Trabajo Práctico 1 - *Scheduling*

Sistemas Operativos - Primer cuatrimestre de 2014

Fecha límite de entrega: 21 de abril de 2014, 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`

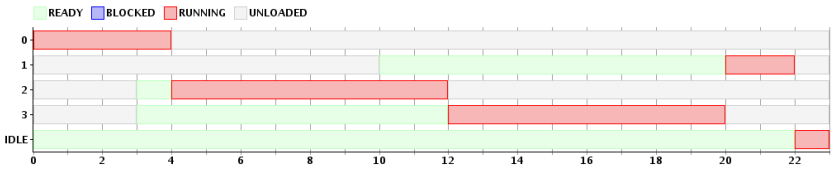
Un *lote de tareas* representa una lista ordenada de tareas numeradas $[0, \ldots, n-1]$ que se especifica mediante un archivo de texto `.tsk`, de acuerdo con la siguiente sintaxis:

- 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 positivo, indican que las tareas definidas a continuación tienen un *release time* igual a tiempo.
- Para tareas en **sistemas de tiempo real**, las líneas de la forma "`$`tiempo", donde tiempo es un número natural, indican que las tareas definidas a continuación tienen un *deadline* igual a tiempo
- Las líneas de la forma "TaskName v_1 v_2 $\cdots$ v_n", donde TaskName es un tipo de tarea y v_1 v_2 $\cdots$ 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):

```
TaskCPU 3
@10:
TaskCPU 1
@3:
*2 TaskCPU 7
```



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 completar, 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[1] 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 Escribir un lote de 3 tareas distintas: una intensiva en CPU y las otras dos de tipo interactivo (`TaskConsola`). Ejecutar y graficar la simulación usando el algoritmo FCFS para 1, 2 y 3 núcleos.

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 que 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.

[1] `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.

Ejercicio 4 Diseñar uno o más lotes de tareas para ejecutar con el algoritmo del ejercicio anterior. Graficar las simulaciones y comentarlas, justificando brevemente por qué el comportamiento observado es efectivamente el esperable de un algoritmo *Round-Robin*.

Ejercicio 5 A partir del artículo

- Fineberg, M.S. and Serlin, O., *Multiprogramming for Hybrid Computation*. Proceedings of the November 14-16, 1967, Fall Joint Computer Conference – 1967.

diseñar e implementar un *scheduler*, `SchedEDF`, basado en el algoritmo *Relative urgency* para **sistemas de tiempo real**. El constructor de la clase debe recibir por parámetro la cantidad de núcleos. Para este ejercicio, utilizaremos una extensión existente del simulador para sistemas de tiempo real con una nueva regla de sintaxis para definir nuestros *lotes de tareas* `.tsk`:

- Las líneas de la forma "`$tiempo`", donde `tiempo` es un número natural, indican que las tareas definidas a continuación tienen un *deadline* igual a `tiempo`.

A su vez, cuando una tarea nueva llega al sistema el simulador ejecturá el método `void load(int pid, int deadline)` del scheduler para notificar a este de la llegada de un nuevo `pid` cuyo *deadline* asociado es `deadline`. Por lo tanto, `SchedEDF` deberá implementar el método load con la nueva signatura.

Parte 3: Evaluando los algoritmos de *scheduling*

Ejercicio 6 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 un (1) ciclo 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).

Ejercicio 7 Elegir al menos dos métricas diferentes. Diseñar un lote de tareas `TaskBatch`, todas ellas con igual uso de CPU, pero con diversas cantidades de bloqueos. Simular este lote utilizando el algoritmo `SchedRR` y una variedad apropiada de valores de *quantum*. Mantener fijo en un (1) ciclo de reloj el costo de cambio de contexto y dos (2) ciclos el de migración. Deben variar la cantidad de núcleos de procesamiento. Para cada una de las métricas elegidas, concluir cuál es el valor óptimo de *quantum* a los efectos de dicha métrica.

Ejercicio 8 Implemente 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). Diseñe y realice un conjunto de experimentos que permita evaluar comparativamente las dos implementaciones de *Round-Robin*.

Ejercicio 9 *SchedEDF* es un un algoritmo de scheduling que cumple con la siguiente propiedad en un procesador con desalojo y con un único núcleo de procesamiento: si una colección de tareas independientes, cada una caracterizada por su release time y deadline, pueden ser asignadas al procesador de alguna manera que todos los trabajos puedan cumplir su deadline a tiempo, *SchedEDF* va a encontar una manera de asignarlos de forma que cumplan con sus deadline.

Diseñar y simular un experimento apropiado para comprobar esta afirmación donde el resto de los schedulers implementados fallen en cumplir con los deadlines de todas las tareas y no lo haga el *SchedEDF*. Comparar las simulaciones de cada scheduler y discutir los resultados obtenidos.

Ejercicio 10 Si bien *SchedEDF* cumple con la propiedad enunciada con un único núcleo de procesamiento, esto no ocurre con más de un núcleo. Diseñe y realice un conjunto de experimentos que demuestre que la propiedad no se cumple con más de un núcleo.

Importante

- En los ejercicios donde se pide diseñar experimentos, asegurarse de documentar y/o justificar los objetivos y decisiones de diseño más importantes (por ejemplo: por qué se decidió usar tal o cual lote de tareas, tal conjunto de valores de *quantum*, etcétera).
- Justificar hipótesis y tesis en base a lo que se desea comprobar y a la evidencia empírica obtenida. En particular, las conclusiones de cada ejercicio deben estar debidamente fundadas en los resultados experimentales **presentados** en la solución del mismo.