

# Trabajo Práctico 1 - *Scheduling*

Sistemas Operativos - Segundo cuatrimestre de 2016

Fecha límite de entrega: 7 de Setiembre de 2016, 23:59hs GMT -03:00

## Parte I – Entendiendo el simulador `simusched`

### Tareas

Una instancia concreta de tarea (*task*) se define indicando los siguientes valores:

- **Tipo:** de qué tipo de tarea se trata; esto determina su comportamiento general.
- **Parámetros:** cero o más números enteros que caracterizan una tarea de cierto tipo.
- **Release time:** tiempo en que la tarea pasa al estado *ready*, lista para ser ejecutada.

### Lotes y archivos `.tsk`

Un *lote de tareas* representa una lista ordenada de tareas numeradas  $[0, \dots, n-1]$  que se especifica mediante un archivo de texto `.tsk`, de acuerdo con la siguiente sintaxis:

- Las líneas en blanco o que comienzan con `#` son comentarios y se ignoran.
- Las líneas de la forma “`@tiempo`”, donde `tiempo` es un número entero, indican que las tareas definidas a continuación tienen un *release time* igual a `tiempo`. Si no se agrega ninguna línea de `@tiempo`, se asume que todas las tareas empiezan en el instante cero.
- Las líneas de la forma “`TaskName v1 v2 ... vn`”, donde `TaskName` es un tipo de tarea y `v1 v2 ... vn` es una lista de cero o más enteros separados por espacios, representa una tarea de tipo `TaskName` con esos valores como parámetro.
- Opcionalmente, las líneas del tipo anterior puede estar prefijadas por “`*cant`”, lo cual indica que se desean `cant` copias iguales de la tarea especificada.

### Ejemplo

El siguiente es un ejemplo de 4 tareas de tipo `TaskCPU` y el diagrama de Gantt asociado (para un scheduler FCFS con costo de cambio de contexto cero y un solo núcleo):



## Definición de tipos de tarea

Los *tipos de tarea* se definen en `tasks.cpp` y se compilan como funciones de C++ junto con el simulador. Cada tipo de tarea está representado por una única función que lleva su nombre y que será el cuerpo principal de la tarea a simular. Esta recibe como parámetro el vector de enteros que le fuera especificado en el lote, y simulará la utilización de recursos. Se simulan tres acciones posibles que puede llevar a cabo una tarea, a saber:

- a) Utilizar el CPU durante  $t$  ciclos de reloj, llamando a la función `uso_CPU( $t$ )`.
- b) Ejecutar una llamada bloqueante que demorará  $t$  ciclos de reloj en completarse, llamando a la función `uso_IO( $t$ )`. Notar que esta llamada utiliza primero el CPU durante 1 ciclo de reloj (para simular la ejecución de la llamada bloqueante), luego de lo cual la tarea permanecerá bloqueada durante  $t$  ciclos de reloj.
- c) Terminar, ejecutando `return` en la función. Esta acción utilizará un ciclo de reloj para completarse (la simulación lo suma en concepto de ejecución de una llamada `exit()`, liberación de recursos, etc), luego del cual la tarea pasa a estado *done*.

## Sintaxis de invocación

Para ejecutar el simulador, tras compilar con `make`, debe utilizarse la línea de comando:

```
./simusched <lote.tsk> <num_cores> <costo_cs> <costo_mi> <sched> [<params_sched>]
```

donde:

- `<lote.tsk>` es el archivo que especifica el lote de tareas a simular.
- `<num_cores>` es la cantidad de núcleos de procesamiento.
- `<costo_cs>` es el costo de cambiar de contexto.
- `<costo_mi>` es el costo de cambiar un proceso de núcleo de procesamiento.
- `<sched>` es el nombre de la clase de scheduler a utilizar (ej. `SchedFCFS`).
- `<params_sched>` es una lista de cero o más parámetros para el scheduler.

## Graficación de simulaciones

Para generar un diagrama de Gantt de la simulación puede utilizarse la herramienta `graphsched.py`, que recibe por entrada estándar el formato de salida estándar de `simusched`, y a su vez escribe por salida estándar una imagen binaria en formato PNG.

Para generar un diagrama de Gantt del uso de los cores puede utilizarse la herramienta `graph_cores.py`, que recibe por entrada estándar el formato de salida estándar de `simusched`, y a su vez escribe por salida estándar una imagen binaria en formato PNG. Requiere la biblioteca para python matplotlib (<http://matplotlib.org>)

## Ejercicios

**Ejercicio 1** 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<sup>1</sup> entre  $bmin$  y  $bmax$  (inclusive). Luego ejecutará 1 segundo 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.

**Ejercicio 2** Explore utilizando el siguiente grupo de tareas:

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

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*.

**Ejercicio 3** 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 pseudoaleatoriamente. 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*

## Parte II: Extendiendo el simulador con nuevos *schedulers*

Un algoritmo de *scheduling* se implementa mediante una clase de C++ (una nueva subclase que herede de `SchedBase`). A continuación se describe la API correspondiente.

Para ser un *scheduler* válido, una tal clase debe implementar al menos tres métodos: `load(pid)`, `unblock(pid)` y `tick(cpu, motivo)`.

Cuando una tarea nueva llega al sistema el simulador ejecutará el método `void load(pid)` del scheduler para notificar al mismo de la llegada de un nuevo `pid`. Se garantiza que en las sucesivas llamadas a `load` el valor de `pid` comenzará en 0 e irá aumentando de a 1.

Por cada *tick* del reloj de la máquina el simulador ejecutará el método `int tick(cpu, motivo)` del scheduler. El parámetro `cpu` indica qué CPU es el que realiza el tick. El parámetro `motivo` indica qué ocurrió con la tarea que estuvo en posesión del CPU durante el último ciclo de reloj:

---

<sup>1</sup>man 3 rand

- **TICK:** la tarea consumió todo el ciclo utilizando el CPU.
- **BLOCK:** la tarea ejecutó una llamada bloqueante o permaneció bloqueada durante el último ciclo.
- **EXIT:** la tarea terminó (ejecutó `return`).

El método `tick()` del scheduler debe tomar una decisión y luego devolver el `pid` de la tarea elegida para ocupar el próximo ciclo de reloj (o, en su defecto, la constante `IDLE_TASK`). El scheduler dispone de la función `current_pid()` para saber qué proceso está usando el CPU.

Por último, en el caso que una tarea se haya bloqueado, el simulador llamará al método `void unblock(pid)` del scheduler cuando la tarea `pid` deje de estar bloqueada. En la siguiente llamada a `tick` este `pid` estará disponible para ejecutar.

## Ejercicios

**Ejercicio 4** 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.

**Ejercicio 5** 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.

**Ejercicio 6** 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:

- ¿Qué problema están intentando resolver los autores?
- ¿Qué algoritmo generan para resolverlo?
- Explicar brevemente las pruebas realizadas

**Ejercicio 7** En base a lo anterior, defina dos nuevos algoritmos de scheduling. Ambos del tipo *shortest job first*.

- a) Uno no reentrante. Tomará la tarea que menos tiempo de ejecución 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 y los **Quantums** de cada uno de ellos. 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**.

### Ejercicio 8 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.

**Ejercicio 9** 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.

## Informe

El informe **DEBE** contener las siguiente secciones:

- Carátula (1 carilla)
- Índice (1 carrilla)
- **1 sección** por ejercicio

## Código

**DEBEN** modificar el Makefile entregado para que soporte los siguientes *targets*

- `make ejercicio1`
- ...
- `make ejercicio9`

Donde cada *target* **DEBE** generar los gráficos presentados en el informe para el ejercicio indicado.

El código **DEBE** estar comentado.

## Entrega

La entrega se realizará por correo electrónico a la dirección **sisopdc@gmail.com** cuyo asunto **DEBE** decir:

### [TP1] Entrega TP1

y cuyo cuerpo **DEBE** contener los datos de cada integrante:

*Apellido<sub>1</sub>, Nombre<sub>1</sub>, LU<sub>1</sub>, Correo Electrónico<sub>1</sub>*  
*Apellido<sub>2</sub>, Nombre<sub>2</sub>, LU<sub>2</sub>, Correo Electrónico<sub>2</sub>*  
*Apellido<sub>3</sub>, Nombre<sub>3</sub>, LU<sub>3</sub>, Correo Electrónico<sub>3</sub>*

En la entrega **DEBERÁN** adjuntar únicamente:

- El documento del informe (en **PDF**).
- El código fuente **completo junto con el Makefile modificado**.

**NO** incluir código compilado.