Práctica 1: Timer sleep

David Flores Peñaloza (dflorespenaloza@gmail.com)
Jorge Luis García Flores (jorgel_garciaf@ciencias.unam.mx)
Angel Renato Zamudio Malagón (renatiux@gmail.com)

Fecha de entrega: Domingo 18 de febrero de 2018

1. Hilos de ejecución (threads)

1.1. La definición de thread

El objetivo principal de un sistema operativo es gestionar los recursos de una arquitectura específica. Los recursos más importantes son el tiempo de ejecución (CPU) y la memoria principal (RAM), y aunque en una arquitectura basada en una máquina de Turing no puede existir un recurso sin el otro, se considera que el recurso más valioso es el tiempo de ejecución. Existen muchas formas de administrar el tiempo de ejecución, sin embargo la más utilizada para arquitecturas de este tipo se apoya del concepto de *thread* o hilo de ejecución. En términos simplistas un thread es la unidad mínima de procesamiento que el sistema operativo puede gestionar.

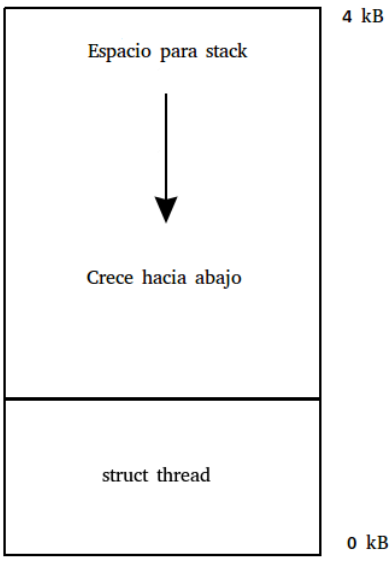
El primer paso es definir de forma puntual cuáles son las características que debe tener un thread, para ello debemos definir el *process control block* o bloque de control de proceso (PCB), en el caso de Pintos se encuentra definido en el archivo *src/threads/thread.h* de la siguiente forma:

```
1  struct thread
2  {
3      tid_t tid;                  /* Identificador del thread */
4      enum thread_status status;  /* Estado del thread. */
5      char name[16];              /* Nombre */
6      uint8_t *stack;             /* Stack pointer. */
7      int priority;               /* Prioridad. */
8      struct list_elem allelem;   /* Nodo para all_list */
9
10     struct list_elem elem;      /* Nodo para ready_list */
11
12 #ifdef USERPROG
13     uint32_t *pagedir;          /* Tabla de paginacion. */
14 #endif
15
16     unsigned magic;             /* Para detectar stack overflow. */
17 };
```

Una vez definido el bloque de control de proceso es necesario definir en qué lugar de la memoria se van a alojar dichas estructuras, también es necesario definir el esquema que se va a utilizar para almacenar variables locales y como se van a hacer los saltos entre funciones. Aunque prácticamente todos los sitemas operativos utilizan un sistema de variables y saltos de función basados en *stack*, es pertinente señalar que en las etapas iniciales del diseño del sistema operativo es posible optar por alugun otro esquema.

En el esquema basado en *stack* todos los threads compraten el mismo código de ejecución y utilizan un segmento de memoria propio de cada thread donde almacenan las variables locales y guardan registro de los saltos entre funciones, a este segmento de memoria se le conoce como *stack*. En el caso particular de Pintos, cada thread se aloja en un segmento de memoria de 4KB en el cual se almacena el

bloque de control del proceso y el *stack*, el siguiente diagrama muestra a grandes rasgos su estructura.



Como se puede apreciar en el diagrama anterior, el bloque de control de proceso (*struct thread*) se coloca al inicio del bloque, el resto del espacio es utilizado para almacenar el *stack* de ese thread. El bloque de control de proceso se coloca en forma inversa a como se definió, de tal forma que el campo *tid* se almacena en el fondo del bloque (la posición cero) y le campo *magic* se almacena en el límite del área correspondiente al *stack*. Si el campo *magic* se inicializa con un valor constante y el *stack* se desborda, el campo *magic* se sobre escribe y deja de contener el valor inicial, comprobando el valor del campo *magic* podemos revisar si el *stack* se desborda.

1.2. El calendarizador (scheduler)

Una vez que definimos de forma concreta lo que es un *thread*, tenemos que diseñar el mecanismo para asignar el tiempo de procesamiento que cada *thread* debe utilizar, a éste mecanismos se le conoce como calendarizador o *scheduler*. El calendarizador debe ser capaz de dar el control del procesador a un *thread* durante un tiempo determinado y cuando el tiempo termine entonces quitarle el control para darlo a otro thread, para conseguir éste objetivo en necesario tener una infraestructura que nos permita administrar a los *threads*.

De momento lo único que debemos saber del calendarizador es que se ejecuta a intervalos regulares para cambiar el thread que se encuentra en ejecución por otro thread. Para que un thread de kernel sepa cuando se terminó su tiempo de ejecución es necesario hacer uso del *hardware*, en este caso utilizamos un dispositivo llamado *timer*. El *timer* es un dispositivo que genera un interrupción hacia el procesador cada determinado tiempo, el tiempo se configura al iniciar el sistema operativo y en el caso de pintos es de 10 milisegundos. A dicho intervalo de tiempo se le conoce como *TICK*.

Como se mencionó anteriormente la principal función del dispositivo *timer* es notificar al thread en ejecución cuando su tiempo de ejecución ha terminado. Sin embargo el controlador del *timer* también realiza otras funciones, un caso particular es la función *timer_sleep* la cual se encarga de dormir al thread que la invoca un número específico de *ticks* (un *tick* es el intervalo de tiempo que tarda el *timer* en interrumpir al procesador). A continuación se muestra la función.

```
1 void timer_sleep (int64_t ticks)
2 {
3   int64_t start = timer_ticks ();
4
```

```
5   ASSERT (intr_get_level () == INTR_ON);
6   while (timer_elapsed (start) < ticks)
7     thread_yield ();
8 }
```

Lo que hace la función es guardar registro del *tick* en el que se invocó la función y después entra en un ciclo para revisar si ha pasado el tiempo necesario, en cada ejecución de ciclo se invoca a la función *thread_yield*, esta función forma al thread actual en la *ready_list* y pone en ejecución a otro thread. Eventualmente el thread que se formó se volverá a ejecutar por la acción del calendarizador, en ese momento volvera a revisar si el tiempo que pasó es igual al que el thread necesitaba estar inactivo, si esto sucedió, el ciclo se rompe y el thread continua su ejecución, en caso contrario el thread vuelve a invocar *thread_yield*.

El procedimiento anterior es una forma muy sencilla de bloquear a un thread determinado tiempo, a las estrategias de éste tipo se les conoce como espera ocupada (*busy wait*) pues lo que hacen es esperar a que ocurra algo pero periódicamente revisan si la condición se cumple. Aunque esta estrategia es muy sencilla tiene varios inconvenientes:

1. Estrictamente hablando el thread nunca se bloquea.
2. El thread desperdicia tiempo de procesador solamente en preguntar si la condición se cumple.
3. Si hay muchos threads en la *ready_list* la activación del thread puede tardar mucho.

1.3. Requerimientos

Para esta práctica debes cambiar la implementación de *timer_sleep* para que no se utilice una espera ocupada. Es decir, es necesario bloquear realmente al thread en lugar de volver a formarlo.

1.3.1. Pruebas

1. alarm-single
2. alarm-multiple
3. alarm-simultaneous
4. alarm-zero
5. alarm-negative

1.3.2. Preguntas

1. ¿Por qué en Pintos los procesos de kernel se llaman *threads* en lugar de procesos?
2. ¿Para qué sirve el *stack* de un proceso?
3. En Pintos cada thread utiliza un bloque de 4KB, en el cual se almacena su *PCB* y el *stack* del thread. ¿Dónde se almacenan las variables globales?
4. Si suponemos que solamente existe un thread en ejecución y dicho thread se bloquea (utilizando la función *thread_block*). ¿De que forma se podrá despertar el thread si el único en ejecución?

1.3.3. Consejos

Es necesario reemplazar el ciclo que invoca a la función *thread_yield* por una única invocación a la función *thread_block*, la función *thread_block* funciona de forma similar a la función *thread_yield*, pues también invoca al calendarizador pero a diferencia de *thread_yield* el thread que invoca al calendarizador ya no ser forma en la *ready_list*, con lo cual se garantiza que el thread no volverá a ejecutarse hasta invocar *thread_unblock*.

Para poder despertar a cada thread en el momento adecuado es necesario tener acceso a su bloque de control de proceso (*struct thread*) y también es necesario conocer el tiempo que lleva dormido o que aun

le falta por despertar. Para tener acceso a los threads que se duermen en la función *timer_sleep* podemos utilizar una lista la cual debemos declarar en el archivo *timer.c* e inicializar en *timer_init*, después, se debe agregar el bloque de control de proceso a la lista justo antes de invocar *thread_block*. Para conocer el tiempo que le falta a cada thread para despertarse podemos modificar el bloque de control de proceso (*struct thread*) agregando un campo para guardar el tiempo, éste valor se debe asignar justo antes de bloquear el thread.

Una vez que tenemos a los threads dormidos en una lista y que conocemos el tiempo que se desean dormir, necesitamos despertar en el momento oportuno a cada uno de ellos. Necesitamos una sección de código que se ejecute cada *tick*, por ello debemos modificar la función *timer_interrupt*. En ésta función debemos recorrer la lista de threads dormidos y hacer lo siguiente:

1. Decrementar en uno la variable que guarda el tiempo que necesita dormir el thread.
2. Si el tiempo que necesita dormir el thread es cero, entonces debemos despertar al thread utilizando la función *thread_unblock*. También necesitamos eliminar al thread de la lista de dormidos.
3. Si el tiempo que necesita dormir el thread no es cero, continuamos con el siguiente thread.

Las listas en pintos se implementan en el archivo *src/lib/kernel/list.c*, en el archivo también podemos encontrar ejemplos de como utilizar una lista. Al momento de dormir al thread utilizando *thread_block* puede ocurrir un error pues es necesario ejecutar la función con las interrupciones apagadas. A continuación un ejemplo de como garantizar que las interrupciones están apagadas:

```
1 int old = intr_set_level(INTR_OFF);
2 thread_block();
3 intr_set_level(old);
```

1.3.4. Puntos extras

La implementación que se propone en la sección de consejos se puede mejorar en dos formas. En la primera, en vez de almacenar el tiempo que a cada thread le queda por dormir, almacenamos el *tick* exacto donde el thread se va a despertar e insertamos de forma ordenada, en función del *tick* donde se debe despertar el thread, en la lista de dormidos. De ésta forma los threads que se despertarán primero son aquellos que están al frente de la lista, por lo que no es necesario iterar sobre toda la lista de threads, únicamente revisamos hasta que haya un thread que no se deba despertar en el *tick* actual.

La segunda forma de mejorar tiene que ver con no utilizar el bloque de control de proceso para almacenar el valor con el que determinamos si cada thread debe despertarse, para ello es necesario crear una nueva estructura que contenga dicho valor y un apuntador al bloque de control de proceso, ésta nueva estructura es la que se va a insertar en la lista de dormidos. Para poder apartar espacio para cada una de las estructuras sin necesidad de utilizar memoria dinámica (*malloc*), podemos declarar una variable del tipo de la estructura directamente en la función *timer_sleep* pues aunque se quede guardada localmente en el stack sabemos que no se borrará pues al bloquear el thread no habrá ningún salto a otra función hasta que se despierte el thread.

1.4. Entregables

La práctica se debe enviar al correo **renatiux@gmail.com** con el asunto **PRÁCTICA 1**, se debe de enviar un archivo comprimido tar.gz el cual debe contener los archivos que se modificaron y un archivo README.txt. Por ejemplo si modificaron los archivos *timer.c* y *thread.h* la estructura del archivo debe ser:

```
|— README.txt
|— src
  |— devices
     |— timer.c
  |— threads
```

|— thread.h

El archivo README.txt debe contener el nombre completo de los integrantes del equipo y las respuestas de las preguntas.