Sistemas Operativos. Pintos: "MLFQ Scheduling"

Ricchy Alain Pérez Chevanier
Angel Renato Zamudio Malagón

1. Objetivo

El objetivo de esta práctica es que el alumno modifique la implementación del calendarizador de prioridades que realizó en la práctica anterior, de tal forma que evite que haya hambruna.

2. Introducción

Uno de los inconvenientes de un calendarizador de prioridades es la hambruna (starvation), esta situación se presenta cuando un thread de alta prioridad tarda mucho tiempo en terminar su ejecución o nunca lo hace, lo cual provoca que los demás threads no se ejecuten. Aunque el problema de la hambruna es importante, no se puede prescindir de forma tan drástica de un calendarizador de prioridades, por lo cual necesitamos un mecanismo para poder seguir teniendo un calendarizador de prioridades, pero que no genere hambruna.

La característica más importante de un calendarizador de prioridades permite que un thread de alta prioridad se ejecute inmediatamente, por lo que una buena solución al problema de la hambruna debe conservar dicha característica. La idea principal es un calendarizador que permita que un thread con la más alta prioridad sea el que se ejecute, pero conforme pasa el tiempo el thread comience a disminuir su prioridad, de forma inversa, con un thread de baja prioridad, conforme pasa el tiempo, debe subir su prioridad, con ello se evita la hambruna.

Una primer aproximación a esta solución consistiría en tener un contador por cada thread que nos indique cuantos *ticks* se ha ejecutado cada thread, la prioridad del thread se calcularía en función del contador. Esta aproximación tan simplista tiene algunos inconvenientes, por ejemplo, un thread puede haberse ejecutado durante mucho tiempo en un período que ocurrió en un momento muy lejano.

2.1. Definición de *load_average*

El indicador *load_average* nos sirve para saber cual es la carga actual de todo el sistema, más concretamente, *load_average* estima el número de threads listos para su ejecución en el último minuto. El indicador *load_average* es global, al inicio del sistema se debe inicializar en 0 y **una vez por segundo** se debe calcular de la siguiente forma:

$$load\_avg = \frac{59}{60} \cdot load\_avg + \frac{1}{60} \cdot ready\_threads$$

La variable *ready_threads* es el número de threads que están en ejecución o listos para ejecución, sin incluir el thread *idle*.

2.2. Definición de *recent_cpu*

Necesitamos una medida que indique que tanto tiempo un thread a utilizado el CPU *recientemente*, dicha medida la llamaremos *recent_cpu*. El *recent_cpu* debe cumplir con la característica de que entre más recientemente se haya usado el CPU, esa cantidad debe pesar más que la misma cantidad de uso en un instante previo. Para ello utilizamos una función exponencial de tipo *moving average*, la cual tiene la siguiente forma:

$$x(0) = f(0)$$
$$x(t) = a \cdot x(t-1) + f(t)$$

El coeficiente a se define como:

$$a = \frac{k}{k+1}$$

El valor *x(t)* es el *moving average* en el tiempo t >= 0, *f(t)* es la función que se desea promediar y k controla el factor de decaimiento. Entre mayor sea k el moving average decaerá más lento. Si iteramos el cálculo de la función obtenemos lo siguiente:

$$x(1) = f(1)$$
$$x(2) = a \cdot f(1) + f(2)$$
$$.$$
$$.$$
$$.$$
$$x(t) = a^{t-1}f(1) + a^{t-2}f(2) + ... + af(t-1) + f(t)$$

El valor de *f(t)* tiene un peso de 1 en el tiempo t, un peso de a en el tiempo t + *1*, un peso de a^2 en el tiempo t + *2* y así sucesivamente. Cada thread debe tener su propio *recent_cpu*, el cual debe ser 0 al momento de crear el thread, cada que el thread se ejecute un *tick* se debe incrementar en 1 para ese thread, esto pasa cada que el *timer* interrumpe al procesador. Adicionalmente, **cada segundo** se debe calcular el valor del *recent_cpu* para todos los threads, esto se realiza de la siguiente forma:

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

2.3. Definición de *nice*

El factor de bondad o *niceness* de un thread determina que tanto debe ceder su tiempo de procesador, esto es útil pues como las prioridades se calcularán dinámicamente, el factor de bondad puede ser utilizado para hacer que un thread permanezca más o menos tiempo en ejecución. Si el valor de *nice* es 0, no afecta en nada al thread. Un valor positivo para *nice*, con un máximo de 20, decrementa la prioridad del thread. Un valor negativo, con un mínimo de -20, incrementa la prioridad del thread.

2.4. Cálculo de la prioridad

Una vez definidos los indicadores *recent_cpu*, *load_avg* y *nice*, lo último que tenemos que hacer es definir la prioridad. Cada thread comienza con una prioridad inicial en el momento en el que se crea, después, **cada cuatro *ticks*** se debe calcular la prioridad de la siguiente forma:

$$priority = PRI\_MAX - \frac{recent\_cpu}{4} - 2 \cdot nice$$

2.5. Aritmética de punto fijo

Las formulas necesaria para calcular la prioridad son números reales, desafortunadamente, Pintos no soporta operaciones de punto flotante en modo kernel pues en caso contrario volvería mas lento el funcionamiento del kernel. La mayoría de los sistemas operativos tienen la misma limitación. Esto implica que las operaciones con valores reales tienen que ser simuladas utilizando enteros.

La idea principal es dividir los bits de un entero en dos partes, una parte entera y una parte decimal. Una estructura que funciona bien en términos de precisión para los números que se utilizan en las formulas es utilizar los 17 bits más significativos para representar la parte entera y los restantes 14 bits para la parte fraccionaria.

Para almacenar los valores en punto fijo se utiliza un entero de 32 bits, es necesario definir operaciones para transformar un entero en formato normal a un entero en formato de punto fijo y viceversa, también es necesario definir las operaciones aritméticas para poder operar con números en formato de punto fijo. A continuación se muestra una tabla con la definición de las operaciones como si se implementaran en lenguaje C, considerando que n es un entero en formato normal, x y y tienen formato de punto fijo y $f = 2^{14}$.

OPERACIÓN	IMPLEMENTACIÓN
Convierte n a punto fijo	n*f
Convierte x a entero (truncando)	x / f
Suma x con y	x + y
Resta y a x	x - y
Suma x con n	x + n
Resta n a x	x - n
Multiplica x por y	((int64_t) x) * y / f
Divide x entre y	((int64_t) x) * f / y
Multiplica x por n	x * n
Divide x entre n	x / n

3. Forma de Trabajo

Pasos a seguir para trabajar en esta práctica:

1. Debes de tener una solución para la práctica anterior en la rama `proyecto01/priority-scheduling` o `main`.
2. A partir de la rama que contiene la solución de la práctica anterior crea una nueva rama llamada `proyecto01/mlfq-scheduling` y posiciónate en ella.
3. En el repositorio reemplaza el archivo `src/threads/Make.vars` con el que entregamos junto con este documento.
4. En el repositorio copia el archivo `execute-tests-assignment-03` en la ruta `src/threads`. Debe de quedar al final en `src/threads/execute-tests-assignment-03`.
5. Agrega al repositorio en el directorio raíz el archivo `designdoc-mlfq-scheduling.md`.
6. Agrega al repositorio en el directorio `src/threads` los archivos `fixpoint.h`, `fixpoint.c`, `mlfqs-calculations.h` y `mlfqs-calculations.c`.
7. Agrega al repositorio en el directorio `src` el archivo `Makefile.build`. Este archivo es que el logra que los archivos que agregaste en el punto anterior puedan ser añadidos al proceso de compilación de pintos.
8. Haz commit de estos cambios y sube (*push*) la rama al repositorio.

9. Crea un *Pull Request* para la rama `proyecto01/mlfq-scheduling` hacia la rama que contiene la solución a la práctica anterior. Este *Pull Request* debe de contener 8 archivos en la sección de *Files Changed*, que son exactamente los que introdujiste en los pasos anteriores.

10. En la sección *Checks* del Pull Request, asegúrate que las pruebas de las primeras dos prácticas siguen pasando (*Alarm Clock* y *Priority Scheduling*).

11. Una nota importante es que para pasar las pruebas vamos a utilizar la máquina virtual `qemu`, pues `boch` puede tener algunos problemas.

En el archivo `readme.md` del repositorio se incluyen instrucciones de cómo compilar el código y de cómo ejecutar algunas pruebas. Para ejecutar todas las pruebas de esta práctica, puedes acceder al contenedor, posicionarte en el directorio `src/threads` y ejecutar la instrucción `./execute-tests-assignment-03`.

Tu solución tiene que ser incremental, es decir, debes de pasar las pruebas de la práctica anterior y también las pruebas específicas de esta práctica, de hecho en Github se van a ejecutar como pasos independientes.

4. Actividades

Implementar un calendarizador avanzado que utilice el criterio de actualización de prioridades descrito anteriormente.

4.1. Implementación (5 pts)

- mlfqs-load-1
- mlfqs-load-60
- mlfqs-load-avg
- mlfqs-recent-1
- mlfqs-block
- mlfqs-fair-2
- mlfqs-fair-20
- mlfqs-nice-2
- mlfqs-nice-10

Para que las pruebas no fallen, aparte de realizar correctamente los cálculos, es necesario implementar las siguientes funciones, la cuales son utilizadas por las pruebas para poder determinar el estado del calendarizador:

int thread_get_nice (void)
Regresa el valor *nice* del thread actual.

void thread_set_nice (int)
Asigna el valor *nice* al thread actual.

int thread_get_recent_cpu ()
Regresa el valor del *recent_cpu* del thread actual multiplicado por 100.

int thread_get_load_avg ()
Regresa el valor del *load_avg* multiplicado por 100.

4.2. Documento de Diseño (5 pts)

Contestar todas las sección del documento de diseño que se entrega junto con este documento.

5. Consejos

Cada una de las formulas requiere que se calcule en diferentes momentos, el lugar más indicado para hacerlo es en la función *thread_tick* pues esta función se ejecuta cada tick. Para determinar los ticks que son múltiplos de un segundo se puede usar la condición *timer_ticks () % TIMER_FREQ == 0*.

Junto con este documento entregamos algunos archivos con la implementación de las operaciones de punto fijo `fixpoint.h` y `fixpoint.c`, en el header file incluimos el tipo alias `fixpoint_t` para que lo utilices para definir las variables *load_avg* y *recent_cpu*. Así como también la infraestructura para el cálculo de *load_avg*, *recent_cpu* y *priority* en los archivos `mlfqs-calculations.h` y `mlfqs-calculations.c`.

Declarar e inicializar todas las constantes que aparecen en las formulas al inicio del archivo *src/threads/thread.c* de ésta forma se garantiza que la conversión de formato normal a punto fijo solamente se realizará una vez.

Mantener las variables *recent_cpu* y *load_avg* en formato de punto fijo, únicamente convertirlas a formato normal en las funciones *thread_get_recent_cpu* y *thread_get_load_avg*.

Es una buena idea comenzar por implementar el cálculo de *load_avg*, tratando de que las pruebas *mlfqs-load-1* y *mlfqs-load-60* pasen. Este es un buen punto de inicio pues los demás cálculos utilizan *load_avg*.

6. Entrega

En Google Classroom agrega el enlace del `Pull Request` que contiene tu solución y en donde se deben de observar los resultados de Github Actions al correr las pruebas. También marca la tarea como entregada en Google Classroom.