Answer 1

a) Yes, we can easily conclude that we are able to schedule this task-set using RM on a single processor machine. We can do this by using the following equation: $\sum_{i=1}^{m} \frac{C_i}{T_i} \leq m \times (2^{\frac{1}{m}} - 1)$, where C_i is the Worst Case Scenario for task i, and T_i is the period at which a new job is released for task i, and finally m is the number of tasks in the task-set *(TABLE 1.1, PAGE 10)*.

If we apply this equation we get:

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq m \times (2^{\frac{1}{m}} - 1) \equiv \frac{4.5}{19} + \frac{4.2}{22} + \frac{3}{9} \leq 3 \times (2^{\frac{1}{3}} - 1) \equiv 0.77462 \leq 0.77976$$

As the equation holds up, we know the task-set is scheduleable in RM.

b) With this new task-set *(TABLE 1.2, PAGE 10)*, if we run the equation as we did above, we get: $0.94138 \leq 0.75682$, which isn't true, so the equation doesn't hold up. However, for RM this equation isn't a necessary condition, rather a sufficient one. As such, the only way to actually know if this is scheduleable is to attempt to schedule it.

Upon attempting to schedule this altered task-set, starting at 0ms, and going up to 100ms *(SCHEDULE 1.1, PAGE 11)*, there was never a single missed deadline when using RM, which means this altered task-set is scheduleable under RM.

c) Again, if we attempt to run the RM equation on this new task-set *(TABLE 1.3, PAGE 10)* we'll get: $1.20805 \leq 0.74349$, which again, doesn't hold, so we once again need to attempt to schedule this new task-set.

This time around, if we attempt to schedule the task-set *(SCHEDULE 1.2, PAGE 11)*, we'll find a missing deadline at 19ms for the "Star Tracking" task.

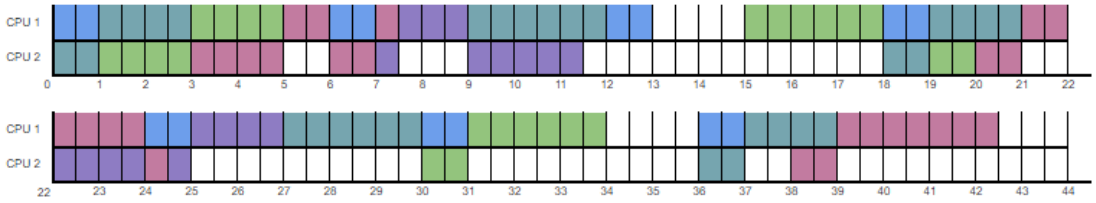
d) Using RM-FF, and the order given to us, then we get the following task-to-CPU assignment:

CPU 1	CPU 2
ST	AC
OC	ISC
GS	

e) If we take into consideration how Global EDF works, then we can quickly see that, for $N = 1$, then it's just a simple EDF schedule. Thankfully, EDF also has an equation to test the schedulabilaty of a task-set, and in EDF's case, the equation is a requirement, the equation is: $\sum_{i=1}^{m} \frac{C_i}{T_i} \leq 1$. If we apply this to the task-set with all 5 tasks *(TABLE 1.3, PAGE 10)*, we get:

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq 1 \equiv 1.20805 \leq 1$$

As we can see, the equation doesn't hold for $N = 1$, however, it might for higher values of N, so let's try scheduling this task-set using Global EDF with $N = 2$:



Queue

Time	Ordered Task Set
0	AC, GS, ISC, ST, OC
1	GS, ISC, ST, OC
2	GS, ISC, ST, OC
3	ISC, ST, OC
4	ISC, ST, OC
5	ST, OC
6	AC, ST, OC
7	ST, OC
8	OC
9	GS, OC
10	GS, OC
11	GS, OC
12	AC
13	-
14	-
15	ISC
16	ISC
17	ISC
18	AC, GS, ISC
19	GS, ISC, ST
20	GS, ST
21	ST
22	ST, OC
23	ST, OC
24	AC, ST, OC
25	OC
26	OC
27	GS
28	GS
29	GS
30	AC, ISC
31	ISC
32	ISC
33	ISC
34	-
35	-
36	AC, GS
37	GS
38	GS, ST
39	ST
40	ST
41	ST
42	ST
43	-

Task	Arrival Times
ST	0 19 38 57
OC	0 22 44
GS	0 9 18 27 36 45
AC	0 6 12 18 24 30 36 42
ISC	0 15 30 45

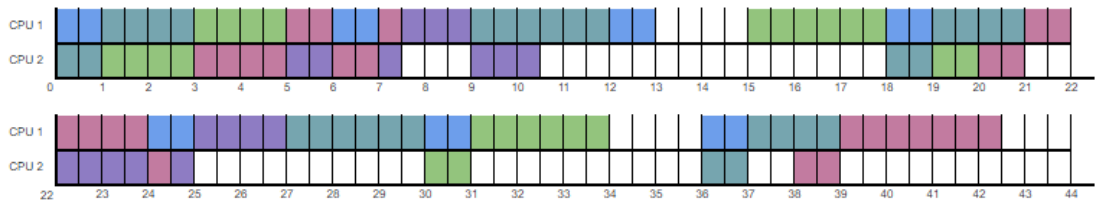
Task	Color
ST	
OC	
GS	
AC	
ISC	

Task	Length
ST	4.5
OC	4.5
GS	3
AC	1
ISC	4

As we can see, the program is scheduleable under Global EDF.

f) If we use Global EDF as a basis on how Global RM would work, then the policy for Global RM would be, maintain a global priority queue of all tasks that need doing, ordering them by the lowest period first, and then whenever a task finishes, or a task arrives, re-calculate the queue order, and pull in however many tasks you can considering the number of CPUs available to you.

Once again, we know that Global RM with $N = 1$, would just be a regular RM schedule, and since we know that it doesn't work on this task set *(TABLE 1.3, PAGE 10)*, we need to try with $N = 2$, so if we draw that schedule we get:



Queue

Time	Ordered Task Set
0	AC, GS, ISC, ST, OC
1	GS, ISC, ST, OC
2	GS, ISC, ST, OC
3	ISC, ST, OC
4	ISC, ST, OC
5	ST, OC
6	AC, ST, OC
7	ST, OC
8	OC
9	GS, OC
10	GS, OC
11	GS
12	AC
13	-
14	-
15	ISC
16	ISC
17	ISC
18	AC, GS, ISC
19	GS, ISC, ST
20	GS, ST
21	ST
22	ST, OC
23	ST, OC
24	AC, ST, OC
25	OC
26	OC
27	GS
28	GS
29	GS
30	AC, ISC
31	ISC
32	ISC
33	ISC
34	-
35	-
36	AC, GS
37	GS
38	GS, ST
39	ST
40	ST
41	ST
42	ST
43	-

Task	Arrival Times
ST	0 19 38 57
OC	0 22 44
GS	0 9 18 27 36 45
AC	0 6 12 18 24 30 36 42
ISC	0 15 30 45

Task	Color
ST	
OC	
GS	
AC	
ISC	

AC
GS
ISC
ST
OC

Task	Length
ST	4.5
OC	4.5
GS	3
AC	1
ISC	4

As we can see, the program is scheduleable under Global RM.

Answer 2

a) The way I see it, we can initialize the items in the $flag$ array to whatever values we'd like. Since in the first iteration of each of the concurrent loops they'll both be set to true before anything else, and who goes first will depend on the $turn$ value.

So if $turn = i$, the i loop will wait for the j loop to turn it's $flag$ value to $false$, and when that is done it'll execute it's order and then set itself to false, and the $turn$ value to j, so that the j loop can iterate and do the same thing. And vice-versa if $turn = j$ initially.

b) Despite Process i's attempts to get hold of the resource, if it never sets $turn = j$, then Process j will never execute at all, so setting the $turn$ value is key so that both processes execute.

c) No, the output displayed here can never happen for a simple reason, since each process, after executing, sets its $flag$ value to false, and then proceeds to switch the value of $turn$ to the other process, this ensures that the processes take turns executing, and never execute more than once before the other process does.

As such, we could never have 2 print statements inside the critical section of the same process in a row, without a breaking statement of being inside the critical section of the other process.

d) Yes this will cause a significant problem, as it'll leave the inside loop $while(flag[j])$, and continue executing the code inside the process. If we were to implement this, we'd get the following output:

Process i: Entering CS
Process j: Entering CS
Process i: Inside CS
Process j: Inside CS

As you can see, the two Inside prints are being done at the same time, which would cause the program to fail its proper use.

e) If we create a shared Semaphore S with the count of 1:

Listing 1: Semaphore Declaration

```
1 Semaphore S = new Semaphore(1);
```

And pass it to both the processes, then we can use the code below to get 2-party mutual exclusion:

Listing 2: 2-Party Mutual Exclusion.

```
1  Process i:
2      Repeat:
3          print_line("Process i: Entering CS");
4          try {
5              S.acquire();
6              /* CRITICAL SECTION -- BEGIN */
7              ...
8              print_line("Process i: Inside CS");
9              ...
10             /* CRITICAL SECTION -- END */
11             S.release();
12         } catch (InterruptedException exc) {
13             print_line(exc);
14         }
15     Forever
```

Answer 3

a) To start, I'd make a common Semaphores for the plates, as well as one for the number of pizza slices:

Listing 3: Semaphore Declaration

```
Semaphore plateS = new Semaphore(numberOfPlates);
Semaphore pizzaS = new Semaphore(8 * initialNumberOfPizzas);
```

With that done, we can implement the algorithm below:

Listing 4: Roomie Algo

```
public void StudyingProcess(int index, Semaphore plateS, Semaphore pizzaS):
    while (True) {
        try {
            plateS.acquire();
            try {
                pizzaS.acquire();
                self.study();
            } catch (InterruptedException noMorePizzaException) {
                print_line(noMorePizzaException);
                plateS.release();
                break;
            }
        } catch (InterruptedException notEnoughPlatesException) {
            print_line(notEnoughPlatesException);
        }
        plateS.release();
    }
}
```

b) If we assume that we already have the two Semaphores that we declared above, then, since as stated we need BOTH a fork and a knife, we can create a single other Semaphore, that will represent a pair of cutlery, since we know we have 3 knifes and 5 forks, then we have at most 3 pairs of cutlery, so:

Listing 5: More Semaphore Declaration

```
1 Semaphore cutleryS = new Semaphore(3);
```

With that done, we can change the implementation above and get the algorithm below:

Listing 6: Roomie Algo 2.0

```
1  public void StudyingProcess(int index, Semaphore cutleryS, Semaphore plateS,
2                              Semaphore pizzaS):
3      while(True) {
4          try {
5              cutleryS.acquire();
6              try {
7                  plateS.acquire();
8                  try {
9                      pizzaS.acquire();
10                     self.study();
11                 } catch (InterruptedException noMorePizzaException) {
12                     print_line(noMorePizzaException);
13                     plateS.release();
14                     break;
15                 }
16             } catch (InterruptedException notEnoughPlatesException) {
17                 print_line(notEnoughPlatesException);
18             }
19             plateS.release();
20         } catch (InterruptedException notEnoughCutleryException) {
21             print_line(notEnoughCutleryException);
22         }
23         cutleryS.release();
24     }
25 }
```

c) If we assume that we already have the Semaphores given above, then we must first create a new Semaphore, that will be used to assure that just 1 order if completed at a time:

Listing 7: Even More Semaphore Declaration

```
1 Semaphore orderS = new Semaphore(1);
```

we first need to make a new method called $orderPizza$, that is called to make the pizza order:

Listing 8: Pizza Ordering Method

```
1  public Semaphore orderPizza(int numberOfPizzas, Semaphore orderS) {
2      while (True) {
3          Pizza newPizza = null;
4          try {
5              orderS.acquire();
6              newPizza = new Pizza(numberOfPizzas);
7          } catch (InterruptedException orderNotPossibleException) {
8              print_line(orderNotPossibleException);
9          }
10         orderS.release();
11         if (newPizza != null) {
12             return new Semaphore(numberOfPizzas*8);
13         }
14     }
15 }
```

The algorithm above will get two arguments, numberOfPizzas (which in this case is always 1, but better to future-proof the code, and the Semaphore declared above that is used to assure only 1 pizza is baked at a time.

With that done, we can change the implementation from berfore and get the algorithm below:

Listing 9: Roomie Algo 3.0

```
public void StudyingProcess(int index, Semaphore cutleryS, Semaphore plateS,
                            Semaphore pizzaS, Semaphore orderS):
    while(True) {
        try {
            cutleryS.acquire();
            try {
                plateS.acquire();
                try {
                    pizzaS.acquire();
                    if (pizzaS.availablePermits() == 0) {
                        pizzaS = orderPizza(1, orderS);
                    }
                    self.study();
                } catch (InterruptedException noMorePizzaException) {
                    print_line(noMorePizzaException);
                    plateS.release();
                    break;
                }
            } catch (InterruptedException notEnoughPlatesException) {
                print_line(notEnoughPlatesException);
            }
            plateS.release();
        } catch (InterruptedException notEnoughCutleryException) {
            print_line(notEnoughCutleryException);
        }
        cutleryS.release();
    }
}
```

Basically what we are doing is, after acquiring our slice of pizza, if the remaining number of available slices (or permits as Java calls them) is 0, that means ours was the last, so we call the $orderPizza()$ method, and assign the value returned from it to $pizzaS$, thus replenishing our stock of pizzas.

The reason we only do this after we have taken our slice is quite simple actually. If we do it after, due to us already having set the $pizzaS$ Semaphore, we are basically wasting the leftover slice. In the case where the acquire function fails, (ie: there are already 0 slices), then we don't even have to worry, since the code throws an exception; and if there are any slices left we also don't worry about it. So our code works as desired.

Auxiliary Materials

All these materials were obtained using Google Spreadsheets. The actual spreadsheet can be found here `https://bit.ly/3p3Kp3d`

TABLE 1.1

Task	Period T (ms)	WCET C (ms)
ST	19	4.5
OC	22	4.5
GS	9	3

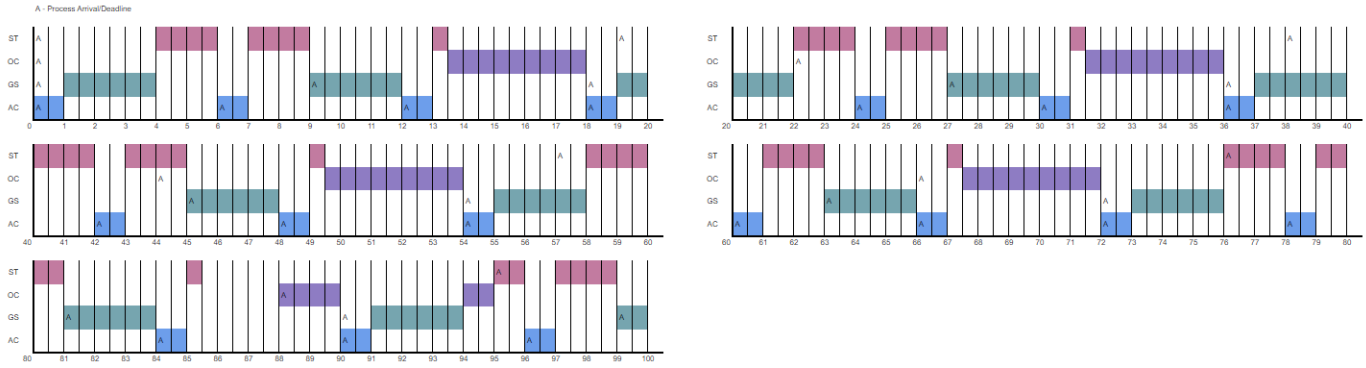
TABLE 1.2

Task	Period T (ms)	WCET C (ms)
ST	19	4.5
OC	22	4.5
GS	9	3
AC	6	1

TABLE 1.3

Task	Period T (ms)	WCET C (ms)
ST	19	4.5
OC	22	4.5
GS	9	3
AC	6	1
ISC	15	4

SCHEDULE 1.1



SCHEDULE 1.2