Trabajo Práctico de Sistemas Operativos

16 de septiembre de 2014

Universidad de Buenos Aires - Departamento de Computación - FCEN

Integrantes:

- Castro, Damián L.U.: 326/11 ltdicai@gmail.com
- Toffoletti, Luis L.U.: 827/11 luis.toffoletti@gmail.com
- Zanollo, Florencia L.U.: 934/11 florenciazanollo@gmail.com

Índice

1. Introducción

Una computadora moderna consiste de uno o más procesadores, memoria, discos, impresoras y uno o varios dispositivos de entrada/salida. Además una computadora ejecuta programas que suelen intentar acceder a estos recursos y generalmente asumiendo que son los únicos que desean utilizarlos. Sin embargo esto casi nunca es cierto y suelen haber conflictos cuando dos procesos (programas en ejecución) acceden al mismo recurso. Para solucionar estas problemáticas se recurre a los sistemas operativos, que son aquellos que se encargan de coordinar el acceso a los recursos por parte de los procesos. Un sistema operativo es un proceso superior a todos los demás y decide, siguiendo alguna normativa, en que momentos los procesos comunes se ejecutan. Se conoce a esta normativa como política de *scheduling*, y no hay una única manera de definirla. Dependiendo de la situación puede que se prefiera políticas que permitan a los procesos mantener ocupado un recurso durante períodos largos de tiempo mientras que otras políticas se valen en desalojar a las tareas y otorgarle los recursos a otra tarea. Se espera que un *scheduler* trate de cumplir los siguientes objetivos:

- Ecuanimidad (*fairness*): que cada proceso reciba una dosis "justa" de CPU (para alguna definición de justicia).
- Eficiencia: tratar de que la CPU esté ocupada todo el tiempo.
- Carga del sistema: minimizar la cantidad de procesos listos que están esperando CPU.
- Tiempo de respuesta: minimizar el tiempo de respuesta percibido por los usuarios interactivos.
- Latencia: minimizar el tiempo requerido para que un proceso empiece a dar resultados.
- Tiempo de ejecución: minimizar el tiempo total que le toma a un proceso ejecutar completamente.
- Rendimiento (throughput): maximizar el número de procesos terminados por unidad de tiempo.
- Liberación de recursos: hacer que terminen cuanto antes los procesos que tiene reservados más recursos.

Como se puede intuir, es imposible cumplir todos los objetivos a la vez, por lo tanto las políticas de *scheduling* tienen sus ventajas y sus desventajas. Es el enfoque de este informe exhibir algunas de las políticas ms conocidas, su implementación en lenguaje *C++* y mostraremos sus pros y sus contras con ejemplos y incluiremos gráficos para ayudar a la comprensión.

1.1. Simulador

Para analizar las diferentes políticas de *scheduling* utilizamos un modelo de computadora simplificado que consiste en los siguientes componentes:

- Un lote de programas a ejecutar.
- Uno o varios núcleos (*cores*) de procesamiento, que se encargarán de hacer los cómputos de los procesos.
- Un único dispositivo de entrada y salida que todos los procesos pueden utilizar si lo desean.

Además contamos con un simulador de ese modelo, que nos permite ejecutar programas o tareas de acuerdo a un *scheduler* dado. El mismo se encarga de imprimir una secuencia de eventos resultantes de la ejecución.

1.1.1. Acciones del simulador

El simulador provee a los procesos con tres acciones posibles:

- $uso\_CPU(n)$ que indica que el proceso utilizará el CPU durante n ciclos de reloj.
- $uso\_IO(n)$ que indica que el proceso utilizará un recurso de entrada/salida durante n ciclos de reloj. Durante este tiempo el proceso no utiliza al procesador, así que el *scheduler* es libre de utilizar ese tiempo en otra tarea.
- *return* que indica que el proceso ya terminó su ejecución. El *scheduler* debe realizar las acciones necesarias para desalojar al proceso.

1.1.2. Definición de *schedulers*

Un *scheduler* válido ofrece las siguientes funcionalidades:

- `load(pid)`: Carga un proceso con número de identificación `pid`.
- `unblock(pid)`: Se realiza cuando un proceso con número de identificación `pid` deja de utilizar un recurso de entrada/salida y desea volver a usar el procesador.
- `tick(cpu,motivo)`: Se ejecuta por cada *tick* del reloj del procesador `cpu` y el `motivo` indica que sucedió con el último proceso que estaba en ejecución. Un `motivo` puede valer tres cosas:
  - `TICK`: indica que el proceso consumió todo el ciclo.
  - `BLOCK`: indica que el proceso pidió acceso a un recurso de entrada y salida y por este motivo la tarea fue bloqueada.
  - `EXIT`: indica que la tarea terminó de ejecutarse.

2. Tareas

2.1. TaskConsola

Algorithm 1 TaskConsola

```
1: procedure TaskConsola(cant_bloqueos, bmin, bmax)
2:     seteo semilla de random
3:     for i ← 0, cant_bloqueos do
4:         random_number ← modulo(rand(), bmax − bmin + 1) + bmin        ▷ [1]
5:         Uso_IO(random_number)
6:     end for
7: end procedure
```

[1]: rand() retorna un número arbitrariamente grande. Tomando su módulo en (bmax - bmin + 1) nos aseguramos que esté entre 0 y (bmax - bmin).
Luego sumamos bmin, resultando un número entre bmin y bmax.

3. Políticas de *scheduling*

3.1. First-Come First-Served (FCFS)

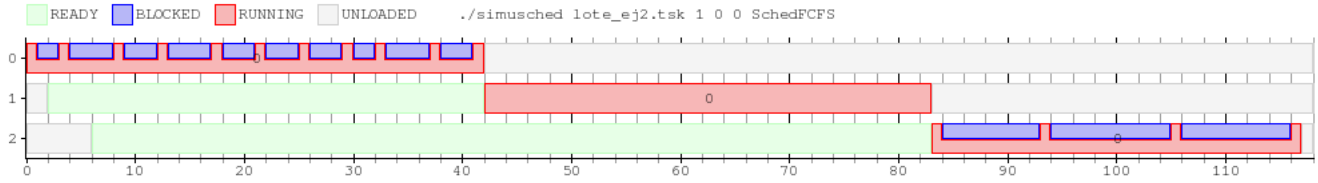
FCFS (First-Come First-Served) un scheduler simple en el cual los procesos son asignados al CPU en el orden en que estos lo requieren.

```
TaskConsola 10 2 4
@2:
TaskCPU 40
@6:
TaskConsola 3 6 11
```

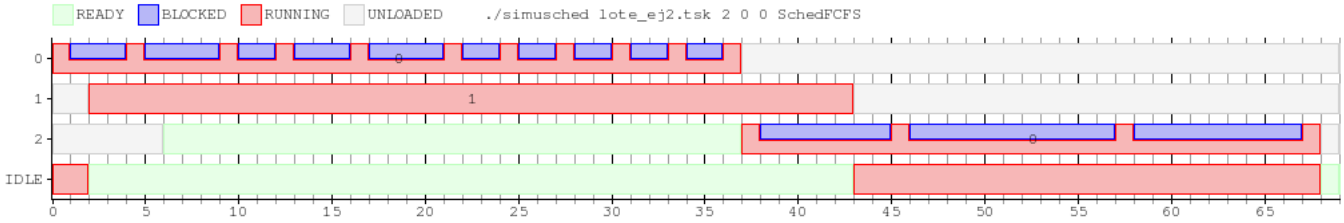
Básicamente hay una sola cola (FIFO) de procesos 'Ready'. Cuando un proceso requiere CPU y éste está libre, se lo deja correr tanto tiempo como quiera y sin interrupciones.

A continuación se muestran los gráficos para 1, 2 y 3 núcleos, usando SchedFCFS para el lote de tareas del cuadro.

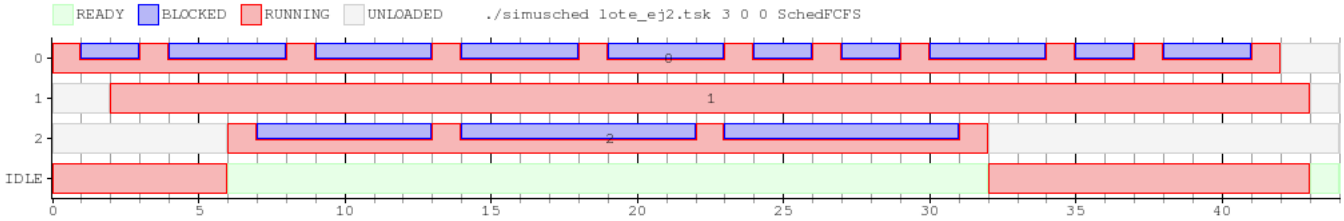
1 Núcleo:



2 Núcleos:



3 Núcleos:



Obs: Los parámetros de costo de cambio de contexto y migración fueron seteados en 0 ya que en este scheduler en particular no son tomados en cuenta. Porque núnca se desalojan las tareas ni se cambian de núcleo.

En los gráficos se hace evidente que a más cántidad de núcleos, mejor rendimiento. Esto se debe a que FCFS no soporta múltitarea por sí sólo (ya que núnca desaloja a las tareas), sino que necesita varios núcleos para lograrlo.

Las desventajas de FCFS son varias:

- Como ya dijimos, no soporta múltitarea.
- Puede generar mucho 'waiting time'.
  Si está ejecutando tareas muy largas las nuevas no entran hasta que éstas terminen.
- No tiene buen rendimiento, a menos que se sepa la duración de las tareas de antemano.
- Carece de 'fairness' i.e. no distribuye el/los procesador/es de forma justa entre las tareas.

3.2. Round-Robin (RR)

RR es un algoritmo de los más viejos, simple, justo y ampliamente usado.

A cada proceso se le asigna un tiempo (quantum) durante el cuál le es permitido correr. Si el proceso sigue corriendo al final del quantum, los recursos le son quitados y se le da paso a otro proceso. Si el proceso se bloquea, pierde el quantum que le quedaba y los recursos son dados a otro proceso.

Implementar RR es simple, sólo se requiere una lista de procesos 'Ready'.
Cuando el proceso usa su quantum es puesto al final de la lista.
Si éste se bloquea, es removido de dicha lista y volverá a ser agregado en cuanto se desbloquee.
En nuestro caso usamos las siguientes estructuras de datos:

Cola de procesos 'Ready'

-Explicar la idea y la funcionalidad de c/u de las estructuras de datos usadas -Hacer pseudocódigo

Algorithm 2 Round-Robin

```
1: procedure LOAD(pid)
2:     encolo la nueva tarea en readyTasks
3: end procedure

4: procedure UNBLOCK(pid)
5:     encolo la tarea desbloqueada en readyTasks
6: end procedure

7: procedure TICK(cpu, motivo)
8:     if el cpu esta ejecutando IDLE then
9:         llamo a NEXT(cpu) para obtener la próxima tarea a ejecutar
10:    else
11:        if el motivo es un TICK then
12:            Actualizo el quantum
13:            if se termino el quantum then
14:                encolo la tarea en readyTasks
15:                renuevo el quantum
16:                llamo a NEXT(cpu) para obtener la próxima tarea a ejecutar
17:            end if
18:        else if el motivo es un BLOCK o un EXIT then
19:            renuevo el quantum
20:            llamo a NEXT(cpu) para obtener la próxima tarea a ejecutar
21:        end if
22:    end if
23: end procedure

24: procedure NEXT(cpu)
25:    if hay tareas esperando en readyTasks then
26:        retornar la primera de la cola readyTasks y quitarla
27:    end if
28: end procedure
```

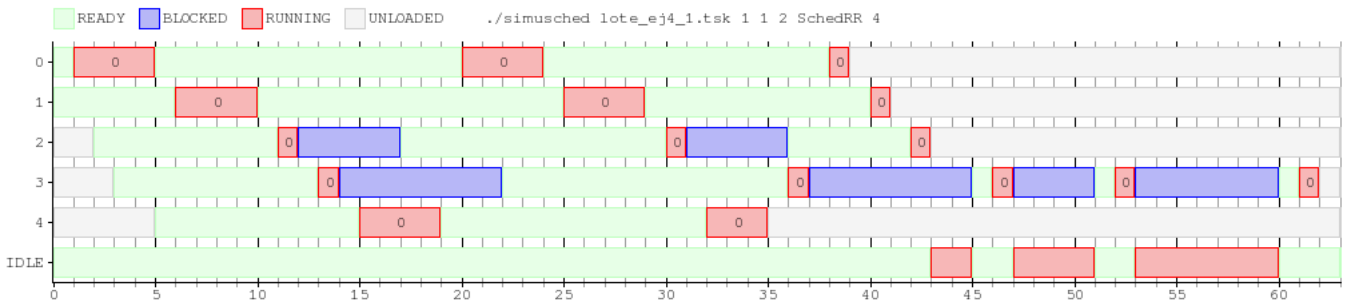
Para las simulaciones elegimos los siguientes costos:

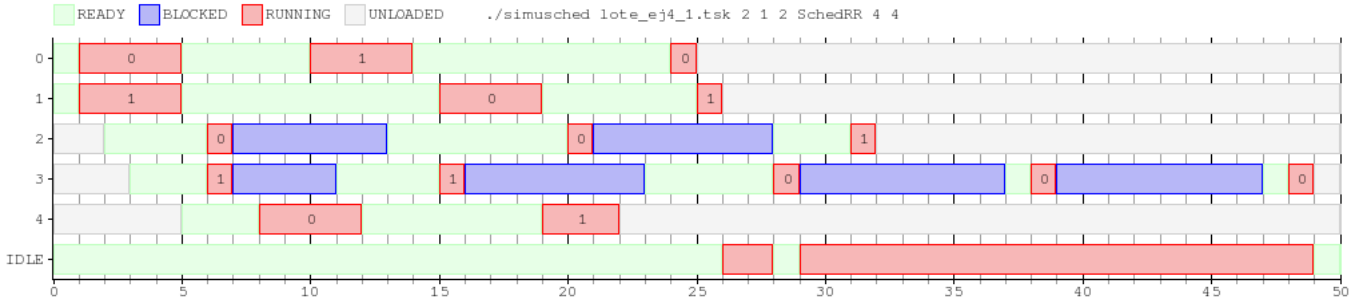
costo de cambio de contexto = 1 : En un CPU real se deben intercambiar estructuras de datos que contienen información de los procesos antes de poder correr la nueva tarea.

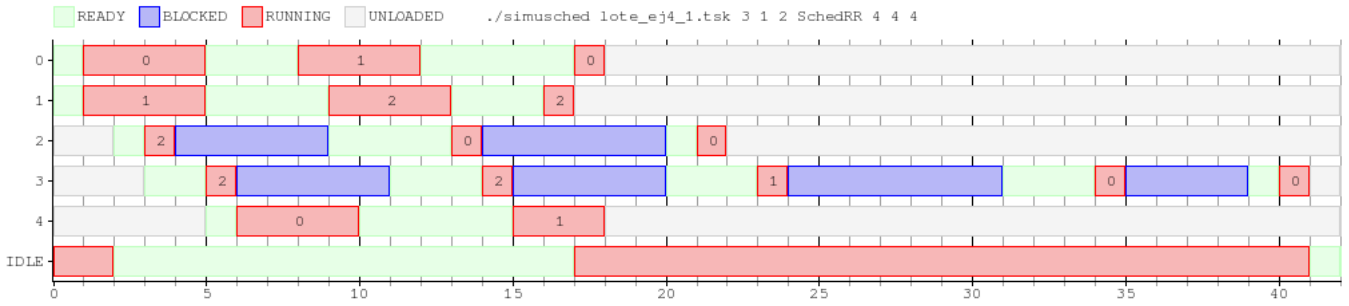
costo de migración de núcleo = 2 : Se deben intercambiar las mismas estructuras de datos que para el cambio de contexto pero la 'distancia' del caché de un núcleo al de otro es mayor que dentro de sí mismo.

!Explicar por qué: -es efectivamente un RR -no siempre es mejor tener más núcleos en un RR con lista global Esto depende de cuándo entren las tareas, que procesadores estaban libres y el quantum de c/u

En estas im todos los cpu tienen quantum = 4 -Hacer otros experimentos con dif quantums y ver cuál queda mejor







4. Discusión

5. Conclusiones