Evaluación 5 Sistemas Operativos I

Procesos

Puntaje máximo: 20

En este proyecto se trabajará directamente con el planificador de Pintos, modificándolo y hasta reemplazándolo por un planificador más avanzado.

1. Introducción

1.1. Hilos

El primer paso es leer y entender el código del sistema de hilos inicial. Pintos ya implementa la creación y completamiento de hilos, un planificador sencillo para hacer cambios de contextos entre hilos, y primitivas de sincronización (semáforos, locks, variables de condición y barreras de optimización).

Parte del este código puede parecer misterioso. Si no se ha compilado y corrido el sistema base, debe hacerlo. Puede leer el código y ver qué sucede. Si lo desea, puedes añadir llamadas a printf() casi en cualquier lugar, y luego recompilar y correr para ver qué sucede y en qué orden. Puede también correr el kernel en modo de depuración y colocar breakpoints en puntos que parezcan interesantes, y examinar la data, etc.

Cuando un hilo es creado, se ha creado un nuevo contexto para ser planificado. Se provee una función para ser ejecutada en este contexto, pasándola como argumento a thread_create(). La primera vez que un hilo es planificado y ejecutado, inicia del principio de la función y ejecuta dentro de ese contexto. Cuando una función termina, el hilo termina. Cada hilo, por tanto, actúa como un mini programa que corre dentro de Pintos, siendo la función pasada a thread_create() la que actúa como el main()[1].

En cualquier momento, exactamente un solo hilo corre y el resto se pone inactivo. El planificador decide cuál hilo correr como próximo. Si ningún hilo está listo para ser ejecutado en cualquier

[1] Punto de entrada.

momento, entonces se ejecuta un hilo especial, idle(). Las primitivas de sincronización pueden forzar cambios de contexto cuando un hilo necesita esperar que otro haga algo adicional.

La mecánica de un cambio de contexto se puede ver en threads/switch.S, que es código x86. En resumen, guarda el estado del hilo actualmente ejecutándose y reestablece el estado del hilo hacia cual se cambia.

Utilizando el depurador GDB, lentamente dé seguimiento a un cambio de contexto para ver qué sucede (revise la sección GDB de la documentación oficial de Pintos). Puede colocar un breakpoint en schedule() para iniciar, y luego continuar paso a paso a partir de ahí. Asegúrese de dar seguimiento a la dirección y estado de cada hilo, y qué procedimientos se encuentran en el call stack para cada hilo. Notará que cuando un hilo llama switch_threads(), otro hilo inicia a correr, y la primera cosa que el nuevo hilo hace es retornar de switch_threads(). Entenderá el sistema de hilos una vez entienda por qué y cómo el switch_threads() que es llamado es distinto al switch_threads() que retorna. Revise la sección Thread Switching, de la documentación oficial de Pintos, para más información.

1.2. Sincronización

La sincronización apropiada es una parte importante de las soluciones a los problemas de concurrencia. Cualquier problema de sincronización puede resolverse fácilmente apagando interruptores: mientras los interruptores están apagados, no hay concurrencia, por lo que no pueden existir condiciones competitivas sobre recursos. Por tanto, es muy tentador resolver todos los problemas de sincronización de esta forma, pero no lo haga. En cambio, utilice semáforos, locks, y variables de condición para resolver la gran mayoría de los problemas de sincronización.

En los proyectos Pintos, la única clase de problemas que son resueltos apagando interruptores es la de coordinar data compartida entre un hilo de kernel y un manejador de interrupción. Debido a que los manejadores de interrupción no pueden dormir, ellos no pueden usar locks. Esto significa que la data compartida entre hilos de kernel y manejadores de interrupciones debe ser protegida dentro de un hilo de kernel a través de apagar interruptores.

En este proyecto sólo necesita acceder a un poco del estado de hilo desde los manejadores de interrupciones. Para el reloj de alarma, el interruptor de tiempo necesita despertar cuando un hilo duerme. En el planificador avanzado, el interruptor de tiempo necesita acceder a unas pocas variables globales y por-hilo. Cuando se accede a estas variables globales desde hilos de kernel, necesita apagar interruptores para prevenir que el interruptor de tiempo interrumpa la correcta ejecución.

Cuando efectivamente apague interruptores, asegúrese de tener cuidado y hacerlo empleando el menor código posible, o pudiera conseguir perder cosas importantes como ticks de reloj o eventos de entrada. Apagar los interruptores también incrementa la latencia de manejo de interruptores, lo cual puede hacer que la máquina se sienta “rara” si durara mucho.

Las primitivas de sincronización en synch.c se implementan apagando interruptores. Puede necesitar incrementar la cantidad de código que corre cuando los interruptores son desactivados, pero debe de todas formas tratar de mantenerlo a un mínimo.

Apagar interruptores puede ser útil al momento de depurar si quiere, por ejemplo, asegurarse que una sección de código NO es interrumpida. Debería remover el código de depuración antes de entregar el proyecto. *No se limite a comentarlo, porque realmente esto causa que el código sea más difícil de leer.*

Tampoco debería hacer busy waiting en su envío. Tenga en cuenta que un loop que llame a thread_yield() es una forma de busy waiting.

2. Productos

Todas las notas están en porciento, que se ajustan al máximo de puntos para dar la nota final.

2.1. Documento de diseño [35 %]

Al final de este documento se encuentra una plantilla de documento de diseño. Esta plantilla debe ser completada. La plantilla completada debe remitirse como un pdf de nombre *design.pdf* precedida por su matrícula sin guiones.

Es recomendable leer y completar lo esencial *antes* de escribir código.

2.2. Reloj de alarma [25 %]

Reimplemente timer_sleep(), definido en devices/timer.c. A pesar de que se suministra una implementación funcional, esta se basa en busy waiting, y ya se comentó sobre busy waiting ... Haga una versión de esta que NO dependa de busy waiting.

Tenga en cuenta que la función **void timer_sleep(int64_t ticks)** suspende la ejecución del hilo que llama hasta que el tiempo haya avanzado por al menos x ticks de reloj. A menos que el sistema esté sin trabajo, el hilo no necesita despertar luego de exactamente x ticks. Basta ponerlo en la cola de listo luego de haber esperado el tiempo requerido.

timer_sleep() es útil para hilos que operan en tiempo real, ejemplo: el parpadeo del cursor en una consola.

El argumento a timer_sleep() se expresa en ticks de reloj, no en milisegundos ni en ninguna otra unidad. Deben haber TIMER_FREQ ticks por segundo (devices/timer.h). El valor por defecto es 100. No es recomendable cambiar este valor, ya que cualquier cambio puede causar que muchos tests fallen.

Otras funciones como timer_msleep(), timer_usleep() y timer_nsleep() existen para dormir durante un número específico de milisegundos, microsegundos o nanosegundos respectivamente pero estas dependen todas de timer_sleep(). Obviamente, no las tiene que modificar.

Si nota algo extraño con el tiempo de delay, revise la opción -r en el programa pintos.

2.3. Planificación por prioridad [30 %]

Implemente planificación por prioridad en Pintos. Cuando un hilo es añadido a la lista de procesos listos y este tiene una prioridad superior a la actualmente en ejecución, el hilo actual debe entregar automáticamente el procesador al nuevo hilo. Similarmente, cuando hilos están esperando por un lock, semáforo o variable de condición, el hilo de mayor prioridad debe ser el primero despertado. Un hilo puede incrementar o reducir su propia prioridad en cualquier momento, pero reducirla de tal manera que cuando ya no tenga la prioridad más alta debe entregar inmediatamente el CPU.

Las prioridades de los hilos oscilan de PRI_MIN(0) a PRI_MAX(63). Los números más bajos corresponden a prioridades inferiores. Por tanto, 0 es la prioridad menor y 63 la mayor. La prioridad inicial de un hilo es pasada como argumento a thread_create(). Si no hay razón para elegir otra prioridad, debe utilizar PRI_DEFAULT (31). Estas macros se encuentran definidas en threads/thread.h, y no debería modificarlas.

Una situación con la planificación por prioridad es la "inversión de prioridad", que deberá tener en cuenta. Considere hilos de prioridad alta, media y baja, H, M, L respectivamente. Si H necesita esperar por L, (por ejemplo, por un lock sostenido por L), y M está en la lista de listos, entonces H nunca conseguirá el CPU debido a que L no conseguirá tiempo de CPU. Una solución parcial a este problema es que H "done" prioridad a L mientras L mantiene el reloj, y luego reclamar la devolución de lo donado cuando L entregue el lock (y H lo adquiera).

Implemente donación por prioridad. Necesitará tomar en consideración las distintas situaciones en las que se requiere donación por prioridad. Sea cuidadoso en el análisis.

Asegúrese manejar múltiples donaciones, en las cuales múltiples prioridades son donadas a un único hilo. Debe también manejar la donación anidada: si H está esperando por un lock que M tiene y M está esperando por un lock que L tiene, entonces M y L deben ser "ascendidas" a la prioridad de H. Si es necesario, puede imponer un límite razonable en la profundidad de la donación de prioridad anidada, por ejemplo de 8 niveles.

Debe implementar donación de prioridad para locks. No necesita implementar donación de prioridad para otras construcciones de sincronización de Pintos. No necesita implementar planificación de prioridad en todos los casos.

Finalmente, implemente las siguientes funciones que permiten que un hilo examine y modifique su propia prioridad. Esqueletos de estas funciones se encuentran en threads/thread.c A continuación, una descripción breve de ellas.

- *void thread_set_priority (int new_priority)*
  Configura la prioridad del hilo actual a new_priority. Si el hilo actual no tiene la prioridad más alta, entonces simplemente entrega el CPU.
- *int thread_get_priority (void)*
  Retorna la prioridad del hilo actual. En la situación de donación de prioridad, retorna la prioridad (donada) más alta. No necesita proveer ninguna interfaz para permitir a un hilo directamente modificar las prioridades de otros hilos.

2.4. Planificador avanzado [10 %]

Implemente un planificador basado en colas multinivel de retroalimentación (multilevel feedback queue)Consultar *Silberschatz, Abraham et al. (2012). Operating System Concepts (9th).New York: Wiley. Epig. 6.3.3, 6.3.6, 6.7.3.*

Similar al planificador de prioridad, el planificador avanzado elige el hilo a correr basado en prioridad. Sin embargo, el planificador avanzado no hace donación de prioridad. Por tanto, recomendamos que primero enfocarse en que el planificador de prioridad funcione, a excepción posiblemente de la donación de prioridad, antes de iniciar trabajo con el planificador avanzado.

Debe escribir su código para permitir elegir la política de planificación de Pintos en TIEMPO DE INICIO. Por defecto, el planificador de prioridad debe estar activo, pero se debe ser capaz de elegir entre ellos a través de la opción -mlfqs de kernel. Pasar esta opción configura thread_mlfqs, declarado en threads/thread.h, al valor verdadero cuando las opciones son evaluadas por parse_options(), que ocurre al principio en main().

Cuando el planificador avanzado es habilitado, los hilos ya no controlan directamente sus propias prioridades. El argumento de prioridad a thread_create() DEBE ser ignorado, así como cualquier llamada a thread_set_priority() y thread_get_priority(). Esta última debe devolver la prioridad configurada por el planificador.

3. Entrega

Todos los productos anteriores deben ser remitidos a la cuenta de correo del profesor en la PUCMM desde su cuenta de correo de estudiante. El nombre del fichero empaquetado *tar.xy* tendrá por nombre *<matrícula sin guión>PDE5.tar.xy* donde debe reemplazar *<matrícula sin guión>* por su matrícula sin guión intermedio.

Por ejemplo, si su matrícula fuese *2021-0777*, el fichero debería llamarse *20210777PDE5.tar.xy* El asunto del correo, siguiendo con la misma matrícula como ejemplo, sería *[20210777] ISC-364 PDE5. Procesos.*

Plantilla de diseño

ISC 364
PROYECTO: PROCESOS
DOCUMENTO DE DISEÑO

ASUNTOS PRELIMINARES

Citar aquí cualquier fuente en línea o fuera de línea consultada para la preparación de su entrega, más allá de la documentación de Pintos, libro de texto, clase, etc.

RELOJ DE ALARMA

ESTRUCTURAS DE DATOS

A1: Copiar aquí la declaración de cada struct nueva o modificada, variable global o estática, "typedef", o enumeración. En 30 palabras o menos, identifique el propósito.

ALGORITMOS

A2: Brevemente describa qué sucede en una llamada a timer_sleep(), incluyendo los efectos del manejador de interrupciones del timer.

SINCRONIZACIÓN

A4: ¿Cómo se evitan las condiciones de competencia cuando múltiples hilos llaman a timer_sleep() simultáneamente?

DECISIONES DE DISEÑO

A6: ¿Por qué se eligió este diseño? ¿En qué formas es superior a otro diseño considerado?

PLANIFICACIÓN POR PRIORIDAD

ESTRUCTURAS DE DATOS

B1: Copiar aquí la declaración de cada struct nueva modificada, variable global o estática, 'typedef', o enumeración. En 30 palabras o menos, identifique el propósito.

B2: Explique la estructura de datos utilizada para dar seguimiento a la donación de prioridad. Utilice arte ASCII para diagramar una donación anidada.

ALGORITMOS

B3: ¿Cómo se asegura que el hilo de mayor prioridad esperando un lock, semáforo, o variable condicional sea el que se despierte primero?

B4: Describir la secuencia de eventos cuando una llamada a lock_acquire() causa una donación de prioridad. ¿Cómo se maneja esta donación anidada?

B5: Describir la secuencia de eventos cuando lock_release() es llamado en un lock que un hilo de mayor prioridad está esperando.

SINCRONIZACIÓN

B6: Describir una condición de carrera potencial en thread_set_priority() y explicar cómo su implementación la evita. ¿Se puede utilizar un lock para evitar esta carrera?

DECISIONES DE DISEÑO

B7: ¿Por qué se eligió este diseño? ¿En qué formas es superior a otro diseño considerado?

PLANIFICADOR AVANZADO

ESTRUCTURAS DE DATOS

C1: Copie aquí las declaraciones de cada nueva o modificada “struct”, variable global o estática, “typedef”, o enumeración. Identifique su propósito en 25 palabras o menos.

ALGORITMOS

C2: Suponga que los hilos A, B y C tienen valores nice de 0, 1 y 2 respectivamente. Cada uno tiene un valor ‘recent_cpu‘ de 0. Llene la siguiente tabla mostrando la decisión de planificación, la prioridad y valor ‘recent_cpu‘ para cada hilo luego de los ticks de reloj dados:

tick	recent_cpu			prioridad			thread
reloj	A	B	C	A	B	C	a ejecutar
0	0	0	0	63	61	59	A
4							
8							
12							
16							
20							
24							
28							
32							
36							

C3: ¿Alguna ambigüedad en la especificación del planificador hace que los valores en la tablas sean inciertos? Si es así, ¿qué regla utilizó para resolver estas ambigüedades? ¿Esto es compatible con el comportamiento de su planificador?

C4: Considere la manera en la que divide el costo de planificación entre código dentro y fuera del contexto de interrupción. ¿Qué tan probable es que esta afecte el rendimiento?

JUSTIFICACIÓN

C5: Critique brevemente su diseño, apuntando ventajas y desventajas en sus decisiones. Si tuviera tiempo extra para trabajar en esta parte del proyecto, ¿cómo elegiría refinar o mejorar su diseño?