SISTEMAS OPERATIVOS

SCHEDULING

1) Las ráfagas de CPU son aquellas que realizan lecturas y escrituras a archivos y/o memoria. Más precisamente:
   Ráfagas CPU: Van del 0 al 2, 11 al 13, 21 y 22.
   Ráfagas E/S: Las restantes. Las que van del 3 al 10 y al 14 y 20.

   (Suponiendo que los tiempos están en milisegundos) Las ráfagas de CPU duran 3 milisegundos (y la última dura 2 ms) y las ráfagas de E/S duran 8 y 7 milisegundos respectivamente.

2) Como P_1 se bloquea fácilmente leyendo de la red, es conveniente utilizar un sistema de *scheduling* que alterna frecuentemente entre cada proceso. Sino, si esperáramos a terminar cada tarea para pasar a la próxima, podríamos demorarnos mucho tiempo. Lo mismo sucede con P_2, que requiere de un alto consumo de CPU. Si la misma se la "prestáramos" por un corto tiempo, debería de poder ejecutar cierta parte de sus ráfagas de CPU.

   Básicamente el algoritmo que más garpa usar es el *Round Robin*. Nos asegura *fairness* (re cipayo) a la hora de ejecutar cada proceso.
   Como P_2 tiene ráfagas de CPU prolongadas, seguro pueda aprovechar todo su *quantum*. También P_0 podría ejecutarse a la perfección, ya que como tiene ráfagas cortas de E/S, su *quantum* puede terminarse antes de tiempo y pasar al siguiente proceso más ágilmente.

   NOTA: No sé como funciona el sistema con P_1. Yo asumí que si se bloquea leyendo de la red, su *quantum* sigue hasta terminarse y luego pasar a la siguiente tarea. Quizás luego de bloquearse da por terminado su proceso y pasa al siguiente hasta estar *ready*.

3) Es un *scheduler* sin desalojo (*non preemptive*).
   REMINDER: Lo que tiene este tipo de *scheduling* es que una tarea se desaloja voluntariamente (ya sea por una *syscall* o porque está esperando una ráfaga de E/S).

4) Las que me acuerdo actualmente, las que generan *starvation* son:
   - Prioridad
   - SJF
   - SRTF
   - Colas de multinivel

   Prioridad y Colas de multinivel generan inanición por el mismo motivo: Siempre ejecutan primero los procesos de mayor nivel, por lo que si recibimos muchas tareas de alto nivel constantemente, los procesos de "menor importancia" no se ejecutan.

En SJF (*Shortest Job First*) pasa algo similar, pero con tareas de muy corta duración. Si recibimos constantes procesos con un tiempo muy cortito, estos siempre se ejecutan primero, y los más largos que están al fondo de la "fila" terminan sin ejecutarse.
SRTF es la versión *preemptive* de SJF. Genera *starvation* por el mismo motivo básicamente.

5) a) Puede generar inanición sobre los procesos. En *Round Robin*, el *scheduler* siempre ejecuta la tarea próxima que está en la cola de *readys*. En este sistema, si tenemos, por ejemplo, constantemente el P1 y P2 en la cola ready tal que:

P1 P1 P1 P2 P2 P2 P1 P2 P2 P1 P2 P1 … P3 P4

Los procesos 3 y 4 no acabarán por ejecutarse.

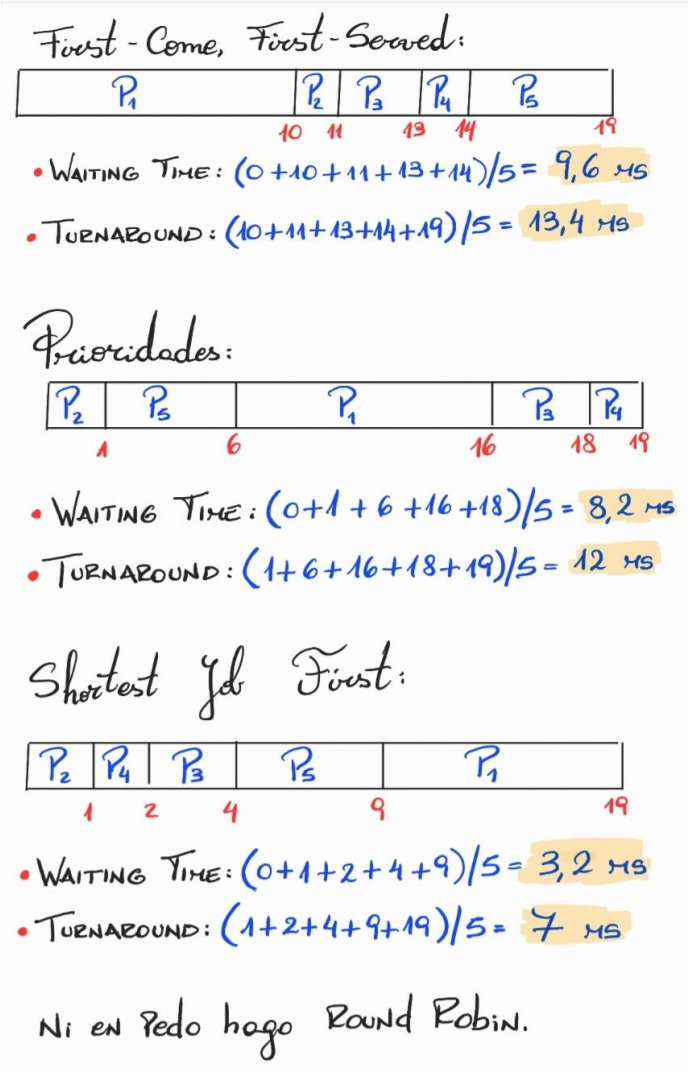
b) Puede interpretarse como ventaja que si un proceso aparece muchas veces en la cola, terminará su ejecución antes de lo estimado. Aún así, puede que se intente ejecutar una tarea que todavía no está *ready* pero aún así aparece en la lista (no sé que pasa en ese caso). Otra desventaja es la inanición que nombramos en el ítem anterior.

c) Si no queremos agregar entradas de más a la cola *ready* pero queremos mantener la idea de que el proceso termina antes de lo esperado, podríamos considerar aumentar el tamaño del *quantum* de cada tarea.

6)

Tuve que googlear que era un diagrama de Gantt, no estoy muy orgulloso de mí.

No evalué *Round Robin*, pero me imagino que el más rápido es **SJF**.



7) Veamos:

- *Waiting Time:* (0 + (3 + (10 - 4)) + 4 + 8 + 15) / 5 = **7,2 ms**
- *Turnaround:* (3 + 15 + 8 + 10 + 20) / 5 = **11,2 ms**

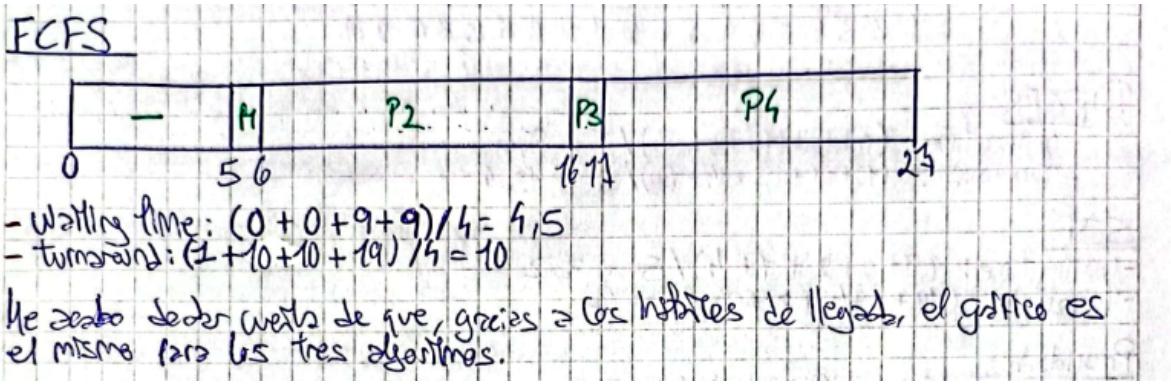
Notar que las sumas están en el orden P1 - P2 - P3 - P4 - P5. Como P2 es desalojado en un momento, su Waiting Time considera el tiempo que tardó en iniciarse por primera vez y lo que tardó en volverse a iniciar desde que terminó su primera ejecución. Luego, su Turnaround consiste básicamente en todo el tiempo que le llevó terminar su ejecución.

b) El *scheduler* es del tipo **SRTF**, ya que atiende los procesos en base al nivel de privilegio y también incluye desalojos de tareas.

8) a) (8 + 12 + 13) / 3 = **11 ms**
b) (8 + 9 + 13) / 3 = **10 ms** // Hay que tener en cuenta el orden de llegada
c) (2 + 6 + 14) / 3 = **7,2 ms** // No sé si la *idle* cuenta como una tarea más o no

9) Que lo parió viejo, justo apagué la tablet y no pienso dibujar en Paint. Me imagino que para estos ejercicios hay que tener en cuenta el tiempo de llegada, cosa de dejar la tarea *idle* activa en el caso de que no haya ningún proceso para ejecutar.

Miren que lindo el gráfico de Gonza:



CRÉDITOS A GONZA GUERRERO

10) No te la puedo creer, esta práctica es todo dibujos.

d) Considerando los distintos tipos de procesos que tenemos, muchas veces es preferible utilizar la política de *Round Robin* antes que *SJF*. Veamos porqué:
Como el *scheduling SRTF* toma SIEMPRE el proceso con menor tiempo de ejecución, es posible que recibamos muchos procesos del tipo “real-time”, que son tareas de muy corta duración (sucede lo mismo con las interactivas, pero depende de la capacidad de respuesta esperada).

A lo que iba con esta explicación, es que la política de SRTF siempre tenderá a atender a los procesos *real-time*, dejando a las tareas de tipo *interactivas* y *batch* de lado, generando *Starvation*.

Es importante destacar que la política de *Round Robin* brinda equilibrio a la hora de ejecutar las tareas que se encuentran en estado *ready*.

11) Uh, medio garronazo. Sigo sin la tablet, estoy resolviendo toda la guía de un tirón. Pero bueno, asumo que lo que hay que hacer es armar las dos colas de prioridad basándonos en el orden de llegada de cada tarea. Luego, si hay 2 colas de prioridad distintas, ejecutamos las tareas que están en la de más alto nivel (la de número más bajo) hasta que se acaben.
Por ejemplo, acá tenemos P1 y P2, que van a la cola 1 → Procesamos estas tareas (cada una con un *quantum* de 1 segundo) hasta terminarlas. Después, ejecutamos las tareas de la cola 2. Hay que estar atentos otra vez al tiempo en el que llega cada proceso.

12) Em… ¿ *Multilevel Feedback Queue* ? ¿O tengo que diseñar YO un nuevo algoritmo? No me importa, la MFQ funciona también con colas de prioridad, pero a medida que las tareas de menor prioridad no son tenidas en cuenta, estas suben su prioridad poco a poco, así es más probable que sean ejecutadas.

No deja en *Starvation* a los intensivos de CPU ya que estos se siguen ejecutando. La diferencia es que los otros aumentan su prioridad, es decir, si tenemos muchíííííísimos procesos de uso intensivo de CPU y otros de E/S, los de E/S aumentan su prioridad hasta llegar al nivel 1, así se ponen en fila y también son ejecutados.

13)

- **FIFO:** Es un poco aleatorio, depende del orden de llegada acá. Aún así, los procesos más cortos son desfavorecidos, ya que si llegan primero muchas tareas de larga duración, hasta no terminar de ejecutarse, no le dan lugar a otras.
- **Round Robin:** Favorece a las más cortas (en general, a todas). Como se le asigna una misma cantidad de tiempo de CPU a cada tarea, todos los procesos esperan su turno durante una cantidad de tiempo equilibrada.
Para cumplir esto último, es OBLIGATORIO tener un quantum que no sea ni muy largo ni muy corto. Algo equilibrado.
- **Multilevel Feedback Queue:** No estoy muy seguro del lugar en donde se ubican las tareas más cortas. Asumo que en los niveles más altos, así que este algoritmo les viene bárbaro.

14) En este ejercicio entiendo que las tareas de menor nivel también tienen que ser tenidas en cuenta al momento de estar atendiendo una alarma.

Consistiría en una *Multilevel Feedback Queue*. En el caso de llegar un proceso de alarma, este pasaría inmediatamente a la cola de mayor prioridad en la que solo pueden encolarse alarmas de forma inmediata.
Veamos: Tenemos colas de prioridad 2 en adelante. Luego, tenemos una cola de prioridad 1 a la que solo pueden ingresar las alarmas apenas aparecen. Si las otras tareas más pequeñas también requieren ser atendidas, aumentan su prioridad hasta lograr su cometido.

15) NO QUEREMOS perjudicar excesivamente el *throughput* del sistema. Esto significa que no queremos disminuir la cantidad de procesos terminados por unidad de tiempo.

→ Utilizaría un *scheduler* con *Round Robin*.
Como tenemos procesos interactivos, es importante dar un rápido tiempo de respuesta para darle una sensación de inmediatez al usuario. Además, los trabajos de procesamiento de datos tienen que leer archivos muy grandes, así que no es muy conveniente usar FIFO para este sistema. Veamos porqué:
Los archivos son muy grandes, entonces si el primer proceso que tenemos es una lectura de archivo, hay que esperar una gran cantidad de tiempo a que el proceso termine. Luego, si durante la lectura del archivo llega un proceso interactivo, el usuario no recibirá respuesta alguna, pues primero tendrá que leerse el archivo, hacer una pequeña suma y luego grabarla.

Entonces, la elección de *Round Robin* me parece óptima ya que mientras los archivos se procesan, el *scheduler* también puede atender los procesos interactivos que el usuario genere.

16) Tenemos que tener en cuenta los procesos *real-time* y los interactivos. Como hay botones de brillo y contraste, y otros de zoom-in y zoom-out, tenemos que tener en cuenta estas tareas participativas.
Además, si queremos permitir una edición “en vivo” de las imágenes, es obligatorio que las tareas de *real-time* devuelvan el resultado esperado dentro de sus limitaciones de tiempo.

Como queremos darle prioridad a las tareas *real-time* e *interactivas*, usaría un *scheduler* de tipo *Multilevel Feedback Queue*. En él, las tareas mencionadas se ubicarían en las colas de prioridad más altas.
A su vez (no sé si así funciona MFQ también), cada cola funcionaría de forma *Round Robin*, para dar una sensación de simultaneidad a la hora de apretar los botones de edición y/o zoom.

17) La idea parece ser una especie de “*polling*”. Hay que observar constantemente la cámara de seguridad, para que la misma pueda correr el algoritmo de detección de objetos.

Queremos que las políticas de *scheduling* sean justas, por lo que posiblemente opte por un *scheduler* del tipo *Round Robin*. Me queda ver qué hacer con el resto de la estructura.
Hay que tener en cuenta que a la noche, los procesos de vídeos activos son muy pocos y cortos, entonces quizás aquí se puede implementar una "cola nocturna" (nada sarpado ni sexual) que funcione con políticas FIFO.
También me faltó mencionar las alarmas → Tienen que tener la prioridad más alta, así nos aseguramos de cumplir con el *deadline* pedido.

Entonces, resumiendo, tenemos:
Multilevel Feedback Queue de 3 niveles

1) Para las alarmas.
2) Para las cámaras y el algoritmo de detección. (Funciona con RR).
3) Para la actividad nocturna. (Funciona con FIFO).

Hay que ver cómo resolver el órden de ejecución de las alarmas. Cada una tiene un *deadline* estricto imperdible, así que habría que ordenarlas considerando primero el MENOR *deadline* y por último el mayor.