Homework 3 Solution

4.15

a) Processes = 6

b) Threads =2

Pid = fork(); The parent process p before the if statement creates one process p1. fork(); The parent process p in the if statement creates one p2.

After the if statement, parent process p, process p1 and process p2 will execute fork(); creating three new processes. 3 process p3, p4, p5 is created by p, p1 and p2.

Hence, 6 unique processes (p, p1, p2, p3, p4, p5) will be created.

Thread creation is done in if block. Only child process p1 is executed in the if block. Therefore, process p1 will be created one thread.

In the if block one process p2 is created using fork(). Therefore, process p2 will also create a thread. Hence, 2 unique threads will be created.

5.20

A. Only one race condition is observed in the program code that is on the variable number_of_processes.

B. The acquire() function should be placed in the beginning of function call. Whereas, acquire() must be put upon entering each function. release() must be put just before exiting each function.

C. No, it would not help. The reason is because the race occurs in the allocate process() function where number of processes is first tested in the if statement, yet is updated afterwards, based upon the value of the test. it is possible that number of processes = 254 at the time of the test, yet because of the race condition, is set to 255 by another thread before it is incremented yet again.

5.32

```
int sumid=0;
int waiting=0;
semaphore mutex=1;
semaphore OKToAccess=0;

get_access(int id)
{
    sem_wait(mutex);
    while(sumid+id > n)
{
        waiting++;
        sem_signal(mutex);
        sem_wait(OKToAccess);
        sem_wait(mutex);
    }
    sumid += id;
    sem_signal(mutex);
}

release_access(int id)
{
    int i;
    sem_wait(mutex);
    sumid -= id;
    for ( i=0; i < waiting;++I )
 {
        sem_signal ( OKToAccess );
    }
    waiting = 0;
    sem_signal(mutex);
}
main( )
{
    get_access(id);
    do_stuff();
    release_access(id);
}
```

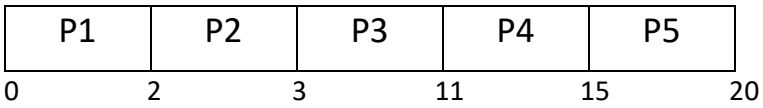
5.35

```
monitor AlarmClock
int now=0;
condition wakeup;

wakeme (int n)
{
        int alarm;
        alarm = mow +n;
        while(now<alarm) wakeup.wait (alarm);
        wakeup.signal;
}

tick()
{
        now =now +1;
        wakeup.signal;
}
```

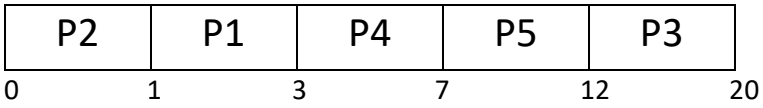
6.16

Process	Burst Time	Priority
P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

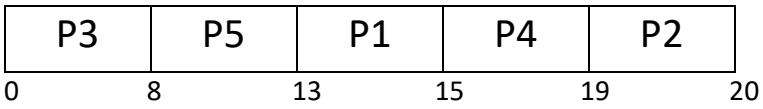
FCFS



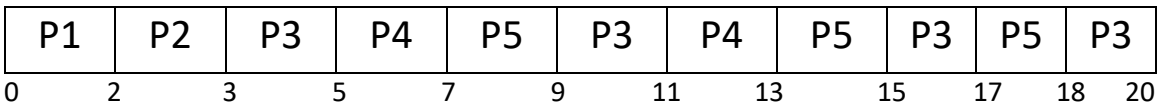
SJF



Non-preemptive priority (a larger number implies a higher priority)



RR (q=2)



b. Turnaround Time

	FCFS	SJF	Priority	RR
P1	2	3	15	2
P2	3	1	20	3
P3	11	20	8	20
P4	15	7	19	13
P5	20	12	13	18

c. Waiting time

	FCFS	SJF	Priority	RR
P1	0	1	13	0
P2	2	0	19	2
P3	3	12	0	12
P4	11	3	15	9
P5	15	7	8	13

d. Minimum average waiting time

SJF has Minimum average waiting time, and it equals (1+0+12+3+7)/5 = 4.6

7.8

A. Deadlock cannot occur because preemption exists.
B. Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as those of process.