Trabajo Práctico 1 - *Scheduling*

Sistemas Operativos - Primer cuatrimestre de 2017

Fecha límite de entrega: Miércoles 19 de abril de 2017, 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`

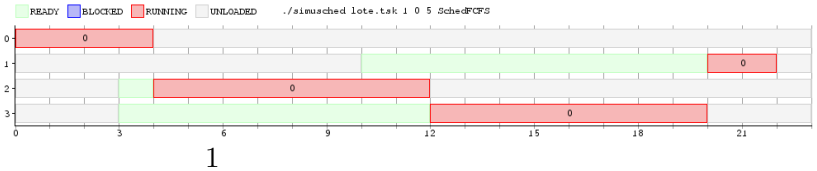
Un *lote de tareas* representa una lista ordenada de tareas numeradas $[0, \ldots, n-1]$ que se especifica mediante un archivo de texto `.tsk`, de acuerdo con la siguiente sintaxis:

- 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, 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 $\cdots$ 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`)

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[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. 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 3
@6:
TaskConsola 5 1 4
@8:
TaskCPU 8
```

Ejecutar y graficar la simulación usando el algoritmo `FCFS` para 1 y 2 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*.

Ejercicio 3 Programar un tipo de tarea `TaskPajarillo` que reciba tres parámetros: *cantidad_repeticiones*, *tiempo_cpu* y *tiempo_bloqueo*. Una tarea de este tipo deberá realizar realizar *cantidad_repeticiones* veces los ciclos de:
tiempo_cpu y luego *tiempo_bloqueo* como llamadas bloqueantes. Uno seguido del otro.

Dé un set de 3 tareas de este tipo.

Para 2 y 3 cores y un cambio de contexto de 2 ciclos calcular la *latencia*, el *waiting time* (por tarea y promedio) y el *throughput* de ejecución de las tareas.

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:

[1] `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 El scheduler *SchedMistery* fue creado por docentes investigadores de nuestra materia y ha sido destacado en la última publicación de `ACM - SIGOPS`, `Operating Systems Review`. Desde entonces, numerosos investigadores de todo el mundo nos han contactado para pedirnos su código fuente. Sin embargo, su código no aparece en ninguno de los repositorios de la materia y nadie parece recordar quiénes habían estado detrás de su implementación.

Se les pide experimentar con dicho scheduler (aprovechando que hemos conseguido el código objeto) y replicar su funcionamiento en *SchedNoMistery*. Graficar como máximo tres lotes de tareas utilizados en los experiementos y explicar en cada uno por separado qué características de SchedMistery identificaron con ese lote. Nota: El scheduler **funciona sólo para tareas de tipo TaskCPU**.

HINT: Utilice la función `std::vector<int>* tsk_params(int pid);` que, dado el pid de una tarea, devuelve la lista de parámetros con que fue cargada.

Ejercicio 6 Definir un nuevo tipo de tarea, `TaskPriorizada`, que utilice sólo *CPU*. Su primer parámetro será su prioridad (más bajo más prioritario y de 1 a 5) el segundo será la cantidad unidades de tiempo *CPU*.

El scheduler será de tipo *Preemptive Shortest Job First* ($PSJF$). Es un scheduler SJF, pero con la particularidad que si entra una tarea más prioritaria o con la misma prioridad pero más corta que la que está en ejecución, ésta tomará el procesador (en el tick siguiente) desalojando a la anterior y haciéndola más prioritaria (hasta 1). Además deberá tener distintas colas de prioridad tomando la prioridad y la duración de la tarea (total de *CPU*, no restante de ejecución), en ese orden.

Realice pruebas de ejecución que muestren las características esperadas de $PSJF$.

Ejercicio 7 Compare para 1 y 2 cores los schedulers *SchedMistery*, *PSJF* y *Round-Robin*, 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.

Informe

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

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

Código

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

- `make ejercicio1`
- ...
- `make ejercicio7`

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 a través del formulario que pueden encontrar en la siguiente URL: `http://www.dc.uba.ar/materias/so/2017/c1/entregas/tp1`.

Deberán completar el número de LU de todos los integrantes del grupo, y subir un único archivo comprimido que **DEBERÁ** contener **únicamente**:

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

NO incluir código compilado.

Al recibir correctamente una entrega, el sistema enviará un mail de confirmación a todos los integrantes del grupo.

Si es necesario, podrán realizar múltiples envíos, en cuyo caso el corrector solo tendrá en cuenta el último envío que se haya realizado antes de la fecha límite de entrega.