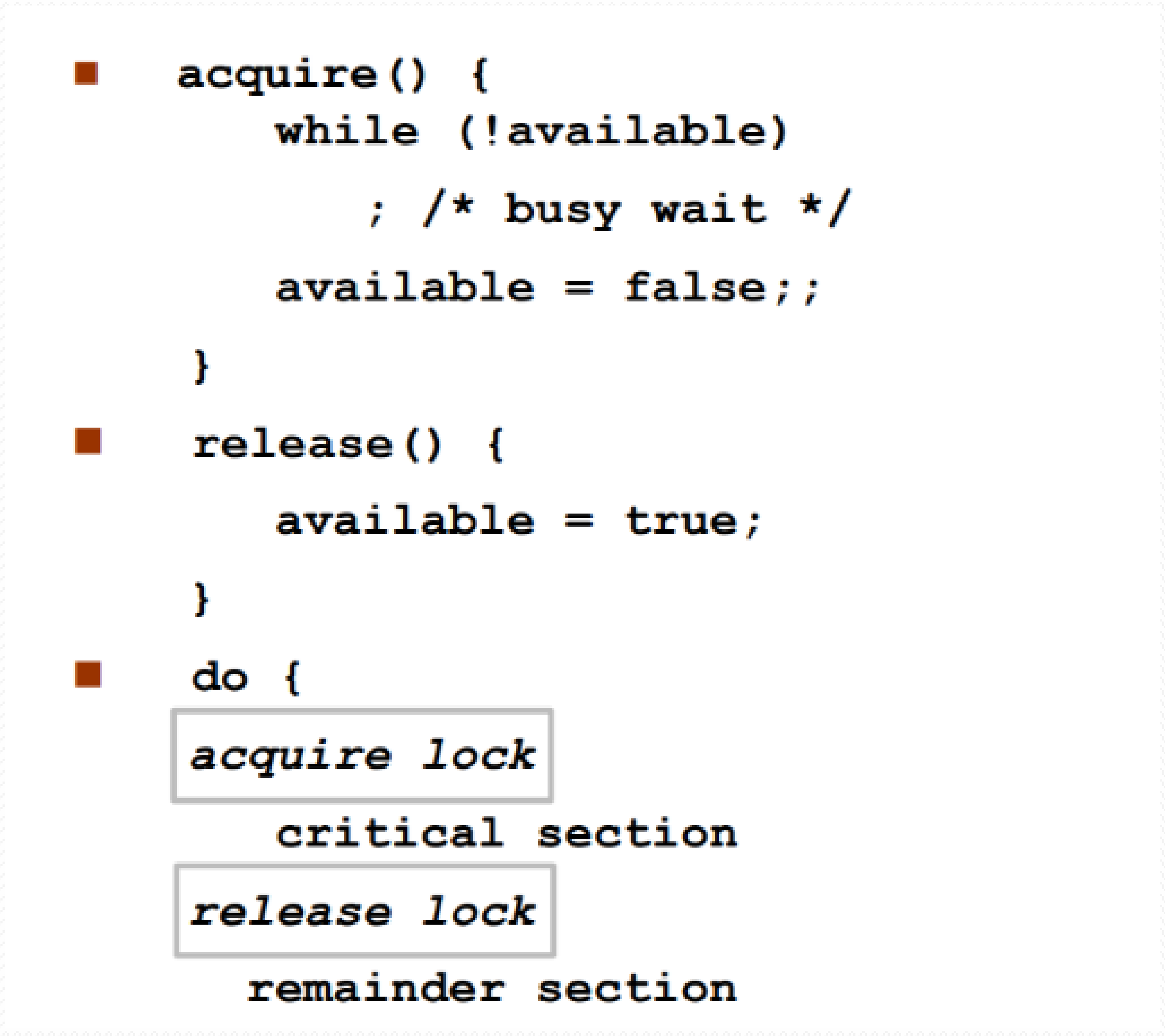
- To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n − 1, 0, ..., i − 1).

- It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n − 1 turns.

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**

Mutex locks Contd..



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

Semaphores

- Discovered by Dijkstra
  - Synchronization tool that does not require busy waiting

- Semaphore or mutex variable **S** integer variable
- Two operations:
  - Wait()
  - Signal()
- Less complicated

Semaphore – Contd…

General Structure of critical section using semaphores

```
do {
        Wait(S)
                critical section
        Signal(S)
                remainder section
} while (TRUE);
```

Definition-signal

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

Definition-Wait

```
wait(S)
{
      while(S<=0)
        ; //no-op
      S--;
}
```

Operations on semaphores (wait and signal / Down and Up)

- Two operations:
  - **block()** – place the process in the waiting queue
  - **wakeup()** – remove one of the processes in the waiting queue and place it in the ready queue

```
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);
   }
}
```

Semaphore usage

- Counting semaphore
  - Values are unrestricted
- Binary semaphore
  - Values can range only between 0 and 1
- We are using binary semaphores

Mutual Exclusion implementation with semaphores

```
do {
        Wait(mutex)
                critical section
        Signal(mutex)
                remainder section
} while (TRUE);
```

Example 1:

- Semaphore is initialized with the number of resources available.
  - Semaphore is having 10 printers
  - Each process Pi, that needs the resource perform the wait() operation on semaphore [thereby decrementing the count]
  - When the process Pi, releases the resources, it performs signal() operation [incrementing the count value]
  - Count = 0 all resources are used
  - Count = n all processes released the resources

Example 2:

- Assume 2 concurrent running processes
  - P1 with statement S1;
  - P2 with statement S2;

- s2 is executed after s1 has completed
  - P1 and p2 share a common semaphore
  - **Synch = 0**

```
S1;
Signal(synch);
```

Fig. Statements in P1

```
wait(synch);
S2;
```

Fig. Statements in P2

Disadvantages of semaphores

- **Busy waiting**
  - (ie) when a process is in critical section, the other process loop indefinitely.
  - Wasting its time

Semaphore Implementation with no Busy Waiting

- Two operations:
  - **block()** – place the process in the waiting queue
  - **wakeup()** – remove one of the processes in the waiting queue and place it in the ready queue

```
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);
    }
}
```

Deadlocks

- Definition:
  - Two or more processes waiting indefinitely for an event to be completed is called as Deadlock.

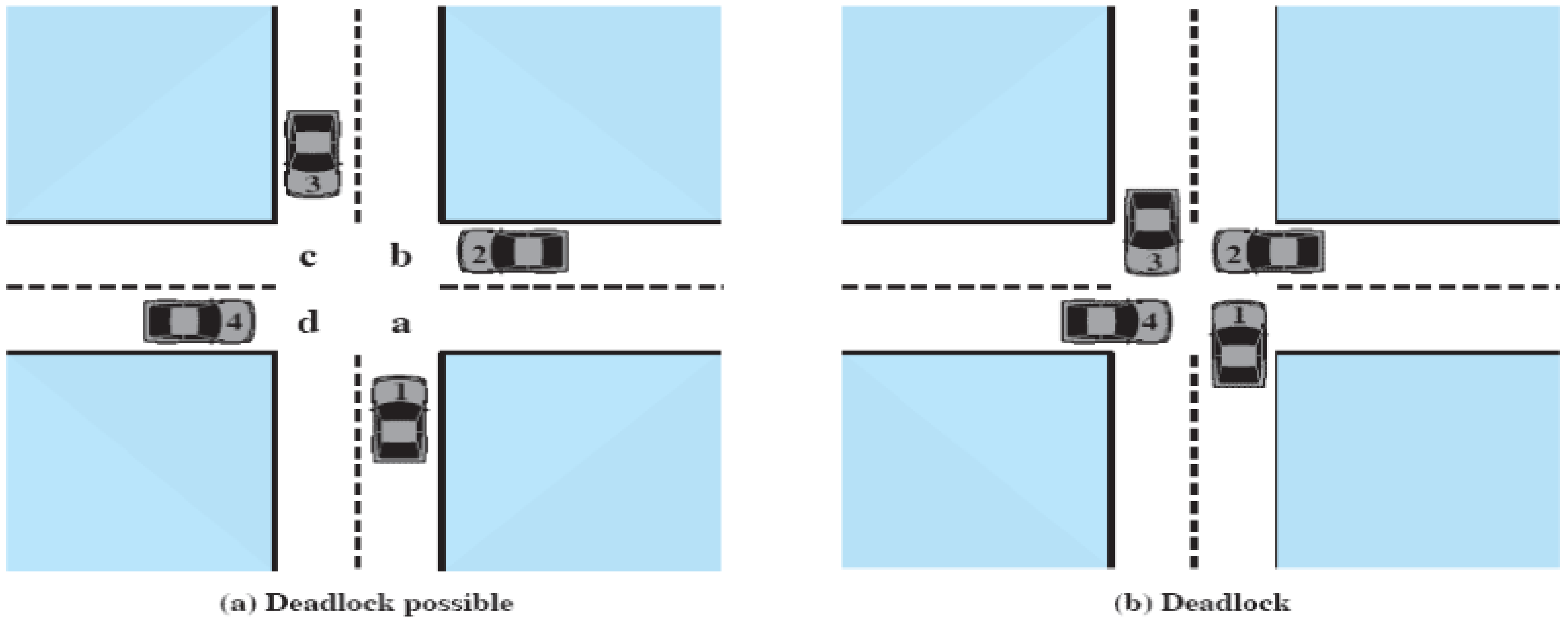


Figure 6.1 Illustration of Deadlock

Example

- System has 2 tape drives.
- P_1 and P_2 each hold one tape drive and each needs another one.

- Consider 2 processes P0 and P1.
  - semaphores *A* and *B*, initialized to 1

```
     P0                        P1
wait (A);                  wait(B)
wait (B);                  wait(A)
signal(B);               signal(A)
signal(A);                signal(B)
```

P0 and P1 are deadlocked

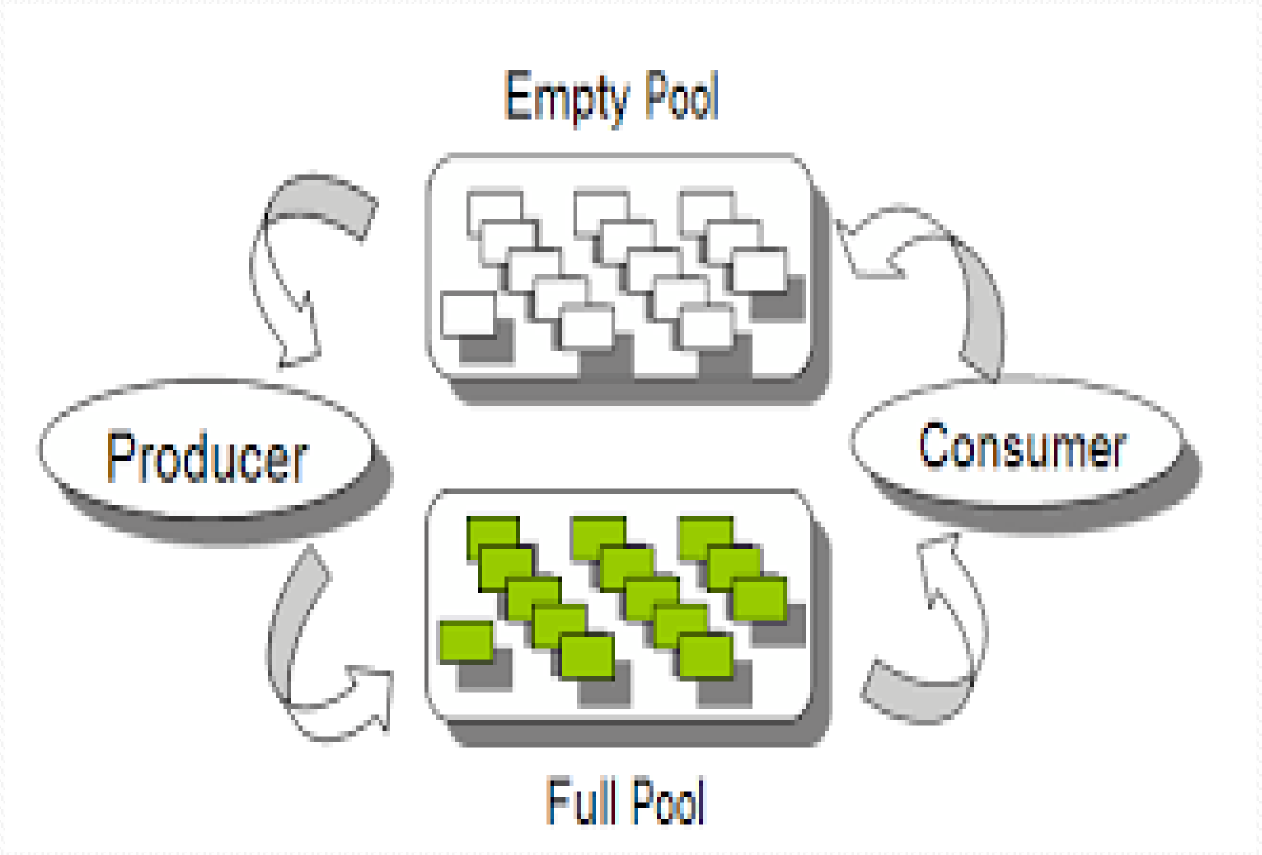
Classical problems of Synchronization

- Bounded-Buffer Producer/Consumer Problem
- Readers and Writers Problem
- Dining Philosophers Problem (Monitor)

Bounded-Buffer Producer/Consumer Problem

- Shared data: semaphore full, empty, mutex
- Initially: full = 0, empty = n, mutex = 1

where n is the buffer size



Empty → Empty Buffer
Full → Full Buffer

Producer → make buffer full
Consumer → empty the buffer

Explanation

- Consider a pool contains “n” buffers and each buffer can hold an item. The shared variable “mutex” provides the required mutual exclusion for the buffer pool, “empty” and “full” are semaphore variables used to count the number of empty and full buffers.
- Need to initialize the mutex variable as ‘1’, empty=‘n’ and full=‘0’.
- According to the production and consumption by the producer and the consumer, the variables empty and full will get modified.
- With the help of “wait” and “signal” method of semaphores, the bounded buffer problem can be handled properly.

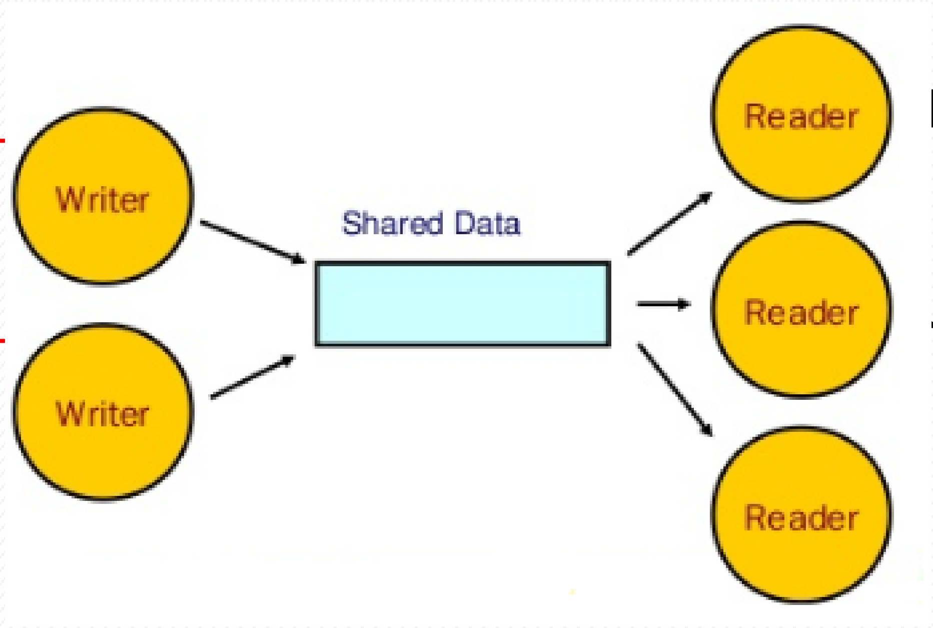
The structure of the Producer process

```
do  {

  //   produce an item in nextp

            wait (empty);
            wait (mutex);

 //  add the item to the  buffer

             signal (mutex);
             signal (full);
        } while (TRUE);
```

The structure of the consumer process

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

// remove an item from  buffer to nextc

            signal (mutex);
            signal (empty);

//consume the item in nextc

      } while (TRUE);
```

- A data set is shared among many 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
  - But Only one writer can write
- Shared data: re
  mutex, rw_mutex
- Initially: utex
  = 1, rw_mutex = 1,
  int readcount =0



Explanation

Consider a situation of having concurrent read and write operation over a common resource like database. In which, many users wants to read and write on the same database. If many users are concurrently perform read operation, it will not create any problem. Whereas if a write operation and any other operation(may be read or write) are concurrently performed on the common field may leads to inconsistencies in the database content.

This synchronization problem is called as “reader-writers” problem. The order in which the read and write operation performed may leads to starvation if they are not synchronized properly.

The structure of the
Reader process

```
do {
          wait (mutex) ;
          readcount ++ ;
          if (readcount == 1)
              wait (rw_mutex) ;

          signal (mutex)

        //reading is performed

           wait (mutex) ;
         readcount  - - ;
         if (readcount  == 0)
              signal (rw_mutex) ;
        signal (mutex) ;
    } while (TRUE);
```

The structure of the
Writer process

```
do {
         wait (rw_mutex) ;

        //    writing is performed

            signal (rw_mutex) ;
  } while (TRUE);
```

The structure of reader and writer process is given in the above figure.

“rw_mutex” semaphore variable is shared among readers and writer processes.

“rw_mutex” act as a mutual exclusion semaphore for writer processes.

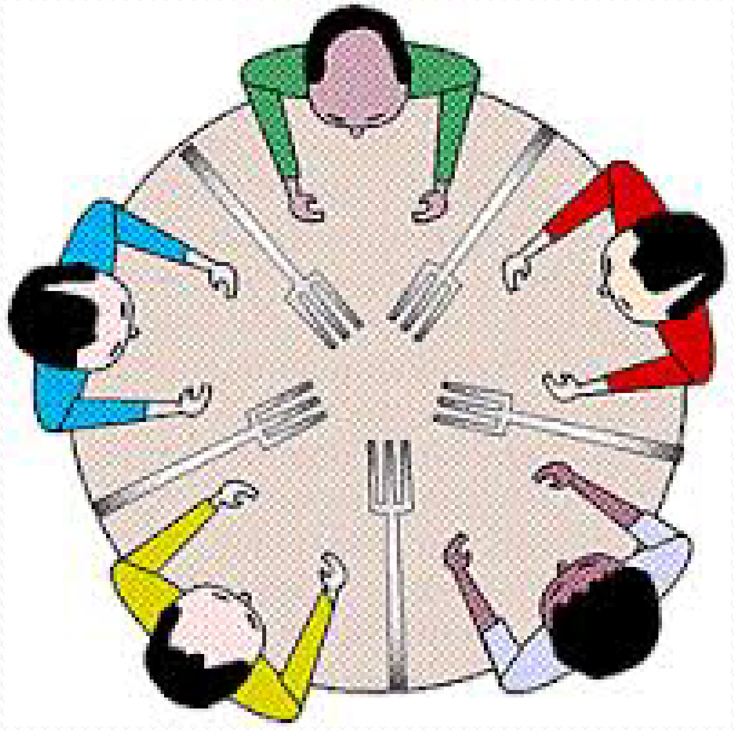
“read_count” variable get updated whenever a new reader process will come or when it gets completed.

“read_count” will specify the number of current read processes waiting for the resource.

With the help of “wait” and “signal” methods of semaphore, the synchronization between readers and writers processes can be achieved.

Dining-Philosophers Problem

- Five philosophers spend their lives thinking and eating
- Circular table with 5 chairs, 5 philosophers, 5 plates and 5 chopsticks
- Each philosopher need 2 chopsticks to eat. After eating, they drop chopsticks
- One person cannot pickup chopstick (ie) already in hand of a neighbour
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



The structure of the Philosopher

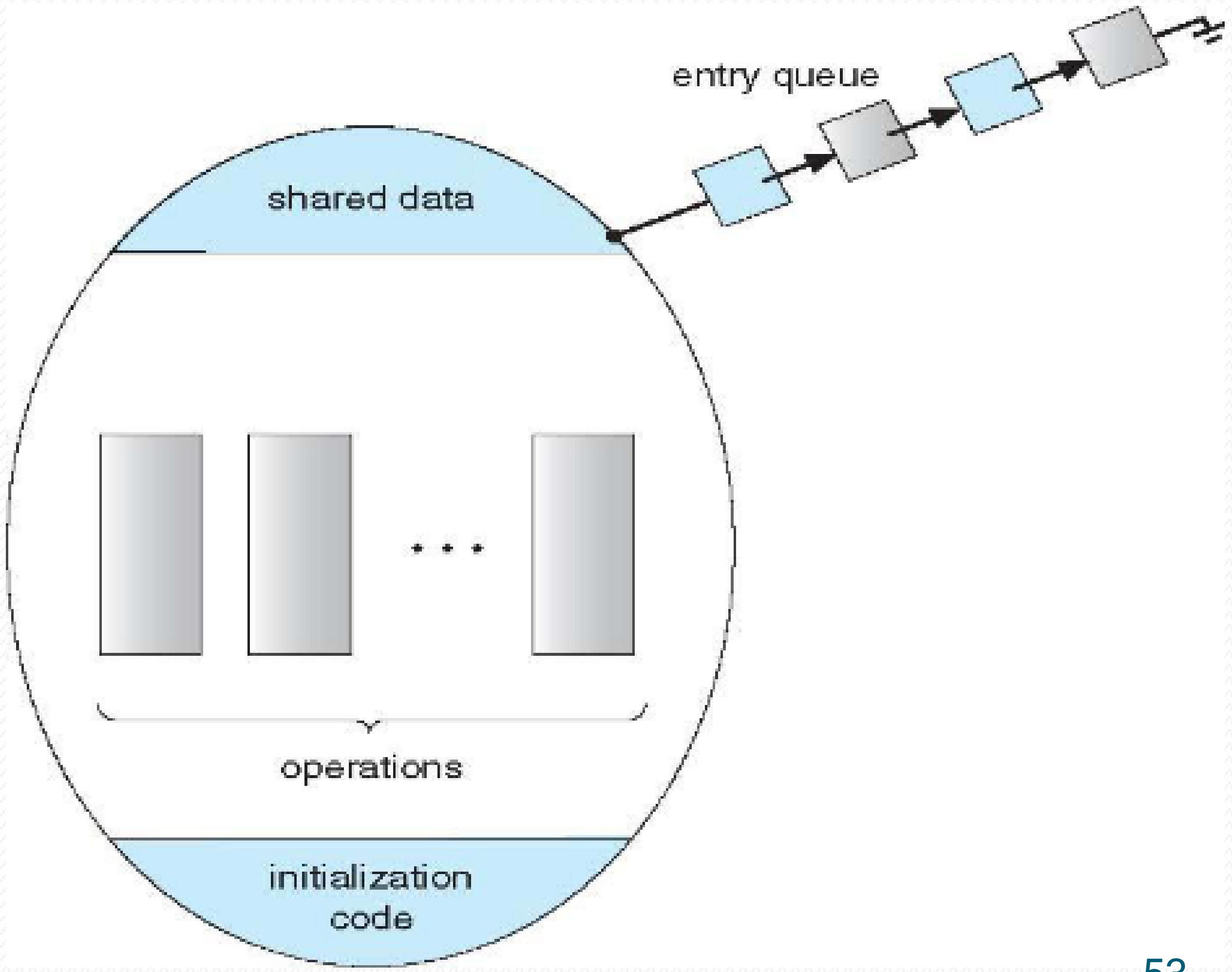
```
do  {
        wait ( chopstick[i] );
                   wait ( chopStick[ (i + 1) % 5] );

                         //  eat

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

              //  think

} while (TRUE);
```

To avoid deadlock in Dining Philosopher problem

- At most 4 philosophers to sit simultaneously
- Allow philosophers to pick chopsticks, only when both chopsticks are available
- Use an asymmetric solution
  - Odd philosopher picks up left chopstick first
  - Even philosopher picks up right chopstick

Monitors

- Monitor is **Abstract data type**, where the internal variables only accessible by code within the procedure
  - Programming languages like PASCAL, C# implement the concept of monitor
  - Only one process is active within the monitor at a time
  - Not powerful



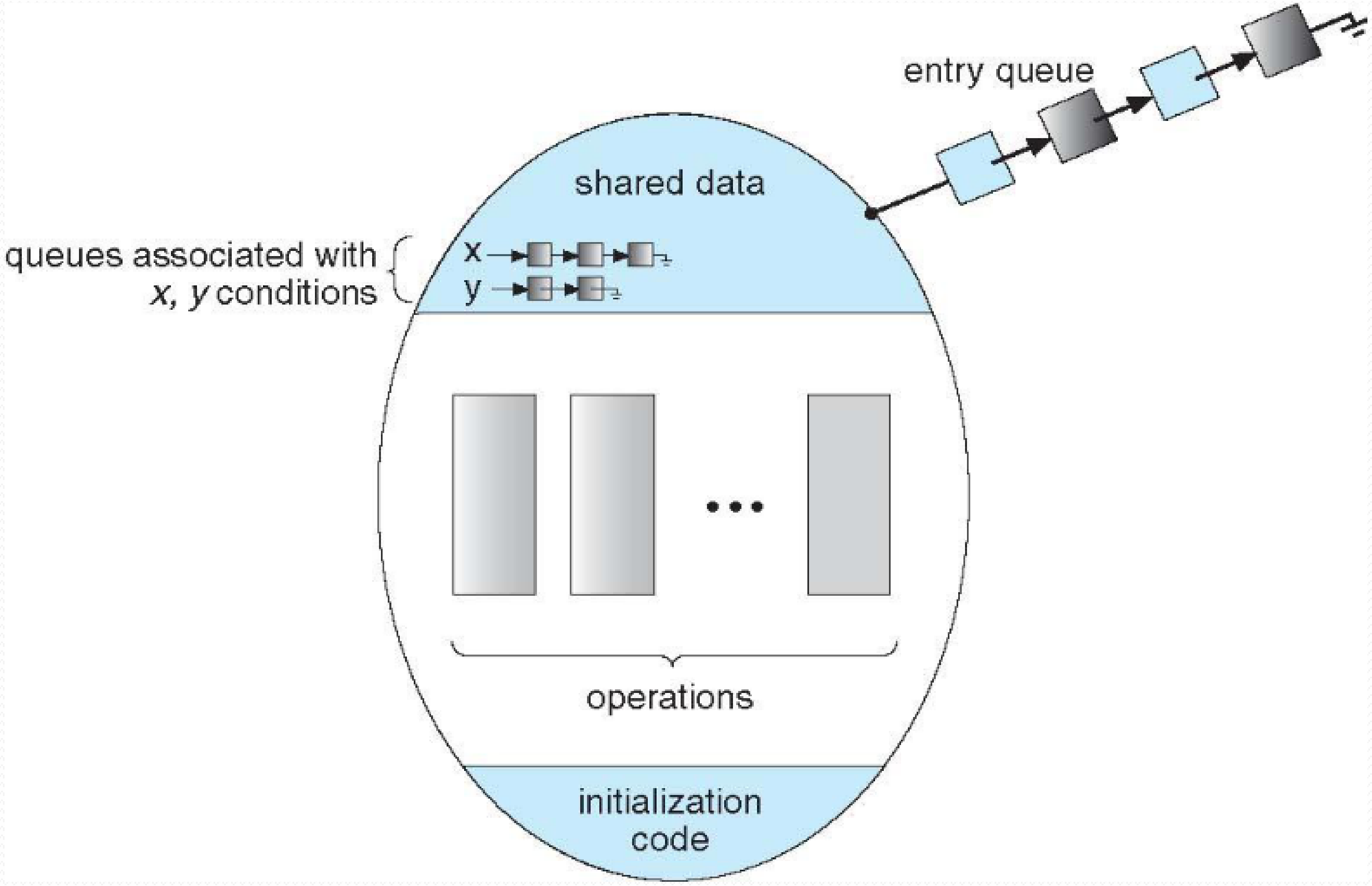
Variables & Operations

- Two variables of data type condition is used

  condition x, y;

- Two operations
  - **x.wait ()**
    - A process that invokes the operation until x.signal ()
  - **x.signal ()**
    - Resumes one of processes that invoked x.wait ()

Monitor with Condition Variables

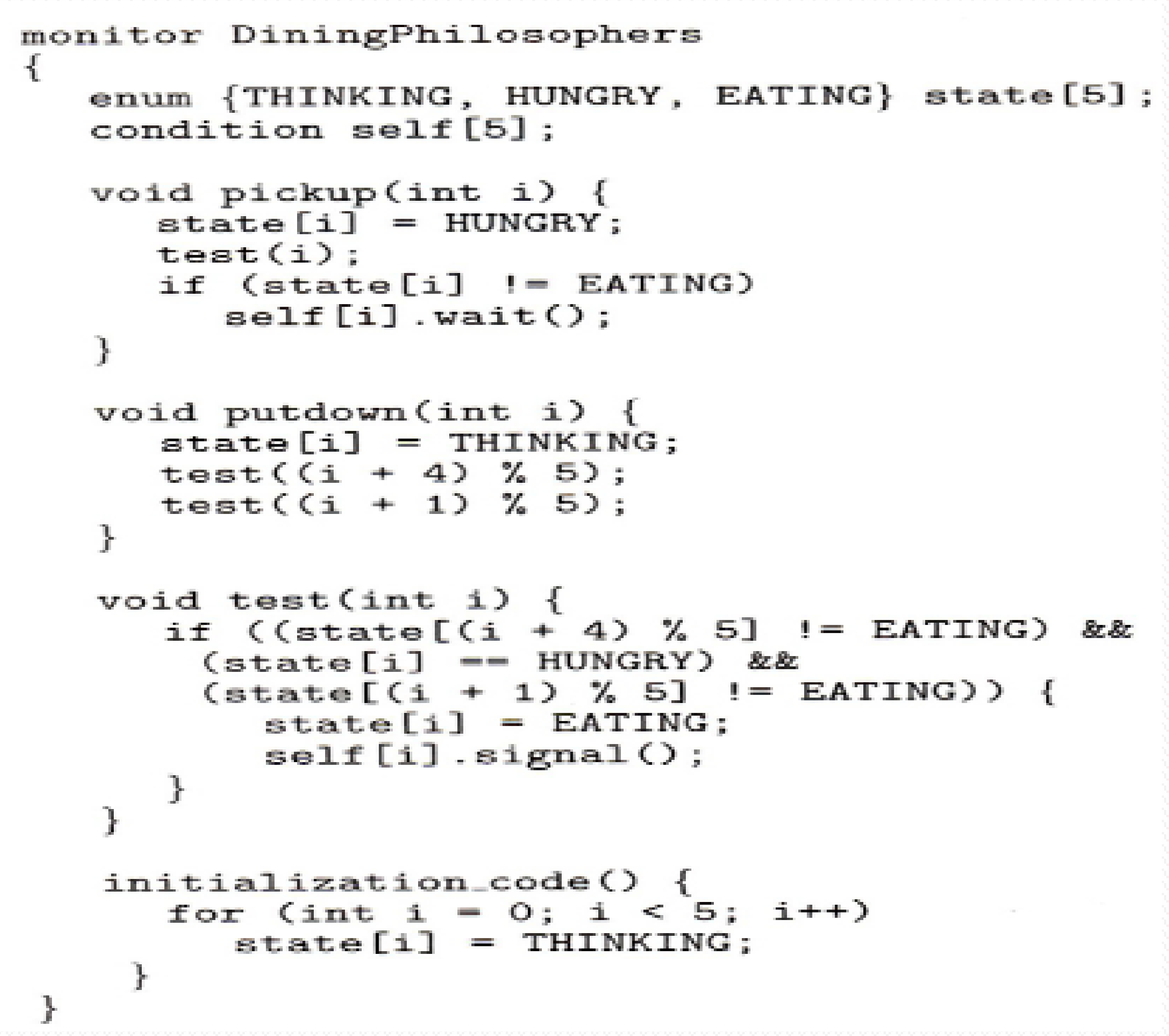


Monitor (syntax)

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
type monitor-name = monitor
          variable declarations
          procedure entry P1 :(…);
                    begin … end;
          procedure entry P2(…);
                    begin … end;
                       ⋮
          procedure entry Pn (…);
                    begin…end;
          begin
                    initialization code
          end
```

Monitor solution for
Dining Philosopher problem



```
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((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();
      }
   }

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

General structure of a Philosopher i using Pickup() and Putdown()

Syntax:

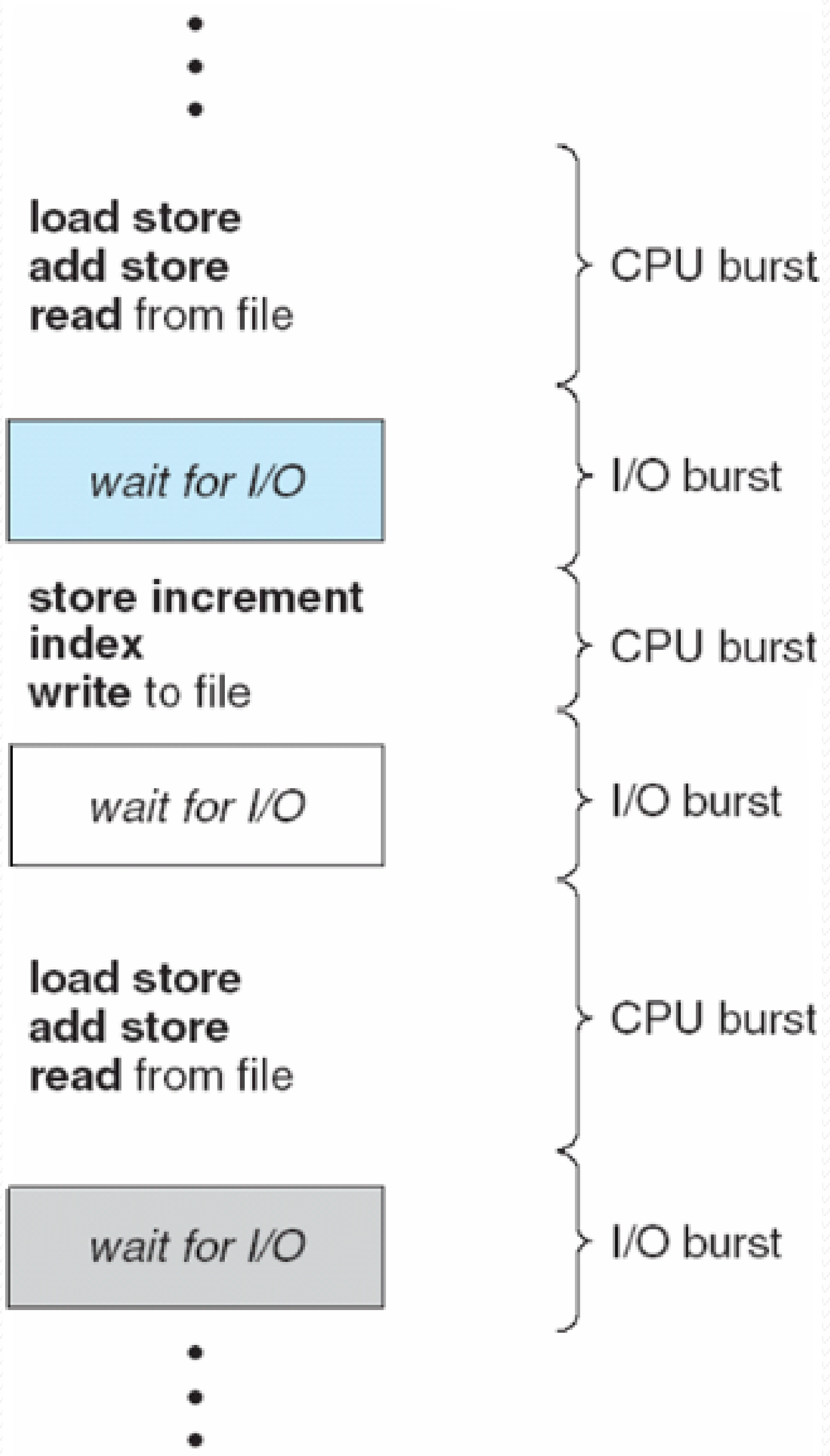
```
DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);
```

Note : This solution ensures that **no two neighbors** are eating simultaneously and that no deadlocks will occur.

PROCESS SCHEDULING

Basic Concepts

- Maximum CPU utilization is obtained with multiprogramming
- Process execution consists of a *cycle* of a **CPU time burst** and an **I/O time burst**
  - Processes alternate between these two states (i.e., CPU burst and I/O burst)
  - Eventually, the final CPU burst ends with terminate execution



Preemptive:

The CPU is allocated to the process, if any higher priority process come it releases the CPU and get the service once the higher priority process completes.

Non Preemptive:

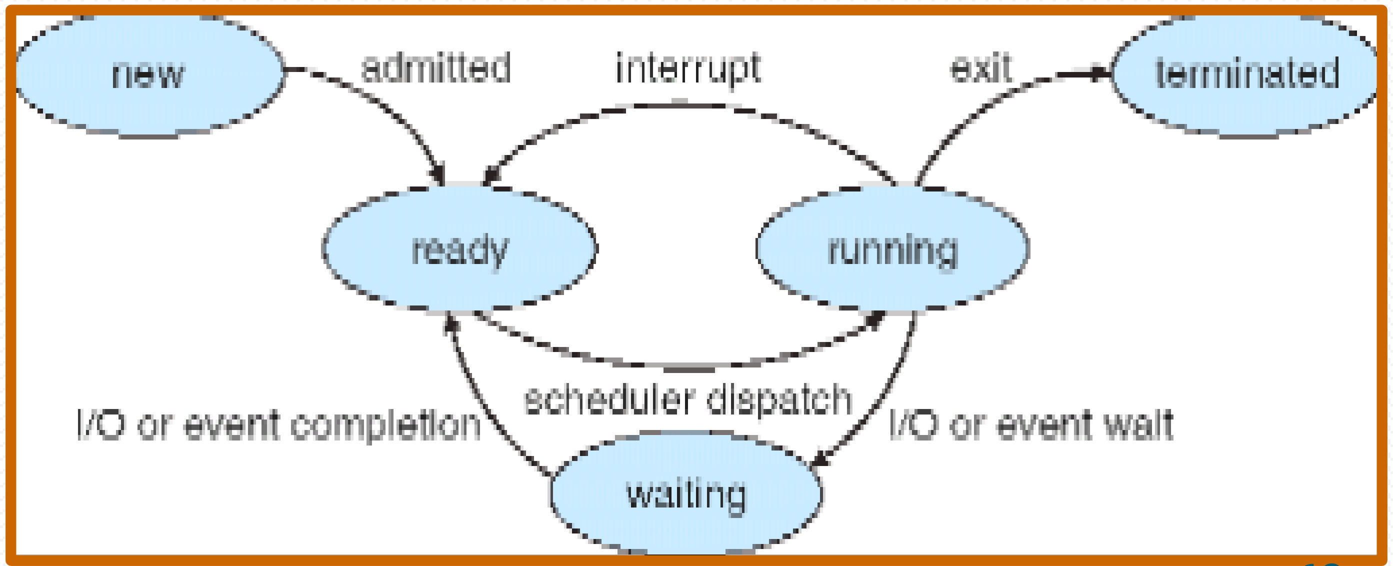
- Once the CPU is allocated to the process, the process keeps the CPU until it releases the CPU either by terminating or switching to waiting state.

CPU Scheduler

- The CPU scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them

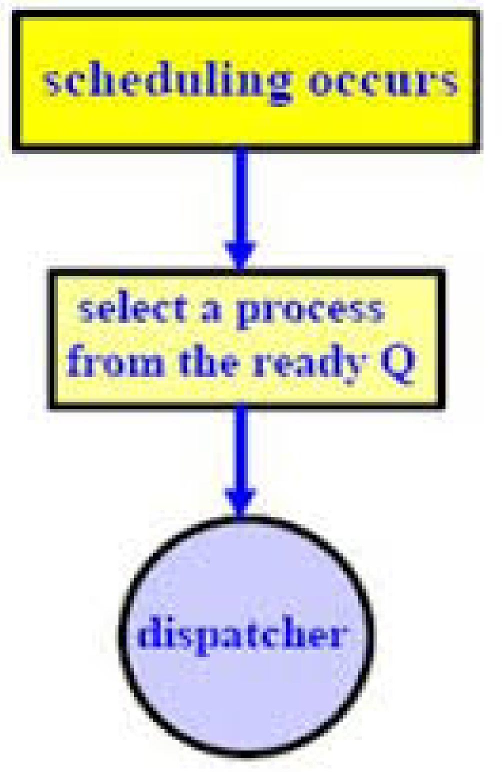
Ready Queue CPU

- When CPU scheduling takes place?
  1. (N) A process switches from **running** to **waiting** state
  2. (P) A process switches from **running** to **ready** state
  3. (P) A process switches from **waiting** to **ready** state
  4. (N) A processes switches from **running** to **terminated** state
- Circumstances 1 and 4 are **non-preemptive**
- Circumstances 2 and 3 are **pre-emptive**



Dispatcher

- The dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program



- The time it takes for the dispatcher to stop one process and start another process is called **dispatch latency**

Scheduling Criteria

- Different CPU scheduling algorithms have different properties
  - **CPU utilization** – keep CPU as busy as possible
    - CPU utilization ranges from 0% to 100%
    - Lightly loaded system 40%
    - Heavily loaded system 90%
  - **Throughput** = Number of processes completed /Unit time
  - **Response time** – amount of time it takes from when a request was submitted until the first response occurs
  - **Waiting time** – the amount of time the processes has been waiting in the ready queue
  - **Turnaround time** – amount of time to execute a particular process from the time of submission through the time of completion

Scheduling Algorithms

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-First (SJF) Scheduling
   - Simultaneous arrival times
   - Varied arrival times
   - Preemptive SJF with varied arrival times = Shortest-remaining time First (SRT) Scheduling
3. Priority Scheduling
   - Preemptive & non preemptive
4. Round robin scheduling
5. Multi-level Queue Scheduling
6. Multilevel Feedback Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

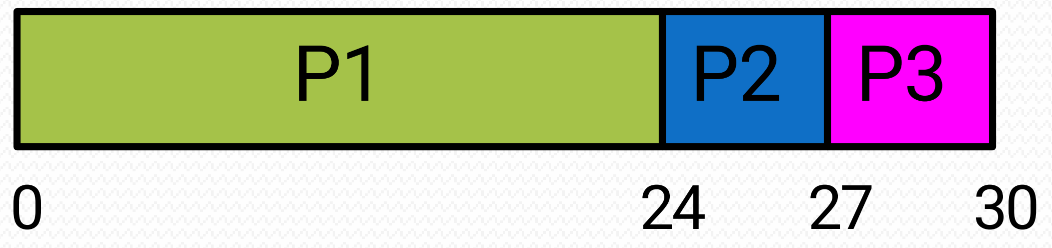
- The first entered job is the first one to be serviced.

- Example: Three processes arrive in order P1, P2, P3.
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3

- Draw the Gantt Chart and compute Average Waiting Time and Average Completion Time.

First-Come, First-Served (FCFS)

- Example: Three processes arrive in order P1, P2, P3.
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3
- Waiting Time
  - P1: 0
  - P2: 24
  - P3: 27
- Completion Time
  - P1: 24
  - P2: 27
  - P3: 30
- Average Waiting Time: (0+24+27)/3 = **17**
- Average Turnaround time: (24+27+30)/3 =**27**



Convoy effect (2 mark)

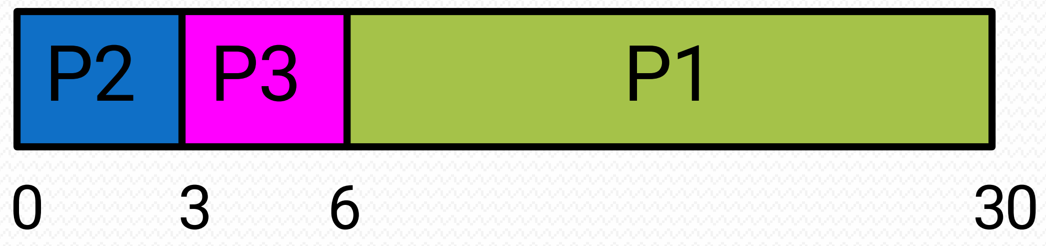
All the other processes wait for one long process to finish its execution

First-Come, First-Served (FCFS)

- What if their order had been P2, P3, P1?
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3

First-Come, First-Served (FCFS)

- What if their order had been P2, P3, P1?
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3
- Waiting Time
  - P2: 0
  - P3: 3
  - P1: 6
- Turn-around Time
  - P2: 3
  - P3: 6
  - P1: 30
- Average Waiting Time: (0+3+6)/3 = 3 (compared to 17)
- Average turn-around Time: (3+6+30)/3 = 13 (compared to 27)



Gnatt Chart

FIFO (First In and First Out) or FCFS

Advantages:

- Simple

Disadvantages:

- Short jobs get stuck behind long ones
- There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
- Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.
-

Shortest-Job-First (SJF) Scheduling
(simultaneous arrival ie. all jobs arrive at the same time)

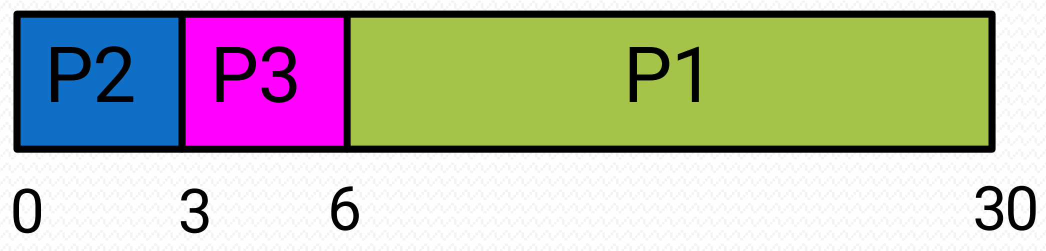
Example 1

- P1 burst time: 24
- P2 burst time: 3
- P3 burst time: 3

- Waiting Time
  - P2: 0
  - P3: 3
  - P1: 6

- Turn-around Time
  - P2: 3
  - P3: 6
  - P1: 30

- Average Waiting Time: (0+3+6)/3 = 3
- Average turn-around Time: (3+6+30)/3 = 13



Gnatt Chart

Shortest-Job-First (SJF) Scheduling

Here come the concept of arrival time.

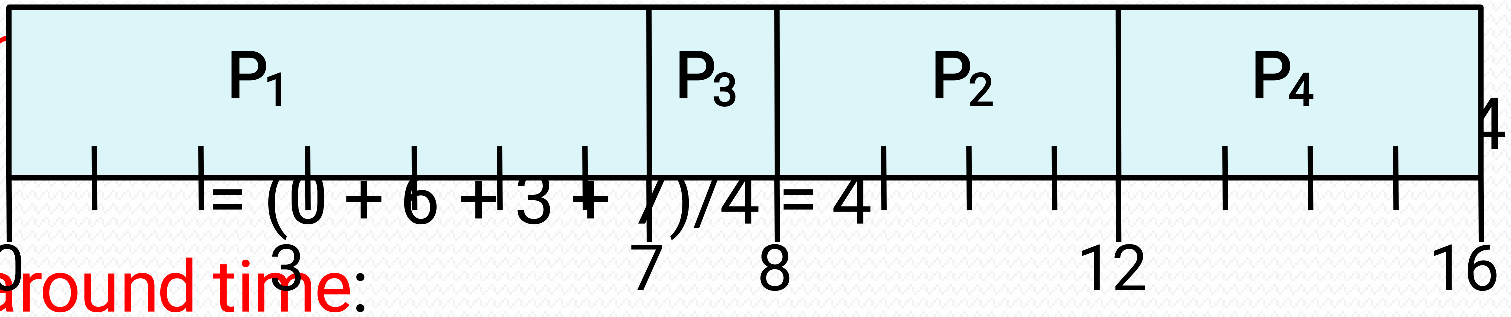
SJF (non-preemptive, varied arrival times)

Example 2

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- Average waitin

Gnatt Chart



= (0 + 6 + 3 + 7)/4 = 4

- Average turn-around time:

= ((7 – 0) + (12 – 2) + (8 - 4) + (16 – 5))/4

= (7 + 10 + 4 + 11)/4 = 8

Waiting time : sum of time that a process has spent waiting in the ready queue

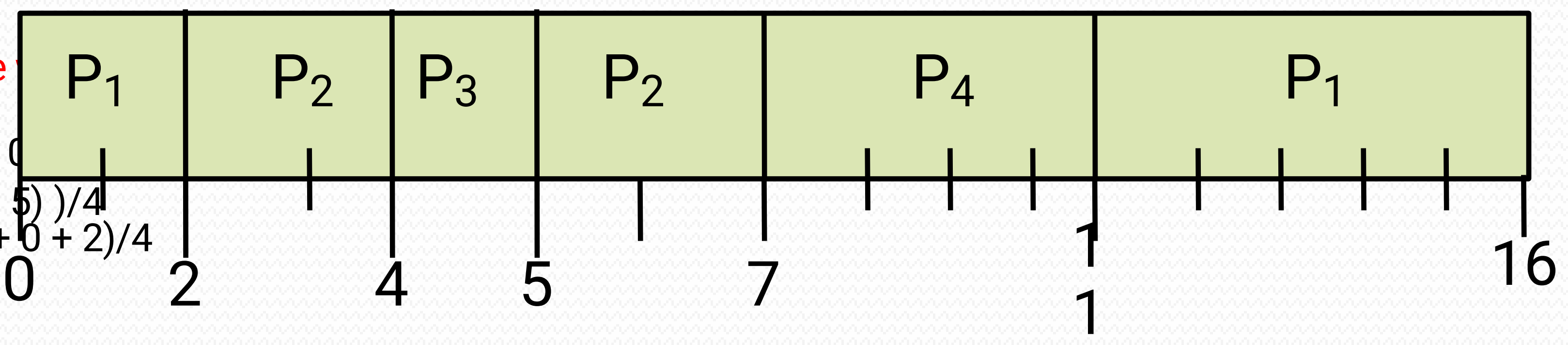
Shortest-remaining time First (SRT) Scheduling

Preemptive SJF with varied arrival times

Example 3

Process	Arrival Time	Burst Time
P₁	0.0	7
P₂	2.0	4
P₃	4.0	1
P₄	5.0	4

Gnatt Chart



- Average
- = ([(0 – 0
-   (7 – 5))/4
  = 9 + 1 + 0 + 2)/4
- = 3
- Average turn-around time = (16-0) + (7-2) + (5-4) + 11-4)/4 = 7

Shortest-Job-First (SJF) Scheduling

Pros and Cons

Advantages:

- Works based on the next process CPU burst
- It gives optimal waiting time

Disadvantages:

- Long jobs get stuck behind short ones

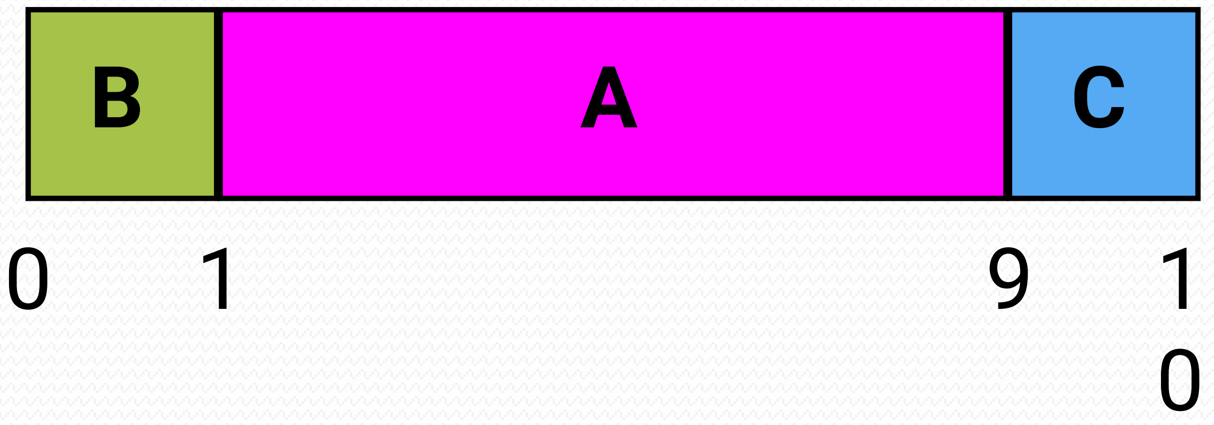
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

Priority Scheduling (non –Preemptive)

- A priority number (integer) is associated with each process (smallest integer = highest priority)

Process	Burst Time	Priority
A	8	2
B	1	1
C	1	3



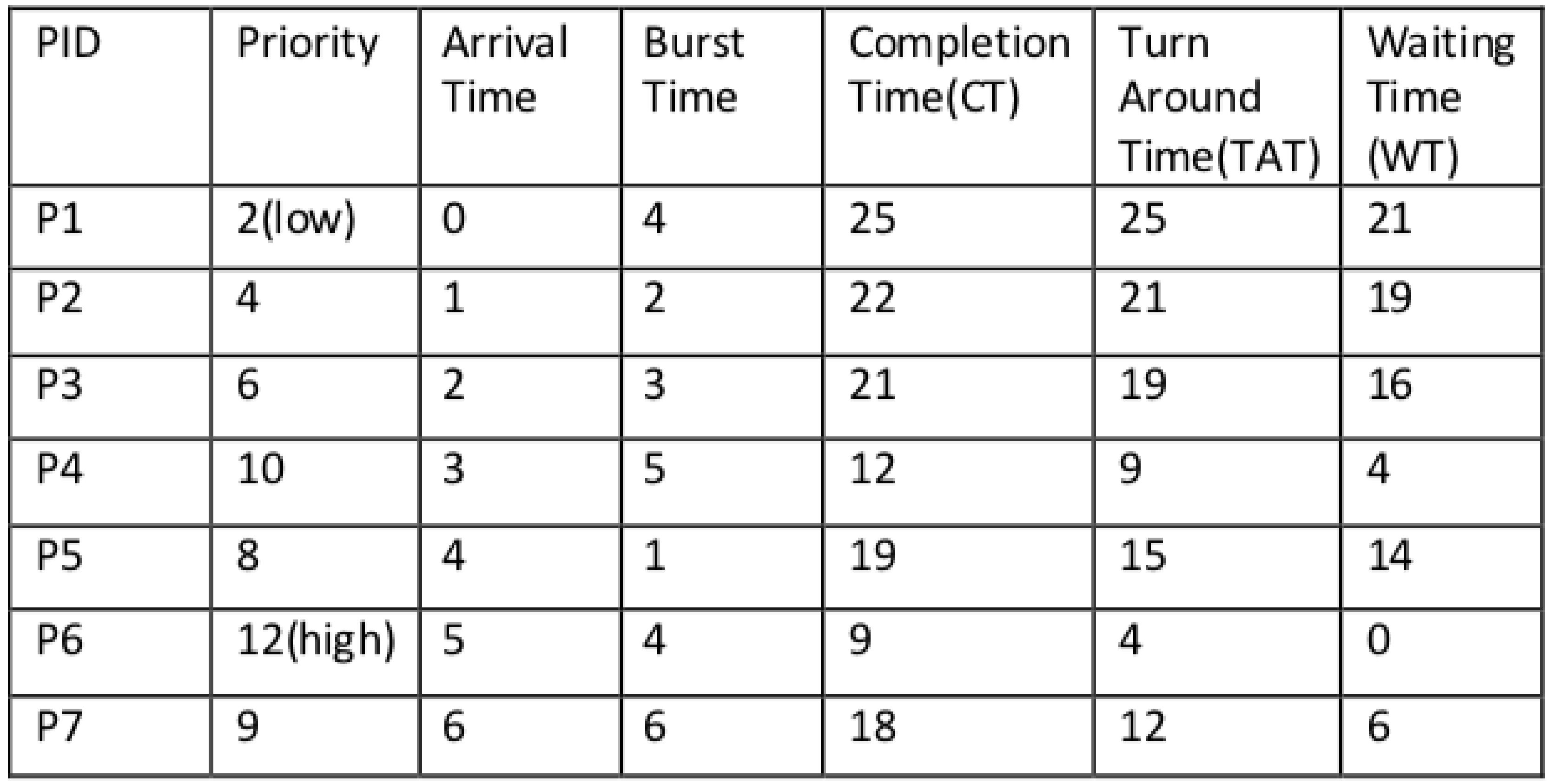
Gnatt Chart

- **Avg Wait Time (0 + 1 + 9) / 3 = 3.3**

Priority Scheduling

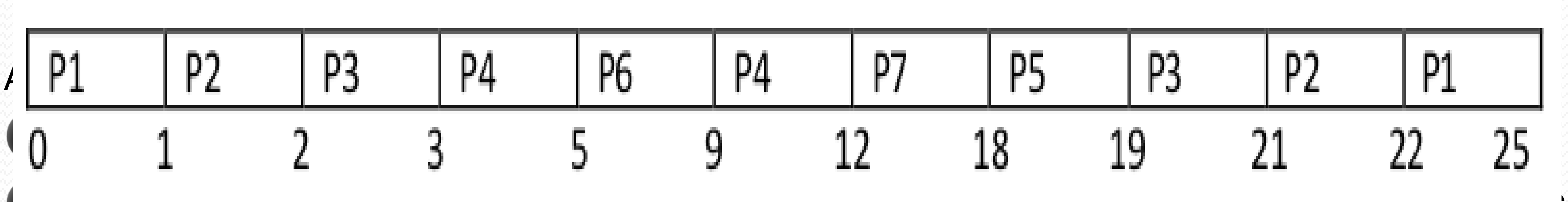
(Preemptive)

- Consider the example with seven process.



PID	Priority	Arrival Time	Burst Time	Completion Time(CT)	Turn Around Time(TAT)	Waiting Time (WT)
P1	2(low)	0	4	25	25	21
P2	4	1	2	22	21	19
P3	6	2	3	21	19	16
P4	10	3	5	12	9	4
P5	8	4	1	19	15	14
P6	12(high)	5	4	9	4	0
P7	9	6	6	18	12	6

- Gantt chart



- It is a situation in which the continuous arrival of higher priority process keeps the lowest priority process always in waiting state. The waiting process will starve (in other words, the deadline of the waiting process will never meet). We can resolve the starvation problem in the priority scheduling with the help of Aging technique.
- **Aging Technique**
- In Aging technique, the priority of every lower priority processes has to be increased after a fixed interval of time.

Priority Scheduling

Pros and Cons

Advantages:

- Higher priority job executes first

Disadvantages:

- Starvation ie. low priority processes never execute.

- To overcome the above problem “AGING” → the priority of a process is increased

Round Robin (RR) Scheduling

- In the round robin algorithm, each process gets a small unit of CPU time (a *time quantum*), usually 10-100 ms.

- After this time has elapsed, the process is preempted and added to the end of the ready queue.

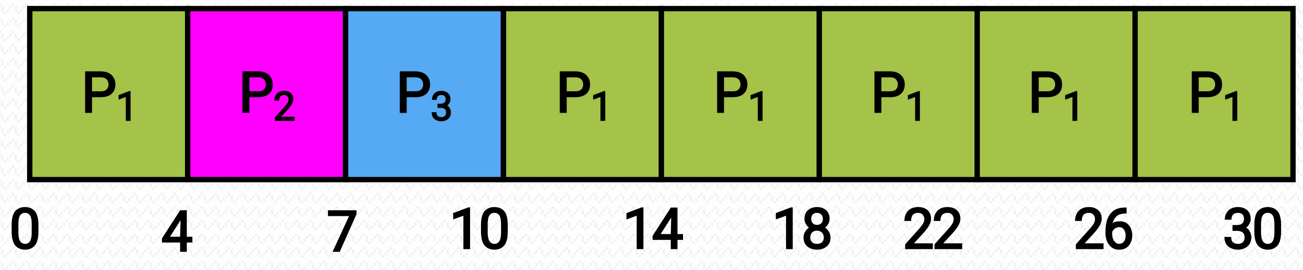
- Performance of the round robin algorithm
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ q must be greater than the context switch time; otherwise, the overhead is too high

Example of RR with Time Quantum = 4

Example 1

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



Average turn around time is larger than SJF
But more context switching

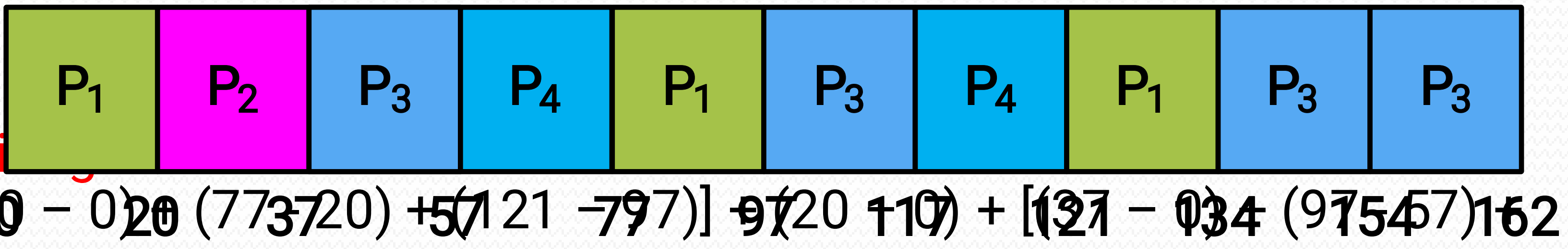
Average waiting time =(6+4+7)/3 = 5.6 ms

Example of RR with Time Quantum = 20

Example 2

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

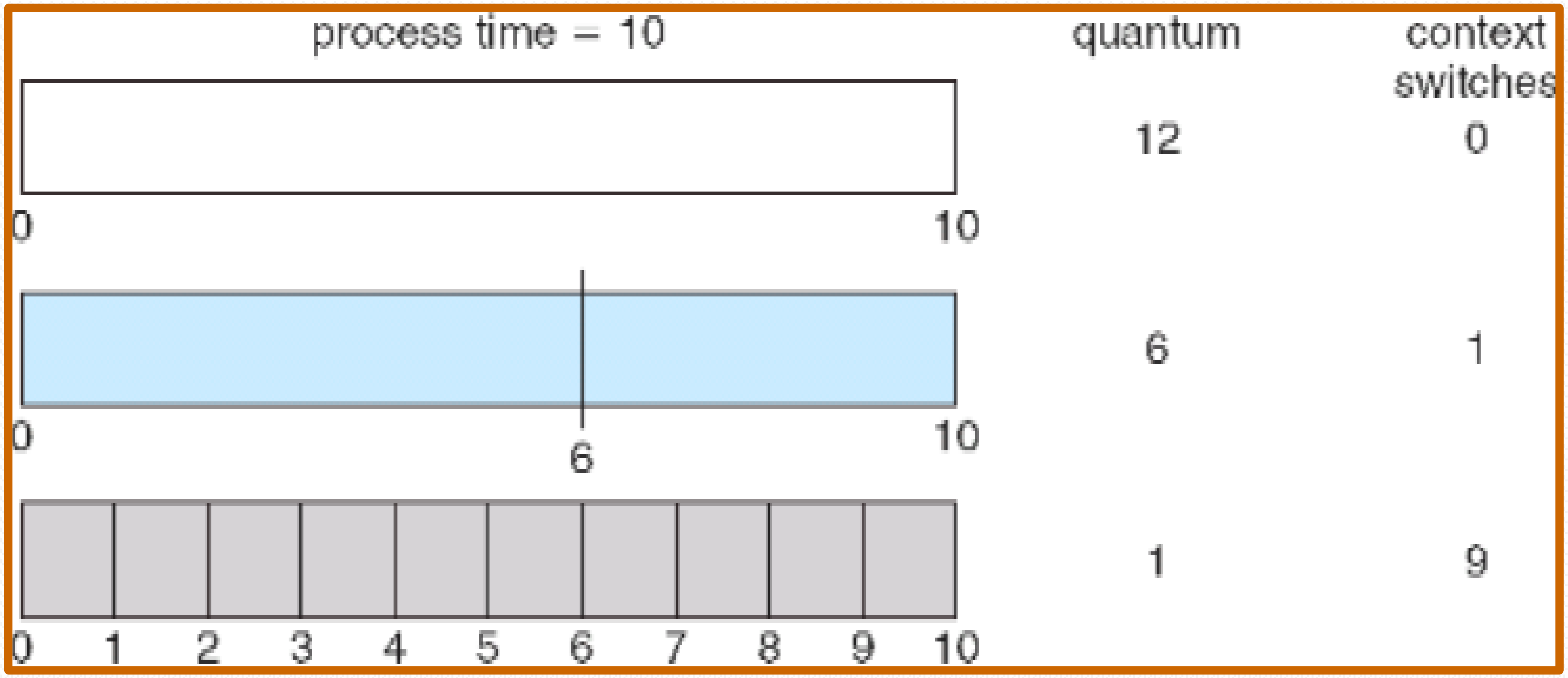
Gantt chart is:



- Average wai

  = ([(0 – 0) + (77 – 20) + (121 – 97)] + (20 – 0) + [(37 – 0) + (97 – 57) + (134 – 117)] + [(57 – 0) + (117 – 77)]) / 4
  = (0 + 57 + 24) + 20 + (37 + 40 + 17) + (57 + 40)) / 4
  = (81 + 20 + 94 + 97)/4
  = 292 / 4 = 73
- Average turn-around time = (134 + 37 + 162 + 121) / 4 = 113.5

Time Quantum and Context Switches



Round Robin (RR) Scheduling
Pros and Cons

Advantages:

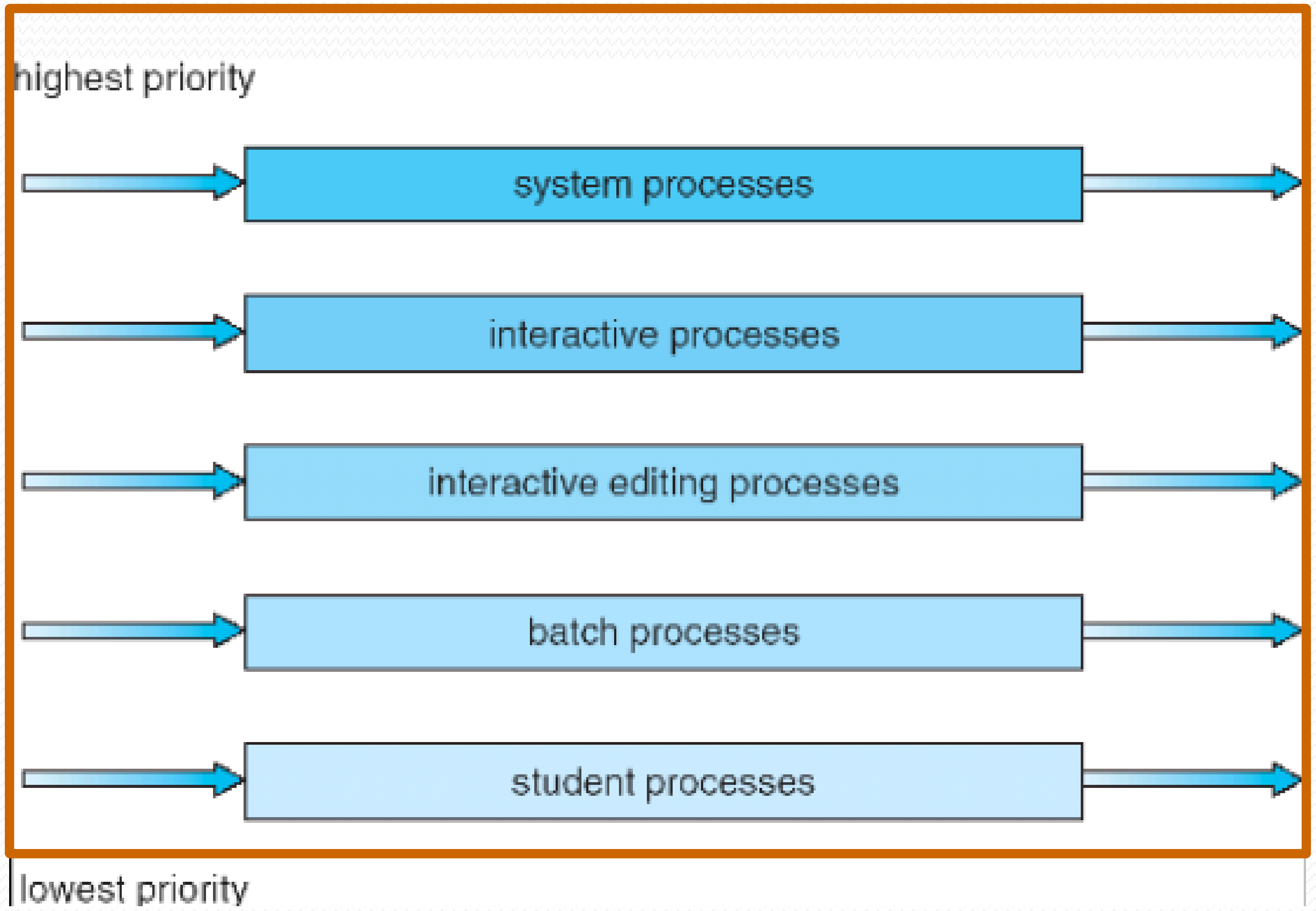
- Fair for smaller tasks

Disadvantages:

- More context switching

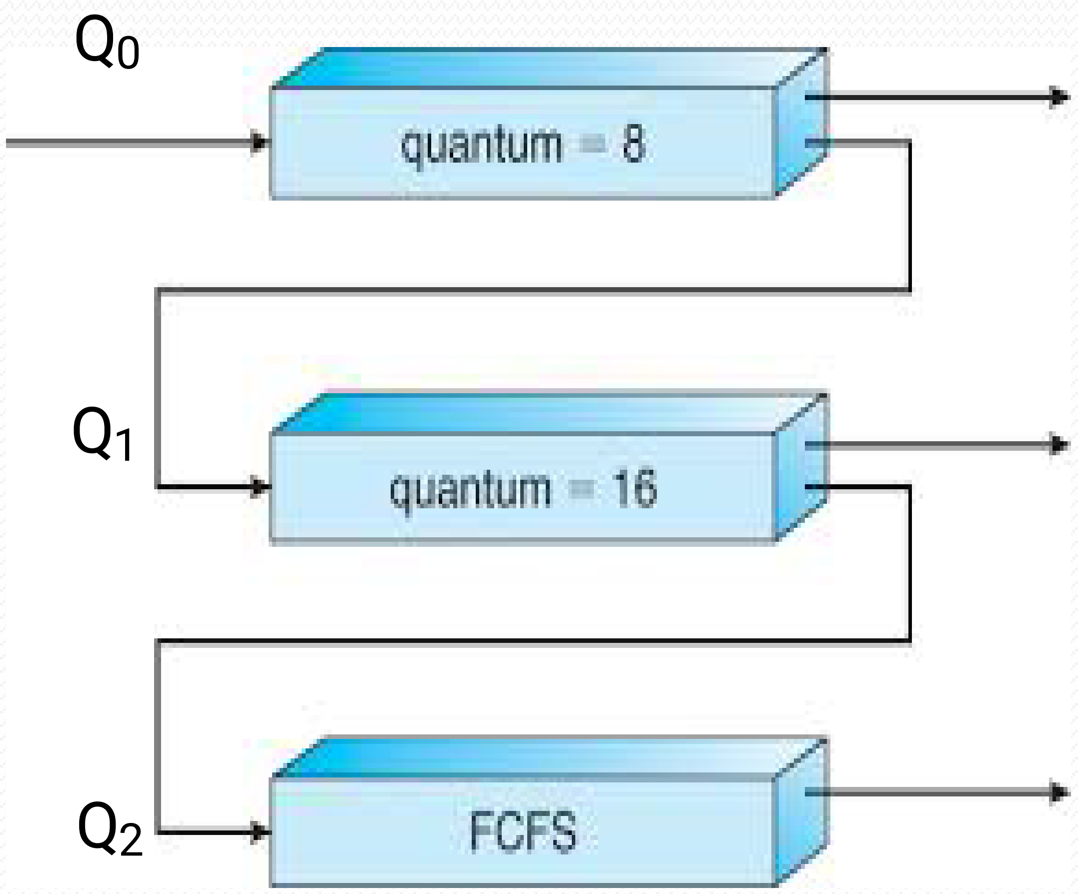
Multi-level Queue Scheduling

- Multi-level queue scheduling is used when processes can be classified into groups
  - For example, foreground (interactive) processes and background (batch) processes
    - 80% of the CPU time to foreground queue using RR.
    - 20% of the CPU time to background queue using FCFS



Multilevel Feedback Queue Scheduling

- In multi-level feedback queue scheduling, a process can move between the various queues;
  - A new job enters queue Q_0 *(RR)* and is placed at the end. When it gains the CPU, the job receives 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to the end of queue Q_1.
  - A Q_1 (RR) job receives 16 milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 (FCFS).

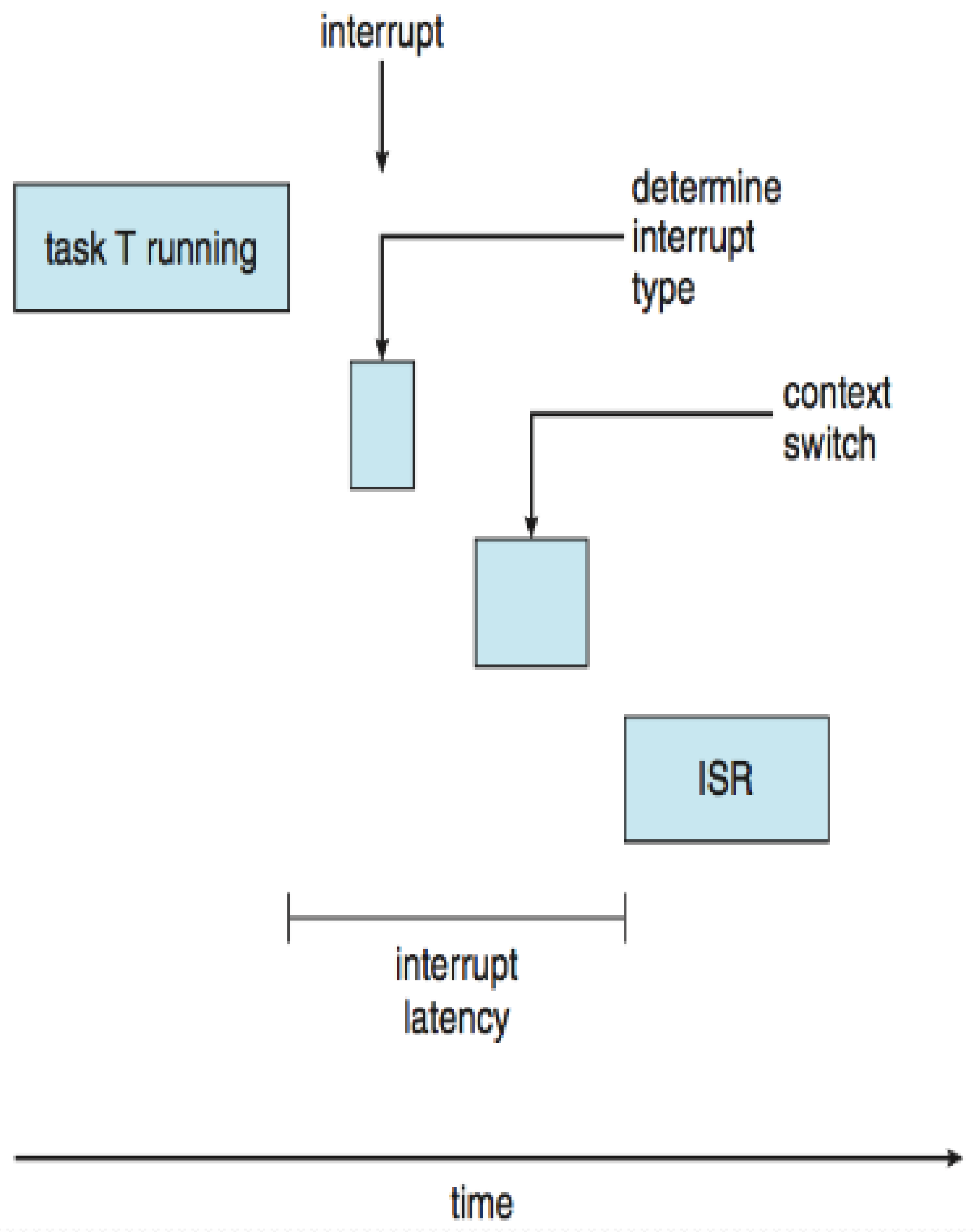


- **CPU scheduling for real-time operating systems involves special issues**.
- In general, we can distinguish between soft real-time systems and hard real-time systems.
- **Soft real-time systems** provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes.
- **Hard real-time systems** have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.
- In this section, we explore several issues related to process scheduling in both soft and hard real-time operating systems

- Two types of latencies affect performance

1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt

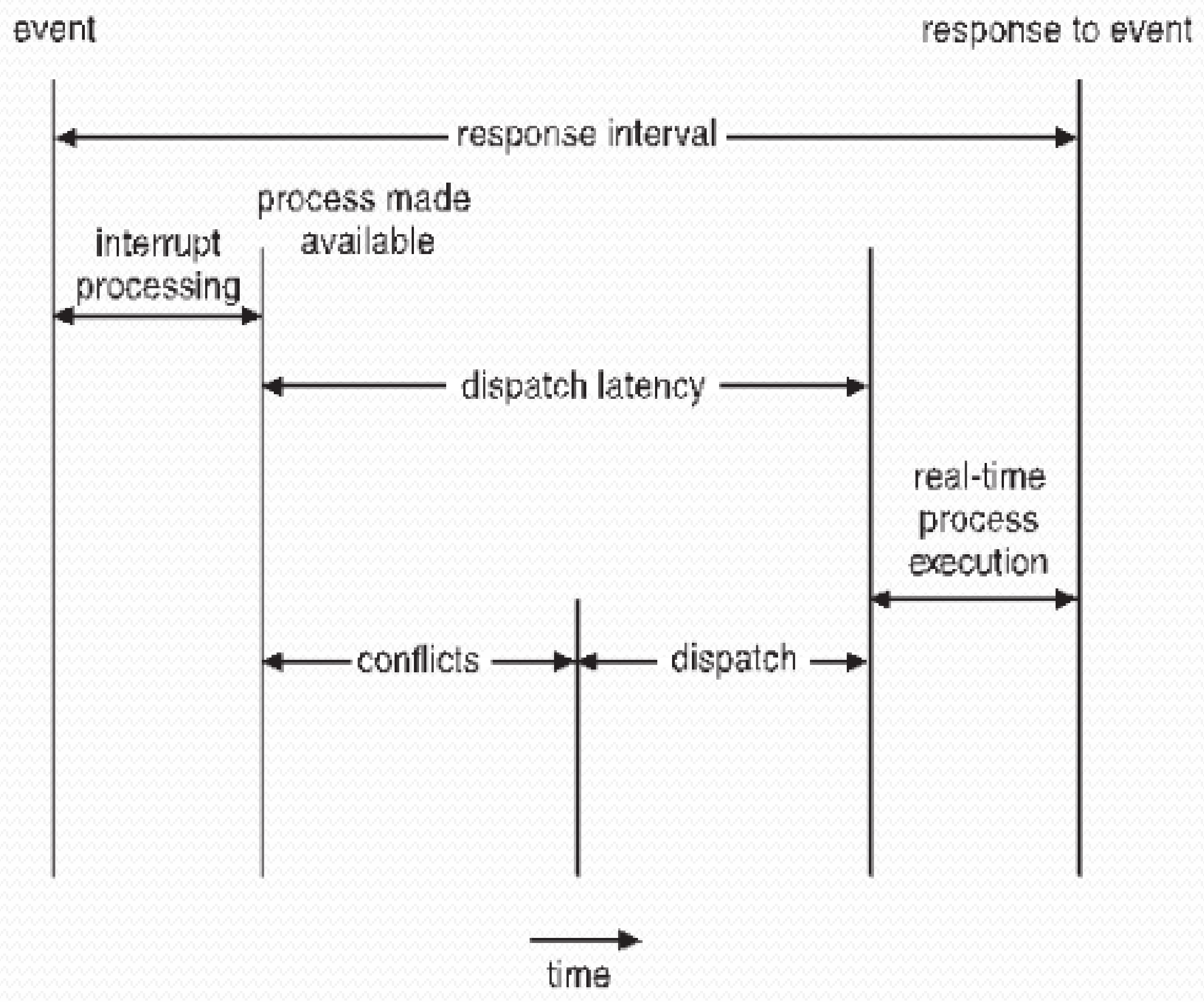
1. **Dispatch latency** – time for schedule to take current process off CPU and switch to another



Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode

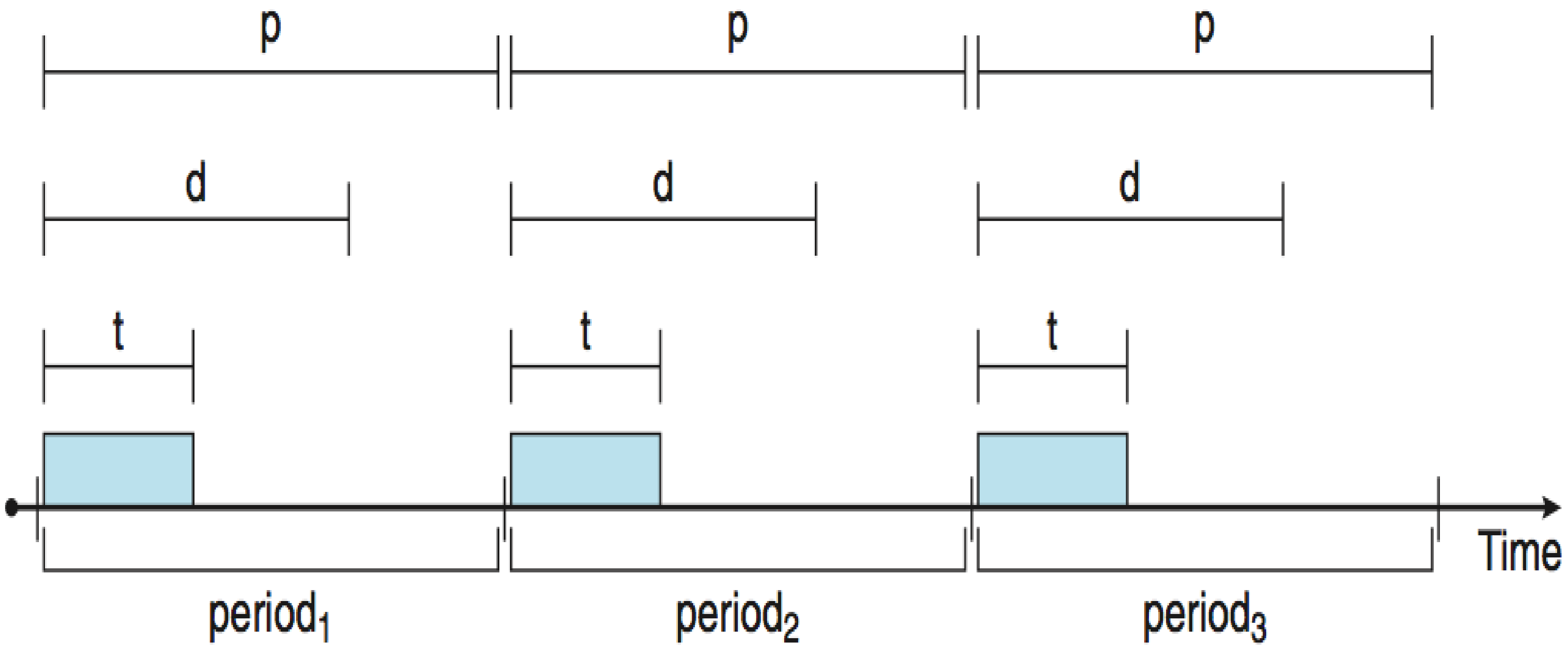
1. Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

- The most important feature of a real-time operating system is to **respond immediately to a real-time process as soon as that process requires the CPU**.
- As a result the scheduler for a real-time operating system must support a **priority-based algorithm with preemption**.
- Recall that priority-based scheduling algorithm **assign each process a priority based on its importance;** more are assigned higher priorities than those deemed less important.
- If the scheduler also supports preemption, a process currently running on the CPU will preempted if a higher-priority process becomes available to run.

- Before we proceed with the details of the individual schedulers, how we must **define certain characteristics of the processes that are to be scheduled**.
- Periodic processes require the CPU at specified intervals (periods).
- **p is the duration of the period.**
- **d is the deadline** by when the process must be serviced.
- **t is the processing time.**
- The **relationship of the processing time, the deadline, and the period can be expressed as $0 \le t \le d \le p$.**


p
p
p
d
d
d
t
t
t
Time
period$_1$
period$_2$
period$_3$

Rate Monotonic Scheduling

- The rate-monotonic scheduling algorithm schedules periodic tasks using static priority policy with preemption.

- If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower priority process.

- Upon entering the system, **each periodic tasks and Priority inversely based on its period.**

- **Shorter periods = higher priority;**
- **Longer periods = lower priority**

- **The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often.**

- Furthermore, rate-monotonic scheduling assumes that the processing time of A periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same

Rate Monotonic Scheduling

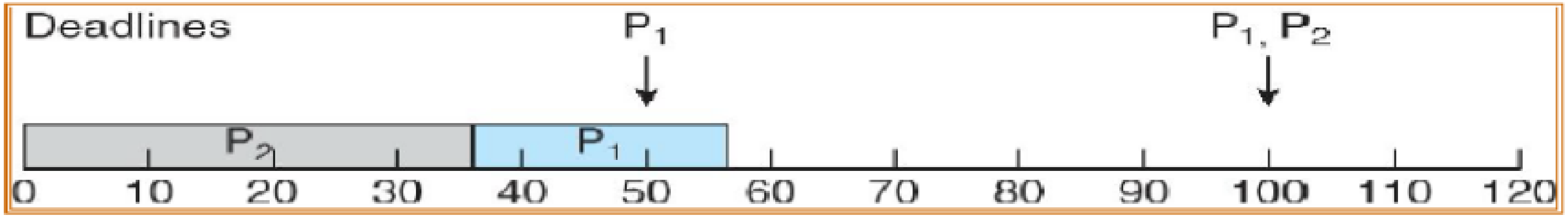
- **Let's consider an example. We** have **two processes, P1 and P2**.
- The **periods for P1 and P2 are p1=50, p2=100**. **The processing times are t1=20, t2=35.**
- **The deadline for each it complete its CPU burst by the start of its next period**.
- We must ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines.
- If we measure the **CPU utilization of a process Pi as the ratio of its burst to its period ti/pi.**
- The **CPU utilization of P1 is 20/50=0.40 and that P2 is 35/100 = 0.35**, for a **total CPU utilization of 75percent**.
- Therefore, it seems we can schedule these tasks in such a way that both **meet their deadlines and still leave the CPU with available cycles.**

Rate Monotonic Scheduling

Example(Why not Priority scheduling)

- Suppose we **assign P2 a higher priority than P1**.
- The execution of P1 and P2 in this situation is shown in the below Figure.
- As we can see, P2 starts execution first and completes at time 35.
- At this point, P1 starts; it completes its CPU burst at time 55.
- However, the first deadline for P1 was at time 50, so the scheduler has caused **P1 to miss its deadline.**

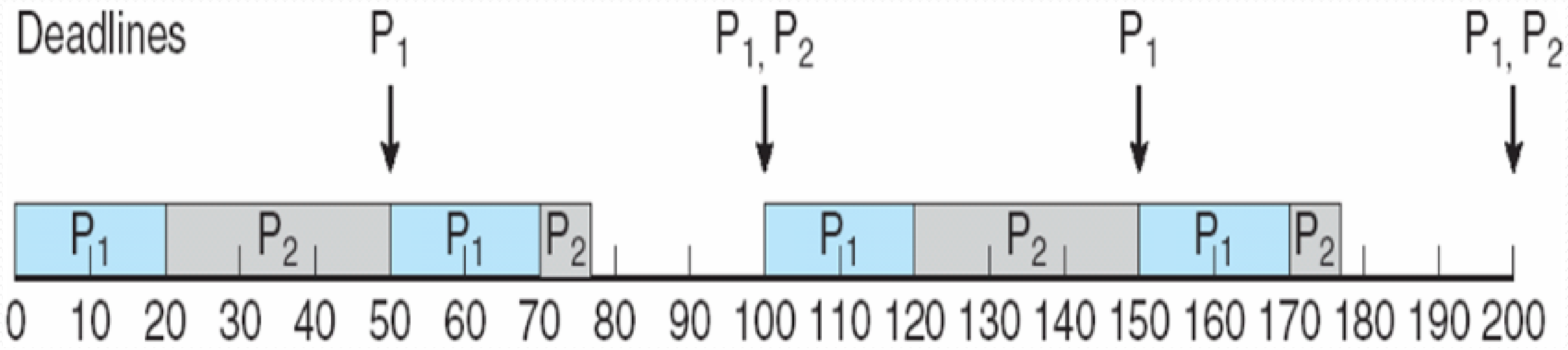
Scheduling of tasks when P2 has a higher priority than P1



Rate Monotonic Scheduling

Now suppose we use rate-monotonic scheduling, in which we assign **P1 a higher priority than P2 because the period of P1 is shorter than that of P2.**

- The execution of these processes in this situation is shown in the below Figure.
- P1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline.
- P2 starts running at this point and runs until time 50.
- At this time, it is preempted by P1, although **it still has 5 milliseconds remaining in its CPU burst.**
- P1 completes its CPU burst at time 70, at which point the scheduler resumes P2.



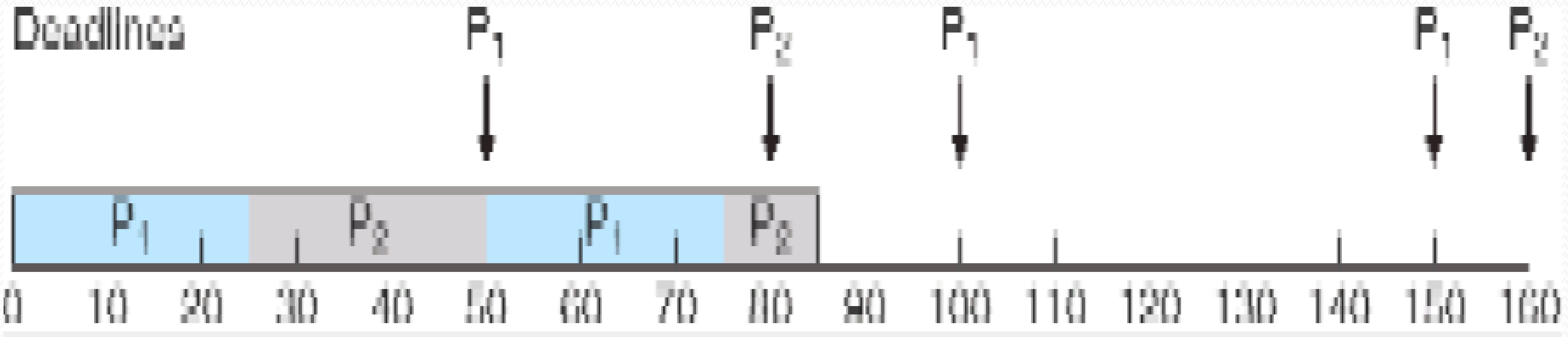
Missed Deadlines with Rate Monotonic Scheduling

- Let's next examine **a set of processes that cannot be scheduled using the rate Monotonic algorithm.**
- Assume that process P1 has a period of **p1 = 50** and a CPU burst of **t1=25**.
- For P2, the corresponding values are **p2= 80 and t2= 35**.
- **Rate-monotonic scheduling would assign process P1 a higher priority as it has the shorter period.**
- The total CPU utilization of the two processes is (25/50)+(35/80)=0.94, and it therefore seems logical that the two processes could be scheduled an leave the CPU with 6 percent available time.

Missed Deadlines with Rate Monotonic Scheduling

- Below figure shows the scheduling processes P1 and P2. Initially P1, runs until it completes its CPU burst at time 25.
- Process P2 then begins running and runs until time 50, when it is preempted by P1.
- At this point, P2, still has 10 milliseconds remaining in its CPU burst. Process P1 runs until time 75; consequently, P2 finishes its burst at time 85, after the deadline for completion of its CPU burst at time 80.

p1=50, p2=80 t1=25,
t2=35



Earliest Deadline First Scheduling (EDF)

- Earliest-deadline-first (EDF) scheduling dynamically **assigns priorities according to deadline.**

- The **earlier the deadline, the higher the priority; the later the deadline, the lower the priority.**

- Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process.

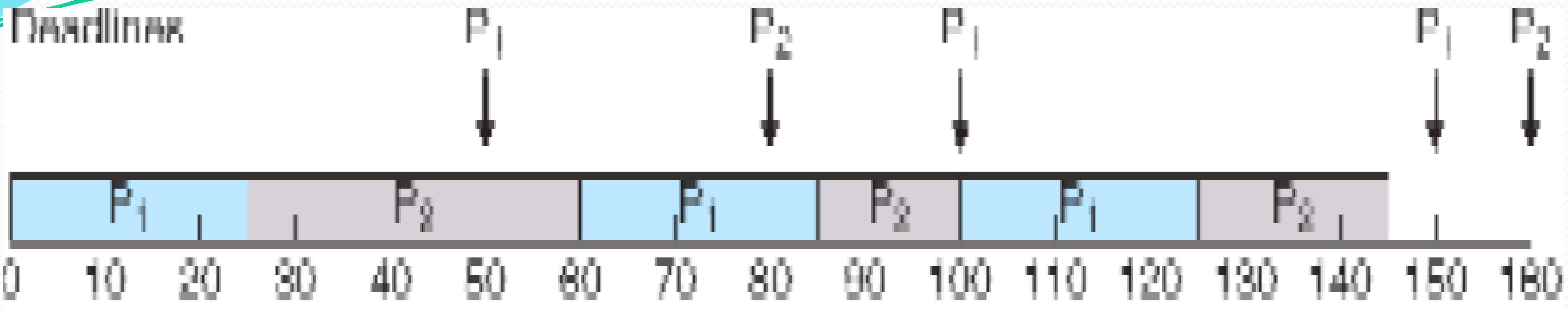
- Note how this differs from **rate-monotonic scheduling, where priorities are fixed.**

- **To illustrate EDF scheduling, we again schedule the processes which failed to meet deadline requirement under the rate-monotonic scheduling.**

Earliest Deadline First Scheduling (EDF)

- Recall that P1 has values of **p1=50 and t1=25 and that values of p2 = 80 and t2= 35**.
- The EDF scheduling of these processes in below figure. **Process P1 has the earliest deadline, so its initial priority is higher than that of process P2.**
- Process P2 begins running at the end of the CPU burst for P1. However, whereas **rate-monotonic scheduling allows P1 to preempt P2 at the beginning of its next period at time 50, EDF scheduling allows process P2 to continue running.**
- P2 now has a higher priority than P1, because its next deadline (at time 80) is earlier than that of P1 (at time 100).
- Thus, **both P1 and P2 meet their first deadlines**. Process P1 again begins running, at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100.
- P2 begins running at this point, only to be preempted by P1 at the start of its next period at time 100. P2 is preempted because P1 has an earlier deadline (time 150) than P2(time 160).
- At time 125, P1 completes its CPU burst and P2 resumes execution, finishing at time 145 and meeting its deadline a well.
- The system is idle until time 150, when P1 is scheduled to run once again.

Earliest Deadline First Scheduling (EDF)

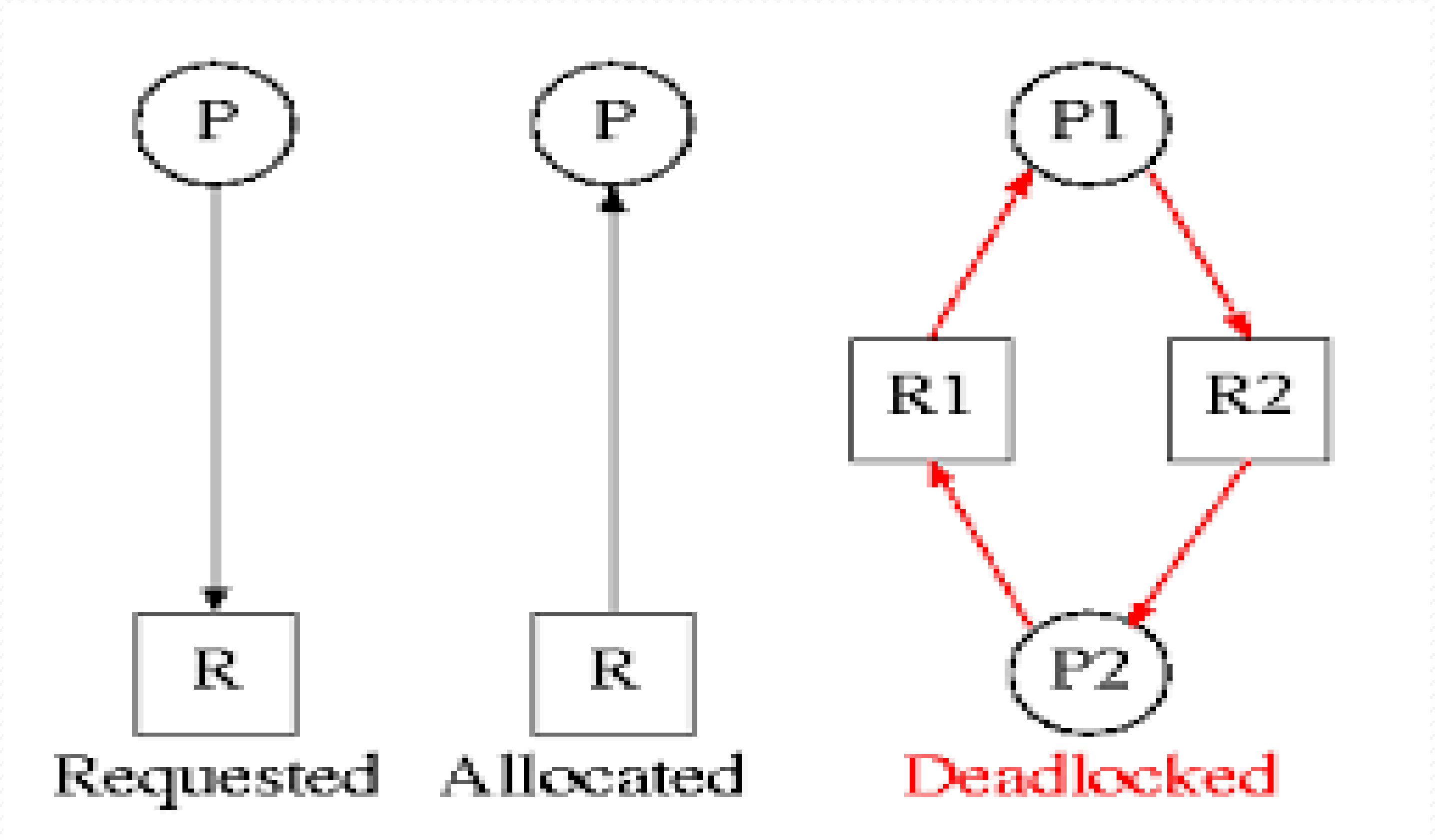


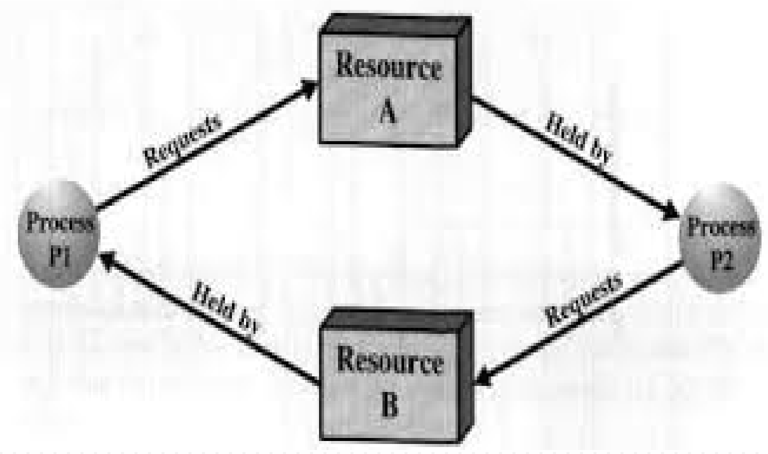
- Unlike the rate-monotonic algorithm, **EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst.**
- The **only requirement is that a process announce its deadline to the scheduler when it becomes runnable.**
- The appeal of EDF scheduling is that it is theoretically optimal -- theoretically, it can schedule processes so that each process **can meet its deadline requirements and CPU utilization will be 100 percent**.
- In practice, however, it is **impossible to achieve this level of CPU utilization** due to the cost of context switching between processes and interrupt handling.

Deadlocks

Deadlocks

- Assume 2 process, P1 and p2.
- When p1 process is holding resource R1 and requesting for resource R2, where it is hold by process P2. This state is **DEADLOCK**.





System Model

- Assume resource types $R_1, R_2, \ldots, R_m$

    CPU cycles, memory space, I/O devices

    - Each resource type R_i has *1 or more* instances

- Each process utilizes a resource as follows:
- Request

```
The process requests the resource.
If (resource == available)
    Grant the resource
else
    Wait
```

- Use
    - The process use the resource
- Release
    - The process release the resource

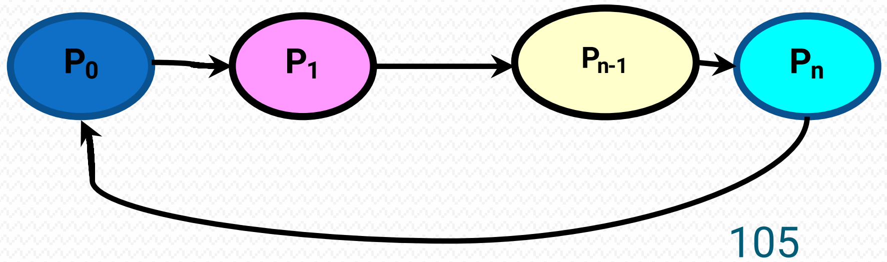
Deadlock Characterization

Repeated University Question

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:**
  - Only one process at a time can use a resource. If another process requests, they need to wait.
- **Hold and wait:**
  - A process holding at least one resource is waiting to acquire additional resources which is 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 $\{P_0, P_1, \ldots, P_0\}$ of waiting processes

  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
  P_n is waiting for a resource that is held by P_0



Resource-Allocation Graph

Deadlocks are described in terms of directed graph called Resource Allocation Graph.

Graph consists of a set of vertices *V* and a set of edges *E*.

Request edge:

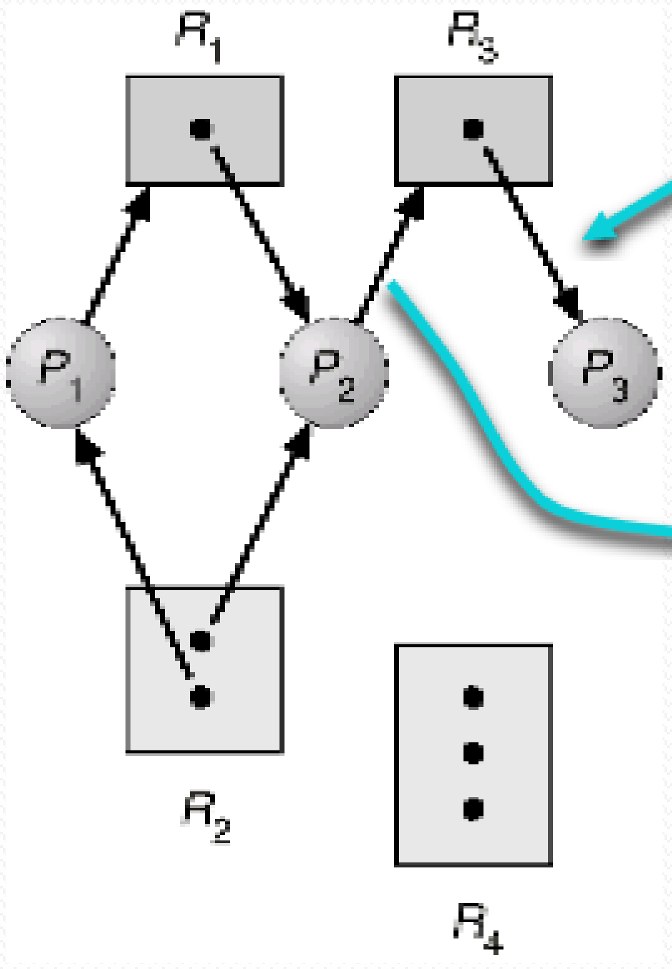
- It is a directed edge from P_1 to resource type R_j

  $P_1 \rightarrow R_j$

Assignment edge:

- It is a directed edge from R_j to resource type P_1

  $R_j \rightarrow P_1$



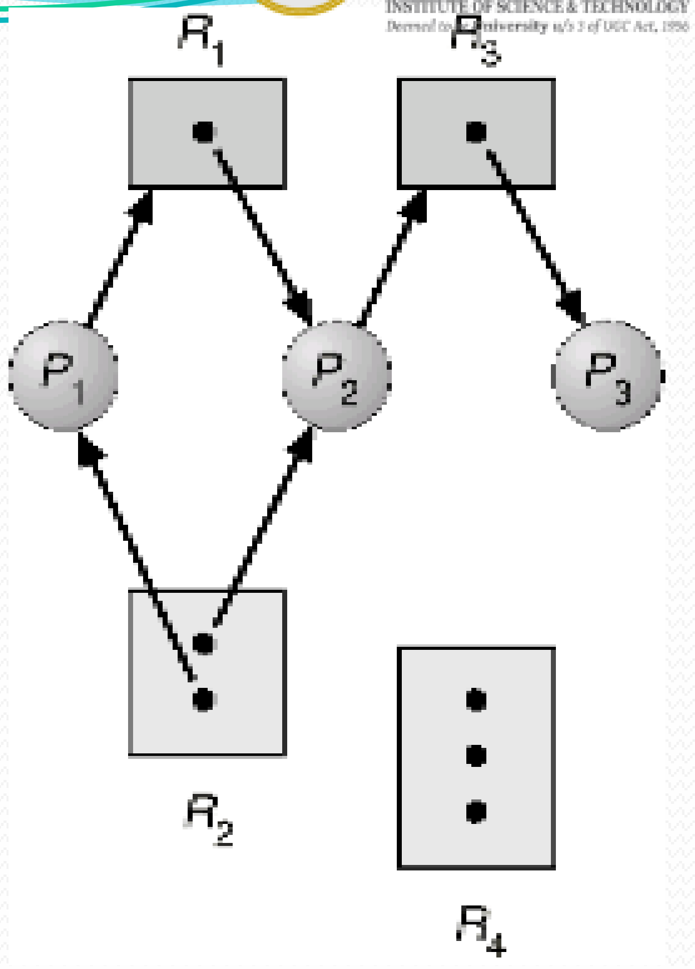
R3 Assigned to P3

P2 Requests R3

Note:
If resource type has more than 1 instance, its indicated by a dot within the rectangle.

Details

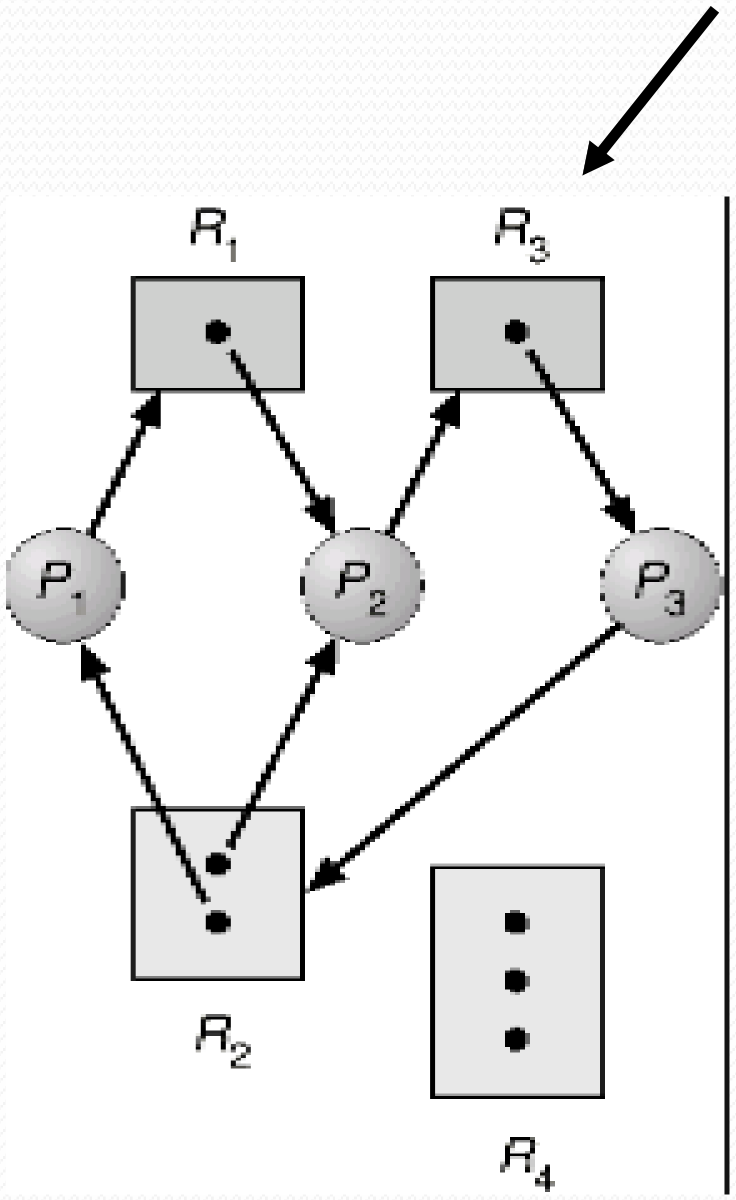
- The resource allocation graph consists of following sets:
  - P ={ P1,P2,p3}
  - R ={ R1, R2, R3, R4}
  - E = { p1 □ R1, P2 □ R3, R1 �� P2, R2□ P2, R2□ P1, R3□ P3}
  - P = Process; R = Resources; E = Edges.
- Resource Instance
  - One instance of resource type R1
  - Two instance of resource type R2
  - One instance of resource type R3
  - Three instance of resource type R4

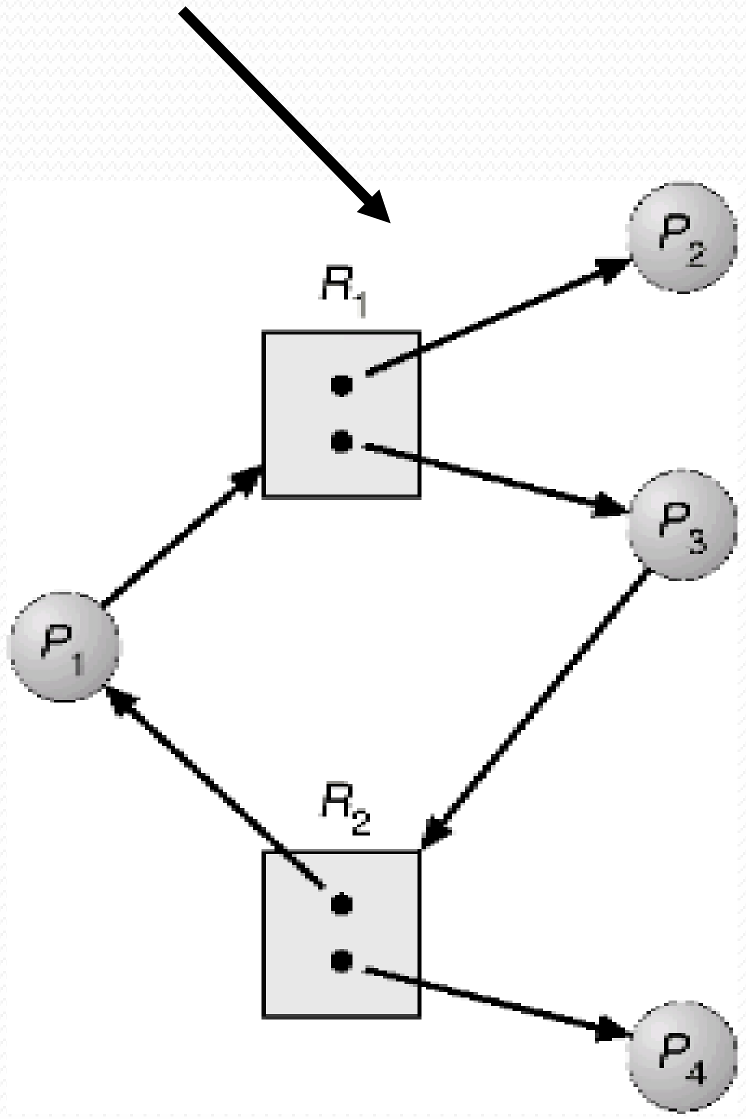


Examples

Resource allocation graph with a deadlock.

Resource allocation graph with a cycle but no deadlock.





HOW TO HANDLE DEADLOCKS ? (or) Methods for handling deadlocks.

1 There are three methods:

Most Operating systems do this!!

Ignore Deadlocks:

2

Ensure deadlock **never** occurs using either

- **Prevention** :
  - Prevent any one of the 4 conditions never happens.
- **Avoidance** :
  - Allow all deadlock conditions, but calculate cycles and stop dangerous operations..

3 **Allow** deadlock to happen. This requires using both:

● **Detection**	Know a deadlock has occurred.
● **Recovery**	Regain the resources.

Deadlock Prevention

Do not allow one of the four conditions to occur.

Mutual exclusion:

- Read only files are good examples for sharable resource
  - Any number of users can access the file at the same time.
- Prevention not possible, since some devices like are non-sharable.

Hold and wait:

- Collect all resources before execution
- A sequence of resources is always collected at the beginning itself.
- Utilization is low, starvation possible.

Deadlock Prevention – Contd...

No preemption:

- If the process is holding some resources and requests another resource (that cannot be immediately allocated to it), then all the resources that the process currently holding are preempted.

Circular wait:

- R = { R1,R2...Rm} □ set all resource types.
- We define a function,
- For example:
  F(tape drive) = 1
  F(disk drive) = 5
  F(printer) = 12

F: R □ N
N = natural number

- Each process requests resources in an increasing order of enumeration (ie) **$F(R_j) > F(R_i)$**

When we try to avoid deadlock

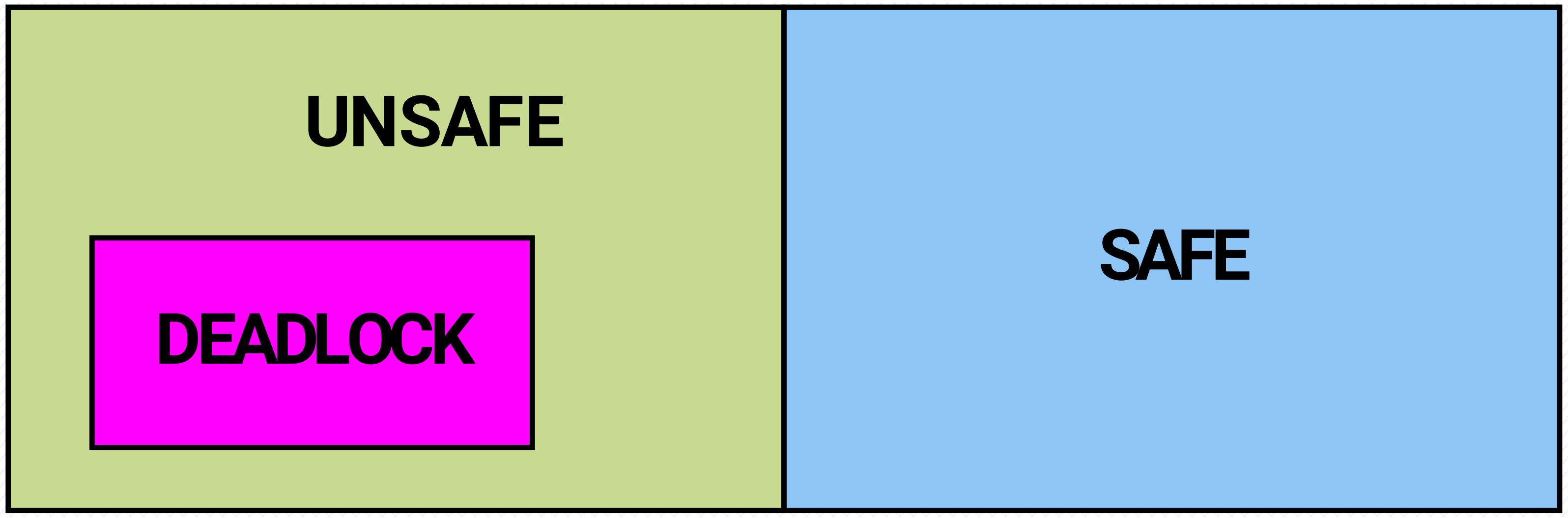
$\downarrow$

Utilization is less and system throughput is low

- An alternative method for avoiding deadlocks is to require additional information about how much resources are to be requested.

Safe State

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.



Only with luck, the processes avoid deadlock.

OS can avoid deadlock.

Safe State

- A system is said to be in safe state, when we allocate resources so that deadlock never occurs.
  - A system is in safe state, only if there exists safe sequence.

Deadlock Avoidance - Example

EXAMPLE:
There exists a total of 12 resources and 3 processes.

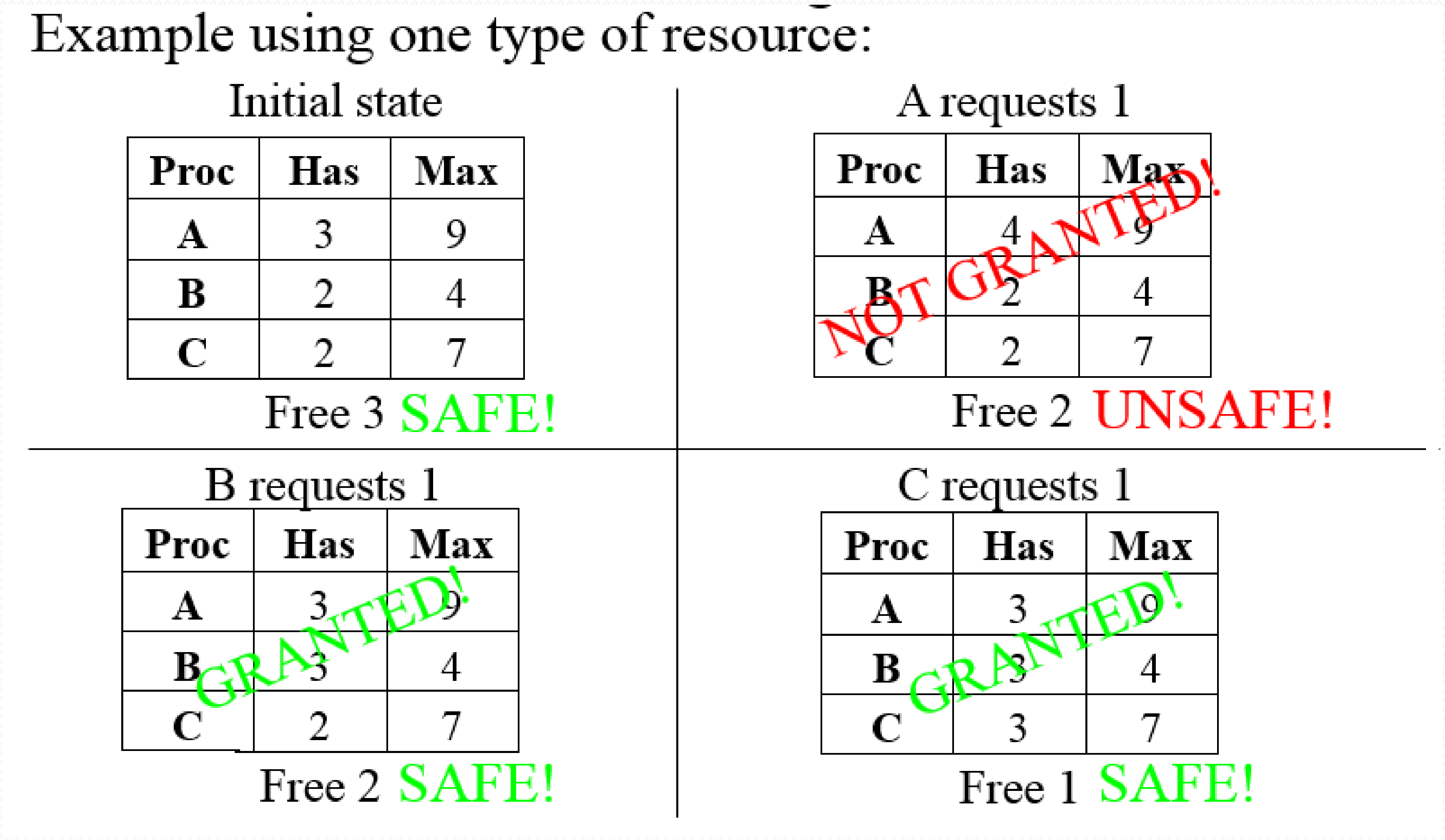
At time t0 , system is in safe sta

At time t1, < p1, p2, p0 > is a sequence.

Suppose p2 requests and is given one more resource. What happens then?

Process	Max Needs	Allocated	Current Needs
P0	10	5	5
P1	4	2	2
P2	7	3	4

Examples



Example using one type of resource:

Initial state

Proc	Has	Max
A	3	9
B	2	4
C	2	7

Free 3 SAFE!

A requests 1

NOT GRANTED!

Proc	Has	Max
A	4	9
B	2	4
C	2	7

Free 2 UNSAFE!

B requests 1

GRANTED!

Proc	Has	Max
A	3	9
B	3	4
C	2	7

Free 2 SAFE!

C requests 1

GRANTED!

Proc	Has	Max
A	3	9
B	3	4
C	3	7

Free 1 SAFE!

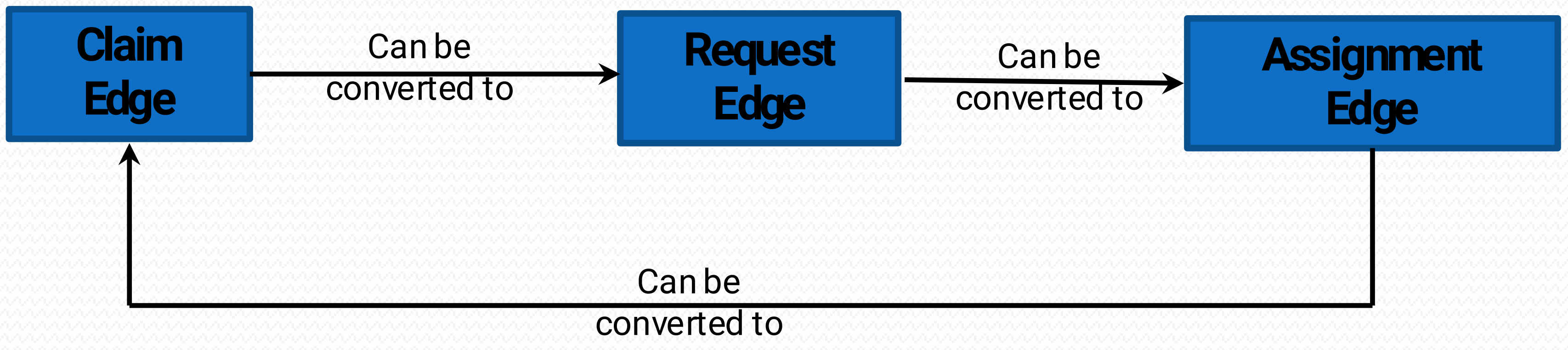
Avoidance algorithms

- For a single instance of a resource type, use a Resource-allocation Graph
- For multiple instances of a resource type, use the Banker's Algorithm

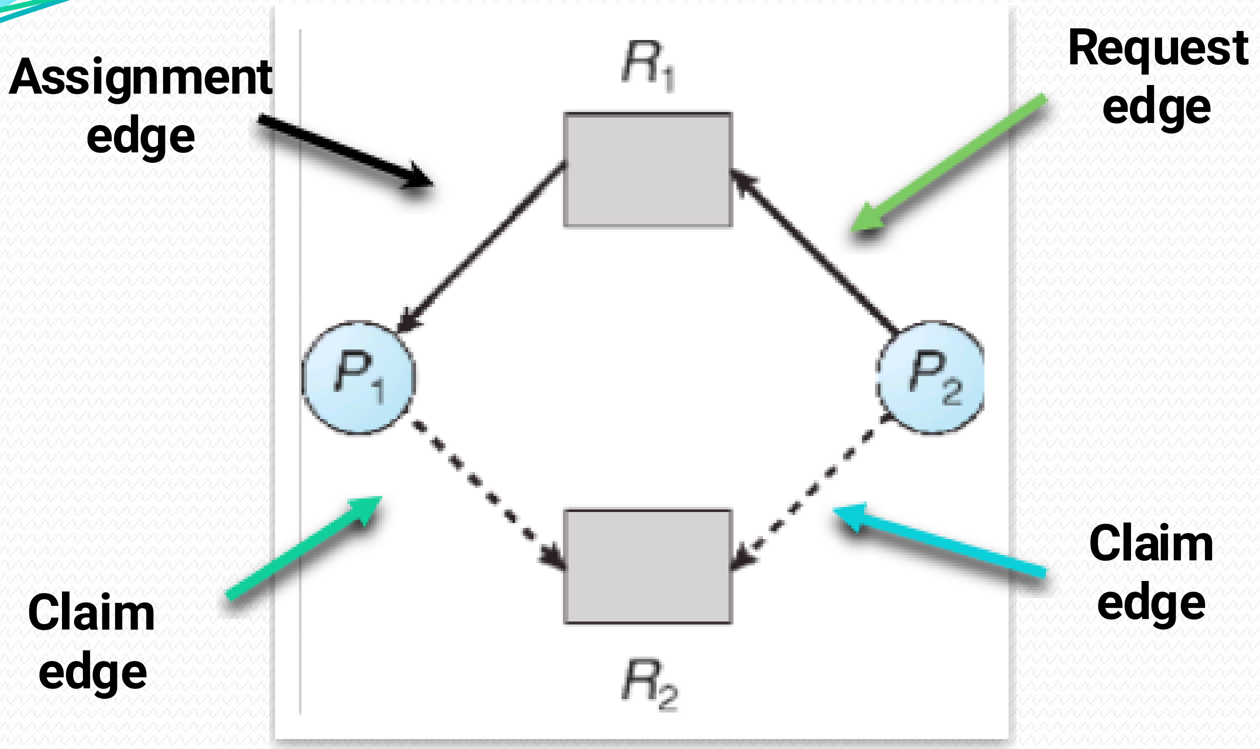
- Introduce a new kind of edge called a Claim Edge

Claim edge $P_i \dashrightarrow R_j$ indicates that process P_j may request resource R_j; which is represented by a dashed line.

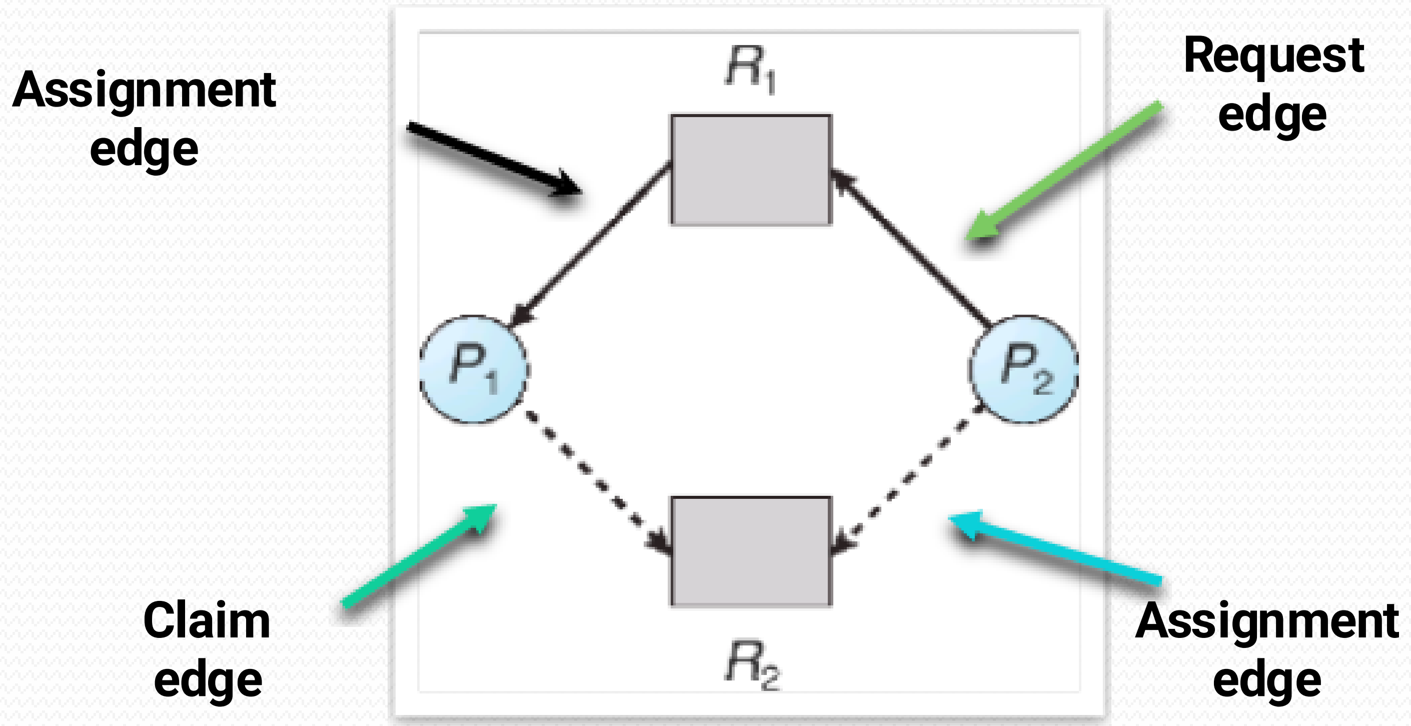
- A claim edge converts to a request edge when a process **requests** a resource
- A request edge converts to an assignment edge when the resource is **allocated** to the process
- When a resource is **released** by a process, an assignment edge reconverts to a claim edge.



Resource-Allocation Graph with Claim Edges



Unsafe State In Resource-Allocation Graph



Banker’s Algorithm

- Applicable for **multiple** instances of a resource type.
  - Its less efficient than Resource-Allocation Graph

- When a process requests a resource, the system determines whether the allocation of resources will lead to safe state.
  - If it lead to safe state allocate resources
  - If not safe state don’t allocate resources

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**: $n \times m$ matrix. If ***Max* [*i,j*] = *k***, then process P_i may request at most k instances of resource type R_j.
- **Allocation**: $n \times m$ matrix. If **Allocation[*i,j*] = *k*** then P_i is currently allocated k instances of R_j.
- **Need**: $n \times m$ matrix. If ***Need*[*i,j*] = *k***, then P_i may need k more instances of R_j to complete its task.

$$\boxed{\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]}$$

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 error.

2. If $Request_i \leq Available$, go to step 3.
   Otherwise P_i must wait, since resources are not available

3. Assume that resources are allocated:

$$Available = Available - Request;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5 instances), *C* (7 instances)

Snapshot at time T_0:

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

Example (Cont.)

- The content of the matrix *Need* is defined to be Need = *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 $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (ie, (1,0,2) ≤ (3,3,2) ⇒ true

	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.

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

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

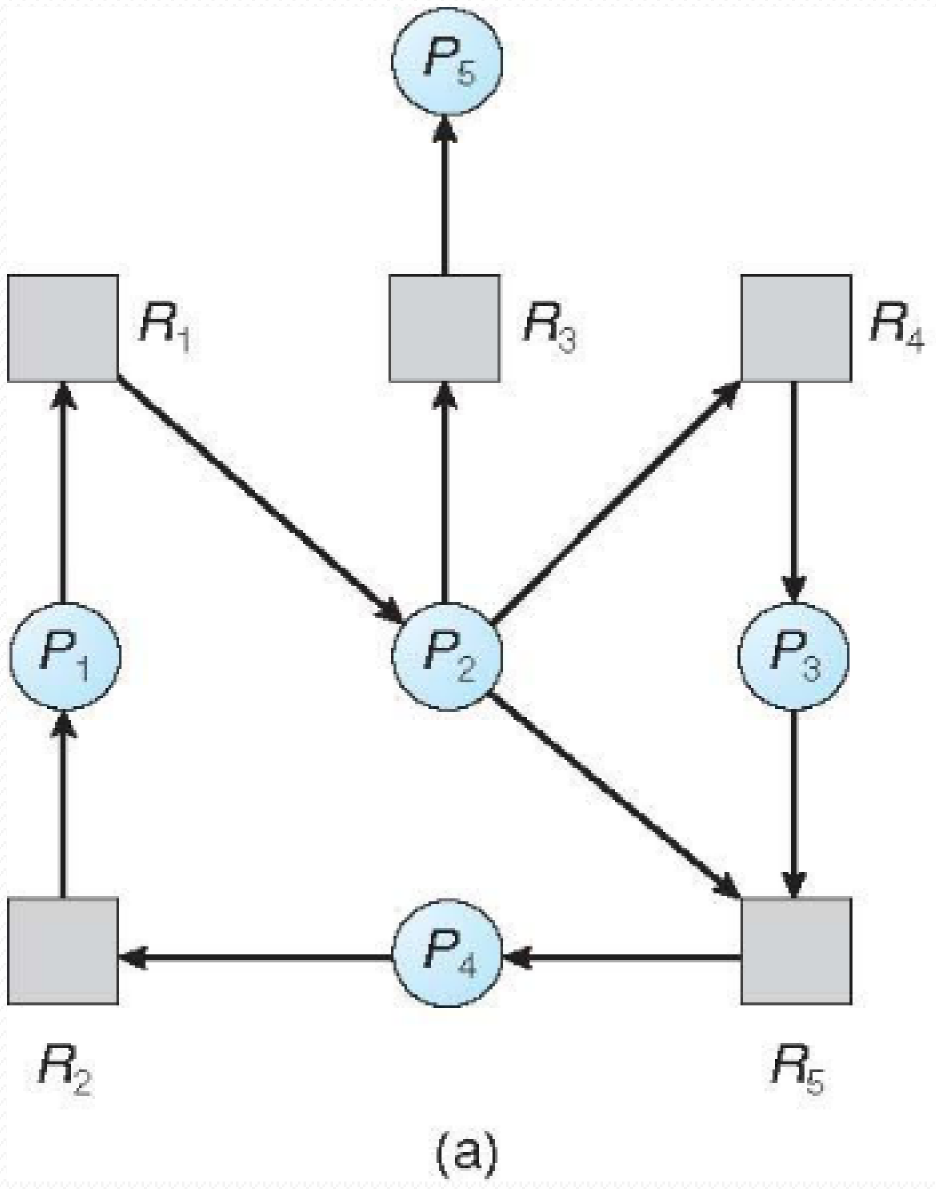
Single Instance of Each Resource Type

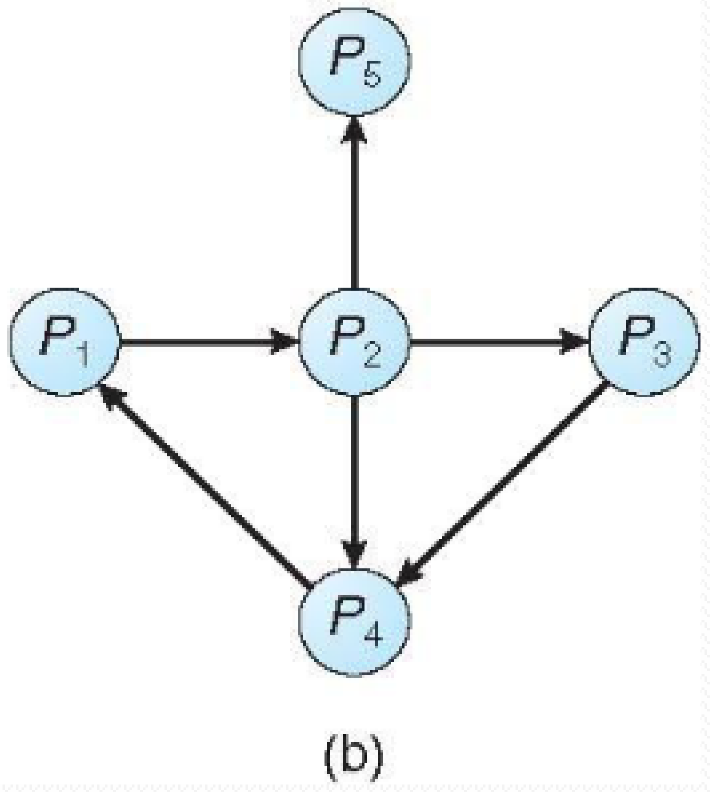
- Maintain wait-for graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

- 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 currently allocated.

- **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

(a) *Work = Available*

(b) For *i* = 1,2, …, *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

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, P_i is also deadlocked

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*

No Deadlock

- ***P_2* requests an additional instance of type *C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
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

Recovery from Deadlock

1. 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 – Contd...

2. Resource Preemption

- Selecting a victim – which resource or which process to be preempted? minimize cost
- Rollback – return to some safe state, restart process for that state ie. Rollback the process as far as necessary to break the deadlock.
- Problem: starvation – same process may always be picked as victim, include number of rollback in cost factor
  - Ensure that process can be picked as a victim only finite number of times.

References

Refer silberschatz, galvin " operating system concepts" 9th edition

- CPU scheduling and policies – pg no:201-216
- Realtime and deadline –pg no: 223 to 230
- Process synchronization- pg no:253-275
- Deadlocks –pg no:311-334

(Can also refer to learning resources mentioned in the syllabus)