Principios de Sistemas Operativos: Calendarizadores

Alejandro González Alvarado

e-mail: ale2793@hotmail.com

Esteban Calvo Lopez

e-mail: estebancalvo2004@hotmail.com

Abstract: *The main goal of this project is to design and implement a set of four CPU Scheduling algortihms for the PintOS operating system. The four algorithms to implement are: First Come First Serve (FCFS), Shortest Job First (SJF), Round Robind (RR) and Multilevel Feedback Queue (MLFQS). All these algorithms must be implemented using C as programming language. Aditionally, we implemented some tests to verify the efficiency and efectiveness of our scheduling algorithms.*

PALABRAS CLAVE: cpu, scheduling, PintOS, FCFS, SJF, RR, MLFQS, tests.

1 INTRODUCCIÓN

Este proyecto tiene como propósito principal, el diseño y la implementación de cuatro algoritmos calendarizadores de CPU, los cuales son: First Come First Serve, Shortest Job First, Round Robin y Colas Multinivel.

Antes de iniciar a programar estos algoritmos, primero fue necesaria una investigación a fondo del funcionamiento de los hilos dentro del sistema operativo PintOS, ya que es necesario entender como se calendarizan inicialmente los procesos y cual es su ciclo de vida, para luego poder implementar nuestras versiones de los algoritmos anteriormente mencionados.

La implementación de estos algoritmos se realizó utilizando el lenguaje de programación C, el cual es comúnmente utilizado para la implementación de sistemas operativos debido a su robustez y sus capacidades de manejo de memoria. Además que C puede ser compilado y ejecutado en una gran cantidad de procesadores distintos.

Para el diseño de estos algoritmos, nos basamos en las descripciones del libro Operating System Concepts, 9th ed. el cual detalla lo suficiente el funcionamiento de cada uno de estos algoritmos. El calendarizador por Colas Multinivel es un caso especial, pues para su implementación se siguió el diseño utizado por el 4.4BSD Scheduler.

Una vez desarrollados los diferentes calendarizadores, se debe ejecutar una serie de pruebas que nos permita cuantificar el desempeño de dichos calendarizadores.

2 DESCRIPCIÓN DEL PROBLEMA

Para el presente proyecto, se deben crear cuatro algoritmos calendarizadores distintos para el sistema operativo PintOS. Éstos algoritmos deben ser implementados en el lenguaje C (utilizado por PintOS) y se debe seguir el estándar de código publicado por Richard Stallman en “The GNU coding standards”, específicamente las secciones 5 y 6. Además, debe proveerse una forma de elegir el tipo de calendarizador con el que se quiere correr PintOS.

Adicionalmente, se debe crear una plataforma de pruebas que nos permita elegir los tipos y la cantidad de hilos a correr en el procesador. La cantidad máxima de hilos que podrán ejecutarse son 25.

También, se debe implementar un programa de comparación que sea capaz de comparar al menos cuatro archivos de logs distintos. Donde cada log corresponde a la salida de una prueba de la plataforma de pruebas anteriormente detallada. Este algoritmo de comparación de mostrar los siguientes datos:

- Comparación entre los tiempos de espera promedio de cada corrida.
- Algoritmo más rápido.
- Porcentaje de tipo de hilos involucrados.

Finalmente, debe implementarse un bootloader que permita ejecutar PintOS con un set pre-establecido de pruebas, además debe existir la posibilidad de elegir el calendarizador que se desea utilizar.

3 DETALLES DE LA IMPLEMENTACIÓN

3.1 Calendarizador FCFS:

First Come First Serve es el algoritmo calendarizador más sencillo de implementar, pues simplemente se le va asignando el procesador a los threads en el mismo orden en que lo solicitaron. Para lograr este comportamiento, no se debió agregar nada nuevo ya que PintOS hace push_back a todos los hilos que se van creando y desbloqueando en el entorno de los procesos. Por lo que el último proceso que solicitó ingresar a la lista de espera, es colocado al final de la misma.

La principal dificultad de su implementación, se debió al comportamiento apropiativo que tiene PintOS por defecto. Este algoritmo es no apropiativo por lo que los hilos no deberían de entregar constantemente el procesador que les fue asignado. Debido a que los hilos

requieren del uso de la función timer_sleep() para que puedan ser calendarizados, tuvimos que realizar una re-implementación completa de dicha función. En la nueva implementación, los hilos son bloqueados mediante un semáforo haciendo sema_down() y se establece un wakeup_time de acuerdo al parámetro de entrada de la función timer_sleep(), luego, en cada timer_interrupt() se verifica que el wakeup_time de cada hilo haya pasado. De ser así, hace un sema_up() a todos los hilos que ya pueden correr. Como la lista de espera está ordenada, la iteración se realiza de forma ordenada y los hilos se van despertando y ejecutando de acuerdo a su orden de llegada. De esta forma se ejecuta el calendarizador siguiendo el algoritmo First Come First Serve.

3.2 Calendarizador SJF:

Este algoritmo asocia y calendariza un proceso de acuerdo a su tiempo de CPU-Burst. En primera instancia, no se conoce la duración del CPU-Burst para un hilo dado. Por lo que es necesario desarrollar un método para realizar estimaciones y así calcular futuras aproximaciones. Para esto, se creó una función llamada *calc_new_total_exec(int old, int recent)* que es la encargada de calcular las aproximaciones. Para las aproximaciones se utiliza la fórmula:

$$\tau_n = \alpha t_n + (1-\alpha)\tau_n$$

Donde τ_{n_1} contiene la siguiente aproximación, t_n guarda la estimación anterior y τ_n es la duración reciente. Para α se utilizó un valor de 0.5.

Esta función es llamada cuando finaliza cada hilo, y además, luego de obtener la aproximación, se procede a actualizar las aproximaciones de los hilos activos que sean iguales al de la última aproximación calculada. Esta actualización se realiza utilizando el método t*hread_foreach(get_total_exec_each, NULL),* cuya función implícita, se encarga de la actualización de las estimaciones buscando en la lista *all_all_lis*t.

Al finalizar el re-ordenamiento, se ordena la lista de espera de acuerdo a la estimación de su CPU-Burst.

3.3 Calendarizador RR:

PintOS cuenta por defecto, con un calendarizador de tipo Round Robin, por lo que toda la infraestructura necesaria para su implementación ya está creada. Para nuestro proyecto, efectuamos la ejecución de este algoritmo de una manera levemente diferente.

En nuestra implementación, cada hilo posee una variable que lleva el conteo de cuantos “ticks” del procesador ha recibido. Cuando esta variable alcanza un valor mayor o igual al quantum, que en nuestro caso elegimos un valor de 2, el hilo que se está ejecutando actualmente cede el procesador. Estos procedimientos se realizan dentro de la función thread_tick(), que corre en un contexto de interrupciones externas, por lo que es llamada cada vez que ocurre una interrupción del timer. Para ceder el procesador, se activa la bandera yield_on_return, la cual a su vez se encarga de llamar a la función thread_yield().

3.4 Calendarizador MLFQS:

Este algoritmo generalmente particiona la lista de espera de los hilos, en múltiples colas de espera, con procesos de distintos tipos. En nuestro caso, creamos múltiples colas de espera diferenciadas por la prioridad de cada hilo. Las prioridades van de 0 (PRI_MIN) hasta 63 (PRI_MAX), por lo que se cuenta con 64 colas distintas. Para calcular la prioridad, utilizamos la fórmula:

$$priority = PRI\_MAX - (recent\_cpu/4) - (nice*2)$$

Dónde recent_cpu, es una estimación del tiempo de CPU que ha utilizado el hilo recientemente. *nice* es un parámetro que establece que tan “bueno” debería ser un hilo con los demás hilos. Un valor de *nice* alto, causa que el hilo esté soltando constantemente el CPU que le fue asignado. Para el cálculo de recent_cpu, se utiliza la fórmula:

$$recent\_cpu = \frac{2*load\_avg}{2*load\_avg+1}*recent\_cpu + nice$$

Donde load_avg, es una media móvil que estima el número de hilos que estuvieron listos para correr durante el último minuto. Para calcular el load_avg, se utiliza la fórmula:

$$load\_avg = (59/60)*load\_avg + (1/60)*ready\_threads$$

Para cada una de estas fórmulas, se implementaron las funciones refresh_priority(), refresh_cpu() y refresh_load_avg() respectivamente.

En cuánto a las actualizaciones que se requieren una vez por segundo, la comprobación se hace dentro de la función thread_tick(), con la condición timer_ticks() % TIMER_FREQ == 0. Si se cumple, se efectúan las respectivas actualizaciones llamando a las funciones refresh_cpu() y refresh_load_avg(). Adicionalmente, cada 4 ticks, se llama la función refresh_priority() que actualiza la prioridad del thread que se está ejecutando actualmente.

A la hora de calendarizar, si existen múltiples threads con una misma prioridad, estos son ejecutados en orden Round Robin hasta que finalicen todos los threads con esa prioridad.

4 JUSTIFICACIÓN

4.1 Calendarizador FCFS:

Decidimos re-implementar la función timer_sleep() debido a la naturaleza apropiativa que presentaba. Al fijarnos en la estructura y documentación de los

semáforos, observamos que estos guardan una lista de hilos que esperan por el mismo semáforo. Entonces lo que se hizo, fue agregar **todos** los hilos que llaman a la función timer_sleep() a la lista de espera del semáforo que llamamos timer_sema, y se le aplica un sema_down al semáforo del hilo correspondiente. De esta forma, eliminamos el thread_yield() y nos deshacemos de la apropiatividad que representa el uso de timer_sleep().

Al eliminarse la naturaleza expropiativa y el Round Robin de los threads en PintOS, estos
pueden ejecutarse en orden con respecto a su wakeup_time y su orden de llegada. Al eliminarse la expropiatividad, los hilos no ceden su CPU hasta que hayan finalizado. De esta manera se implementa efectivamente un calendarizador de tipo FCFS.

4.2 Calendarizador SJF

Consideramos que nuestra implementación es bastante eficiente pues los cálulos y el re-ordenamiento de la lista de espera sólo se hace una vez cada vez que un hilo finaliza. Por lo que se reduce significativamente el overhead necesario para el algoritmo, en comparación si realizáramos éstos cálculos en cada timer_tick por ejemplo. En lo que respecta al cálculo de la aproximación, utilizamos la fórmula:

$$\tau_n = \alpha t_n + (1-\alpha)\tau_n$$

Dónde elegimos un valor de α = 0.5, lo cual provoca que la última estimación y el valor de la última duración tengan el mismo peso a la hora de calcular la nueva estimación, lo cual consideramos apropiado pues se consideran ambos valores igual de importantes. Existe un caso especial que se da cuando la estimación anterior es 0, en este caso no realizamos el cálculo de la aproximación sino que simplemente decimos que la primera estimación corresponde a la duración que tardó en completarse el algoritmo la primera vez.

4.3 Calendarizador RR

Generalmente, un ***quantum*** de tiempo está definido entre 10 y 100 milisegundos, su elección varía dependiendo del sistema operativo que lo implemente y de los requerimientos que existan.

PintOS realiza TIMER_FREQ interrupciones por segundo. Por defecto, TIMER_FREQ posee un valor de 100, lo cual se traduce en 100 interrupciones del timer por segundo. Decidimos mantener este valor ya que lo consideramos adecuado para la cantidad de interrupciones que ocurren por segundo en el sistema, además de ser un número con el cual es sencillo trabajar.

En lo que respecta al quatum , definimos su valor con 7, es decir, ccada vez que un hilo se ejecuta durante 7 “ticks”, éste cede el procesador al siguiente hilo en la lista de espera.

Este valor fue elegido por dos razones, la primera es que se mantiene un valor promedio de 15 cambios de contexto por segundo, lo cual consideramos apropiado para PintOS ya que este sistema no corre generalmente una gran cantidad de hilos simultáneamente, además que para las pruebas se correrán un máximo de 25 hilos, por lo que en 2 segundos, el procesador le ha sido asignado a todos los hilos al menos una vez. La otra razón, es que realizamos múltiples pruebas con distintos valores de ***quantum*** y el valor de 7 nos brindada regularmente los menores tiempos de ejecución. Cabe resaltar, que todas las pruebas fueron hechas utilizando hilos de tipo CPU-Bounded únicamente.

4.4 Calendarizador MLFQS:

Decidimos implementar el calendarizador por colas multinivel de esta forma pues este es el diseño recomendado por la Universidad de Stanford para implementar el algoritmo de olas multinivel en PintOS. Además, que así lo indica la especificación del proyecto. Este diseño corresponde al calendarizador utilizado por el sistema operativo 4.4 BSD.

Consideramos que ésta implementación es bastante eficiente, pues fue utilizada para un sistema operativo real. Además, se ajusta a las capacidades de PintOS, ya que éste cuenta con una lista de espera ready_list predefinida y lo que se hace es dividir esta lista de espera en 64 listas de prioridad distintas. De ésta forma, los hilos se van ejecutando de acuerdo al valor de su prioridad, y, si existen múltiples hilos con el mismo valor de prioridad, éstos se ejecutan utilizando el algoritmo de calendarización Round Robin. Así, no se deben crear múltiples colas de espera, todas con hilos diferentes, sino que se maneja una sola lista y la particionamos por prioridades.

4.5 Pruebas

Para los hilos de tipo I/O-Bounded, se utilizaron lecturas a memoria con la función inl(uint16_t port), la cual retorna 32 bits leídos de la dirección port. Decidimos utilizar lecturas a memoria, ya que incialmente PintOS no cuenta con gran variedad de funciones I/O, por lo que las funciones a elegir están limitadas. Además, decidimos no usar escrituras a memoria como operaciones I/O ya que una escritura a la dirección incorrecta podría generar un comportamiento no deseado en nuestra sistema.

En cuanto a los hilos CPU-Bounded, se implementaron sumas hasta un valor objetivo, de esta manera es más sencillo controlar la cantidad de ejecuciones que efectúa cada hilo.

5 Comparación de los hilos

El proceso de comparación del desempeño de los diferentes algoritmos de calendarización, se realiza obteniendo y analizando los tiempos de espera promedio para cada calendarizador, así como su duración de ejecución medida en ticks del procesador.

Para la comparación, se creó un programa que parsea un log con los siguientes campos:

- Algoritmo
- Duración
- Tiempo de espera promedio de todos los hilos
- Porcentaje de hilos CPU-Bounded
- Porcentaje de hilos I/O-Bounded

De acuerdo a estos valores, el algoritmo comparador determina cual fue el más rápido, así como también, cuál fue el que tuvo menor tiempo de espera promedio para todos sus hilos.

Para comparar los algoritmos, se ejecutó la misma prueba para los diferentes. La cual consistía de 12 hilos, de los cuales 1 es I/O Bounded y 11 son CPU Bounded.

El siguiente resultado corresponde a la salida del **comparador**:

El calendarizador más rapido fue MLFQS con un tiempo de ejecución de 580 ticks de procesador. Se utilizó un set de hilos de tipo CPU-Bound: 92% y I/O-Bound: 8%.

El calendarizador con menor tiempo promedio de espera fue RR con un promedio de 26 segundos. Se utilizó un set de hilos de tipo CPU-Bound: 92% y I/O-Bound: 8%.

Como se puede observar, el menor tiempo de espera lo brinda el algoritmo Round Robin, lo cual concuerda con lo esperado pues la finalidad del algoritmo, es reducir los tiempos de espera de cada hilo.

6 CONCLUSIONES

- El algoritmo calendarizador FCFS es el más sencillo de implementar pero también es el más lento.
- Para el algoritmo SJF, la dificultad se encuentra en calcular efectivamente las aproximaciones de duración de cada hilo. No basta con que los hilos indiquen cuánto van a durar, ya que este valor puede no ser exacto.
- En cuánto al algoritmo Round Robin, es uno de los mejores calendarizadores pues su implementación es sencilla y requiere de poco overhead.
- La eficiencia de un calendarizador Round Robin depende completamente del *quantum* que se elija.
- Un calendarizador de colas multinivel es apropiado para utilizarlo en un sistema operativo real, pues este ejecuta muchos procesos de distintos tipos. Por lo que una buena categorización permitiría una calendarización eficiente.
- Un algoritmo calendarizador no es simplemente una función que se encarga de calendarizar los procesos, sino que su implementación se encuentra en varias partes del manejo de hilos del sistema operativo
- Los hilos I/O Bounded poseen ráfagas de ejecución mucho menores a los CPU Bounded.t

7 Referencias

Tanenbaum. Sistemas Operativos: Diseño e Implementación. Prentice Hall. 1988.

Peterson. Operating Sytem Concepts. Addison Wesley, Second Edition. 1985.