# Práctica 1: Timer sleep

*David Flores Peñaloza (dflorespenaloza@gmail.com)*
*José Nieves Morán (nieves@ciencias.unam.mx)*
*Angel Renato Zamudio Malagón (renatiux@gmail.com)*

**Fecha de entrega: Domingo 16 de febrero de 2020**

## 1. Procesos

En un sistema de computo, el procesador (CPU) es considerado el recurso mas importante, por lo cual, una de las principales tareas del sistema operativo consiste en administrar el uso del procesador. Existen muchas formas de administrar uso del procesador, sin embargo la más utilizada para la mayoría de las arquitecturas se apoya del concepto de proceso. En términos simplistas un proceso es la unidad mínima de procesamiento que el sistema operativo puede gestionar. Silberschatz[1] define a un proceso como:

> Informalmente, un proceso es un programa en ejecución. Un proceso es más que el código de un programa, el cual es conocido como la sección *text*. Un proceso también incluye la actividad actual, representada por el valor del *program counter* y los registros de procesador. Un proceso generalmente incluye la pila de ejecución (*process stack*), la cual contiene datos temporales.

### 1.1. Bloque de control de proceso (PCB)

El primer paso es definir de forma puntual cuáles son las características que debe tener un proceso. En términos generales podemos decir que un proceso consta de un código y un conjunto de datos. Los sistemas operativos utilizan una estructura llamada Bloque de Control de Proceso (*process control block*), que sirve para definir las características de un proceso. En el caso de Pintos el Bloque de Control de Proceso (PCB) 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 };
```

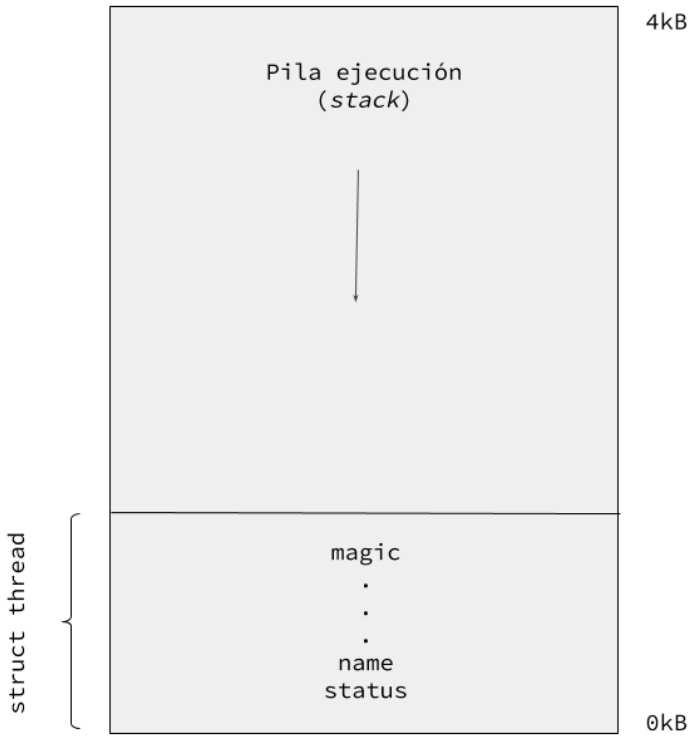
Podemos observar que en Pintos el PCB se llama *thread*, haciendo referencia a un hilo de ejecución y no a un proceso. Hay diferencias claras entre un proceso y un hilo, a partir de este punto nos referiremos a hilos de ejecución en lugar de procesos y al llegar a procesos de usuario veremos las diferencias entre ambos conceptos.

Otra observación que podemos hacer es que en el PCB de Pintos no hay una referencia directa al

código que el hilo debe ejecutar. Lo anterior se debe a que los hilos en Pintos comparten el mismo código y que un hilo, cuando termina de ejecutarse, siempre se bloquea en el mismo punto del código.

La otra parte importante que se mencionó en la definición de proceso es el conjunto de los datos. En este caso el PCB de Pintos hace referencia explicita de los datos locales y lo hace mediante el apuntador *stack*, que hace referencia a la pila de ejecución. La pila de ejecución de un proceso o hilo sirve para almacenar variables locales y los saltos que realizan entre las funciones invocadas. En cuanto a las variables globales, no hay una referencia explicita en el PCB pues dichas variables se colocan en la memoria en tiempo de carga, ya que sus referencias son estáticas y se definen por el enlazador (*linker*).

El siguiente paso consiste en definir en que lugar de la memoria se va a almacenar un hilo y cuanto espacio debe ocupar. En el caso particular de Pintos, cada thread se aloja en un segmento de memoria de 4KB en el cual se almacena el PCB y la pila de ejecución. El siguiente diagrama muestra a grandes rasgos la estructura del segmento.




El bloque de control de proceso (*struct thread*) se coloca al inicio del bloque, el resto del espacio es utilizado para almacenar la pila de ejecución del hilo. El PCB 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 el campo *magic* se almacena en el límite del área correspondiente a la pila de ejecución. Por lo anterior, el campo *magic* sirve para detectar cuando la pila de ejecución se desborda, la idea es que se inicialice el campo *magic* con un valor constante, si dicho valor cambia significa que la pila de ejecución se desbordó.

## 1.2. El primer hilo

Como se mencionó anteriormente, el concepto de hilo o proceso tiene el objetivo de enmascarar la arquitectura específica de hardware en una estructura más manejable para el propio sistema operativo. Cuando el equipo de computo se enciende, no existe ningún proceso o hilo, si consideramos que incluso el sistema operativo consta de uno o varios hilos, entonces es necesario que el mismo sistema sea el encargado de crear el primer hilo del sistema.

El proceso de crear un primer hilo a partir del hardware nativo implica un amplio conocimiento de

la arquitectura específica, muy particularmente del conjunto de instrucciones del procesador. La tarea implica crear una primer estructura de hilo, pero además, dejar el sistema en condiciones para que ese hilo se comience a ejecutar. Usualmente a este proceso se le conoce como *boot process*, en Pintos comienza en el archivo *src/threads/loader.S*. El objetivo del archivo es cargar en memoria principal el código del Kernel y transferir el control, a continuación se muestra la parte que transfiere el control al código del Kernel:

```
1   sub %ax, %ax
2   mov %ax, %ds
3   mov %ax, %ss
4   mov $0xf000, %esp
5
6   mov $0x2000, %ax
7   mov %ax, %es
8   mov %es:0x18, %dx
9   mov %dx, start
10  movw $0x2000, start + 2
11  ljmp *start
```

Primero se inicializan los registros DS y SS en cero (líneas 1 a 3). Luego, el registro ESP (*stack pointer*) en la líneas 4, de esta forma se define que la pila de ejecución inicia en la dirección 0xf000. Después saltamos a la dirección 0x2000, que es donde comienza el código del Kernel (líneas 6 a 11). El código que se invoca se encuentra definido en el archivo *src/threads/start.S*. El archivo contiene en su mayor parte una inicialización de la memoria virtual y al final salta a la función *main*, la cual está programada en lenguaje de alto nivel. A partir de aquí, la inicialización de Pintos se realiza en lenguaje C. La función *main* está definida en el archivo *src/threads/init.c* y se encarga de inicializar la mayor parte del Kernel. La parte que nos concierne en este momento se realiza mediante la función *thread_init*, definida en el archivo *src/threads/thread.c*:

```
1  void
2  thread_init (void)
3  {
4    ASSERT (intr_get_level () == INTR_OFF);
5
6    lock_init (&tid_lock);
7    list_init (&ready_list);
8    list_init (&all_list);
9
10   initial_thread = running_thread ();
11   init_thread (initial_thread, "main", PRI_DEFAULT);
12   initial_thread->status = THREAD_RUNNING;
13   initial_thread->tid = allocate_tid ();
14 }
```

La primer parte importante ocurre en la línea 10, mediante la función *running_thread* obtenemos el apuntador al PCB del hilo inicial. Podemos observar que en realidad no existe aun una estructura PCB en la memoria principal, revisemos la función *running_thread*:

```
1  struct thread *
2  running_thread (void)
3  {
4    uint32_t *esp;
5
6    asm ("mov %%esp, %0" : "=g" (esp));
7    return pg_round_down (esp);
8  }
```

Lo que ocurre es que se copia el valor de registro ESP en una variable local, recordemos que el registro ESP contiene el valor de apuntador de la pila de ejecución (*stack pointer*. También debemos recordar que el registro se inicializó en la dirección 0xf000, y que cada que se declara una variable local o se invoca una función, el valor se incrementa o decrementa. El punto clave ocurre en la línea 7, lo que hacemos es redondear el valor del registro ESP tomando como referencia el tamaño de una página (4KB). La razón de hacer esto es que estamos tomando un bloque de 4KB a partir del valor del registro ESP, y lo utilizaremos para almacenar el primer hilo.

## 1.3. El dispositivo timer

El dispositivo *timer* tiene la función de interrumpir al procesador en intervalos de tiempo regulares. El controlador del dispositivo se implementa en el archivo *src/devices/timer.c*. El controlador a parte de manejar las interrupciones realiza otras tareas, un caso particular es la función *timer_sleep* la cual hace que el hilo que la invoca espere un número específico de *ticks*. Recordemos que 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 (línea 3), 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 agrega al hilo actual en la *ready_list* y pone en ejecución a otro hilo. Eventualmente el hilo que cedió el procesador se volverá a ejecutar por la acción del calendarizador, en ese momento el hilo volverá a revisar si el tiempo que pasó es igual al que necesitaba esperar. Si esto sucedió, el ciclo se rompe y el hilo continua su ejecución, en caso contrario el hilo vuelve a invocar *thread_yield*.

El procedimiento anterior es una forma muy sencilla de simular el bloqueo de un hilo durante un determinado tiempo. A las estrategias de este tipo se les conoce como espera ocupada (*busy wait*), pues lo que hacen es utilizar un ciclo para detener al hilo hasta que se cumpla una condición. Aunque esta estrategia es muy sencilla tiene varios inconvenientes:

1. Desde el punto de vista teórico el hilo nunca se bloquea.
2. El hilo desperdicia tiempo de procesador solamente en preguntar si la condición se cumple.
3. Si hay muchos hilos en la *ready_list* la activación del hilo puede tardar mucho.

## 1.4. 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.

### 1.4.1. Pruebas

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

### 1.4.2. Preguntas

1. ¿Como se da cuenta un thread que su *stack* se desbordó?
2. Describe el proceso de *booting* (inicialización) de Pintos.
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. ¿Qué función tiene el thread idle de Pintos?

### 1.4.3. 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);
```

### 1.4.4. 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.

### 1.5. 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.

## Referencias

[1] Abraham Silberschatz. *Operating System Concepts Essentials*. 2011.