DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I - Scheduling

16 / 9 / 2014 Sistemas Operativos

Integrante	LU	Correo electrónico
Straminsky, Axel	769/11	axelstraminsky@gmail.com
Chapresto, Matias		matiaschapresto@gmail.com
Torres, Sebastian	723/06	sebatorres1987@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. Parte I: Entendiendo el simulador *simusched*

1.1. Introducción

El objetivo de esta parte es familiarizarse con el simulador *simusched*, el cual sirve para ver el comportamiento de distintos lotes de procesos bajo distintas políticas de scheduling. Adicionalmente se puede especificar la cantidad de cores a disposición de los procesos, y los costos de ciertas acciones como hacer un cambio de contexto, o cambiar un proceso para que se ejecute en otro core.

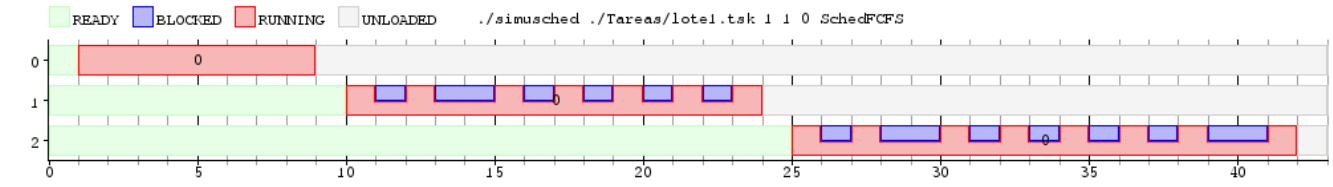
1.2. Ejercicio 1

El objetivo de este ejercicio es implementar una tarea de tipo **TaskConsola**, la cual debe simular ser una tarea interactiva. Para esto, la tarea realiza n llamadas bloqueantes, cada una con una duración al azar entre *bmin* y *bmax*, ambos especificados por parámetro. La implementación de esta función es bastante directa, y básicamente consiste en inicializar el generador de números aleatorios con el parámetro *time(NULL)*, es decir, con la fecha actual al momento de ejecutarse la función. Luego, se realizan n llamadas bloqueantes con una duración al azar entre *bmin* y *bmax*, mediante la fórmula $(rand()\,\%(bmax - bmin)) + bmin$, utilizando la función `uso_IO`. Para más detalles, consultar la implementación en el archivo *tasks.cpp*.

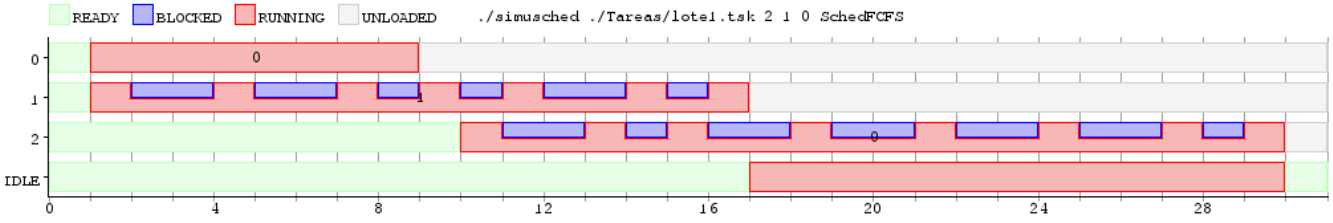
1.3. Ejercicio 2

El objetivo de este ejercicio es ejecutar un lote de tareas, una intensiva en CPU y las otras 2 de tipo interactivo (**TaskConsola**), con la política de scheduling **FCFS**, y observar y graficar los resultados, variando la cantidad de cores, con un costo de cambio de contexto igual a 1, y costo de migración igual a 0.

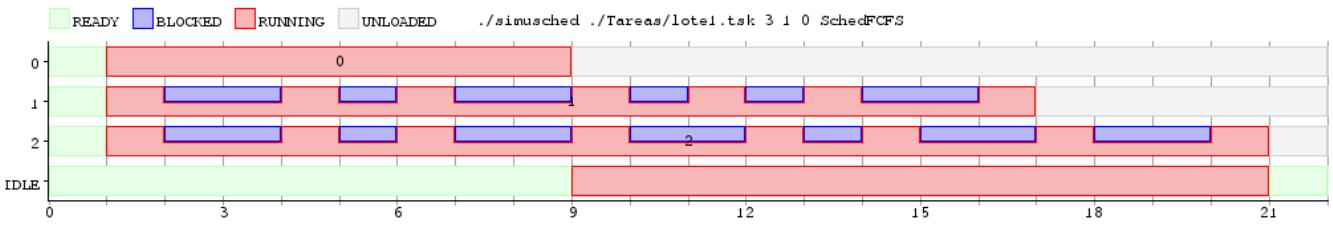
El lote de tarea que utilizamos es el *lote1.tsk*. A continuación se pueden ver los gráficos:



1 core.



2 cores.



3 core.

2. Parte II: Extendiendo el simulador con nuevos schedulers

2.1. Introducción

En esta sección se extiende el simulador con un nuevo algoritmo de scheduling, *Round Robin*, y se lo testea con diversos lotes de tareas.

2.2. Ejercicio 3

El objetivo de este ejercicio es implementar la polÃtica de scheduling *Round Robin*. La función más importante es `tick(cpu, motivo)`, cuya implementación se describe a continuación: si el motivo es **TICK** o **BLOCK**, se aumenta el contador de ticks del core correspondiente. Si este contador supera el quantum del core, se vuelve a poner el contador en 0, se encola la tarea actual y comienza a ejecutarse la siguiente tarea en la cola; caso contrario, se sigue ejecutando la tarea actual.

Si el motivo es **EXIT**, sencillamente se devuelve la proxima tarea en la cola, sin encolar nuevamente la tarea actual. En caso de no haber mÃ¡s tareas, se ejecuta **IDLE_TASK**.

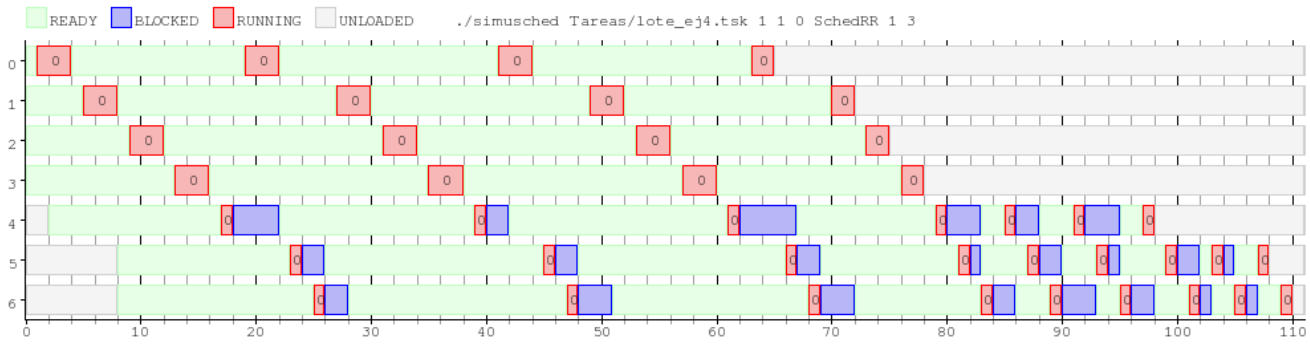
Para mÃ¡s detalles, consultar la implementación en el archivo *sched_rr.cpp*.

2.3. Ejercicio 4

En esta parte nos vamos a focalizar en la utilización de la política de scheduling implementada anteriormente para mostrar cómo se comporta la misma con diversos quantums y cantidad de cores. La intención es mostrar cuan eficiente o ineficiente puede ser una misma política tan solo con variar el quantum y mostrar que la elección del mismo es muy importante.

Para tal motivo vamos a usar el lote *lote_ej4.tsk* que contiene 4 tareas intensas en CPU y 3 interactivas. Las de CPU llegan en el momento 0, luego una interactiva en el momento 2 y otras dos mas en el momento 8.

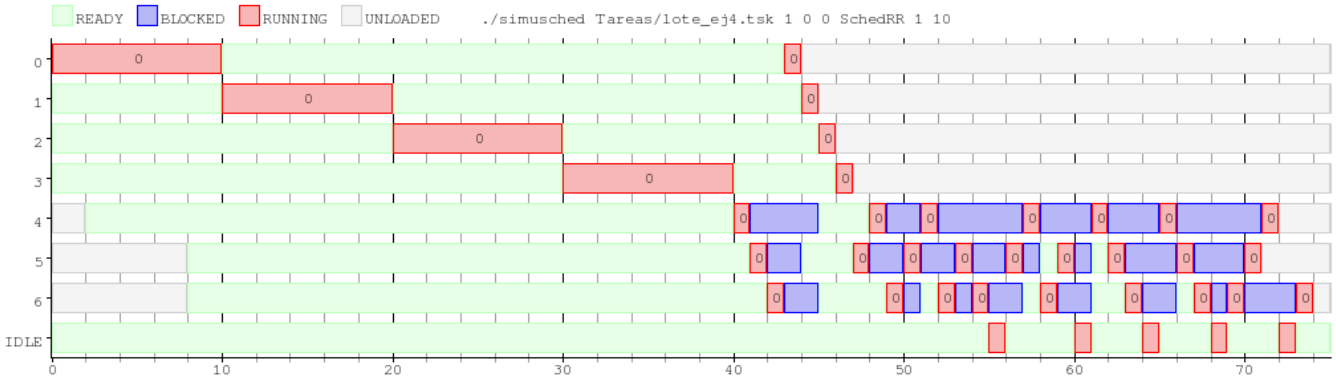
Como primer política vamos a tomar un RR con quantum 3, costo de cambio de contexto de 1 y un sólo un core. El procesamiento de dicho lote arroja el siguiente gráfico:



Cores = 1, Quantum = 3, CS = 1.

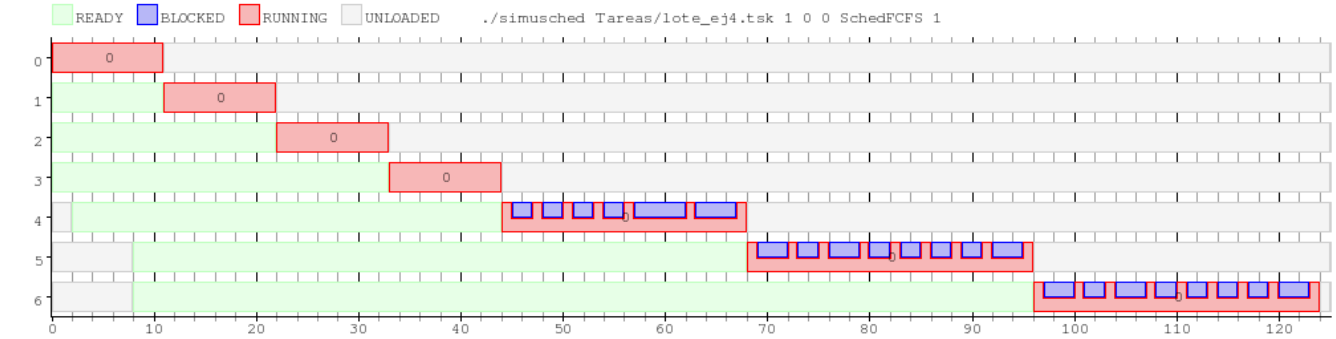
Como podemos apreciar, se ejecutan en una primera pasada las tareas de cpu (de la 0 a la 3 inclusive) que no tienen bloqueos. Luego se ejecuta la tarea 4, que llegó en el momento 2. Luego en lugar de ejecutarse la 5 o la 6 (que llegaron en el momento 8) vuelve a ejecutarse la tarea 0, luego la 1 y recién luego de esta se ejecuta la número 5. Esto ocurre porque cuando termina de ejecutarse por primera vez la tarea 0 vuelve a ser encolada porque su quantum terminó y todavía queda más por procesar. Cuando esto ocurre, todavía las tareas 5 y 6 no llegaron a la cola. Esto sucede también con la tarea 1. A mitad de la primer ejecución de la tarea 2 es cuando llegan las tareas 5 y 6 y por eso en la segunda corrida (momento 13) las tareas 5 y 6 se ejecutan después de las 0 y 1. Conforme pasan los ticks vemos como el gráfico va siguiendo un cierto patrón de tipo escalera, característico de Round Robin, ya que usa un esquema de despacho de tareas circular.

El siguiente experimento que queremos mostrar es qué sucede cuando el quantum de un esquema de RR es muy grande. A priori un quantum muy grande no sería muy conveniente ya que se estaría aproximando a un esquema de FCFS (en el caso de tareas intensas en CPU). A continuación vemos un expermiento con un quantum de 10.



Cores = 1, Quantum = 10, CS = 0.

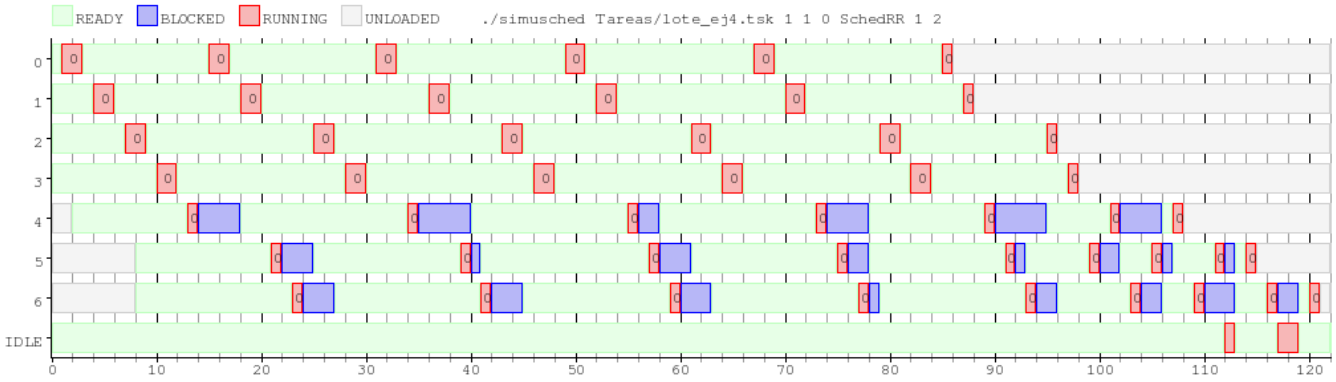
Y a continuación el mismo lote de tareas procesado con FCFS.



Cores = 1, Quantum = 10, CS = 0.

Como se puede apreciar, un esquema de RR con un quantum alto tiende a comportarse de una manera similar al FCFS con la importante salvedad de que RR es *starvation-free*, debido a que el quantum puede ser muy grande pero es finito.

Por otro lado, un quantum muy pequeño tampoco es bueno. Esto se debe a que, si asumimos un costo de cambio de contexto mayor a 0, se va a desperdiciar mucho tiempo en cambiar de contexto y eso entorpecerÃa el rendimiento general del sistema.



Cores = 1, Quantum = 2, CS = 1.

Si comparamos este último gráfico con el primero de la sección vemos que todas las tareas tardan más en finalizar su ejecución y la cantidad de tiempo en cambios de contexto es mayor, lo cual se traduce en un menor rendimiento.

3. Parte III: Evaluando los algoritmos de scheduling

3.1. Introducción

En esta sección se evaluan las políticas de scheduling implementadas, utilizando diversas métricas especificadas más adelante.

3.2. Ejercicio 6

El objetivo de este ejercicio es programar un tipo de tarea **TaskBatch**, que durante *total_cpu* ciclos, realize *cant_bloqueos* llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente. La implementación es bastante directa, con la semilla del generador de números pseudoaleatorios inicializada con la fecha del sistema al momento de ejecutar la función. Las llamadas bloqueantes se lanzan si *rand()* devuelve un número impar. Para asegurarnos de que se ejecuten las *cant_bloqueos* llamadas bloqueantes antes de que se terminen los *total_cpu* ciclos, creamos procesos cuya cantidad de llamadas bloqueantes fuese menor o igual a la mitad de los ciclos totales de CPU.

Para más detalles, consultar la implementación en *tasks.cpp*.

3.3. Ejercicio 7

En este ejercicio debemos elegir 2 métricas diferentes y testear un lote de tareas **TaskBatch**, todas ellas con igual uso de CPU pero con diversas cantidades de bloqueos. El lote de tareas utilizado es el *lote3.tsk*.

Las métricas que elegimos fueron:

- Turnaround
- Waiting Time

Definidas en [Sil1] como:

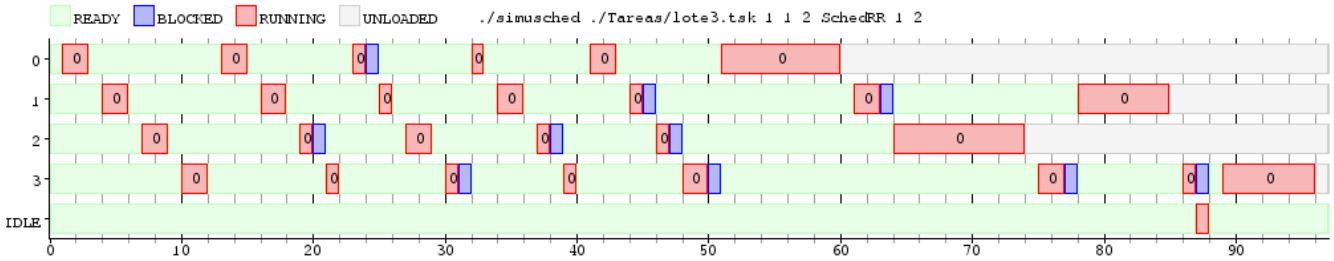
Turnaround: Es el intervalo de tiempo entre el momento en que el proceso comienza a ejecutarse por primera vez, hasta el momento en que el mismo termina. Es decir, es la suma de los períodos usados en esperar datos de memoria, en estar encolado en la "ready queue", ejecutandose en la CPU, y haciendo E/S.

Waiting Time: Es el tiempo que un proceso se pasa encolado en la "ready queue".

Elegimos estas métricas ya que, con el Turnaround, tenemos una visión global de cómo se comportan los procesos, mientras que con el Waiting Time podemos observar cómo un hecho más puntual, el tiempo que los procesos pasan encolados, impacta en el tiempo de ejecución total del proceso.

Para calcular el desvío standard utilizamos la fórmula de [WikSD]

A continuación se pueden observar los resultados de la experimentación:



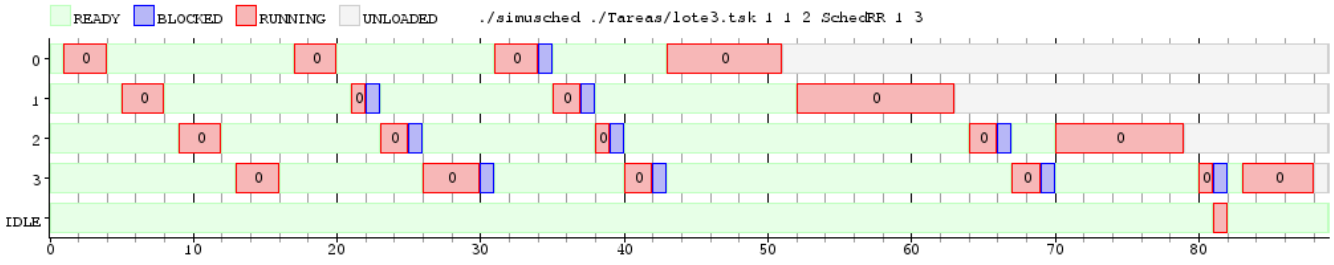
1 core, quantum = 2.

Turnaround:
- P_0: 59
- P_1: 81
- P_2: 67
- P_3: 86
- Promedio: 73,25

Waiting Time:
- P_0: 42
- P_1: 67
- P_2: 55
- P_3: 75
- Promedio: 59,75

DS: 10,78

DS: 12,47



1 core, quantum = 3.

Turnaround:
- P_0:
- P_1:
- P_2:
- P_3:
- Promedio:
- DS:

Waiting Time:
- P_0:
- P_1:
- P_2:
- P_3:
- Promedio:
- DS:

3.4. Ejercicio 8

3.5. Ejercicio 9

3.6. Ejercicio 10

4. Referencias

[Sil1] A. Silberschatz, *Operating System Concepts, 4º Ed., 1994, págs 135-136.*

[WikSD] http://en.wikipedia.org/wiki/Standard_deviation#Basic_examples