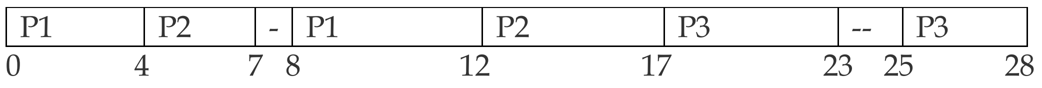
SJF

Process	Priority	Arrival Time	1st CPU Burst	I/O Burst	2nd CPU Burst
P1	2	0	4	4	4
P2	1	3	3	3	5
P3	0	8	6	2	3

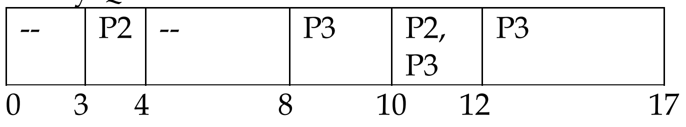
Using **Shortest Job First algorithm**, show the **Gantt chart** of processes executing in the CPU. Also, calculate the **average waiting time**.

Solution:

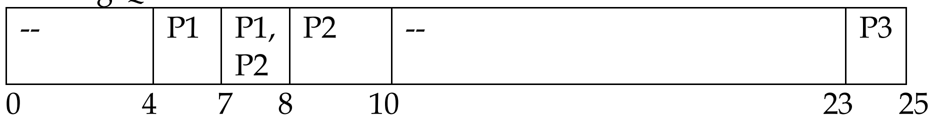
Gantt Chart:



Ready Queue:



Waiting Queue:



Waiting time: P1 = 0
P2 = (4-3) + (12-10) = 3
P3 = 17-8 = 9
Average = (0 + 3 + 9) / 3 = 4

PP

Process	Priority	Arrival Time	1st CPU Burst	I/O Burst	2nd CPU Burst
P1	2	0	4	4	4
P2	1	3	3	3	5
P3	0	8	6	2	3

Using **Preemptive Priority scheduling algorithm**, show the **Gantt chart** of processes executing in the CPU. Also, calculate the **average waiting time**.

Solution:

Gantt Chart:

P1	P2	P1	--	P3	P2	P3	P2	P1

0 3 6 7 8 14 16 19 22 26

Ready Queue:

--	P1	--	P2	P2, P1	P1	P1, P2	P1

0 3 6 9 11 14 16 19 22

Waiting Queue:

--	P2	P1, P2	P1	--	P3

0 6 7 9 11 14 16

Waiting time: P1 = (6-3) + (22-11) = 14
P2 = (14-9) + (19-16) = 8
P3 = 0
Average = (14 + 8 + 0) / 3 = 7.33

A unisex bathroom is shared by men and women. A man or a woman may be using the room, waiting to use the room, or doing something else. They work, use the bathroom and come back to work. The rule of using the bathroom is very simple: *there must never be a man and a woman in the room at the same time; however, people with the same gender can use the room at the same time.*

```
Man Thread                                Woman Thread
void Man(void)                            void Woman(void)
{                                         {
   while (1) {                               while (1) {
      // working                                // working
      // use the bathroom                       // use the bathroom
}                                         }
```

Declare semaphores and other variables with initial values, and add `Wait()` and `Signal()` calls to the threads so that the man threads and woman threads will run properly and meet the requirement. Your implementation should not have any busy waiting, race condition, and deadlock, and should aim for maximum parallelism.

A convincing correctness argument is needed. Otherwise, you will receive no credit for this problem.

Answer: This is a simple variation of the reader-priority readers-writers problem. More precisely, we allow the "writers" to write simultaneously. Therefore, the writers have the same structures as the readers. We need to maintain two counters, one for the males `MaleCounter` and the other for the females `FemaleCounter`. Of course, we need two Mutexes `MaleMutex` and `FemaleMutex` for mutual exclusion. In addition, there is a semaphore `BathRoom` to block the males (*resp.*, females) if the room is being used by the females (*reap.*, males). Note that the male thread and female thread are symmetric.

```
int       MaleCounter = 0, FemaleCounter = 0;   // male and female counters
Semaphore MaleMutex = 1, FemaleMutex = 1;       // male and female counters
Semaphore BathRoom = 1;                         // the bathroom is empty initially

Male Thread                     Female Thread

while (1) {                     while(1) {
   // working                      // working

   MaleMutex.Wait();               FemaleMutex().Wait();       // update counter
      MaleCounter++;                  FemaleCounter--;
      if (MaleCounter == 1)           if (FemaleCounter == 1)  // if I am the first
         BathRoom.Wait();                BathRoom.Wait();      //    yield to other
   MaleMutex.Signal();             FemaleMutex.Signal();

   // use the bathroom             // use the bathroom

   MaleMutex.Wait();               FemaleMutex.Wait();         // update counter
      MaleCounter--;                  FemaleCounter--;
      if (MaleCounter == 0)           if (FemaleCounter == 0)  // if I am the last one
         BathRoom.Signal();              BathRoom.Signal();    //    let the other group know
   MaleMutex.Signal();             FemaleMutex.Signal();
}                               }
```

Assume that there are three resources, A, B, and C. There are 4 processes P_0 to P_3. At T_0 we have the following snapshot of the system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	1	0	1	2	1	1	2	1	1
P_1	2	1	2	5	4	4	▨	▨	▨
P_2	3	0	0	3	1	1	▨	▨	▨
P_3	1	0	1	1	1	1	▨	▨	▨

1.Create the need matrix.

	Need		
	A	B	C
P_0	1	1	0
P_1	3	3	2
P_2	0	1	1
P_3	0	1	0

2. Is the system in a safe state? Why or why not?
In order to check this, we should run the safety algorithm. Let's create the work vector and finish matrix:

Work vector	Finish matrix	
2	P_0	False
1	P_1	False
1	P_2	False
	P_3	False

$Need_0$ (1,1,0) is less than work, so let's go ahead and update work and finish:

Work vector	Finish matrix	
3	P_0	True
1	P_1	False
2	P_2	False
	P_3	False

$Need_1$ (3,3,2) is not less than work, so we have to move on to P_2.

$Need_2$ (0,1,1) is less than work, let's update work and finish:

Work vector	Finish matrix	
6	P_0	True
1	P_1	False
2	P_2	True
	P_3	False

$Need_3$ (0,1,0) is less than work, we can update work and finish:

Work vector	Finish matrix	
7	P_0	True
1	P_1	False
3	P_2	True
	P_3	True

We now need to go back to P_1. $Need_1$ (3,3,2) is not less than work, so we cannot continue. Thus, the system is not in a safe state.

To illustrate this algorithm, we consider a system with five threads T_0 through T_4 and three resource types *A*, *B*, and *C*. Resource type *A* has seven instances, resource type *B* has two instances, and resource type *C* has six instances. The following snapshot represents the current state of the system:

	Allocation	***Request***	***Available***
	A B C	*A B C*	*A B C*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $<T_0, T_2, T_3, T_1, T_4>$ results in ***Finish***[*i*] == *true* for all *i*.

Suppose now that thread T_2 makes one additional request for an instance of type *C*. The ***Request*** matrix is modified as follows:

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by thread T_0, the number of available resources is not sufficient to fulfill the requests of the other threads. Thus, a deadlock exists, consisting of threads T_1, T_2, T_3, and T_4.

Consider the following page reference string (15 memory references) in a demand paging virtual memory environment (repeated in tables):

1	2	3	4	2	1	5	6	2	1	2	3	7	5	3	2	1	2	3	6

16% Calculate how many page faults would occur, the success rate and failure rate for each of the following replacement algorithms, We have 3 frames F1-F3 and all frames are initially empty.

a. **Optimal (OPT) replacement** (5%):

Page#➔	1	2	3	4	2	1	5	6	2	1	2	3	7	5	3	2	1	2	3	6
Frame1	1	1	1	1			1	1				1	7	5			1			6
Frame2		2	2	2			2	2				2	2	2			2			2
Frame3			3	4			5	6				3	3	3			3			3
Fault?	+	+	+	+			+	+				+	+	+			+			+

11 Faults

b. **LRU replacement** (5%)

Page#➔	1	2	3	4	2	1	5	6	2	1	2	3	7	5	3	2	1	2	3	6
Frame1	1	1	1	4		4	4	6		6		3	3	3		3	3			3
Frame2		2	2	2		2	2	2		2		2	2	5		5	1			6
Frame3			3	3		3	5	5		1		1	7	7		2	2			2
Fault?	+	+	+	+		+	+	+		+		+	+	+		+	+			+

14 Faults

3. **FIFO with 3 frames:** (5%)

Page#➔	1	2	3	4	2	1	5	6	2	1	2	3	7	5	3	2	1	2	3	6
Frame1	1	1	3	3	2	2	5	5	2	2		3	3	5	5	2	2		3	3
Frame2		2	2	4	4	1	1	6	6	1		1	7	7	3	3	1		1	6
Fault?	+	+	+	+	+	+	+	+	+	+		+	+	+	+	+	+		+	+

18 Faults