

# Taller de *Scheduling*

Sistemas Operativos - Segundo cuatrimestre de 2017

Martes 29 de Agosto de 2017

## 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  $v_1 v_2 \dots v_n$` ”, donde `TaskName` es un tipo de tarea y  $v_1 v_2 \dots v_n$  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>)

Como ejemplo de uso, podríamos tener:

```
./simusched lote.tsk 1 0 5 SchedFCFS | ./graphsched.py > imagen.png
```

## Ejercicios

**Nota:** Los tiempos serán siempre medidos en “ciclos”.

**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). 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.

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

```
TaskCPU 10
@5:
TaskConsola 5 1 3
@6:
TaskConsola 5 1 4
@8:
TaskCPU 8
```

Ejecutar y graficar la simulación usando el algoritmo **FCFS** para 1 y 3 núcleos con un cambio de contexto de 2 ciclos. Calcular la *latencia*, el *waiting time* de cada tarea en los tres gráficos y el *throughput*.

## 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:

- **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.

---

<sup>1</sup>man 3 rand

## Ejercicios

**Ejercicio 3** 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. Diseñar 3 lotes de tareas para experimentar con el Scheduler y comprobar su correcto funcionamiento.

**Ejercicio 4** Implementen un scheduler *Round-Robin* que no permita la migración de procesos entre núcleos (`SchedRR2`). La asignación de CPU se debe realizar en el momento en que se produce la carga de un proceso (load). El núcleo correspondiente a un nuevo proceso será aquel con menor cantidad de procesos activos totales (`RUNNING + BLOCKED + READY`). Debatan un escenario real donde la migración de núcleos sea beneficiosa y uno donde no (piensen específicamente qué métricas de comparación vistas en la materia mejorarían en cada caso). Diseñen un lote de tareas en nuestro simulador que represente a cada uno de esos escenarios y grafique su resultado para cada implementación. Calculen y comparen en cada gráfico las métricas que pensaron.

**Ejercicio 5** Compare para 1 y 2 cores los schedulers *FCFS*, *Round-Robin con migración* y *Round-Robin 2 sin migración*, todos con cambio de contexto de 1 ciclo. El *Round-Robin* con quantum de 5 ciclos.

- Genere sets de pruebas que muestren las ventajas y desventajas según los próximos ítems.
- Calcular la *latencia* y el *waiting time* por tarea y promedio.
- Calcular *throughput*.
- Obtenga conclusiones.

Contando con la implementación de los schedulers *FCFS*, *Round-Robin con migración* y *Round-Robin sin migración*, debatan 3 posibles escenarios en los que cada uno de estos schedulers funcione mejor que los demás según las métricas de *waiting time* y *turnaround*. Diseñen un lote para cada escenario, grafiquen la ejecución con cada scheduler y calculen el *waiting time* y el *turnaround* para cada uno en cada caso.