Sistemas Operativos 2020-2, Práctica 04:

Pintos, Scheduler de Prioridades

Luis Enrique Serrano Gutiérrez (luis@ciencias.unam.mx)
Juan Alberto Camacho Bolaños (juancamacho@ciencias.unam.mx)
Ricchy Alaín Pérez Chevanier (alain.chevanier@ciencias.unam.mx)

Fecha de Entrega: 04 de Mayo de 2020

1 Objetivo

El objetivo de esta práctica es que el alumno implemente un `scheduler de prioridades` dentro de `pintos`, además de resolver las distintas problemáticas que surgen de dicha implementación sobre el comportamiento de otros componentes del sistema operativo.

2 Introducción

Gracias a la práctica anterior ahora ya tienes un mejor entendimiento de cómo funciona el módulo de `threads` en `pintos` y también del manejo de interrupciones, pues de hecho modificaste el `interrupt handler` del `timer` como parte de tu solución.

Durante las sesiones de laboratorio hemos visto a grandes rasgos cómo funciona la implementación por omisión del `scheduler` de `pintos`, es un `Round Robin`, i.e. cada proceso tiene la misma cantidad de tiempo para ejecutarse dentro del sistema y son elegidos en orden `FIFO`. El objetivo de esta práctica es modificar el funcionamiento del `scheduler` de `pintos` de tal manera que ahora funcione como un `scheduler de prioridades`. Lo que buscamos es que el `scheduler` garantice que el proceso con mayor prioridad sea el que se esté ejecutando en todo momento. [1]

2.1 Scheduling en Pintos

En `pintos` el scheduling se lleva a cabo dentro del módulo de `threads`, i.e. dentro del archivo

[1] Existen algunos casos especiales en los que esto no es necesariamente cierto, pero de hecho son parte del desarrollo de ésta y de las siguientes prácticas

`thread.c`. La función encargada de realizar todos los pasos que implica una llamada a scheduler es `schedule()`, que por medio de la función `next_thread_to_run()` elige al siguiente proceso que recibirá tiempo de procesador.

Revisando el código podrás darte cuenta si hay procesos listos para ser ejecutado dentro de la lista `ready_list` entonces se escoge el que está al frente[2].

Las únicas funciones públicas del módulo de threads que permiten añadir procesos a la lista `ready_list` son `thread_unblock(...)` y `thread_yield()`. A continuación una breve descripción de cada una:

- `thread_unblock()`: Recibe como parámetro un apuntador a un proceso en estado `THREAD_BLOCK` y lo encola al final de la lista `ready_list`. Esta función puede ser llamada en todo momento, incluso en el contexto de interupciones externas, por ejemplo durante una llamada al `interrupt handler` del `timer` de `pintos`.
- `thread_yield()`: Fuerza un cambio de contexto, hace que el proceso actual ceda el procesador a alguien más dentro de la lista `ready_list`, para ello el proceso actual se encola dentro de dicha lista y después llama a la función schedule(). Es importante mencionar que esta función no puede ser llamada dentro de un `interrupt handler` pues estos deben de ejecutarse por completo antes de que cualquier otro proceso pueda ejecutarse, si quieres obtener el mismo resultado dentro de un `interrupt handler`, tienes que llamar a la función `intr_yield_on_return()`, que lo que hace es posponer la llamada de `thread_yield()` al momento en el que el `interrupt handler` termina de ejecutarse.

Dicho lo anterior las únicas funciones que afectan el contenido del `ready_list` son:

- `thread_unblock(...)`
- `thread_yield()`
- `next_thread_to_run()`

Como parte de tu solución, ahora estas tres funciones deben de asegurar que el `scheduler` se comporte como un `scheduler de prioridades`.

[2]Haciendo list_pop_front de la ready_list

3 Actividades

3.1 Base de código

Recibirás la misma base de código que para la práctica pasada, sólo el script `execute-tests` es actualizado para poder ejecutar las nuevas pruebas que necesitas pasar, sin embargo, tienes que utilizar la solución de tu práctica anterior también como base de esta práctica. De hecho en tu entrega tienes que enviar todos los archivos que modificaste tanto en esta práctica como en la anterior para que nosotros podamos evaluar tu solución en su totalidad.

3.2 Ajustes en la imagen de Docker

Actualizamos la imagen de Docker que utilizamos para compilar y ejecutar el código de las prácticas. Ejecuta el comando `./docker-exec pull-image` antes de .

3.3 Programación

Implementan un `scheduler de prioridades` en `pintos`, es decir, cuando un proceso es añadido a la `ready_list` y este tiene una prioridad mayor a la del proceso que se encuentra actualmente corriendo, el proceso actual debe "inmediatamente"[3] ceder el procesador al nuevo proceso. De forma similar, cuando un proceso está esperando en un candado, semáforo o variable de condición, el proceso en espera con la más alta prioridad debe de ser el primero en despertar. Un proceso puede incrementar o disminuir su propia prioridad en cualquier momento, pero si éste disminuye su prioridad y deja de tener la más alta priodad, debe de "inmediatamente" ceder el CPU.

La prioridad de los hilos de kernel va desde `PRI_MIN (0)` hasta `PRI_MAX (63)`. Los números más bajos corresponden a prioridades más bajas, es decir, la prioridad 0 es la más baja y 63 la más alta. La prioridad inicial de proceso es pasada como argumento a la función `thread_create (...)`. Si no hay otra razón para elegir otra prioridad, utiliza `PRI_DEFAULT (31)`. Los macros `PRI_*` están definidos en el archivo `threads/thread.h` y nunca debes de modificar estos valores.

Para esta práctica hay que hacer las bases del requerimiento `2.2.3 "Priority Scheduling"` del proyecto 01 de `pintos`, para ello limitaremos el conjunto pequeño de pruebas que tienes que pasar.

https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC26

Las pruebas que tendrás que pasar para saber que tu solución está completa, son las siguientes:

[3] Excepto si se encuentra atendiendo una interrupción externa/de hardware, en cuyo casi espera ha que la interrupción sea atendida y después hay un cambio de contexto.

- alarm-priority: Crea diversos threads y los pone a dormir, prueba que dichos threads se ejecuten en el orden que dicta su prioridad.

- priority-change: Se trata de modificar la función thread_set_priority(), de tal manera que si el proceso actual disminuye su prioridad y deja de ser el de mayor prioridad, cede el procesador al thread con mayor prioridad dentro de la lista ready_list.

- priority-preempt: Si el proceso actual crea otro proceso pero con mayor prioridad, tan pronto como sea posible el proceso actual cede el procesador a dicho proceso nuevo; y aunque el proceso nuevo fuerce llamadas al scheduler, éste es el que continua ejecutándose pues sigue siendo el proceso con mayor prioridad.

- priority-fifo: Si hay más de un proceso con la prioridad más alta del sistema, para ellos el scheduler se comporta como un `Round Robin`.

- priority-sema: Referente a semáforos, lo que prueba es que el thread de más alta prioridad que está esperando en un semáforo sea el priomero en salir de su lista de espera.

- priority-condvar[4]: Referente a variables condicionales, una variable condicional también tiene una lista de espera, igual que para la prueba anterior el proceso que debe de abandonar primero la lista de espera de la variable condicional debe de ser siempre el de mayor prioridad.

Al igual que en la práctica anterior, proveemos el script de elevaluación `src/threads/execute-tests`, que ejecutará todas las pruebas de esta práctica y de la anterior, es decir, todas las pruebas que hemos resuelto del proyecto 01.

A continuación la lista de archivos que podrías modificar como parte de tu desarrollo:

- `threads/thread.h`, `threads/thread.c` : Código referente al módulo de `threads` de `pintos`.

- `threads/sync.h`, `threads/sync.h`: Código con la definición e implementación de semáforos, candados y variables de condición para el uso de ellos por el módulo de `threads`.

Recomendamos ampliamente que leas el requerimiento dentro del sitio de Pintos para tener un mejor entendimiento de lo que tienes que hacer. También sugerimos que intentes pasar las pruebas en el orden en el que las estamos enunciando arriba.

[4]Esta prueba es opcional pero si la sacan tendrá un punto extra

Si haces que tu solución meta y saque procesos de la estructura que representa la `ready_list` en tiempo constante y lo justificas bien en tu documento de diseño tendrás un punto extra.

3.4 Documento de Diseño

Igual que en la práctica anterior tendrás que entregar el documento de diseño asociado a tu solución, de nuevo entregamos este documento junto con el archivo `DesignDOC.txt`.

4 Hints

- El proceso `idle_thread` es un caso especial, de hecho tiene prioridad 0; cuando él desbloquea a otros procesos por medio de `thread_unblock(...)`, deja que éste termine su ejecución, es decir, no fuerces un cambio de contexto aunque los procesos que está desbloqueando tengan mayor priodad que él, de otra manera el sistema operativo no podrá iniciar.
- Dado que las prioridades están acotadas entre 0 y 63, es posible utilizar una estructura de datos que en tiempo constante nos diga en qué nivel vamos a insertar un proceso en la `ready_list` de acuerdo a su prioridad.
- Otra opción que ya conoces, es mantener la `ready_list` ordenada, sin embargo insertar en una lista ordenada toma tiempo lineal.
- Ten cuidado cuando quieras forzar un cambio de contexto, pues en el caso de la función `thread_unblock(...)` nada impide que ésta sea llamada dentro del contexto de una interrupción externa; más aun dentro de tu solución a la práctica anterior haces llamadas a `thread_unblock(...)` dentro del `interrupt handler` del `timer`.
- Uno de los requerimientos para pasar algunas de las pruebas, es que cuando un proceso que está corriendo deje de ser el de más alta prioridad, fuerces un cambio de contexto. Lo anterior implica que al momento de que un proceso con mayor prioridad al actual es agregado a la `ready_list`, entonces debe de ceder el procesador, es decir, llama a`thread_yield(...)`. Lo mismo sucede si el proceso actual modifica su priodad por medio de `thread_set_priority(...)`, si este deja de ser el de mayor prioridad, también tiene que forzar un cambio de contexto por medio de `thread_yield(...)`.
- Dentro de `threads/interrupt.c` está la función `intr_context ()` que regresa `true` si el proceso actual está atendiendo la ejecución de una interrupción externa/de hardware, y regresa `false` en caso contrario.
- Atención con los `semáforos`, asegurate que antes de llamar a la función `thread_unblock(...)` estés incrementado el valor del semáforo, de otra manera el semáforo no incrementará su valor antes de que suceda un cambio de contexto.

5 Entrega

Tienes que entregar todos los archivos que modificaste como parte de tu solución (con git es facil darse cuenta que cambió), seguiendo la estructura también descrita en la práctica anterior, y sólo esos archivos.

De nuevo el documento de diseño vale la mitad de la calificación de la práctica.