DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Grupo: 6

Sistemas Operativos
Primer Cuatrimestre 2017

Integrante	LU	Correo electrónico
Emiliano Galimberti	109/15	emigali@hotmail.com
Ignacio M. Fernandez	47/14	nachofernandez.1995@hotmail.com
Pedro Mattenet Riva	428/15	peter_matt@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

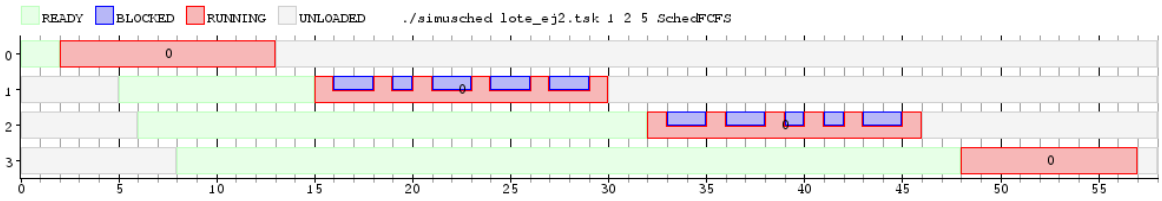
Índice

1. Ejercicio 1

En este ejercicio nos pidieron programar una tarea. La función a programar que simula la tarea tiene como parámetros su identificador de proceso (`pid`) y como segundo parámetro un vector donde cada elemento representa los parámetros de la tarea propiamente dicha. En este caso la tarea `TaskConsola` tiene 3 . El primero es un entero que representa la cantidad de llamados bloqueantes que hace (llamémoslo n) , el segundo y tercero ($bmin$ y $bmax$ respectivamente) son los limites de la duración del bloqueo por cada llamada y nos pide que en cada una sea al azar. Para hacer esto utilizamos como parámetro de la función `Uso_IO` (que es la función que hace el bloqueo) la función `rand`. Para usar esta función hay que darle un rango de valores para que elija números al azar. El rango que pusimos fue modulo $(bmax - bmin) + bmin$. Con esto nos aseguramos los tiempos de bloqueo sean entre $bmin$ y $bmax$. Luego `Uso_IO` esta en un ciclo que se ejecuta n veces , es decir la cantidad de llamadas bloqueantes que debe realizar.

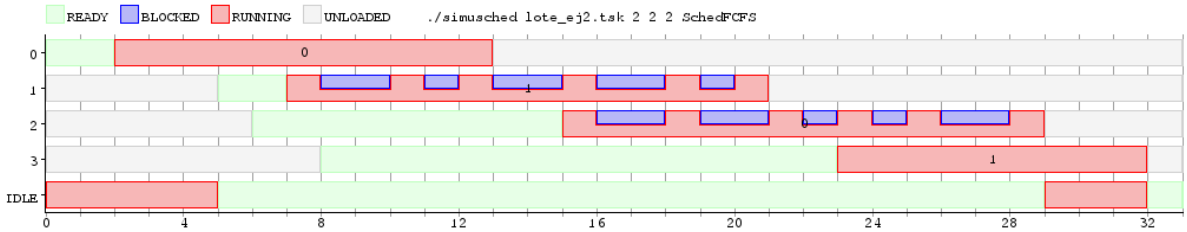
2. Ejercicio 2

ACLARACIÓN: Para calcular el Throughput de los procesos tuvimos en cuenta el ready y se midió sobre el total del tiempo del gráfico. Las pruebas se hicieron en base a `lote_ej2.tsk` Gráfico de Gantt para el scheduler FCFS con 1 núcleo :



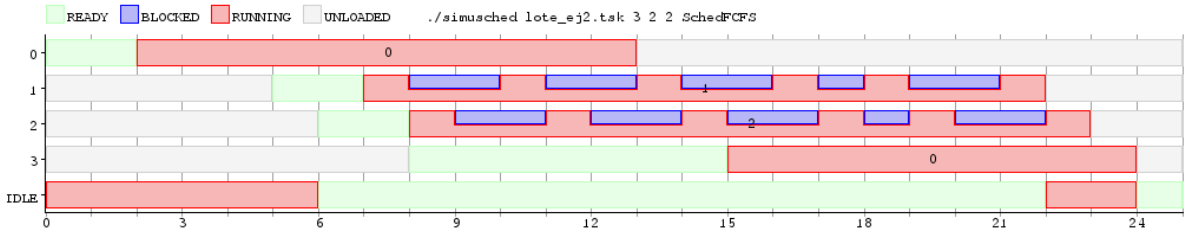
Proceso	Latencia	Waiting Time
0	2	2
1	10	10
2	26	26
3	40	40
Throughput	4/57 = 0.07	

Gráfico de Gantt para el scheduler FCFS con 2 núcleos :



Proceso	Latencia	Waiting Time
0	2	2
1	2	2
2	9	9
3	15	15
Throughput	4/32 = 0.125	

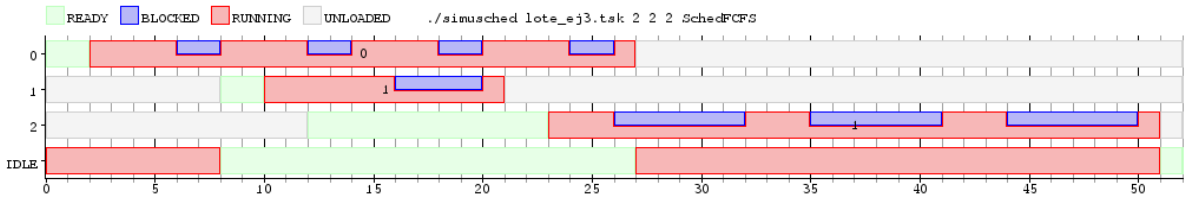
Gráfico de Gantt para el scheduler FCFS con 3 núcleos :



Proceso	Latencia	Waiting Time
0	2	2
1	2	2
2	2	2
3	7	7
Throughput	4/24 = 0.166	

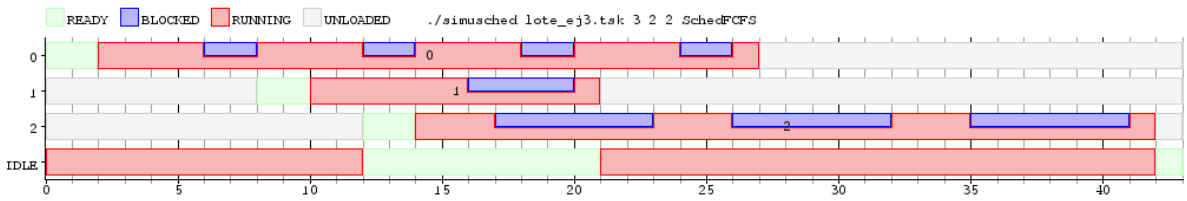
3. Ejercicio 3

En este ejercicio se pidió hacer una tarea. Esta tarea llamada `TaskPajarillo` recibe tres parámetros de entrada. El primero es *cant_repeticiones* que indica la cantidad de veces que debe repetir cierto funcionamiento. El segundo parámetro es *tiempo_cpu* que indica el tiempo de CPU que consume la tarea y por ultimo el tercer parámetro es *tiempo_bloqueo* que indica el tiempo por el cual la tarea va a ser bloqueada. Se utilizaron las funciones `uso_CPU` que como parámetro toma *tiempo_cpu* y luego se utilizo `uso_IO` que toma como parámetro *tiempo_bloqueo* . Dentro de un ciclo que se ejecuta *cant_repeticiones* veces , se realiza primero `uso_CPU` y a continuación `uso_IO`. Para verificar su comportamiento se creo un lote llamado `lote_ej3.tsk` y se lo gráfico con el scheduler FCFS para 2 y 3 núcleos. Veamos los datos que pudimos extraer del siguiente gráfico de Gantt.



Proceso	Latencia	Waiting Time
0	2	2
1	2	2
2	11	11
Promedio	5	5
Throughput	3/51 = 0.05	

Para tres núcleos :



Proceso	Latencia	Waiting Time
0	2	2
1	2	2
2	2	2
Promedio	2	2
Throughput	3/42 = 0.07	

4. Ejercicio 4

Se pidió completar la implementación del scheduler Round Robin.
Siendo n la cantidad de procesadores, utilizaremos las siguientes estructuras de representación:

- q. Una cola global de enteros que representen los procesos en el scheduler.
- $quantum\_cpu$. Un vector de enteros de n posiciones, representando la cantidad parcial de ticks del procesador.
- $end\_quantum$. Un vector de enteros de n posiciones, representando la cantidad de ticks por quantum de cada procesador.

En la cola q almacenamos los `pid` de los procesos de nuestro scheduler. Interactuamos con la misma mediante las operaciones $push$ y pop cada vez que un proceso salía o entraba en ejecución.
El vector $quantum\_cpu$ lo utilizamos para almacenar la cantidad parcial de ticks de cada procesador. Se incrementa cada vez que le llega un tick, y se resetea cada vez que alcanza su quantum.
El otro vector $endquantum\_cpu$ representa el quantum de cada procesador en el cual termina, es decir $quantum\_cpu_i$ guarda el quantum del procesador i pues cada procesador puede tener un quantum distinto.

En el archivo `sched_rr.cpp` se completaron las funciones básicas del scheduler. La función `SchedRR` toma los parámetros del scheduler que son la cantidad de núcleos y sus respectivos quantums. Estos últimos se guardan en el vector $endquantum\_cpu$.

Funciones del scheduler:

- `Load`: simplemente se encola en q el `pid` de la tarea pasada por parámetro.

- `Tick`: La función `tick` es un poco mas compleja.
  1. `EXIT`: Si el motivo pasado por parámetro es un EXIT entonces hay que desencolar de q el primero y poner en $quantum\_cpu_i$ (i es el número del núcleo pasado por parámetro) un 0 pues hay que indicarle que ya terminó su quantum. En el caso de que la cola q estaba vacía solo actualizar $quantum\_cpu_i$ como antes y devolver el `pid` de la idle.
  2. `TICK`: Sin embargo, si el motivo es TICK entonces hay que ver si se llego al quantum del CPU actual para desalojar o no la tarea actual. Además se debe ejecutar las tareas al frente de la cola global en los procesadores que no estén en uso. Por último hay que actualizar el tick CPU actual en $quantum\_cpu$ si la tarea actual sigue corriendo. En el código se puede ver mas en detalle sobre estas decisiones en los comentarios agregados.

Hicimos unos test con el lote `lote_ej3.tsk` y `lote_ej2.tsk`. Haciendo `make ejercicio4` se generaran los graficos de Gantt para esos lotes.

5. Ejercicio 5

En este ejercicio, la consigna consistía en evaluar el funcionamiento del Scheduler Mistery, sin poder ver la implementación del mismo. En cuanto supiéramos como este trabaja, se tendría que implementar un Scheduler NoMistery, el cual tendría que imitar el comportamiento del anterior. Graficamos varias instancias de lotes de tareas usando el simulador, y en base a sus resultados, intentamos hallar como este funciona. Probamos su funcionamiento con el `lote.tsk`. Haciendo `make ejercicio5` se generaran los gráficos de Gantt para los lotes mencionados

El lote pasado por parámetro, tiene cuatro tareas CPU de tiempos: 3,1,1,7,7. Lo primero que nos percatamos es que en cuanto una tarea quiere hace una señal de `exit`, la próxima a ejecutarse es la de menor duración. Y entre dos tareas de igual duración, que ambas estén en ready, no hay criterio para elegir cual ejecutará primero, a excepción de cual se encuentra primero en el lote.tsk (o sea, el menor PID primero). Esto lo probamos también alternando sus tiempos de `load`, y aun así, la primera en ejecutar entre estas dos tareas siempre es la primera en el lote.

Esto elimina nuestra primera suposición de que el Mistery tuviera un criterio de tipo First in-First out, o Last in-Last out. Por ende concluimos que el scheduler Mistery es en esencia un Shortest Job First, uno que deja ejecutar todas las tareas hasta que estas terminen, y al momento de elegir la siguiente, devuelve la que llamara a `exit` en menor tiempo. Para implementarlo, le creamos una cola de procesos al scheduler, a la cual siempre que se haga un llamado a `load`, se agrega el proceso a la cola ordenadamente según su duración pasada por parámetro, y en el caso de encontrar procesos con los que comparta el mismo uso de CPU, el que tiene el mayor pid se almacena más a la izquierda. De esta manera cuando se popean, siempre saldrán en el orden que el scheduler Mistery también cumple.

6. Ejercicio 6

La consigna de este ejercicio pedía hacer una tarea y luego crear un scheduler para ejecutarlas.
La tarea que hay que hacer es `TaskPriorizada` que tiene dos parámetros. El primero es un número que representa la prioridad de la tarea y el segundo es el tiempo de uso del CPU. El código de esta tarea es muy simple, solo utiliza la función `uso_CPU` que toma como parámetro el segundo parámetro de la tarea (tiempo de CPU).
El scheduler que se pidió crear es un *Preemtive Shortest Job First* (PSJF) ya que las tareas tienen una prioridad como parámetro y solo podrán ejecutarse tareas de este tipo.
La clase PSJF tiene la siguiente estructura que refleja el comportamiento de un scheduler *Preemtive Shortest Job First* :

1. Un vector de triplas llamado `queue_prior_time` donde la tripla representa:
    - `pid` de la tarea en la primera coordenada.
    - prioridad de la tarea en la segunda coordenada.
    - tiempo de consumo de CPU de la tarea en la tercer coordenada.

La ventaja que nos da esta estructura es que se pueden utilizar algoritmos de ordenamiento para ordenar por prioridad (segunda coordenada) y en caso de que dos tareas tengan la misma prioridad , se ordenan por tiempo de CPU (tercer coordenada).

Funciones del scheduler:

- `Load`: Esta función se encarga de ordenar cada tarea que llega al scheduler ordenando con el criterio pedido. Se tomó la decisión de ubicar la tarea mas prioritaria al final del vector para aprovechar la función `pop_back`. Esto quiere decir que la tarea al final del vector es la que va a ejecutar el scheduler. Para ordenarlo utilizamos una variante del algoritmo de ordenamiento `bubblesort` haciendo swaps entre las triplas ordenando primero por prioridad y luego por tiempo de CPU.
- `Tick`: Para esta función se separó en varios casos según el motivo m pasado por parámetro que puede ser `EXIT` o `TICK`. No podrá ser `BLOCKED` pues las tareas `TaskPriorizada` no se bloquean.
    1. `EXIT`: En este caso hay que fijarse si $queue\_prior\_time$ esta vacía o no. En caso de estar vacía simplemente devolver el pid de IDLE. Si no esta vacía entonces hay que devolver el pid del ultimo elemento del $queue\_prior\_time$ y sacarlo.
    2. `TICK`: Hay que fijarse primero si la $queue\_prior\_time$ no esta vacía y si hay un CPU libre. En ese caso hay que devolver el pid del ultimo elemento del $queue\_prior\_time$ y sacarlo. En caso contrario hay que devolver el pid de la tarea mas prioritaria. Eso implica seguir ejecutando si no hay una mas prioritaria o desalojarla y ejecutar la correspondiente.

Para probar su funcionamiento utilizamos el lote `lote_ej6.tsk`. Haciendo `make ejercicio6` se ven los gráficos de Gantt de los lotes mencionados.

7. Ejercicio 7

En este ejercicio realizamos comparaciones entre varios tipos de schedulers ($NoMistery$, $PSJF$ y $Round-Robin$) a modo de experimentación con el fin de mostrar los distintos rendimientos de cada sistema. También se generaron sets de prueba para mostrar ventajas y desventajas de los distintos schedulers frente a la misma entrada.

7.1. Lote 1:

Para el primer experimento cargamos varias tareas al mismo tiempo con distintos tiempos de `CPU` (en un orden creciente), todas van a tener la misma prioridad. Nuestra hipótesis es que los schedulers NoMistery y PSJF van a comportarse de manera similar. En cambio en el RR vamos a notar un menor tiempo de latencia, mayor tiempo de waiting time y menor throughput (**latencia y waiting time en tiempo promedio**). El motivo de este comportamiento se debe al costo que se paga en cada desalojo del scheduler. Veamos si se cumple nuestra hipótesis con la siguiente tabla de datos que obtuvimos del diagrama de Gantt corriendo el lote `lote_ej7_t1.tsk` para el RR y NoMistery y `lote_ej7_t1_prior.tsk` para el PSJF.

	NoMistery	PSJF	RR
Avg. Latencia	12.8	12.8	11.4
Avg. Waiting	12.8	12.8	17.6
Throughput	5/38 = 0.131	5/38 = 0.131	5/41 = 0.123

Los datos de la tabla de arriba comprueban nuestra hipótesis. Esto se debe a que el RR va alternando entre las tareas ejecutadas de una manera mas ”justa”, es decir, intenta repartir el uso del procesador de manera mas equitativa y podemos observar que las tareas tienen menos tiempo de espera entre que llegan al estado READY, y luego pasan a RUNNING.
En cuanto a las similitudes entre NoMistery y PSJF, podemos confirmar que actúan de manera idéntica, lo cual es lógico considerando que uno es un scheduler de tipo SJF, y el otro es un PSJF donde las prioridades son todas iguales. También se puede ver como los tiempos de cambio de contexto, perjudicaron al RR obteniendo el peor throughput y waiting time de los 3 schedulers.
Luego probamos los mismos lotes pero ejecutando los schedulers con 2 cores. Nuestra hipótesis es que

la latencia va a bajar en todos los schedulers (el RR seguirá teniendo la menor) asi como también el waiting time. Sin embargo,el RR tendrá el mayor Throughput en comparación con los otros dos. Veamos la siguiente tabla para verificar estas conjeturas.

	NoMistery	PSJF	RR
Avg. Latencia	5.4	5.4	5
Avg. Waiting	5.4	5.4	6.2
Throughput	5/23 = 0.217	5/23 = 0.217	5/21 = 0.238

Los datos de la tabla de arriba verifican nuestras hipótesis. Efectivamente en general se redujeron todas las mediciones, y también el RR tuvo el mejor throughput de los tres, a pesar de haber sido el peor en la evaluación con un procesador. Esto indicaría que a pesar de los cambios de contexto extra que realiza, aun así demora menos en realizar la misma cantidad de tareas. Esto ocurre ya que el conjunto de tareas tienen la suma de sus duraciones mejor repartida entre ambos procesadores, mientras que tanto para el NoMistery y el PSJF, al ser schedulers que postergan la ejecución de las tareas mas costosas, estas se ejecutan mientras no queda mas trabajo para hacer en el segundo procesador. Esta mala repartición de los costos entre procesadores causa que el RR se torne más efectivo en cuanto a terminar un número de tareas en menor tiempo.

7.2. Lote 2:

El propósito de este test era evaluar las diferencias del funcionamiento entre los schedulers, dado el caso que tenemos lotes en los cuales los tiempos de llegada de las tareas van variando. Esto es porque a diferencia del NoMistery y el RR, el PSJF desaloja tareas cuando se agrega una nueva con menor costo de procesamiento. Tomamos como hipótesis que seria una competencia más justa entre el RR y el PSJF, corriendo con un solo procesador, ya que este último tendría más cambios de contexto que en el lote anterior.
Como el lote inserta procesos de bajo costo a medida que se ejecuta la simulación, tendremos la certeza de que el PSJF tendrá el menor tiempo promedio de latencia, ya que las tareas menos costosas, son inmediatamente ejecutadas en cuanto se cargan. También podemos confirmar observaciones del lote anterior, como el RR teniendo el peor throughput y waiting time.
Aun conservaremos las prioridades de las tareas del PSFJ iguales, ya que en este caso no es algo que pretendemos evaluar. Veamos si se cumple nuestra hipótesis con la siguiente tabla de datos que obtuvimos del diagrama de Gantt ejecutando el lote `lote_ej7_t2.tsk` para el RR y NoMistery y `lote_ej7_t2_prior.tsk` para el PSJF.

	NoMistery	PSJF	RR
Avg. Latencia	14.6	5.8	10.8
Avg. Waiting	14.6	14.8	25.8
Throughput	5/46 = 0.109	5/48 = 0.104	5/51 = 0.099

Tal como se supuso, la latencia del PSJF fue mucho menor al resto de los schedulers. Además, tanto en throughput como waiting time, el RR obtuvo los peores resultados en ambas mediciones. El PSJF tuvo apenas más waiting time que el NoMistery, gracias a los cambios adicionales de contexto que tuvo por los desalojos al llegar tareas con menor uso del CPU.
Luego probamos los mismos lotes pero corriendo los schedulers con 2 cores. Podemos suponer con seguridad que la latencia se reducirá en los tres casos, y también que el PSJF conservará el primer puesto en cuanto al tiempo de latencia. Aun así, es probable que el RR tenga el mejor throughput, ya que es posible que el PSJF al final de la simulación, mantenga un procesador ejecutando la tarea de mayor costo, mientras que el otro se mantiene en idle, algo que no mostraría una buena repartición del costo total de todos los procesos entre ambos procesadores.

	NoMistery	PSJF	RR
Avg. Latencia	4.2	1.4	3.4
Avg. Waiting	4.2	5.4	6.6
Throughput	5/28 = 0.178	5/29 = 0.172	5/28 = 0.178

Efectivamente, la latencia se redujo para los tres schedulers, siendo el PSJF todavía el mejor de los tres, y el RR siguiéndole, al igual que los resultados obtenidos para la prueba con un core. De todas formas, la hipótesis de que el RR superaría al PSJF respecto al throughput medido, no se ve realmente confirmada, ya que a pesar de haber obtenido un mejor resultado, esta diferencia es apenas notable. Además, el NoMistery, aun siendo un SJF, obtuvo un throughput idéntico al RR, lo cual desmiente la suposición nuestra de que para el lote usado, sería mejor ir alternando de proceso en proceso, desalojándolos según un quantum.