INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

LABORATORIO: Sistemas Operativos

Práctica 12

Simulación de un pequeño *scheduler* de un sistema de cómputo: los estados de un proceso

LosDos

Integrantes:

Amanda Velasco Gallardo - 154415
Carlos Octavio Ordaz Bernal - 158525

Fecha(s) de elaboración de la práctica:

03 de mayo de 2019

Introducción

El tiempo de un proceso al momento de su ejecución se divide en ráfagas, éstas pueden ser de CPU (*CPU Bursts*) o de entradas o salidas (*I/O Bursts*). El calendarizador o *scheduler* es el encargado de asignar los recursos a cada uno de los procesos que se encuentran en la memoria esperando a ser ejecutados. Una *CPU Bursts* es un conjunto de instrucciones que son ejecutadas mientras el proceso se encuentra en el estado de `RUNNING`. Las *I/O Bursts*, es una cantidad de tiempo continuo en el que se espera por la terminación de una operación de entrada o salida, es decir, el tiempo en que permanece en el estado `WAITING`.

Existen dos diferentes grupos de algortimos que los *scheduler* utilizan para la asignación de recursos, los cuales son: *non-preemptive* y *preemptive*. Los primeros, son aquellos en los que, una vez asignado el uso de CPU a un proceso sólo puede ser interrumpido cuando termina su ejecución o se inicia una entrada o salida. Los segundos, son aquellos en los que el tiempo de CPU puede ser interrumpido durante la ejecución.

El algoritmo más sencillo es *First-Come, First Served* (*FCFS*). Los procesos pueden llegar de los estados `NEW` y `WAITING`. La cola de espera es del tipo *First In First Out* (*FIFO*), donde el proceso con mayor precedencia es el primero en salir de la cola. La estructura del algoritmo es parecida a una cola `M/M/1`.

Un monitor es una estructura de alto nivel que provee sincronización segura entre los *threads* de una aplicación concurrente. La máquina virtual de `Java` permite que una aplicación pueda tener varios `threads` de ejecución corriendo concurrentemente. Un monitor tiene su propia clase y sus atributos sólo son accesibles desde los propios métodos de instancia del monitor. Un *thread* o proceso sólo puede tener acceso a un monitor invocando el método del mismo, uno a la vez; en consecuencia, sólo se puede estar ejecutando un monitor a la vez. En otras palabras, un monitor provee exclusión mutua entre los procesos concurrentes que lo invocan.

En `Java`, el modificador `synchronized` permite ejecutar código sincronizado para bloquear un objeto, sin necesidad de invocar el método sobre el otro objeto. Si se invoca este método desde un *thread*, el objeto queda bloqueado para otros *threads*. Si otro invoca otro método `synchronized`, sobre el mismo objeto, éste quedará suspendido hasta que se libere el bloqueo.

Desarrollo

Para la implementación de esta solución, en primer lugar se definió la manera de representar los estados por los que pasan los procesos. En nuestro código los estados de `NEW`, `RUNNING`, `WAITING`, y `TERMINATED` son representados como *threads*. El estado `READY` no se representó mediante un *thread* sino que solamente se implementó con una cola. El estado `WAITING` también utilizó una cola; ambas eran comunes a todos los elementos del código.

En segundo lugar, se hizo la clase `ProcesoG`, la cual tiene como atributos el nombre del proceso, la cantidad de *bursts* asignada, el tiempo de servicio, y el tiempo de espera. En tercer lugar, se definió la clase principal llamada `Scheduler` cuyos atributos son una cola de procesos en estado `READY`, una cola de procesos en estado `WAITING`, una bandera *boolean* para indicar si el CPU está ocupado o no, y dos enteros que indican la cantidad de procesos creados y la cantidad de procesos terminados.

El método `main` de la clase `Scheduler` es el encargado de leer del archivo `cmdbatch.txt` los comandos `run` de cada uno de los programas. Por cada línea se extrae el nombre del programa y el número de *bursts*, se crea un *thread* del tipo `TNew` y se ejecuta.

A partir de la ejecución del *thread* `TNew`, se incrementa la cantidad de procesos creados, se encola el proceso en la cola de `READY` y se crea un nuevo *thread* `TRun` para el estado `RUNNING`.

En el *thread* `TRun` se verifica que el CPU no esté en uso, en caso contrario se espera a que éste se desocupe. Una vez libre, se extrae el primer elemento en la cola de `READY`, se calcula un `Math.random()` para la duración del *burst* y se manda a espera con el comando `sleep()`. Una vez terminado el tiempo de ejecución, se actualiza el tiempo de servicio del proceso y se decrementa en uno el número de *bursts*. Por último, se pone en cola de `WAITING` y se libera la ocupación del CPU. Para el caso de que se trate del último *burst* que el proceso debe ejecutar, éste es mandado a un *thread* `Terminator`.

Para el caso del *thread* `TWaiting`, de manera análoga a `TRun`, se calcula el tiempo que dura la operación de entrada o salida. Al finalizar, se actualiza el tiempo de espera, y el proceso se devuelve a la cola de `READY`.

Para el último de los estados de los procesos, el*thread* `Terminator` recibe el proceso saliente del estado `RUNNING` y crea unos `String` que son un resumen de la información del proceso, y se incrementa en uno la cuenta de los procesos terminados. Dicha información se manda a un método que es el encargado de escribir en el archivo `bitacora.txt`.

Finalmente, por cada cambio de estado en los procesos también se manda a imprimir la información en el archivo `bitacora.txt` y en la misma ventana de comandos. Para terminar la ejecución del `main` se revisa que la cuenta de procesos terminados sea igual al número de procesos creados en el estado `NEW` e imprime un mensaje de salida.

Conclusiones

La presente práctica nos permitió conocer más a fondo la forma en la que el algoritmo *FCFS* funciona. Programar desde cero todas las clases de los distintos estados de los procesos nos ayudó a entender la interacción de cada uno de ellos con el resto de los mismos y con el sistema operativo. También, reforzamos el uso de los monitores como herramientas para sincronizar de manera segura los *threads* de una aplicación concurrente. Finalmente, hicimos uso de las herramientas adquiridas a lo largo del segundo parcial para la programación y manejo de *threads* dentro de la máquina virtual de `Java`.

Referencias

- Ríos, J. (2019). Notas del curso de Sistemas Operativos. Recuperado el 08 de mayo de 2019, del sitio web: Comunidad ITAM.