DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico I

Scheduling

Sistemas Operativos

Integrante	LU	Correo electrónico
Cámera, Joel Esteban	257/14	joel.e.camera@gmail.com
Lavia, Alejandro Norberto	43/11	lavia.alejandro@gmail.com
Guttman, Martín David	686/14	

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Ejercicio 1

1.1. Enunciado

Programar un tipo de tarea **TaskConsola**, que simulará una tarea interactiva. La tarea debe realizar *n* llamadas bloqueantes, cada una de una duración al azar entre *bmin* y *bmax* (inclusive). Luego ejecutará 1 tic cada vez. La tarea debe recibir tres parámetros: *n*, *bmin* y *bmax* (en ese orden) que serán interpretados como los tres elementos del vector de enteros que recibe la función. Explique la implementación realizada y grafique un lote que utilice el nuevo tipo de tarea.

1.2. Resolución

Para el ejercicio programamos la tarea TaskConsola de la siguiente forma:

```
void TaskConsola(int pid, vector<int> params) {
        // params: n bmin bmax

        // Entero que va a tomar el valor aleatorio entre bmin y bmax
        int random_num;

        // Pasamos los valores de params a variables mas declarativas
        int n = params[0];
        int bmin = params[1];
        int bmax = params[2];

        // Iteramos n veces
        for(int i = 0; i < n; i++){

                // Generamos una duracion al azar para la llamada bloqueante
                random_num = bmin + rand() %(bmax - bmin + 1);

                // Se realiza la llamada bloqueante con la duracion al azar
                uso_IO(pid, random_num);

                // Ejecutamos 1 clock
                uso_CPU(pid, 1);
        }
        return;
}
```

Primero pasamos los valores del vector **params** a variables para que sea mas declarativo el código. Luego iteramos n veces donde cada vez se genera un número al azar entre bmin y bmax y, con ese valor realizamos la llamada bloqueante mediante la función **uso_IO**. Luego ejecutamos un clock con la funcion **uso_CPU**.

Generamos y ejecutamos el siguiente lote, y luego graficamos la simulación usando el algoritmo FCFS con un costo de 3 clocks para cambiar de contexto.

```
TaskConsola 10 5 25
TaskConsola 3 12 17
@3:
TaskConsola 6 1 4
@6
TaskConsola 4 19 30
```

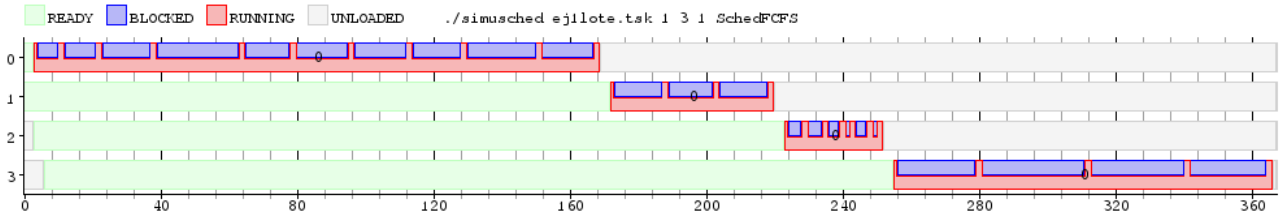

Gráfico generado con el lote.

2. Ejercicio 2

2.1. Enunciado

Ejecutar y graficar la simulación usando el algoritmo **FCFS** para 1 y 2 y 4 núcleos con un cambio de contexto de 2 ciclos. Calcular la *latencia* de cada tarea en los tres gráficos y el *throughput*.

TaskCPU 10
@5:
TaskConsola 5 1 4
@6:
TaskConsola 5 1 2
@8:
TaskCPU 10

2.2. Resolución

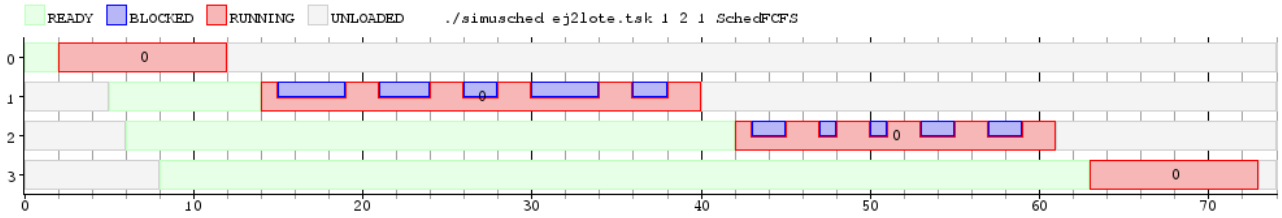

Grafico generado con el lote usando el algoritmo FCFS para 1 núcleo con un cambio de contexto de 2 ciclos.

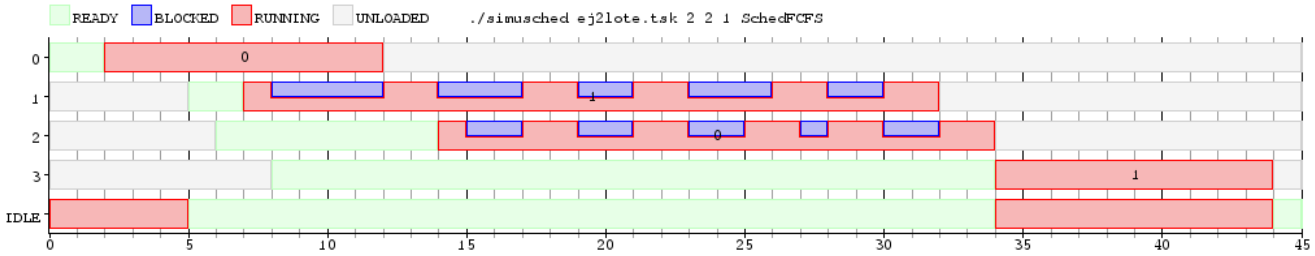

Grafico generado con el lote usando el algoritmo FCFS para 2 núcleos con un cambio de contexto de 2 ciclos.

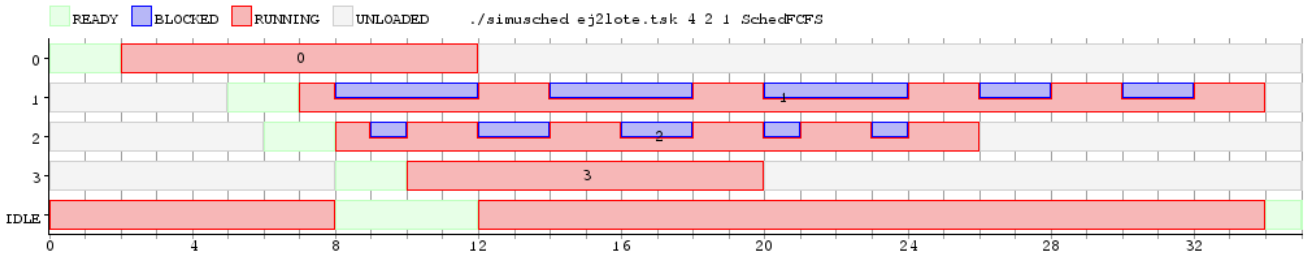

Grafico generado con el lote usando el algoritmo FCFS para 4 núcleos con un cambio de contexto de 2 ciclos.

3. Ejercicio 3

3.1. Enunciado

Programar un tipo de tarea **TaskBatch** que reciba dos parámetros: *total_cpu* y *cant_bloqueos*. Una tarea de este tipo deberá realizar *cant_bloqueos* llamadas bloqueantes, en momentos elegidos seudoaleatoriamente. En cada tal ocasión, la tarea deberá permanecer bloqueada durante exactamente dos (2) ciclos de reloj. El tiempo de CPU total que utilice una tarea **TaskBatch** deberá ser de *total_cpu* ciclos de reloj (incluyendo el tiempo utilizado para lanzar las llamadas bloqueantes; no así el tiempo en que la tarea permanezca bloqueada). Explique la implementación realizada y grafique un lote que utilice 4 tareas **TaskBatch** con parámetros diferentes y que corra con el scheduler **FCFS**.

HINT: Piense en distribuir adecuadamente esas llamadas bloqueantes. Algunos suelen llamarlo *shuffle*.

3.2. Resolución

Para el ejercicio programamos la tarea TaskConsola de la siguiente forma:

```
void TaskBatch(int pid, vector<int> params) {
        // params: total_cpu cant_bloqueos

        // Entero que utlizamos para generar el valor pseudoaleatorio
        int random_num;

        // Pasamos los valores de params a variables mas declarativas
        int total_cpu = params[0];
        int cant_bloqueos = params[1];

        // El total del cpu que se utilice va a ser total_cpu - cant_bloqueos (ya que por
        // cada uno de ellos se utiliza 1 ciclo del cpu) y - 1 ya que return utiliza
        // un ciclo de reloj.
        int cpu_restante = total_cpu - cant_bloqueos - 1;

        // Ciclo mientras todavia haya bloqueos para realizar
        while(cant_bloqueos > 0){
                // Genero un numero aleatorio entre 0 y cpu_restante
                random_num = rand() %(cpu_restante + 1);

                // Si el numero aleatorio es mayor que cero, uso el CPU
                // random_num ciclos.
                if(random_num > 0) uso_CPU(pid, random_num);

                // Descuento random_num ciclos a cpu_restante
                cpu_restante = cpu_restante - random_num;

                // Se realiza la llamada bloqueante
                uso_IO(pid, 2);
```

```
                cant_bloqueos--;
        }
        return;
}
```

Cómo debemos realizar *cant_bloqueos* llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente, lo que hacemos con este código es utlizar el CPU con una cantidad aleatoria de ciclos (entre 0 y *cpu_restante*) y luego se realiza la llamada bloqueante. Esto genera que las llamadas bloqueantes se realicen efectivamente en momentos pseudoaleatoriamente.

La variable *cpu_restante* se genera restandole a *total_cpu* la cantidad de bloqueos más uno. Esto es porque cuando se llama a la función **uso_IO** se utiliza un clock de CPU y el uno restante es porque la función **return** también utiliza un clock de CPU.

Generamos y ejecutamos el siguiente lote, y luego graficamos la simulación usando el algoritmo FCFS con un costo de 3 clocks para cambiar de contexto.

```
TaskBatch 23 1
@5:
TaskBatch 42 2
@6:
TaskBatch 58 3
@8:
TaskBatch 77 4

TaskBatch 33 12
@2:
TaskBatch 50 5
@3:
TaskBatch 70 9
@2:
TaskBatch 27 3
```

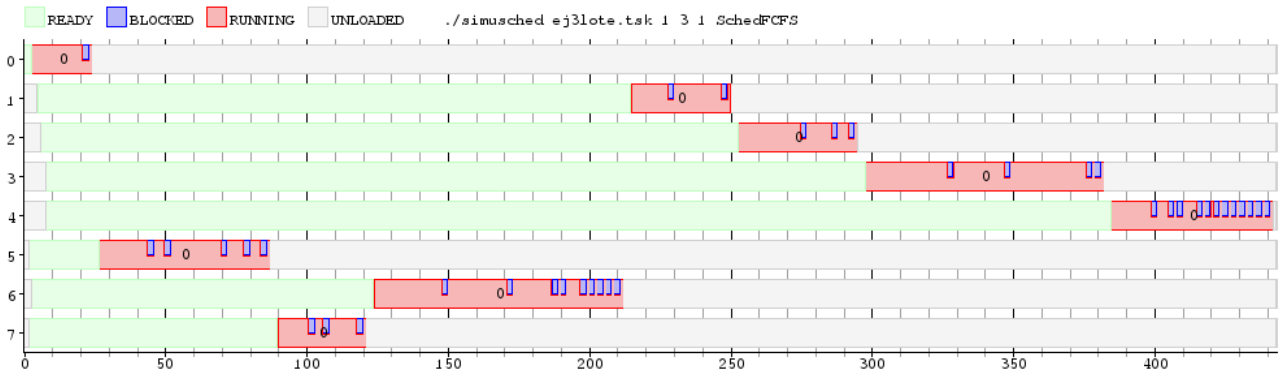

Gráfico generado con el lote.

4. Ejercicio 4

4.1. Enunciado

Completar la implementación del scheduler Round-Robin implementando los métodos de la clase **SchedRR** en los archivos **sched_rr.cpp** y **sched_rr.h**. La implementación recibe como primer parámetro la cantidad de núcleos y a continuación los valores de sus respectivos quantums. Debe utilizar una única cola global, permitiendo así la migración de procesos entre núcleos.

4.2. Resolución

5. Ejercicio 5

5.1. Enunciado

Dado el lote de tareas del ejercicio 2. Ejecutar y graficar la simulación utilizando el scheduler *Round-Robin* con quantum 2, 5 y 10. Con un cambio de contexto de 2 ciclos y un sólo núcleo calcular la *latencia*, el *waiting time* y el tiempo total de ejecución de las tareas para cada quantum.

5.2. Resolución

6. Ejercicio 6

6.1. Enunciado

A partir del artículo:

- Jia Ru and Jack Keung. *An Empirical Investigation on the Simulation of Priority and Shortest-Job-First Scheduling for Cloud-based Software Systems*, 2013 22nd Australian Conference on Software Engineering

Responda:

a) ¿Qué problema están intentando resolver los autores?

b) ¿Qué algoritmo generan para resolverlo?

c) Explicar brevemente las pruebas realizadas

6.2. Resolución

7. Ejercicio 7

7.1. Enunciado

En base a lo anterior, defina dos nuevos algoritmos de scheduling. Ambos del tipo *shortest job first*. Para ambos puede considerar que solamente correran tareas de tipo TaskCPU.

a) Uno no reentrante. Tomará como parámetros la cantidad de procesadores y el tiempo total de ejecución de cada tarea en el lote. Ejecutará la tarea que menos tiempo de cpu necesite. Al terminar de ejecutarla, volverá a elegir la de menor tiempo de ejecución. Será llamado **SJF**.

b) Uno reentrante. Tomará como parámetros la cantidad de procesadores, los **Quantums** de cada uno de ellos y el tiempo total de ejecución de cada tarea en el lote. Un proceso correrá en ese procesador durante ese quantum. Al terminar dicho tiempo, ejecutará la tarea a la que menos tiempo le quede de ejecución (podría seguir ejecutando la tarea actual). Existe una única cola de procesos para todos los procesadores. Lo llamaremos **RSJF**.

7.2. Resolución

8. Ejercicio 8

8.1. Enunciado

Resuelva:

a) Realice tareas de prueba para comparar los schedulers **Round Robin**, **FIFO**, **SJF**, **RSJF**.

b) Compárelas según la **latencia**, **waiting time**, **turnaround**.
Realice gráficos y tablas comparativas para exponer los resultados obtenidos.

c) Escriba una breve conclusión.

8.2. Resolución

9. Ejercicio 9

9.1. Enunciado

Completar la implementación del scheduler *Multilevel Feedback Queue* implementando los métodos de la clase **SchedMFQ** en los archivos **sched_mfq.cpp** y **sched_mfq.h**. La implementación debe utilizar *n* colas con *Round-Robin* en cada una con los parámetros que se detallan a continuación.

- Las n colas se numeran de 0 a $n-1$ siendo 0 la de mayor prioridad.
- El constructor recibe como parámetro n números q_i indicando el *quantum* de la cola i.
- Al iniciar una tarea comienza al final de la cola de mayor prioridad.
- Siempre se ejecuta la primer tarea de la cola no vacía de mayor prioridad. Si esta tarea consume todo su *quantum* sin bloquearse entonces pasa al final de la cola inmediatamente de inferior prioridad (si hay). Si esta tarea se bloquea antes de agotar su *quantum*, entonces (cuando se desbloquee)
- Si todas las colas están vacías se ejecuta **IDLE TASK**.

Realice pruebas y muestre su ejecución.

9.2. Resolución