Sistemas Operativos 2020-2, Práctica 03: Pintos, Interrupciones de Hardware.

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: 23 de Marzo de 2020

"It is not enough to do your best: you must know what to do, and THEN do your best".

W. Edwards *Deming*

Nota: Lee toda la práctica antes de comenzar cualquier actividad referente a lo descrito en este documento

1 Objetivo

El objetivo de esta práctica es introducir al estudiante al sistema de `hilos de kernel`[1] y al uso de interrupciones de hardware en `Pintos`.

2 Introducción

Para poder realizar esta práctica tendrás que conocer los servicios que ofrece el módulo de manejo de *hilos* en `Pintos` y cómo utilizarlos, en general como utilizar las funciones que mueven los hilos entre los estados THREAD_READY, THREAD_RUNING y THREAD_BLOCKED; de igual manera tendrás que aprender a utilizar la peliculuar implementación de listas doblemente ligadas que proporciona la API de Pintos. Finalmente necesitarás comprender ligeramente cómo funciona una interrupción de hardware.
Las partes del sitio de Pintos que tienes que revisar antes de hacer la práctica (y en general el proyecto 01) son:

1. **Introducción**: En esta sección encontrarás generalidades del tipo ¿cómo ejecutar Pintos?, ¿donde está cada módulo de Pintos en al estructura de directorios?, etc

   https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html#SEC1

2. **Referencia de Pintos**: Aquí encontrarás una descripción de cómo funciona Pintos, desde el proceso de Loading hasta el Manejo de Memoria Virtual. Para esta práctica es importante leer las siguientes secciones:

[1] Procesos

https://web.stanford.edu/class/cs140/projects/pintos/pintos_6.html#SEC90

(a) A.1 Loading

   i. A.1.1 The loader
   ii. A.1.3 High-level kernel initialization

(b) A.2 Threads (Revisa el código de Pintos mientras lees esta sección)

   i. A.2.1 struct thread
   ii. A.2.2 Thread Functions
   iii. A.2.3 Thread Switching

(c) A.3 Synchronization (Revisa el código de Pintos mientras lees esta sección)

   i. A.3.1 Disabling Interrupts,
   ii. A.4.3 External Interrupt Handling

(d) Para la parte de listas tienen que revisar el código de la implementación, la idea es entender cómo utilizar toda la insfraestructura que proporciona. Dentro del código del módulo de threads hay ejemplos concretos de cómo insertar y elimnar elementos de una listas por lo que quizás sea una buena guía. Los archivos que contienen la implementación/firmas de listas son[2]:

   i. src/lib/kernel/list.h
   ii. src/lib/kernel/list.c

3. **Herramientas de Debugging**: Aquí encontrarás las diferentes maneras de hacer debugging del código de `pintos`. Las secciones que recomendamos para esta práctica son:

   https://web.stanford.edu/class/cs140/projects/pintos/pintos_10.html#SEC145

   (a) E.1 printf()
   (b) E.2 ASSERT
   (c) E.3 Function and Parameter Attributes
   (d) E.4 Backtraces

[2]En el sitio de Pintos probablemente no encuentres información de cómo utilizar las listas, aunque dentro del código que las define/implementa encontrarás todo lo que necesitas para entender cómo utilizarlas.

3 Práctica

En esta práctica tendrás que implementar el segundo requerimiento (2.2.2) del proyecto 01 de Pintos.

https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC25

En general se trata de hacer una mejor implementación del la función timer_sleep(...) contenida en devices/timer.c.

3.1 ¿Qué hay que hacer?

3.1.1 Instalación de Pintos

Al igual que las prácticas anteriores, utilizarás una imagen de `docker` para poder compilar y ejecutar las pruebas y programas de `pintos`, por lo cual puedes ignorar las secciones de la documentación que explican cómo instalar el entorno de desarrollo necesario.
Los pasos para iniciar tu entorno de trabajo son las siguientes:

1. Instalar y ejecutar `docker` en tu computadoras.[3]
2. Posiciona tu terminal en el directorio `pintos` que descargaste junto con este documento.
3. Descargar la imagen base para ejecutar el proyecto:
   `./docker-exec pull-image`
   Opcionalmente, también puedes construir tu propia imagen utilizando:
   `./docker-exec build-image`
   Si decides utilizar la imagen que construiste, es necesario que modifiques el archivo `docker-exec` en la opción `enter-container`, para que utilice la imagen que construiste localmente.
4. Luego ingresa a la terminal bash del contenedor ejecutando el comando:
   `./docker-exec enter-container`
5. Una vez que estes dentro de la terminal, realiza las siguientes acciones para ejecutar `pintos`:
   (a) Posiciona la terminal en el directorio `threads`:
   `cd /pintos-2020-2/src/threads`
   (b) Compila el código de ese módulo:
   `make`
   (c) Nos movemos hacia el recien creado directorio build y ejecutamos una de las pruebas:
   `pintos run alarm-multiplo`[4]
   (d) Ejecuta todas las pruebas de esta práctica:
   `cd /pintos-2020-2/src/threads`
   `./execute-tests`

[3] Si hiciste las prácticas anteriores, ya deberías de tener instalado este programa.
[4] Por omisión la imagen de docker ejecuta las pruebas utilizando qemu

3.1.2 Programación

El requerimiento `2.2.2 "Alarm Clock"` se trata de reimplementar la función timer_sleep(...) definida en el archivo devices/timer.c. Hay una implementación por omisión de dicha funcion que trabaja correctamente, el problema es que gasta tiempo del procesador mientras el hilo que la llama está dormido, i.e. es espera ocupada. El objetivo de esta práctica es proveer una nueva implementación de timer_sleep(...) en la cual el proceso que la invoque no gaste tiempo de procesador mientras se encuentra dormido, i.e. si un proceso invoca a timer_sleep(...) entonces debe de dejar de ejecutarse y volver a una posible ejecución hasta que en realidad haya transcurrido al menos el número de ticks especificado a timer_sleep(...).
Para implementar tu solución sólo deberías de modificar algunos de los siguientes archivos:

- `threads/thread.h`
- `threads/thread.c`
- `devices/timer.h`
- `devices/timer.c`

En general por cuestiones de modularidad y para ahorrar espacio en el tamaño del `PCB`[5], quizás sea mejor sólo modificar `timer.c`, aunque no significa que no haya otras buenas soluciones que modifiquen otros archivos. Si NO modificas el `PCB`, i.e. struct thread como parte de tu solución, recibirás 1 punto extra.
Una manera de saber que tu implementación está bien es verificando que pase las siguientes pruebas dentro del proyecto 01:

- `alarm-single`
- `alarm-zero`
- `alarm-negative`
- `alarm-multiple`
- `alarm-simultaneous`

Para esto posiciona la terminal en el directorio `threads` y ejecuta el siguiente script:

```
./execute-tests
```

Deberás de ver las siguientes lineas en la salida del script para saber que tu implementación es correcta:

```
pass tests/threads/alarm-single
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/alarm-multiple
```

[5]Process Control Block

```
pass tests/threads/alarm-simultaneous
```

puedes ver cuales de las pruebas requeridas ya pasan conforme vayas haciendo tu implementación. Te recomendamos vayas verificando tu implementación considerando las pruebas en el orden mecionado arriba.

3.1.3 Documento de Diseño

El requerimiento 2.2.1 del proyecto 01 es hacer un documento de diseño, en realidad dicho documento no puede ser completado hasta que tengas una solución para cada uno de los siguientes requerimientos. Dado que sólo hay que dar una solución para el requerimiento `2.2.2 "Alarm Clock"`, entonces unicamente hay que llenar la sección correspondiente en el documento de diseño.
Junto con este documento distribuimos otro llamado `DesignDoc.txt`, que es el que tendrás que llenar y entregar como parte de tu solución.

3.2 Hints

- Recuerda que un thread T_i deja el contexto de ejecución cuando pasa al estado THREAD_BLOCK llamando la función thread_block(), y vuelve al contexto de ejecución cuando algún otro thread T_j llama sobre T_i la función thread_unblock(...).
- Analiza cómo el módulo de threads utiliza un ready_list para quitar y poner threads en ejecución, podrías utilizar una idea similar dentro de tu solución .
- Dentro de devices/timer.c hay una función que llamada timer_interrupt() que representa el interrupt handler del timer, el cual es invocado cada vez que sucede un tick.
- Para evitar problemas de concurrencia entre una función y un interrupt handler de un dispositivo externo es necesario apagar las interrupciones en la función, para lograr esto es necesario enmarcar el código en un segmento como el siguiente:

```
old_level = intr_disable ();
... Aqui va todo el codigo que puede tener
... problemas de concurrencia con una interrupt handler
intr_set_level (old_level);
```

3.3 Notas

- Cuando las interrupciones de hardware están apagadas en el sistema, no hay concurrencia, i.e. sólo el thread de kernel que la atiende se ejecuta y nadie lo puede interrumpir hasta que termine, o cual evidentemente detiene al resto de los procesos.
- Los manejadores de interrupciones externas se ejecutan con las interrupciones apagadas, i.e. sólo un manejardor se ejecuta al mismo tiempo y de hecho no hay más computo efectivo hasta que dicho manejador termina, por tanto los manejadores deben de ser tan breves como sea posible, pues en realiadad no representan ninguna tarea de ningún proceso activo.

- Sólo como complemento al punto anterior, en general es importante matener implementaciones tan eficientes como sea posible dentro del kernel de un sistema operativo, ya que es éste el que utilizan todos los programas de usuario como intermediario ante el hardware de la máquina.
- Utilicen las ideas que vimos en el laboratorio, en general, insertar ordenado en la lista de dormidos y sólo preguntar por el/los proceso(s) al frente de la lista dentro del interrupt handler del timer.
- Sería muy recomendable que utilices un sistema de control de versiones como `git` para hacer cambios versionados dentro de tu código, es necesario hacer una inversión de tiempo para aprender las bases de `git`, pero al final ahorrarás mucho cuando tengas que corregir y quieras regresar a una versión anterior de tu código que sí funciona.

3.4 Entrega

Para la entrega de tu solución sólo requerimos que envies los archivos que modificaste como parte de tu solución. Por ejemplo supongamos que sólo modificaste el archivos `threads/thread.h` y `devices/timer.c`. Entonces tu entrega debe de ser un directorio **src** con la siguiente estructura:

```
src
|__threads
|   |__thread.h
|
|__devices
    |__timer.c
```

El directorio comprimido que tienes que entregar debe de tener una estructura como a la siguiente

```
practica_03
|__DesignDOC.(txt|pdf)
|__src
```

NOTA: El documento de diseño vale el 50% de la calificación de la práctica, así que escríbelo con tanto cuidado como el código mismo.

4 FAQ

1. Para ejecutar una prueba particular con puedes utilizar una instrucción como la siguiente:

   (a) `cd /pintos-2020-2/src/threads`
   (b) `make`
   (c) `pintos run alarm-single`

2. Para ejecutar todas las pruebas que tienes que pasar en esta práctica, ejecuta los siguientes comandos:

(a) `cd /pintos-2020-2/src/threads`

(b) `./execute-tests`[6]

3. El código de las pruebas de `pintos` se encuentra dentro del directorio src/test, para esta parte del proyecto 01 están dentro del directorio src/test/threads, allí podrás ver el código de las pruebas que tienes que pasar.

4. Podrías también leer el FAQ del proyecto 01.

[6]Este script lo creamos nosotros, para que solamente se ejecuten las pruebas requeridas en esta práctica, de otra manera hay que ejecutar el comando "make check SIMULATOR=--qemu" que ejecutaría todas las pruebas de la proyecto 01.