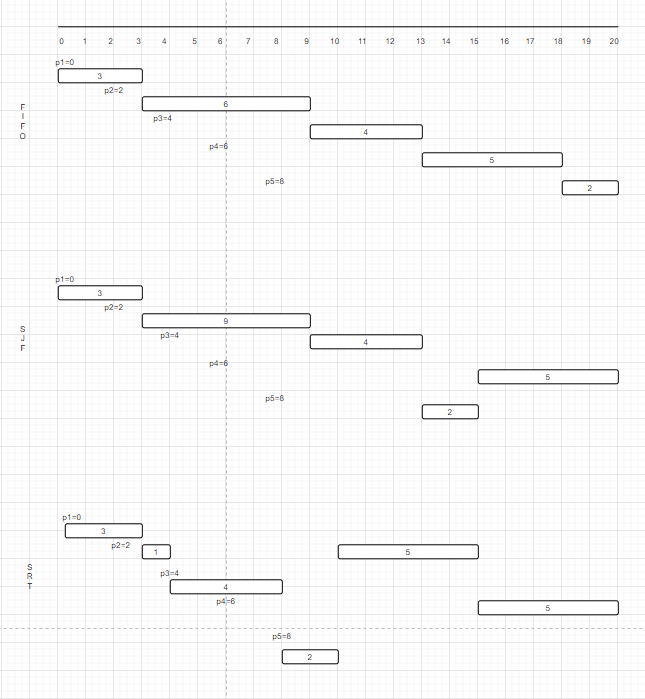
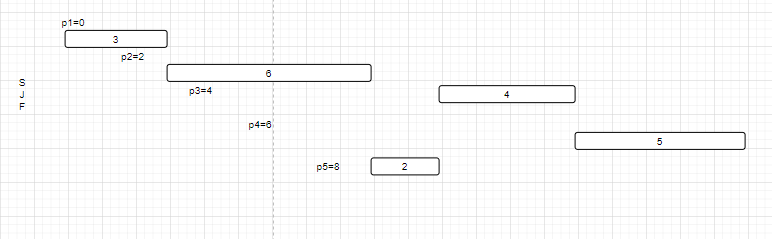
***COIS 3320H Winter 2021 Assignment 2***

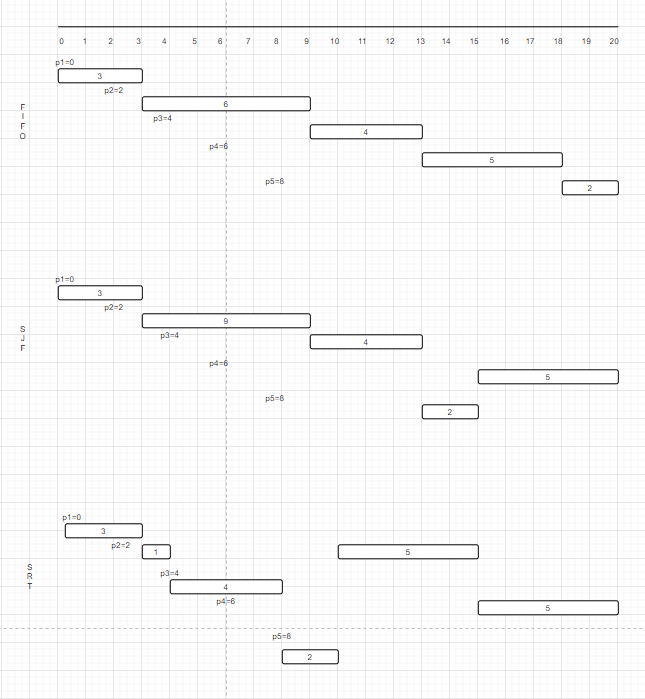
**Q1. Scheduling using FIFO, SJF, and SRT**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | P1 | P2 | P3 | P4 | P5 |
| Arrival Time | 0 | 2 | 4 | 6 | 8 |
| Total CPU Time | 3 | 6 | 4 | 5 | 2 |

1. **For the 5 processes described below draw a timing diagram showing when each process will execute under FIFO, SJF, and SRT.**







I have made all the FIFO, SJF and SRT for one timeline, it just was not clear when I snipped it all together so they are in different parts.

1. **Determine the ATT (Average Turnaround Time) for each scheduling algorithm for the 5 processes described below**

For each process -> waiting time + running time

ATT -> add each process/ number of process (5)

**ATT for FIFO :**

P1 = 0 + 3 = 3

P2 = 1 + 6 = 7

P3 = 5 + 4 = 9

P4 = 7 + 5 = 12

P5 = 10 + 2 = 12

ATT = (3+7+9+12+12)/5 = 43/5 = 8.6

**ATT for SJF :**

P1 = 0 + 3 = 3

P2 = 1 + 9 = 10 //wrong

P3 = 7 + 4 = 9

P4 = 9 + 5 = 14

P5 = 5 + 2 = 7 //wrong

ATT = (3+10+9+12+12 )/5 = 43/5 = 8.6

**ATT for SRT :**

P1 = 0 + 3 = 3

P2 = 6 + 6 = 12 //wrong

P3 = 0 + 4 = 4

P4 = 9 + 5 = 14

P5 = 0 + 2 = 2

ATT = (3+12+4+14+2)/5 = 35/5 = 7

**Q2. Predicting CPU Bursts**

**The following sequence of CPU bursts has been observed: 7, 5, 6, 15, 15, 15.**

1. **Using 7 as the initial estimate, S0, generate the sequence of predictions, Si, for the corresponding observed values Ti, where 1 <= i <=5 and α = 0.8.**

α = 0.8 ; 1 <= i <= 5

Sn+1 = αTn + (1- α)Sn

Ti for ( 0 <= i <= 5) are (7, 5, 6, 15, 15, 15)

For 1 <= i <= 5 are (5, 6, 15, 15, 15)

S0 = 7

For i = 1

S1 = (0.8)T0 + (1 – 0.8)\*S0

**S1 = (0.8)\*7 + (0.2)\*7 = 7**

For i = 2

S2 = (0.8)T1 + (1 – 0.8)\*S1

**S2 = (0.8)\*5 + (0.2)\*7 = 5.4**

For i = 3

S3 = (0.8)T2 + (1 – 0.8)\*S2

**S3 = (0.8)\*6 + (0.2)\*5.4 = 5.88**

For i = 4

S4 = (0.8)T3 + (1 – 0.8)\*S3

**S4 = (0.8)\*15 + (0.2)\*5.88 = 13.176**

For i = 5

S5 = (0.8)T4 + (1 – 0.8)\*S4

**S5 = (0.8)\*15 + (0.2)\*13.176 = 14.635**

1. **Repeat the same predictions for α = 0.5.**

α = 0.5 ; 1 <= I <= 5

Sn+1 = αTn + (1- α)Sn

Ti for ( 0 <= i <= 5) are (7, 5, 6, 15, 15, 15)

For 1 <= i <= 5 are (5, 6, 15, 15, 15)

S0 = 7

For i = 1

S1 = (0.5)T0 + (1 – 0.5)\*S0

**S1 = (0.5)\*7 + (0.5)\*7 = 7**

For i = 2

S2 = (0.5)T1 + (1 – 0.5)\*S1

**S2 = (0.5)\*5 + (0.5)\*7 = 6**

For i = 3

S3 = (0.5)T2 + (1 – 0.5)\*S2

**S3 = (0.5)\*6 + (0.5)\*6 = 6**

For i = 4

S4 = (0.5)T3 + (1 – 0.5)\*S3

**S4 = (0.5)\*15 + (0.5)\*6 = 10.5**

For i = 5

S5 = (0.5)T4 + (1 – 0.5)\*S4

**S5 = (0.5)\*15 + (0.5)\*10.5 = 12.75**

**Q3. Turnaround Time Under Round Robin Scheduling**

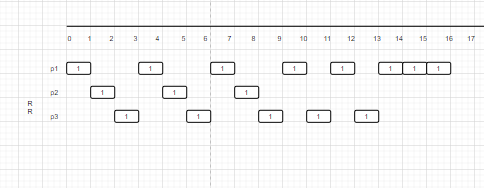
**Three processes, p1, p2, p3, arrive at the same time and start executing using RR scheduling.**

**p1 starts first, followed by p2, and then p3.**

**The respective total CPU times of the 3 processes are 8, 3, 5-time units.**

**The context switching time is negligible.**

1. **Determine the average turnaround time, ATT, when the quantum is Q = 1 time unit.**



Q = 1

For each process -> waiting time + running time

ATT -> add each process/ number of process (3)

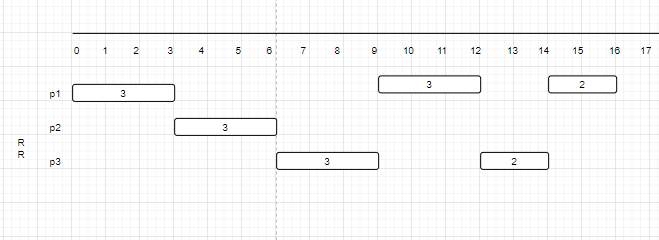
P1 = 8 + 8 = 16

P2 = 5 + 3 = 8

P3 = 8 + 5 = 13

ATT = (16+8+13)/3 = 12.333

1. **Determine the average turnaround time, ATT, when the quantum is Q = 3-time 3units.**



Q = 3

For each process -> waiting time + running time

ATT -> add each process/ number of process (3)

P1 = 8 + 8 = 16

P2 = 3 + 3 = 6

P3 = 9 + 5 = 14

ATT = (16+6+14)/3 = 12

**Q4. Determining Process Execution Time**

**n processes are time-sharing the CPU using RR scheduling, each requiring T ms of CPU time to complete.**

1. **How long will the execution take on a machine with n CPUs?**

There are ‘n’ CPUs. So each one of the ‘n’ processes can be scheduled on each one of the ‘n’ CPUs. Since, 1 process on 1 CPU, all the ‘n’ processes will be executing in parallel.

Therefore, total time = time required by 1 process

= **T ms**

1. **How long will the execution take on a single CPU machine when the context switch overhead is zero?**

One CPU 🡺 all ‘n’ processes on the same CPU

Context switch time = 0

Total time required = Number of processes \* Time required by each process

= n \* T

= **nT ms**

1. **How long will the execution take on a single CPU machine when:**
2. **The length of the time quantum is Q ms**
3. **The time to perform each context switch is S ms**

For this question, the assignment pdf question had the same conditions for both I and II of this part c, so I referenced from zybooks and got the II condition.

Time to perform each Context switch = S ms

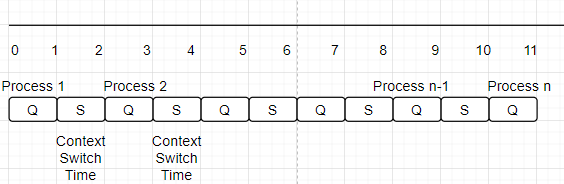
Quantum Time = Q ms

The first process starts, runs for Q ms then there is S ms for context switch

Then, second process runs for Q ms then there is S ms for context switch

This goes on for n processes.

So for one round :



Number of time quantum slots required = T/Q = Number of context switches

* Total time for context switches = S\*(T/Q)
* Total time for context switch for n processes = n\*S(T/Q)
* There fore, total time of execution for n processes = n\*T + n\*S\*(T/Q)

1. **Repeat the previous calculation using n=5, T=10,000, Q=100, S=10**

n\*T = 5\*10,000 = 50,000

(n\*S\*T)/Q = (5\*10 \* 10,000)/100 = 5000

Therefore total execution time = 50,000 + 5,000 = **55,000 ms**

**Q5. Scheduling with MLF**

|  |  |  |
| --- | --- | --- |
| Process | Arrival | Total CPU Time |
| P1 | 0 | 1 |
| P2 | 1 | 3 |
| P3 | 1 | 14 |

**5 Priority Levels**

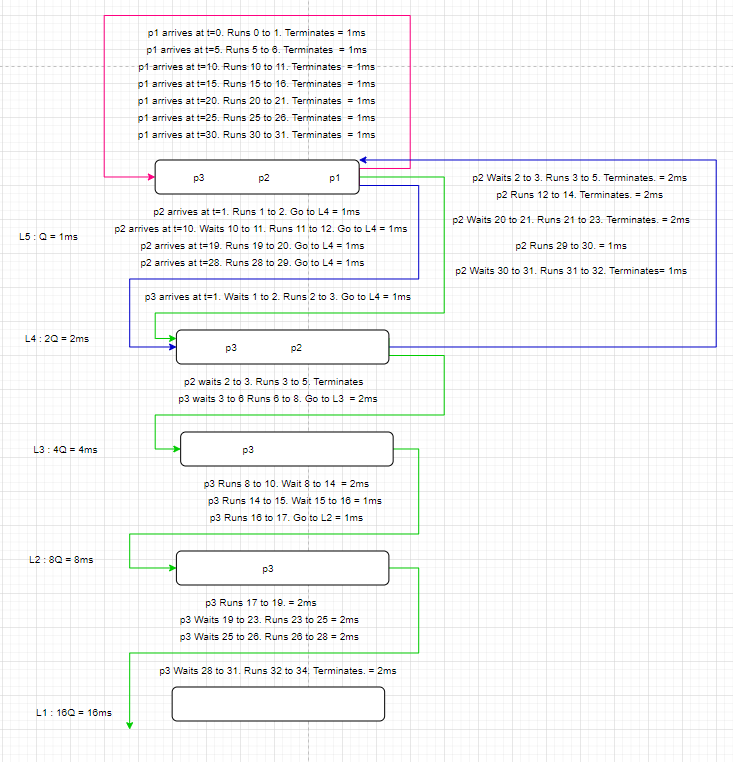
**At level 5 🡪 Q=1ms then doubles 🡪 2Q, 4Q, 8Q…**

**After termination, process p1 blocks for 4 ms and then re-enters the queue again at level 5. Similarly, process p2 blocks for 5 ms and then re-enters the queue again at level 5.**

1. **Draw a timing diagram for the first 33 ms. On each of the 3 lines (one per process) show when the process is running and at which priority level.**

The red line is for p1, blue line is for p2, and green line is for p3

These are the processes happening in MLF for the first 33 ms.



1. **Determine the ATT for each process.**

ATT for each process.

For each process -> waiting time + running time

ATT -> add each process/ number of process (3)

P1 = 0 + 7 = 7ms 🡪 Since, p1 always starts at L5 it has higher priority and never waits.

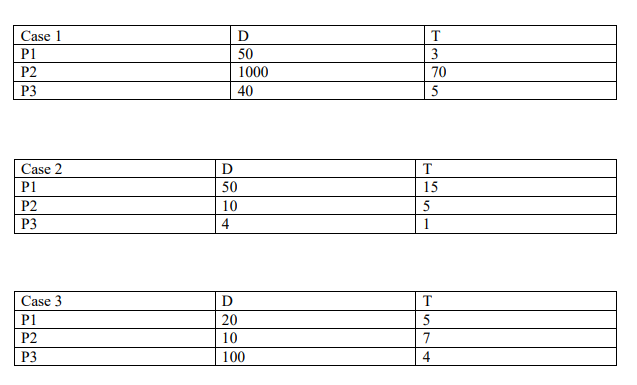
P2 = 4 + 12 = 16ms 🡪 Since p2 arrives at p5, it also has less wait time, and it runs 4 times and hence 4\*3 = 12 ms is the running time

P3 = 20 + 12 = 32ms 🡪 Since p3 finishes at 34ms, therefore, only 12 is the execution time and not 13

ATT = (7+16+32)/3 = 55/3 = 18.333ms

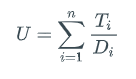
**Q6. Determining Feasible Schedules**

**Three periodic processes with the following characteristics are to be scheduled (D is the period and T is the total CPU time):**



1. **For each case, determine if a feasible schedule is likely to be generated by:**

A schedule is feasible if the deadlines of all processes can be met. The CPU utilization (U) is the sum of individual fractions of CPU times used by each process i.e. Ti/Di where Ti is the total CPU time and Di is the period process I (for each process).



Calculating Ti / Di for each process of each case.

|  |  |  |  |
| --- | --- | --- | --- |
| Cases | Case 1 | Case 2 | Case 3 |
| P1 | 3/50 | 3/10 | ¼ |
| P2 | 7/100 | 1/2 | 7/10 |
| P3 | 1/8 | 1/4 | 1/25 |

Case 1 🡪 U = 3/50 + 7/100 + 1/8 = 0.255  
Case 2 🡪 U = 3/10 + 1/ 2 + 1 /4 = 1.05  
Case 3 🡪 U = 1/ 4 + 7/10 + 1/25 = 0.99

1. **RM**

For a feasible schedule to be generated under RM scheduling, U should be less than 0.7

Therefore,

* Case 1, U = 0.255 🡪 thus, feasible schedule will be generated
* Case 2, U = 1.050 🡪 thus, no feasible schedule will be generated
* Case 3, U = 0.990 🡪 thus, no feasible schedule will be generated

1. **EDF**

For a feasible schedule to be generated under EDF scheduling, U should be less than equal to 1

Therefore,

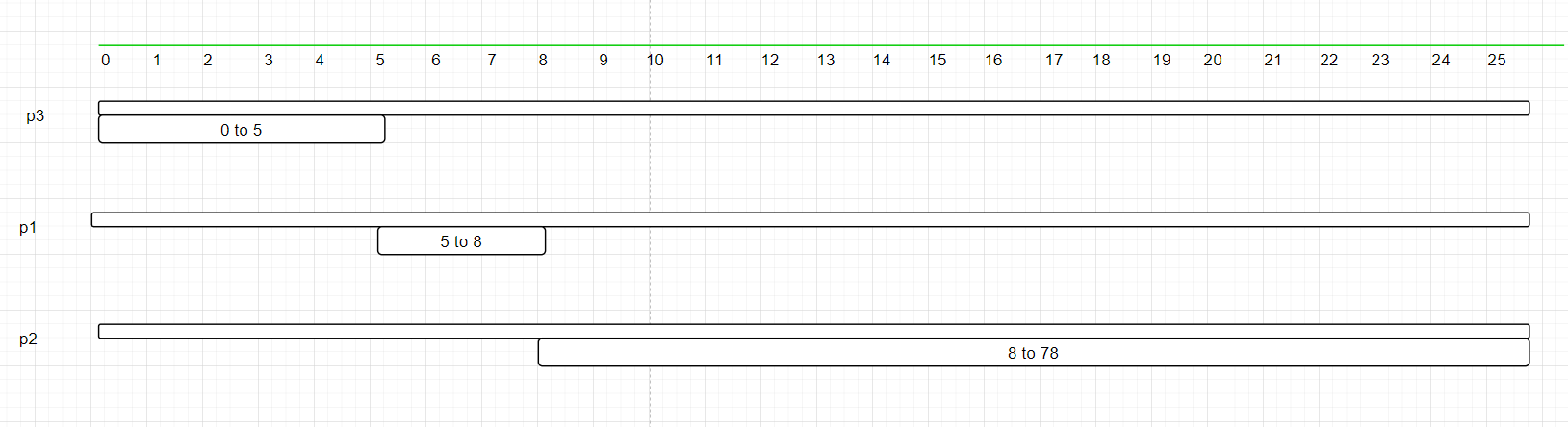
* Case 1, U = 0.255 🡪 thus, feasible schedule will be generated
* Case 2, U = 1.050 🡪 thus, no feasible schedule will be generated
* Case 3, U = 0.990 🡪 thus, feasible schedule will be generated

1. **Draw a timing diagram for the first 25-time units. For each of the 3 cases, show the schedules produced by RM and by EDF.**

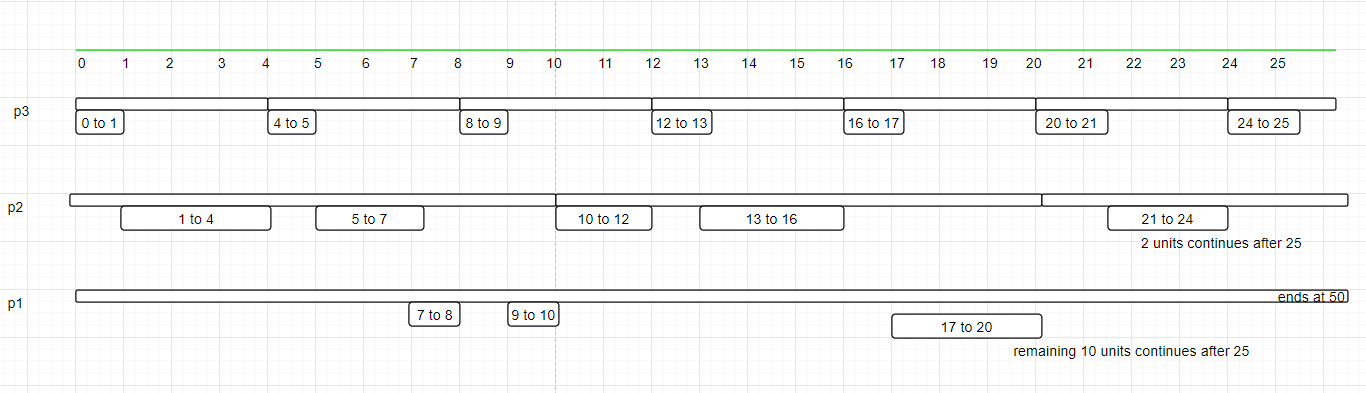
**RM**

Case 1 : The diagram is for only first 25 units, but the periods are much larger than that like 40, 50, 1000. So, their final deadline for each process will be much later and hence the long rectangles

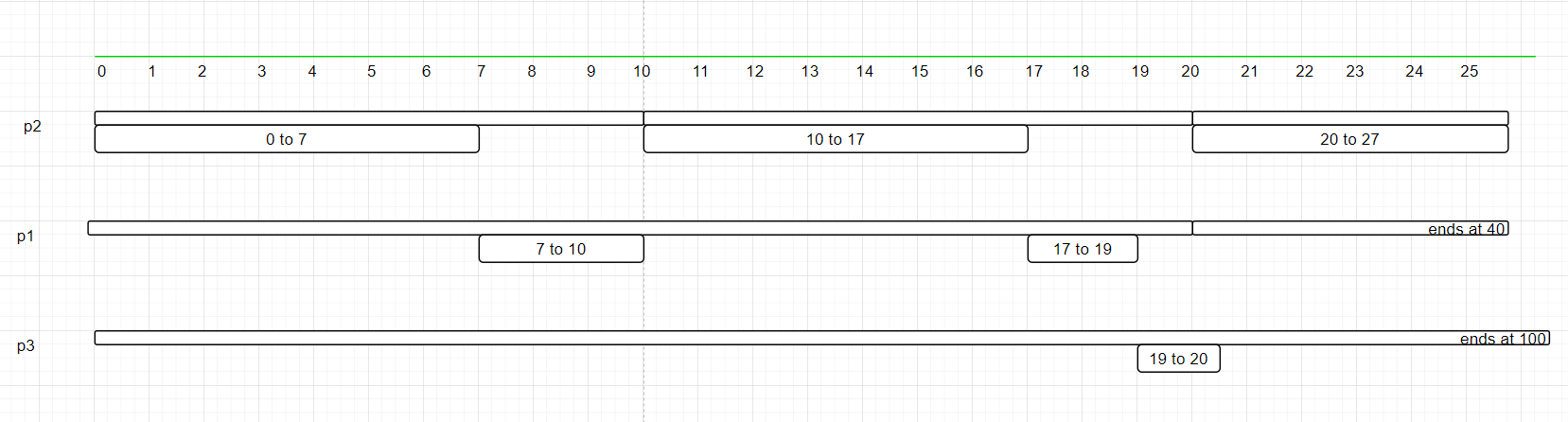
Priority P3 > P1 > P2



Case 2 : Priority P3 > P2 > P1



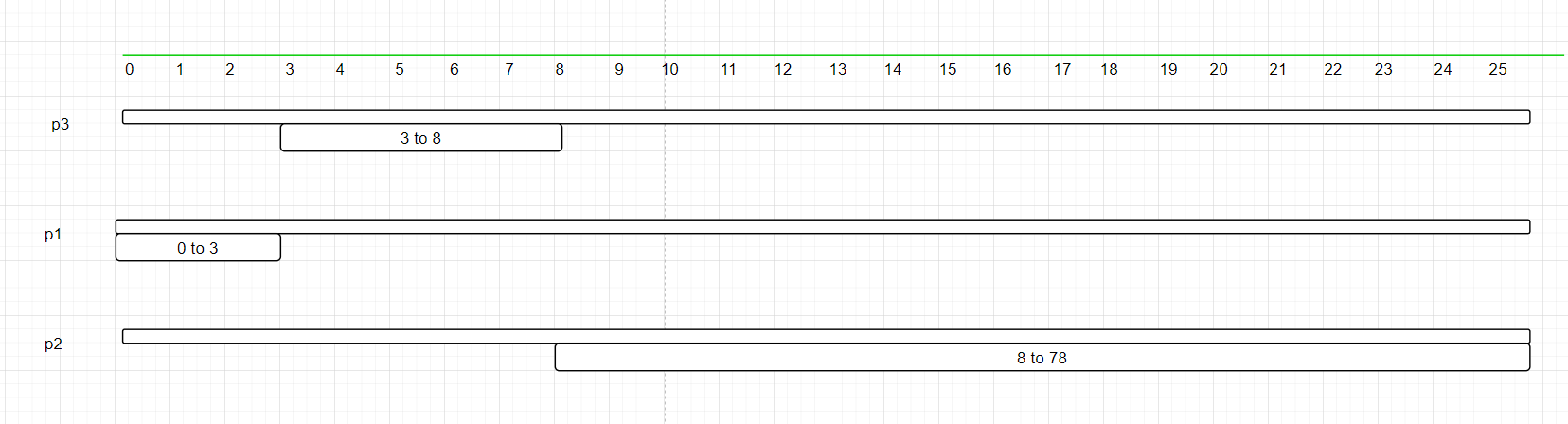
Case 3 : Priority P2 > P1 > P3



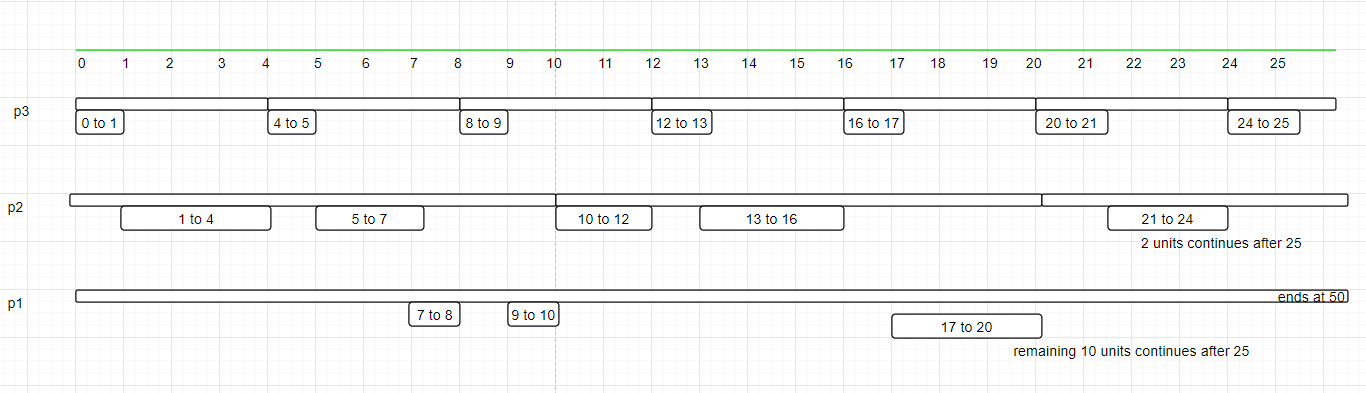
**EDF**

EDF prioritize the shorter remaining execution time.

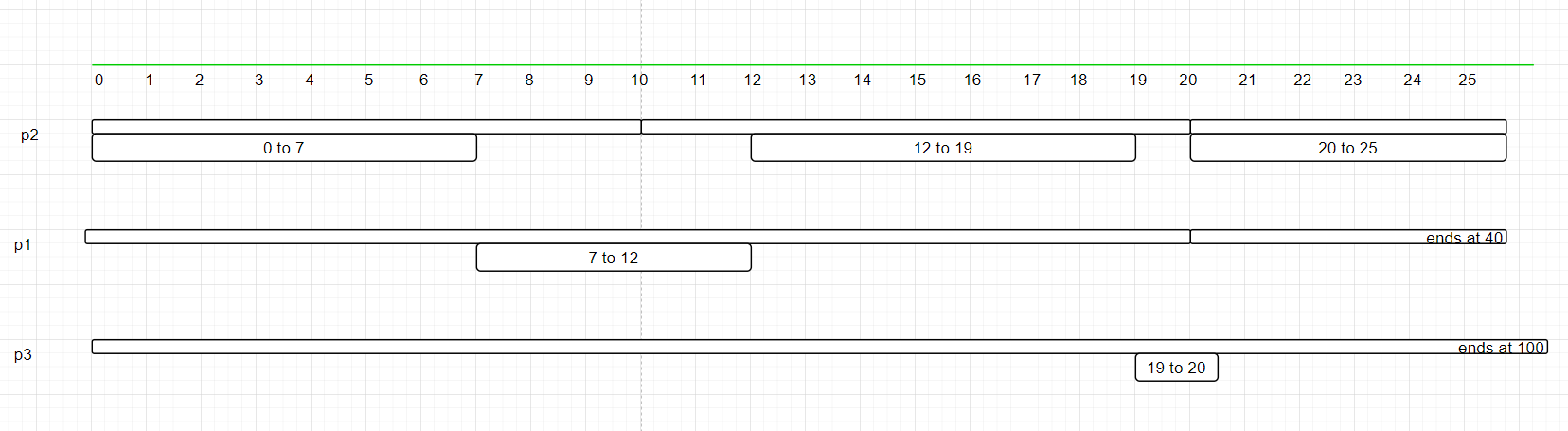
Case 1 : For case 1, p1 takes time 3 units and p3 takes time 5 units, so first p1 runs then p3.



Case 2 :

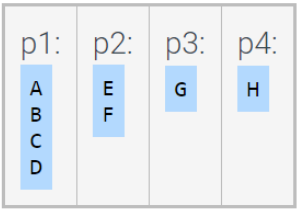


Case 3 : In the beginning technically p1 has less execution time, but p2 does not have enough time period, therefore p2 runs first. However, later, p1 continues running while p2 waits for a few time units.

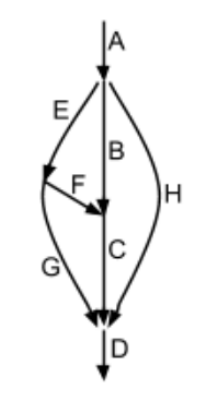


**Q7. Using P and V Operations to Enforce Precedence of Execution**

**Four processes are executing the computations A through H:**



**The computations must be executed in the order given by the following precedence graph**



1. **Insert P and V operations into the code to enforce the prescribed order. For example, since E and H must wait for A to complete, the processes p2 and p4 must each start with a P operation and A must end with 2 corresponding V operations.**

There are probably 5 semaphores a, b, c, d, e

The following statements define which semaphore value is supporting or ensuring which computation to execute.

Semaphore value ‘a’ supports completion of D 🡪 After C executes

Semaphore value ‘b’ supports completion of E 🡪 after A executes

Semaphore value ‘c’ supports completion of G 🡪 after E executes

Semaphore value ‘d’ supports completion of H 🡪 After A executes

Semaphore value ‘e’ supports completion of C 🡪 After F and B executes

*Case 1 : P1*

P(a) > A > V(b) > V(d) > B > P(e) > C > P(a) > P(a) > D

*Case 2 : P2*

P(b) > E > V(c) > F >V(e)

*Case 3 : P3*

P(c) > G > P(a)

*Case 4 : P4*

P(d) > H > V(a)

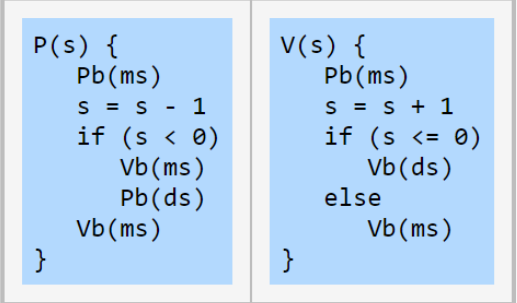
1. **What are the initial semaphore values?**

Initial Semaphore values will be a=1; b=0; c=0; d=0; e=0

At the end of all the computations, semaphore values will be a=0; b=0; c=0; d=0; e=0

**Q8. A Different Implementation of P and V**

**The following implementation of P and V uses busy waiting inside P. Rather than blocking the process when s is less or equal 0, the process is prevented from continuing by executing Pb(ds), where ds is a binary semaphore initialized to 0. A V(s) then unblocks a process by performing Vb on the binary semaphore ds.**



1. **Since the original solution also uses Pb and Vb (implemented using busy waiting), why is the above solution much worse in terms of performance?**

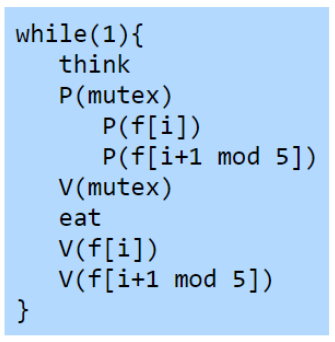
According to the above solution, when s is deducted and goes below zero, instead of the process being blocked, there is another binary semaphore on which Pb is performed. Since this is a binary semaphore, it can only take value 0 or 1, while the counting integer s takes any values.

When s<0 inside P(s), first ms is set back to 1 by Vb(ms) that is ms is released. Next, ds is set to 0 by Pb(ds). And in V(s) operations, when s is less than or equal to 0 after increment, that means there process in waiting list (that were blocked), or in this case, there are processes with semaphore value of ds set to 0, then ds is released by Vb(ds). Otherwise ms is released.

Having multiple semaphores like ds, which can be avoided, creates problems for performance and hence this solution is much worse in terms of performance. Ds semaphore takes up extra memory and also there is no mention of blocking or releasing the processes. There needs to be a condition that for Pb(ds) the process should be blocked and stopped running. A little mistake in wait or signal can lead to synchronization problems. For many processes in the waiting list, the ds semaphore is released the moment the first process from the waiting list is released, making the value of ds back to 1.

**Q9. An inadequate Solution to the Dining Philosophers Problem**

**Each of the five philosophers, p[i], in the dining philosophers problem executes the code:**



1. **Does the solution satisfy all requirements of the dining philosopher’s problem?**

In this approach, mutex acts like a waiter that does not allow all the philosophers to go for the fork at the same time. Let’s say there are the 5 philosophers – p[0], p[1], p[2], p[3], p[4].

Mutex = 1

*First p[0] :*

So, p[0] is able to request left fork

p(mutex) = success

p(f[i]) will work

p(f[i+2 mod 5]) will work

That means p[0] can request right fork and will be granted that.

V[mutex] = success

And eat; The philosopher gets to eat.

*P[1] :*

P(mutex) works

But p(f[i]) = p(f(1)) does not work. Since p[0] is currently eating.

So p[1] is blocked on left fork.

*P[2] :*

p[2] can eat because p[0] is eating and p[1] is blocked. But, because of mutex, p[2] is blocked on mutex at the step p(mutex).

This means only one philosopher can eat at one time.

Therefore, this does not satisfy all the requirements of the dining philosopher’s problem : two philosopher’s opposite to each other, i.e. not adjacent should be able to eat concurrently.