KANAMPALLI HIMAJA - 700732994

ASSIGNMENT - 1

1) Consider the following set of processes, with
5.4a) the length of the CPU burst time given in milliseconds.

Process	Burst time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

Sol: The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

Sol: FCFS [First Come First Served Scheduling]

Gantt chart: (order: P_1, P_2, P_3, P_4, P_5)

P_1	P_2	P_3	P_4	P_5
0 – 2	2 – 3	3 – 11	11 – 15	15 – 20

SJF [Shortest Job First]

order according to burst time: P_2, P_1, P_4, P_5, P_3

P_2	P_1	P_4	P_5	P_3
0 – 1	1 – 3	3 – 7	7 – 12	12 – 20

Non-preemptive priority (larger priority = higher priority) Then : $P_3, P_5, (P_1$ or $P_4), P_2$ is order. P_1 or P_4, P_1 has less burst time.
Therefore order : P_3, P_5, P_1, P_4, P_2

P_3	P_5	P_1	P_4	P_2
0	8	13	15	19 – 20

Round Robin (RR) [quantum = 2]

P_1	P_2	P_3	P_4	P_5	P_3	P_4	P_5	P_3	P_5	P_3
0	2	3	5	7	9	11	13	15	17	18 – 20

Time is divided into quantum and assigned to process.

2) The following processes are being scheduled using a
5.5 preemptive, roundrobin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the System also has an idle task (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 Units. if a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a) Show the scheduling order of the processes using a Gantt chart.

sol: ⇒ higher number = higher relative priority

⇒ P_{idle} - With no CPU resources

⇒ Priority 0 is scheduled whenever no other available processes to run.

⇒ Time quantum = 10 units

According to this, the scheduling order is:

(P_1, idle, P_2, P_3, P_2, P_3, P_4, P_2, P_3, idle, P_5, P_6, P_5)

P_1	idle	P_2	P_3	P_2	P_3	P_4	P_2	P_3	idle	P_5	P_6	P_5
0–20	20–25	25–35	35–45	45–55	55–60	60–75	75–80	80–90	90–100	100–105	105–115	115–120

3) What is the meaning of the term busy waiting?
6.2 What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

[Provide more than that from the publisher's answer, use structure of semaphore and its list from PPT and book to explain].

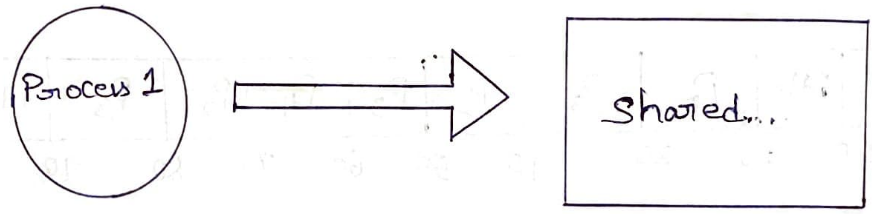
Sol: Busy waiting:

⇒ Busy waiting, also known as spinning, (or) busy looping is a process synchronization technique in which a process / task waits and constantly checks for a condition to be satisfied before proceeding with its execution.

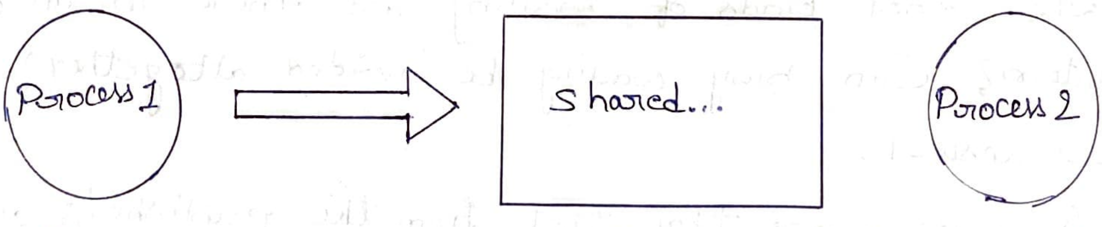
⇒ In busy waiting, a process executes instructions that test for the entry condition to be true, Such as the availability of a lock or resource in the computer system.

⇒ For resource availability, consider a scenario where a process needs a resource for a specific program. However, the resource is currently in use and unavailable at the moment, therefore the process has to wait for resource availability before it can continue. This is what is known as busy waiting as illustrated below:

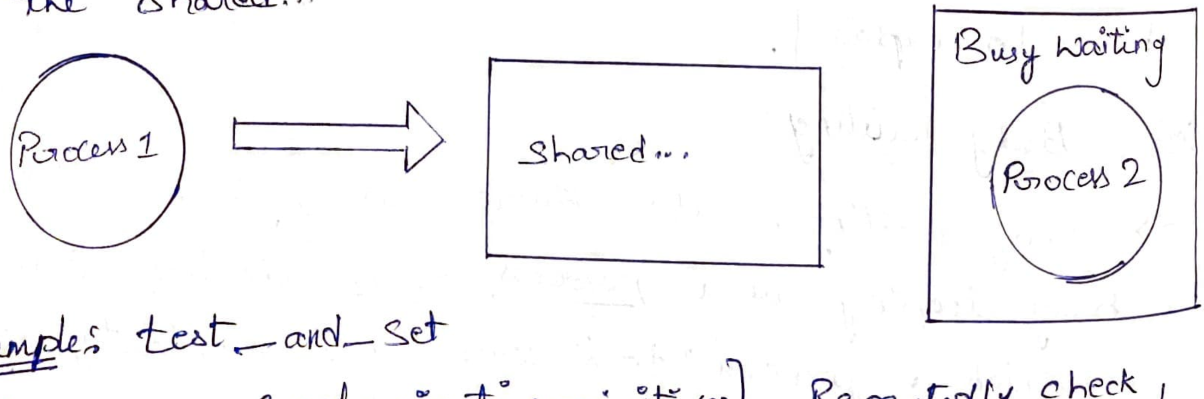
1. Process 1 is using the shared resource.



2. Process 2 is now in need of the shared resources.



3. Process 2 enters busy waiting till it has access to the shared...



Example: test_and_set
[hardware synchronization primitives]. Repeatedly check, work is to check, take resource. Like wait outside of restroom.

other Kinds of waiting in a operating System are:

* Waiting an application
* Waiting for a file
* Waiting for user input
* Waiting for hardware to communicate.

Alternate to busy waiting

* " Blocked waiting " also called sleep waiting. Where a task sleeps an event occur.
* For blocked to work there must be some external agent that can wakes up the task.
* When the event has occured processes also wait when they are ready to run, but the processor is busy executing other task.

Avoid ?

* Busy waiting cannot be avoided altogether.
* Some events cannot trigger a wakeup, for example on Unix a processor cannot " Sleep until a file is modified ", because the operating system does not provide any mechanism to automatically wake up the process when the event occurs some amount of repeated polling is required.
* It can be avoided on uniprocessor machines,
* Unavoided for multiprocessor machines.

Semaphore implementation with no " Busy Waiting "

Each semaphore is associated with a waiting queue. Each entry in a waiting queue has two data items:

1. value [of type integer]
2. Pointer to next record in the list.

And two operations will be performed, they are block & wakeup.

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.

Waiting queue:

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

} semaphore;
```

wait() semaphore:

```
wait (semaphore *S) {
    S->value--;
    if (S->value < 0) {  // no resource available, other process might wait as well
        add this process
          to S->list;
        block();

    }
}
```

signal() semaphore:

```
signal (semaphore *S) {
    S->value ++;
    if (S->value <= 0) {  // there is process wait for the resource
        remove a process
         P from S->list;
        wakeup (P);
} }
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. The two operations are provided by the operating system as basic system calls.

4) (chapter 7) Explain in the Readers-Writers Problem, how writer process and reader process use the semaphores of both rw_mutex and mutex to achieve process synchronization.

Sol: Reader - Writer problem

=> Reader process who only read the shared data.

=> Writer process who may change the data in addition to reading.

=> There is no limit to how many readers can access the data simultaneously but when a writer access the data, it needs exclusive access.

=> There are several variations to reader-writer problem, most centered around relative priorities of readers versus writers.

1) Priority to readers:

=> If no writer then access is granted to the reader.

=> A solution to this problem can lead starvation to writers.

2) Priority to writer:

=> Writer wants access to the data it jumps to the head of queue.

=> all waiting readers are blocked.

⇒ This solution may lead to readers Starvation.

Solution to acheive process synchronization:

⇒ As mentioned in readers & writer process

⇒ read_count: Used by reader process to count the number of readers currently accessing the data.

⇒ Mutex: Semaphore used only by readers to controlled access to read_count

⇒ no_mutex: Used to block and release the writers.

* The first reader to access the data will set the lock and last release to exit will release it.

* Now, the first reader to come along will block on rw_mutex if there is currently a writer accessing the data and that all following readers will only block on mutex for its turn to increment read count

[Three variables are used: mutex, wrt, readcnt to implement solution]

Functions for Semaphore:

- wait(): decrements the semaphore value.
- signal(): increments the semaphore value.

Writer process:

1. Writer requests the entry to critical section.
2. If allowed i.e wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exists the critical section.

```
do{
    // writer requests for critical section
    wait(wrt);

    // performs the write

    // leaves the critical section
```

```
    signal (wrt);

} while (true);
```

Reader process:

1) Reader requests the entry to critical section.
2) If allowed:
* It increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.
* It then, signals mutex as any other reader is allowed to enter while others are already reading.
* After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside; it signals the semaphore "wrt" as now, writer can enter the critical section.
3) If not allowed, it keeps on waiting.

```
do {
  // Reader wants to enter the critical section
  wait (mutex);

  // The number of readers has now increased by 1
  readcnt ++;

  // there is atleast one reader in the critical section
  // this ensure no writer can enter if there is
     even one reader
  // thus we give preference to readers here
  if (readcnt = =1)
     wait(wrt);
  // other readers can enter while this current
  reader is inside
```

```
// the critical section
signal(mutex);
// current reader performs reading here
wait(mutex);   // a reader wants to leave

readcnt--;
// that is, no reader is left in the
critical section,
if (readcnt == 0)
    signal(wrt);        // writers can enter

signal(mutex);  // reader leaves

} while(true);
```

* Some hardware implementations provide specific reader-writer locks, which are accessed using an argument specifying whether access is requested for reading (or) writing.
* Use of reader-writer locks is beneficial for situations in which
(1) Process can be easily identified as either reader or writers.
(2) there are significantly more readers than writers, making the additional overhead of the reader writer lock pay off in terms of increased concurrency of the readers.

5) 8.3) Use Allocation Matrix, Max Matrix, Need Matrix and Available Matrix to explain

Consider the following snapshots of a system:

	Allocation	Max	Available
T_0	0012	0012	1520
T_1	1000	1750	
T_2	1354	2356	
T_3	0632	0652	
T_4	0014	0656	

Answer the following questions using the banker's algorithms ?

(a) What is the content of the matrix Need?

(b) Is the system in a safe state?

(c) If a request from thread T_1 arrives for (0,4,2,0) can be request be granted immediately?

Sol: a) The values of Need for processes P_0 through P_4, respectively are

(0,0,0,0),

(0,7,5,0),

(1,0,0,2),

(0,0,2,0),

(0,6,4,2),

b) The system is in a safe state with Available equal to (1,5,2,0) either process P_0 or P_3 could run. Once process P_3 runs. if releases its resources, which allows all other existing processes to run.

c) The request can be granted immediately. The value of Available is then (1,1,0,0). One ordering of processes that can finish is P_0, P_2, P_3, P_1 and P_4.