CPU Scheduling

Process Management:

A program in execution is called the Process.

A process include:

- Program Counter
- Stack -temporary data
- Data Section
- Heap – memory dynamically allotted

Process- Active

Program -Passive Entity

States of Process:

- New
- Ready
- Running
- Wait
- Terminated

Process Control Block:

- Process State
- Process Number
- Process Counter
- Registers
- CPU scheduling information
- Memory-management information – base and limit register
- Accounting information – amount of CPU time, time limit
- I/O status information – list of IO devices

Process Scheduling

- Process Scheduler
- Job Queue: Set of all processes in the system
- Ready Queue: set of all processes in the residing in main memory, ready and waiting to execute
- Device Queue: set of processes waiting for IO devices

Process Schedulers:

Long-term scheduler (Job scheduler): selects which process should be brought into ready queue

Short term schedulers(CPU scheduler): selects which process has to be executed next and allocats into CPU

Context switch:

Saves the current process and state restore of different process

Process Creation

Parent process created child. Child has the same program and data duplicate of parent and a new program loaded to it

Process identified by pid -process identifier

Parent and child can execute concurrently but parent wait until children terminate

- Fork()---creates new process
- Exec()—used after fork to replace process' memory space with new program

Process termination

It termiates after executing last statement and asks os to delete it using exit()

Parent terminates using abort()

If process terminates, all children must get terminated

Parents process may call wait()---for termination of child process

- Zombie: Process that has terminate but parent has not called wait()
- Orphan: Parent terminated without invoking wait()

IPC-Inter process communication

Shared Memory—region of memory is sharred

Message passing—msgs exchanged btwn coopering process useful for exchanging small amount of dat and time consuming

Synchronization

Blocking is synchronous

Non-blocking is asynchronous

Rendezvous: both sender and receiver are blocking

<u>Thread</u>

A light weight process

Basic unit of CPU utilization

Every process is a program that executes in a single thread of execution

Perform One tasks at a time

One user cannot perform more than one task simultaneously

If a process has multiple threads of control, it can perform more than one task at a time

<u>Five areas present challenges in programming for multicore systems</u>

1) Identifying tasks - find areas that can be divided into separate, concurrent tasks

2) Balance

3) Data splitting

4) Data dependency

5) Testing and debugging

Parallelism

- Data Parallelism: distributing same set of data across multiple computing cores
- Task Parallelism: Distributing tasks across multiple computing cores

Multithreading Models

- User level threads ULT
- Kernel level Threads KLT

A relationship must exist between user and kernel threads

- Many to one: many user-level threads to one kernel
- One to one: user-level thread to kernel-level thread
- Many to many: many ULT to smaller or equal number of KLT

Two-Level model

Many ULT to small or equal number of KLT but allows a ULT to bound to a KLT (Here both many to many model and one to one model is observed)

Preemptive Scheduling

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

Scheduling under 1 and 4 is nonpreemptive. All other scheduling is preemptive

• Consider access to shared data

• Consider preemption while in kernel mode

• Consider interrupts occurring during crucial OS activities

Dispatcher:

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

• Dispatch latency – time it takes for the dispatcher to stop one process

and start another running

Scheduling Criteria:

- CPU utilization – keep the CPU as busy as possible
- Throughput – Number of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue

- o Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time sharing environment)

• Asymmetric multiprocessing – only one processor accesses the

system data structures, easing the need for data sharing

• Symmetric multiprocessing (SMP) – each processor is self-

scheduling, all processes in common ready queue, or each has its own private queue of ready processes

Processor affinity – process has affinity for processor on which it is

currently running

• Soft affinity – a policy of attempting to keep a process running on the same

processor—but not guaranteeing that it will do so

• Hard affinity - allowing a process to specify a subset of processors on which

it may run

Load balancing attempts to keep the workload evenly distributed

across all processors in an SMP system

Push migration

• A specific task periodically checks the load on each processor

• If imbalance then by moving (or pushing) processes from overloaded to idle

or less-busy processors

Pull migration

• When an idle processor pulls a waiting task from a busy processor

Multicore processor - place multiple processor cores on the same

physical chip

Memory stall – when a processor accesses memory, it spends a significant amount of time waiting for the data to become available

Coarse-grained multithreading

• A thread executes on a processor until a long-latency event such as a memory stall occurs

• The cost of switching between threads is high

Fine-grained (or interleaved) multithreading

• Switches between threads at a much finer level of granularity

• The cost of switching between threads is small

Completion Time: Time at which process completes its execution.
Turn Around Time: Time Difference between completion time and arrival time.

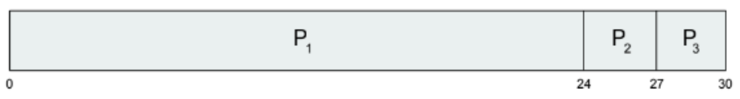
- Turn Around Time = Completion Time – Arrival Time

Waiting Time(W.T): Time Difference between turn around time and burst time.

- Waiting Time = Turn Around Time – Burst Time

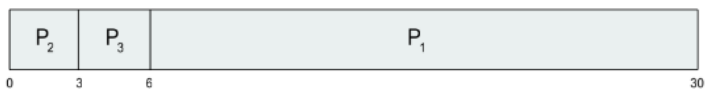
First Come First Serve (FCFS)

• Gantt chart



Process Name	Arrival Time	Burst Time(T_s)	Starting Time	Finishing Time	Response Time	Waiting Time	TAT (T_r)	T_r / T_s
P1	0	24	0	24	0	0	24	24/24 = 1
P2	0	3	24	27	24	24	27	27/3 = 9
P3	0	3	27	30	27	27	30	30/3 = 10
Average			51/3=17	81/3=27	51/3=17	**51/3=17**	81/3=27	20/3 = 6.6

- P2, P3, P1 order

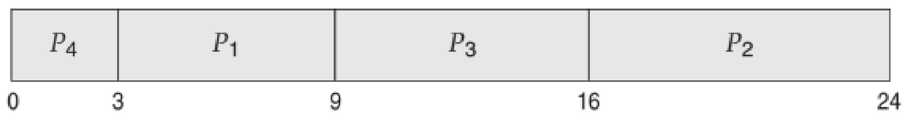


Process Name	Arrival Time	Burst Time(T_S)	Starting Time	Finishing Time	Response Time	Waiting Time	TAT (T_r)	T_r / T_s
P1	0	24	6	30	6	6	30	30/24=1.25
P2	0	3	0	3	0	0	3	3/3=1
P3	0	3	3	6	3	3	6	6/3=2
Average			9/3=3	39/3=13	9/3=3	**9/3=3**	39/3 =13	4.25/3=1.417

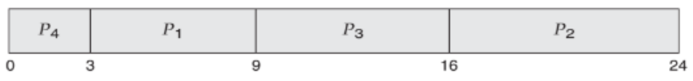
Shortest Job First

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling Gantt chart(Non-preemptive)



- Gantt chart

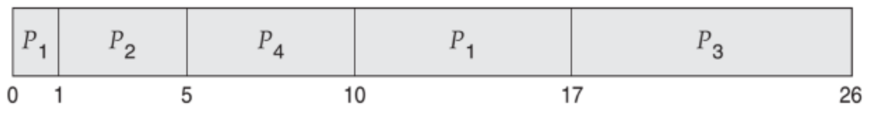


Process Name	Arrival Time	Burst Time(T_s)	Starting Time	Finishing Time	Response Time	Waiting Time	TAT (T_r)	T_r / T_s
P1	0	6	3	9	3	3	9	9/6=1.5
P2	0	8	16	24	16	16	24	24/8=3
P3	0	7	9	16	9	9	16	16/7=2.286
P4	0	3	0	3	0	0	3	3/3=1
Average			28/4=7	52/4=13	28/4=7	28/4=7	52/4=13	7.786/4=1.946

- The SJF algorithm can be either preemptive or nonpreemptive

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF



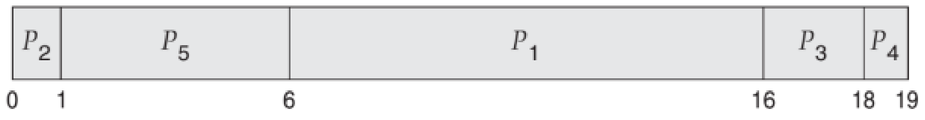
Non-Preemptive SJF

- Gantt chart

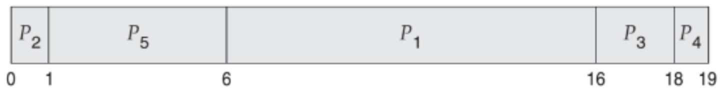
Process Name	Arrival Time	Burst Time(T_s)	Starting Time	Finishing Time	Response Time	Waiting Time	TAT (T_r)	T_r / T_s
P1	0	8	0	8	0	0	8	8/8=1
P2	1	4	8	12	7	7	11	11/4=2.75
P3	2	9	17	26	15	15	24	24/9=2.667
P4	3	5	12	17	9	9	14	14/5=2.8
Average			**37/4=9.25**	**63/4=15.75**	**31/4=7.75**	**31/4=7.75**	**57/4=14.25**	**9.217/4=2.304**

Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



- Gantt chart

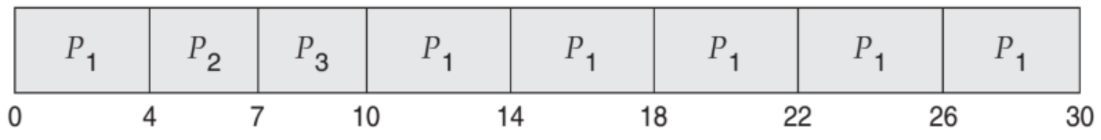


Process Name	Burst Time(T_s)	Priority	Starting Time	Finishing Time	Response Time	Waiting Time	TAT (T_r)	T_r / T_s
P1	10	3	6	16	6	6	16	16/10=1.6
P2	1	1	0	1	0	0	1	1/1=1
P3	2	4	16	18	16	16	18	18/2=9
P4	1	5	18	19	18	18	19	19/1=19
P5	5	2	1	6	1	1	6	6/5=1.2
Average			**41/5=8.2**	**60/5=12**	**41/5=8.2**	**41/5=8.2**	**60/5=12**	**31.8/5=6.36**

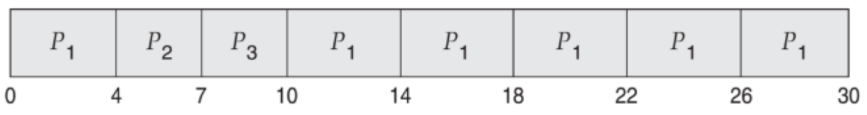
Round Robin

- Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3



- Gantt chart



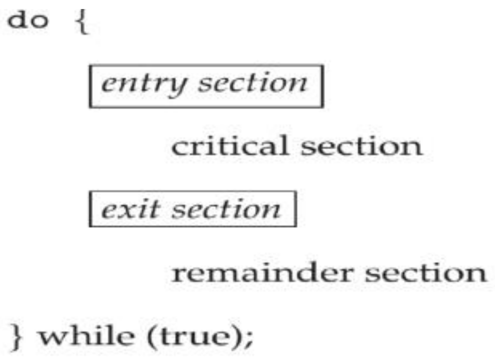
Process Name	Burst Time(T_s)	Starting Time	Finishing Time	Response Time	Waiting Time	TAT (T_r)	T_r / T_s
P1	24	0	30	0	6	6 (30)	6/24=0.25
P2	3	4	7	4	4	4 (7)	4/3=1.33
P3	3	7	10	7	7	7 (10)	7/3=2.33
Average		**11/3=3.67**	**47/3=15.67**	**11/3=3.67**	**17/3=6.33**	**17/3=6.33 (47/3=15.6)**	**3.91/3=1.30**

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Process Name	Arrival Time	Burst Time(T_s)	Starting Time	Finishing Time	Response Time	Waiting Time	TAT (T_r)	T_r / T_s
A	0	3	0	3	0	0	3	1
B	2	6	3	17	1	1+8=9	15	2.5
C	4	4	7	11	3	3	7	1.75
D	6	5	11	20	5	5+4=9	14	2.8
E	8	2	17	19	9	9	11	5.5
Average					**18/5=3.6**	**30/5=6**	**50/5= 10**	**13.55/5 =2.71**

Critical Session Problem

Process Synchronization



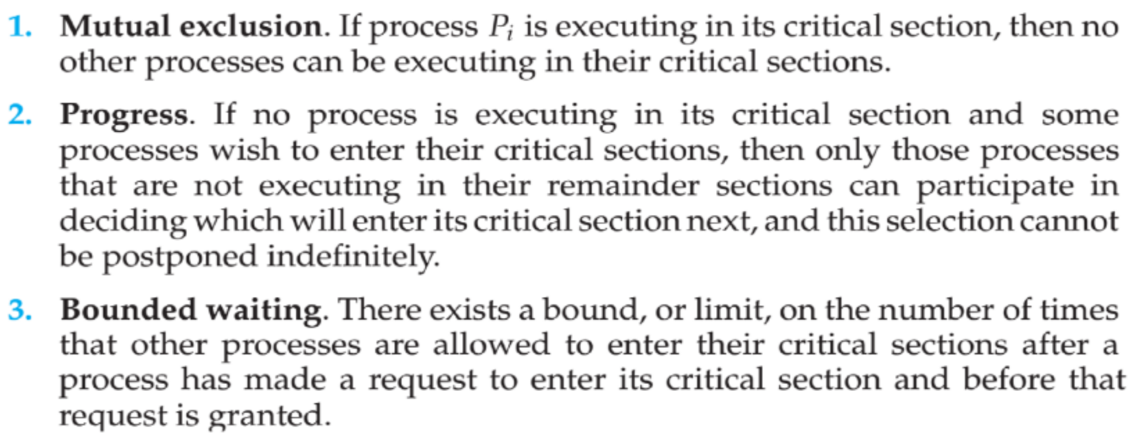
```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

General structure of a typical process P_i.

Critical section – when one process is executing in its critical section,

no other process is allowed to execute in its critical section

Solution :



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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting.** There exists a bound, or limit, 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.

Provable that the three CS requirement are met:

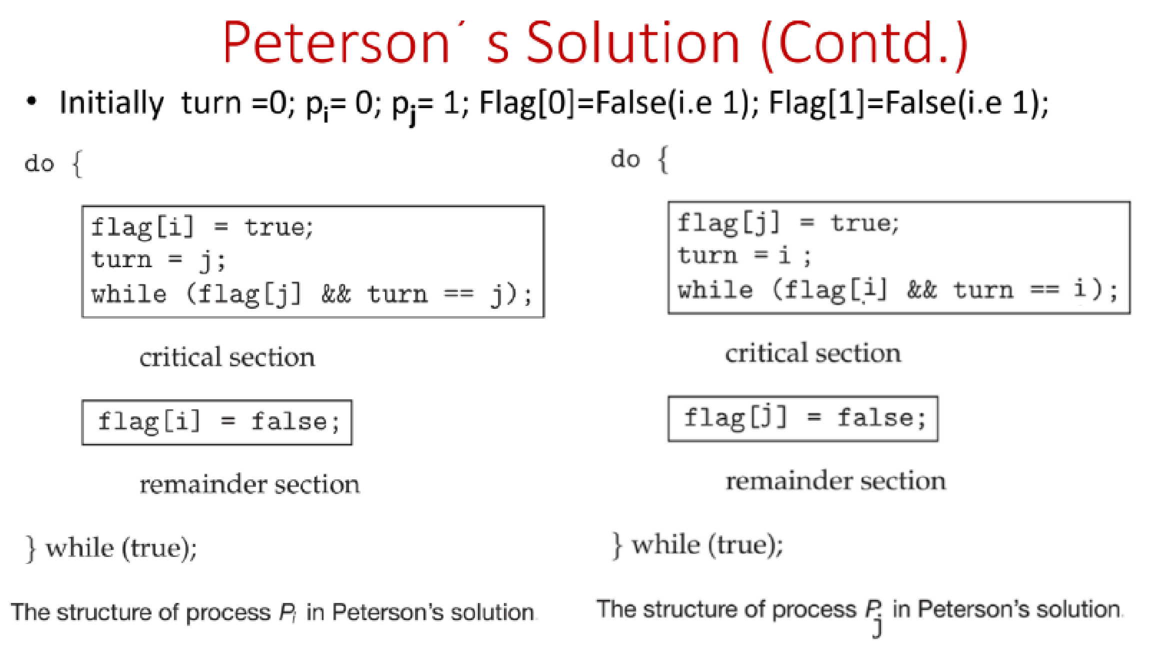
1. Mutual exclusion is preserved

Pi enters CS only if:

either flag[j] = false or turn = i

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met



Peterson´ s Solution (Contd.)

- Initially turn =0; p_i= 0; p_j= 1; Flag[0]=False(i.e 1); Flag[1]=False(i.e 1);

```
do {

     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j);

          critical section

     flag[i] = false;

          remainder section

} while (true);
```

The structure of process P_i in Peterson's solution

```
do {

     flag[j] = true;
     turn = i ;
     while (flag[i] && turn == i);

          critical section

     flag[j] = false;

          remainder section

} while (true);
```

The structure of process P_j in Peterson's solution

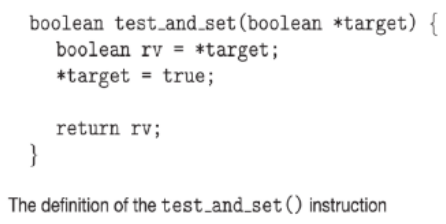
Synchronization Hardware

Locking: protecting critical regions through the use of locks

Two Approaches

Test_and_set()

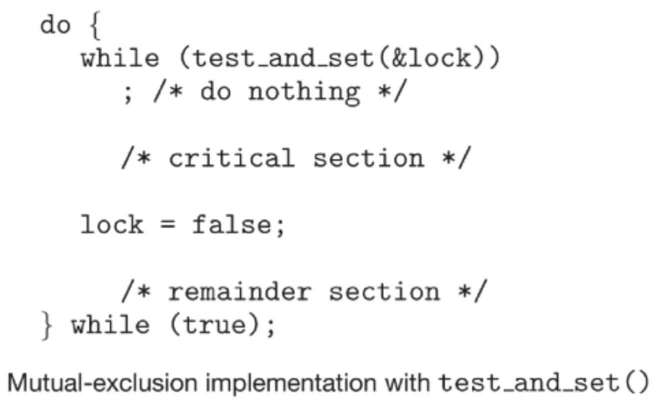
Compare_and_swap()



```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

The definition of the test_and_set() instruction



```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Mutual-exclusion implementation with test_and_set()

Mutex Lock

The definition of acquire() is as follows:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}
```

The definition of release() is as follows:

```
release() {
   available = true;
}
```

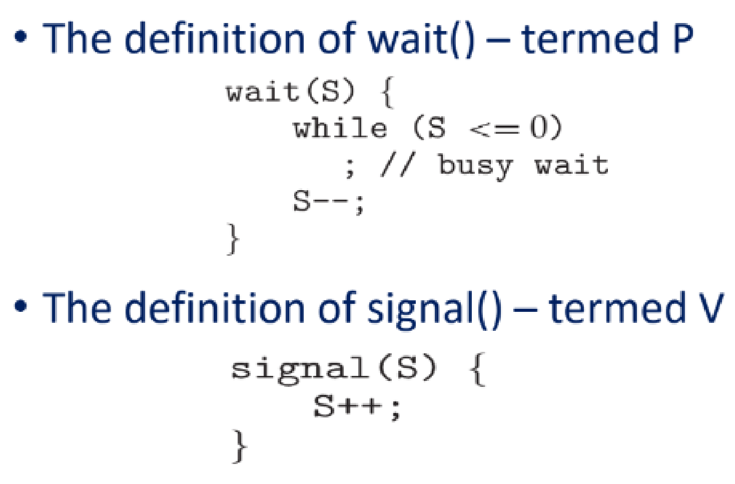
```
do {
    acquire lock
         critical section
    release lock
         remainder section
} while (true);
```

Solution to the critical-section problem using mutex locks.

- Busy waiting – also called as spinlock

<u>Semaphores</u>

A semaphore S is an integer variable that accessed only through two standard atomic operations: wait() and signal();



- The definition of wait() – termed P

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

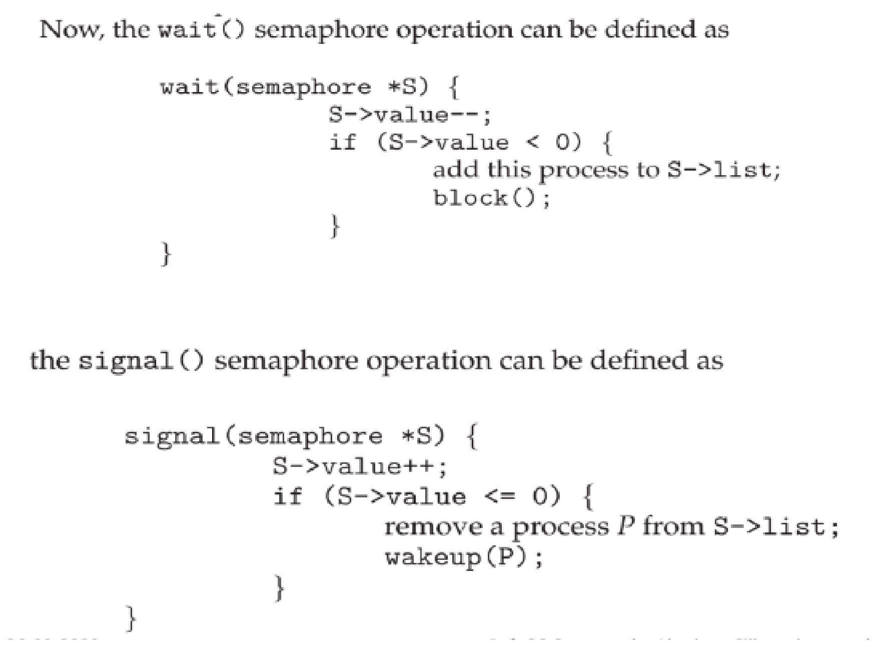
- The definition of signal() – termed V

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

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 it in the ready queue



Now, the wait() semaphore operation can be defined as

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

the signal() semaphore operation can be defined as

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

<u>Deadlocks and Starvation</u>

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes

Starvation – indefinite blocking

• A situation in which processes wait indefinitely within the semaphore

Priority Inversion

The higher-priority process will have to wait for a lower-priority one
to finish

• Solution: priority-inheritance protocol

• all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished

• revert to their original values