Trabajo Práctico - *Scheduling* de tareas

Sistemas Operativos - Primer Cuatrimestre 2011

Fecha de entrega: 18/04/2011 - 23:59hs GMT-0300

Parte 1: Entendiendo el Simulador `simusched`

Una tarea (*Task*) se define indicando los siguientes valores:

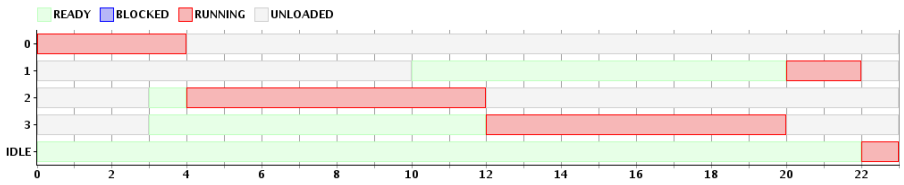
- Tipo: el tipo predefinido de tarea, que determinará su comportamiento.
- Parámetros: cero o más números enteros que caracterizarán la tarea dentro de su tipo.
- Release time: tiempo en que la tarea pasa al estado *Ready*, lista para ser ejecutada.

Un conjunto de tareas o *lote* representa una lista ordenada de tareas numeradas de 0 a $n-1$ y se define en un archivo de texto `.tsk` 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`.
- 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`" que indica que se definen `cant` copias de la misma tarea.

El siguiente es un ejemplo de 4 tareas de tipo `TaskCPU` y el diagrama de Gantt asociado para un scheduler FCFS.

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



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 realizar la tarea de la siguiente manera:

a) Utilizar el CPU por un tiempo de 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`)`. Esta llamada utilizará primero el CPU durante un ciclo de reloj para simular la ejecución de la llamada y luego durante t ciclos de reloj la tarea permanecerá bloqueada.

c) Terminar, ejecutando `return` en la función. Esta acción utilizará un ciclo de reloj para completarse.

Para ejecutar el simulador se debe compilar primero con `make` y luego utilizar la siguiente línea de comando:

```
./simusched <archivo_tareas.tsk> <costo_cs> <sched> [<params_sched>]
```

donde:

- `archivo_tareas.tsk` es el lote de tareas a simular.
- `costo_cs` es el costo del *context-switch* medido en ciclos de reloj.
- `sched` es el nombre de la clase de scheduler a utilizar. Por ejemplo, `SchedFCFS`.
- `params_sched` es una lista de ceros o más enteros que serán pasados como parámetro al scheduler.

Por otro lado, para generar un diagrama de Gantt de la simulación se puede utilizar la herramienta `graphsched.py` que recibe por entrada estandar la salida del simulado y escribe en la salida estandar una imagen en formato PNG.

Ejercicio 1 Escriba un tipo de tarea `TaskCon` que simule una tarea interactiva. La tarea debe realizar n llamadas bloqueantes de una duración al azar entre *bmin* y *bmax* inclusive. La tarea debe recibir tres parámetros: n, *bmin* y *bmax* que serán interpretados como los tres elementos del vector de enteros que recibe la función.

Ejercicio 2 Escriba un *lote* de 3 tareas distintas: una intensiva en CPU y las otras dos de tipo interactivo. Ejecute la simulación para FCFS y haga el diagrama de Gantt de este lote.

Parte 2: Extendiendo el Simulador `simusched`

Un algoritmo de *scheduling* en este simulador se implementa en una clase de C++ que hereda de `SchedBase` e implementa los métodos `load(pid)`, `unblock(pid)` y `tick(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(motivo)` del scheduler. El parámetro `motivo` indica qué ocurrió con la tarea que *ocupaba* el CPU durante el último ciclo de reloj:

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

El método `tick` del scheduler deberá devolver el `pid` que ocupará el próximo ciclo de reloj o la constante `IDLE_TASK`.

El scheduler puede utilizar la función `current_pid()` para saber qué proceso está utilizando 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.

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`. Esta implementación recibe un único parámetro que representa el *quantum* del scheduler.

Ejercicio 4 Diseñar y simular uno o más lotes de tareas con el algoritmo *Round-Robin*. Mostrar y explicar brevemente que el comportamiento es el esperado en base a los diagramas de Gantt generados. Realizar nuevos tipos de tarea de ser necesario para este propósito.

Ejercicio 5 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) se reencola al final de la cola inmediatamente de superior prioridad (si hay).
- Si todas las colas están vacías se ejecuta `IDLE_TASK`.

Parte 3: Evaluando los algoritmos de scheduling

Ejercicio 6 Programar un tipo de tarea `TaskBatch` que tome dos valores *tot* y *blocks*. El tiempo total de CPU que utilice la tarea deberá ser de *tot* ciclos de reloj, incluyendo el tiempo utilizado por las llamadas bloqueantes. La tarea realizará *blocks* llamadas bloqueantes que durarán exáctamente 1 cilco de reloj, en momentos elegidos al azar.

Ejercicio 7 Diseñar un lote de tareas `TaskBatch` de igual uso del CPU pero distinta cantidad de bloqueos. Simular este lote con el algoritmo `SchedRR` con distintos valores de *quantum* y un costo de cambio de contexto de una unidad. Concluir cuál debería ser la *mejor* elección del *quantum* en base a las distintas medidas disponibles. Justificar.

Ejercicio 8 Para el algoritmo implementado en `SchedMFQ`:

a) **Explicar** cómo podría producirse inanición (*starvation*) de una tarea `TaskCPU 20` cargada al inicio.

b) Mostrar un lote de tareas, parámetros iniciales y el diagrama de Gantt asociado para un scheduler con 3 colas en donde se pueda ver que esto ocurre. Implementar nuevos tipos de tarea si es necesario.