Sistemas Operativos 2020-2, Práctica 04: Pintos, Advanced Scheduler

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: 31 de Mayo de 2020

1. Objetivo

El objetivo de esta práctica es que el alumno implemente una segunda versión de un scheduler de prioridades en el cual se elimine la hambruna (Starvation). La idea es simple, hay que aumentar paulatinamente la prioridad de los procesos de baja prioridad, y disminuir la de los procesos de alta prioridad y que consuman mucho tiempo del procesador. Al final eventualmente todos los procesos que puedan ejecutarse lo harán, dicha propiedad se llama fairness.

2. Introducción

La función de un `scheduler` de propósito general es balancear la necesidades de `scheduling` de diferentes procesos. Los procesos que realizan muchas operaciones de I/O requieren un tiempo de respuesta veloz para mantener los dispositivos de entrada y salida ocupados, pero necesitan poco tiempo de procesador. Por otro lado, procesos que utilizan de manera intensiva el procesador requieren que se les asigne el procesador durante periodos prolongados para poder finalizar su tarea. Otros procesos tienen un comportamiento híbrido, con periodos de uso de dispositivos de I/O, y también con picos de uso del procesador, y así estos tienen diferentes necesidades de `scheduling` a través del tiempo. Un scheduler bien diseñado con frecuencia puede manejar procesos con estos requerimientos simultáneamente.

Multiples facetas del `scheduler` requieren que la información de los procesos y del kernel mismo sea actualizada cada cierto número de ticks. En cada caso, estas actualizaciones deben de ocurrir antes de cualquier proceso tenga la oportunidad de ser ejecutado, de tal forma que no exista la posibilidad de que un proceso puede ver los viejos valores del scheduler una vez que haya transcurrido un tick nuevo.

2.1. Contexto del scheduler

El scheduler de prioridades que has implementado hasta el momento tiene un gran problema analizándolo desde el punnto de vista de en un sistemas operativos de propósito general, pues los procesos de baja prioridad podrían nunca obtener tiempo de procesador, dicho fenómeno es conocido como hambruna (Starvation).

Como parte de esta práctica tendrás que continuar la implementación del scheduler de prioridades en el que has trabajado pero ahora debes de hacer que sea más justo (fairness), es decir, los procesos de baja prioridad que no obtienen tiempo de procesador cada cierto tiempo incrementan su prioridad hasta que eventualmente reciben tiempo de procesador debido a su prioridad.

2.2. Fairness

Un scheduler tiene la propiedad fairness si este garantiza que eventualmente cada proceso recibe tiempo de procesador, esta es una propiedad deseable en el scheduler del kernel de un sistema operativo

de proposito general.

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 tus ultimas dos prácticas 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 las anteriores para que nosotros podamos evaluar tu solución en su totalidad.

3.2. Programación

En esta práctica tendrás que implementar la solución del requerimiento "2.2.4 Advanced Scheduler" del proyecto 01 de Pintos.

https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC27

Debes de adecuar tu código para que permita dinámicamente elegir la algoritmo de scheduling de pintos al momento de iniciarlo. Por omisión, el scheduler de prioridades debe de estar activo, pero debemos de ser capaces de activar el `4.4BSD Scheduler` con la opción de kernel `-mlfqs`. Pasar esta opción pone el valor de la variable global `thread_mlfqs` a `true`, esta variable está declarada en `threads/thread.h`, esta opción es detectada por medio de la función `parse_options()`, y esta función es llamada al inicio de la ejecución de `main()`.

Cuando el `4.4BSD Scheduler` está habilitado, los procesos dejan de tener control directo sobre su propia prioridad. El argumento `priority` de `thread_create()` debe de ser ignorado, así como también cualquier llamada a la función `thread_set_priority()`, y la función `thread_get_priority()` debe de regresar la prioridad calculada por el `scheduler`.

Las pruebas que tendrás que pasar como parte de tu solución son:

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

La descripción completa del scheduler que tienes que implementar se encuentra dentro del apéndice "4.4BSD Scheduler" en:

https://web.stanford.edu/class/cs140/projects/pintos/pintos_7.html#SEC131%20-0

3.3. Implementación de Operaciones Aritméticas de Punto Fijo

Básicamente no puedes hacer ningún cálculo para el nuevo `scheduler` si no implementas las operaciones necesarias para manejar números reales con representación de punto fijo. Hay varias maneras de hacer dicha implementación, pero quizás la más sencilla y eficiente se hace en un archivo `.h` y utilizando sólo macros; otra opción es añadir un archivo `.c` con dichas operaciones, sólo hay que tener cuidado de añadir dicho archivo al makefile correspondiente, porque si no nunca va a ser compilado.

3.4. Documento de Diseño

Nuevamente recibirás este documento junto con el DesignDOC.txt que contiene las preguntas y detalles acerca de la implementación que realizarás, recuerda que dicho documento tiene el mismo peso en la calificación que la solución que dés en código al problema en cuestión, así que invierte tiempo suficiente en contestarlo.

4. Notas

4.1. Considederaciones sobre el valor nice

1. El valor nice es un número entero entre -20 y 20, y dicho valor es local para cada proceso.
2. El thread inicial comienza con un valor nice de 0. Cualquier otro thread hereda el valor nice del thread que lo creó.
3. Funciones a implementar:
   a) int thread_get_nice (void): Regresa el valor nice del thread actual.
   b) void thread_set_nice (int new_nice): Asigna un nuevo valor nice al thread actual y recalcula la prioridad del thread actual utilizando el nuevo valor nice asignado. Si el thread actual deja de ser el thread de mayor prioridad entonces cede el procesador al thread de más alta prioridad esperado en la lista ready_list.

4.2. Consideraciones para calcular la prioridad

1. La prioridad de un proceso es calculada al comienzo de su proceso de inicialización, i.e. al comienzo de init_thread(...).
2. La prioridad de cada proceso tiene que ser recalculada cada cuatro ticks.
3. En cualquier situación que implique recalcular la prioridad de un proceso la fórmula que determinará la nueva prioridad es: priority = PRI_MAX - (recent_cpu/4) - (nice*2)
   a) recent_cpu es una estimación del tiempo de procesador que el thread ha utilizado recientemente.
   b) El resultado de la fórmula debe de ser truncado al mayor entero menor o igual.
   c) Esta fórmula da menor prioridad a los threads que recientemente han recibido tiempo de procesador y da mayor prioridad a los que no han recibido tiempo de procesador recientemente, esta es la clave para evitar que en el kernel haya starvation.
4. La prioridad resultante del cálculo tiene que ser ajustada de tal manera que siempre se encuentre en el rango PRI_MIN hasta PRI_MAX.

4.3. Consideraciones sobre el valor recent_cpu

1. El valor recent_cpu es un número real local para cada proceso, e indica cuanto tiempo de procesador ha obtenido recientemente dicho proceso.
2. El valor inicial de recent_cpu es cero para thread inicial, y para el resto de los threads es valor del thread que lo creo.
3. Cada vez que una interrupción del timer ocurre, recent_cpu incrementa su valor en uno para el thread en ejecución, a menos que éste sea el thread idle.
4. Una vez cada segundo debe de ser recalculado el valor recent_cpu para cada proceso con la fórmula:
   recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice

5. El efecto es que el valor recent_cpu estima la cantidad de tiempo de procesador que el proceso ha recibido recientemente, con un rango de decadencia inversamente proporcional al número de threads que compiten por el procesador.

6. El valor recent_cpu puede ser negativo para un thread con valor nice negativo. No consideres que valores negativos de recent_cpu son 0.

7. Implementa la función

   a) thread_get_recent_cpu(): Regresa el valor recent_cpu del thread actual multiplicado por 100 y redondeado al entero más cercano.

4.4. Consideraciones sobre el cálculo del valor load_avg

1. El valor load_avg es global y es un número real (global para los procesos).

2. El valor load_avg es frecuentemente asociado a la carga promedio del sistema. Estima el número promedio de procesos listos para ejecutarse en el último segundo de tiempo.

3. Durante el proceso de inicialización del sistema (system boot), load_avg debe de ser inicializado a 0.

4. Una vez cada segudo, load_avg es recalculado por medio de la siguiente fórmula
   load_avg = (59/60)*load_avg + (1/60)*ready_threads

   a) ready_threads: es el número total de threads en estado ready o running.

5. Implementa la función:

   a) thread_get_load_avg(): Regresa el valor actual del siustema de load_avg multiplicado por 100 y redondeado al entero más cercano.

4.5. Más ...

- Podemos saber cuando ha trancurrido un segundo cada vez que timer_ticks () % TIMER_FREQ == 0
- Cada vez que actualices un valor local o global para los procesos revisa que hayas actualizado primero los valores de los cuales depende. Por ejemplo para actualizar la prioridad de un thread es necesario que primero actualices su valor de recent_cpu.
- Si no tienes una solución para la primera práctica del proyecto uno de Pintos, i.e. eliminar la espera ocupada de la función timer_sleep (), entonces nada va a funcionar pues conceptualmente rompe la intención de los cálculos que involucra el reajuste de prioridades scheduler.

5. Hints

1. La función thread_tick ()[1] es invocada durante cada interrupción del timer como parte del interrupt handler correspondiente. Esta función es el lugar indicado para modificar todos los valores que necesita actualizar el nuevo scheduler.

2. Los valores load_avg y recent_cpu son números reales, así que para poder calcularlos necesitas implementar el simulador de operaciones de números de punto fijo 17.14. Podrías incluso realizar dicha implementación por fuera del Código de Pintos y una vez que estés seguro que funciona utilizarla dentro de Pintos. Te sugerimos hacer pruebas unitarias de la implementación que hagas de números de punto fijo, te recomandamos utilizar operaciones bitwise siempre que sea posible, pues son más eficientes.

[1] Esta función está en thread.c

3. Puedes mantener siempre la cuenta de número ready_threads de tal manera que lo pueda utilizar directamente dentro del cálculo del valor load_avg. Aunque si no quieres enredarte en el intento, mejor cuenta el número de threads dentro de la ready_list y súmale uno para contar al thread actual en ejecución.

4. La clave para pasar todo las pruebas es hacer que la latencia del interrupt handler del timer sea baja, y que te asegures bien de hacer las cuentas.

5. Cuando la prioridad de un thread en estado THREAD_READY cambia durante durante una interrupción del timer, ¿cómo asegurarás que cada thread sigue estándo en su la lista del nivel que le corresponde?. ¿aplica lo mismo para threads en estado THREAD_BLOCKED?

6. Entrega

Sigue los lineamientos de entrega y recuerda sólo entregar los archivos que conforman tu solución junto con tu `DesignDoc`.