Práctica 2: Calendarizador de prioridades

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 25 de febrero de 2018

1. 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 este mecanismo 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 este objetivo es necesario tener una infraestructura que nos permita administrar a los *threads*.

Lo primero que necesitamos es guardar registro de los threads que están listos para ejecutarse, para ello se utiliza un lista doblemente ligada llamada *ready_list* la cual se declara en el archivo *src/thread.c*. El punto de entrada de todo *thread* es cuando se crea, en Pintos la función *thread_create* se encarga de esto, en está función se aparta la memoria necesaria para crear un nuevo thread (4kB), se inicializa la estructura *struct thread* con los valores requeridos y se coloca al *thread* en la lista *ready_list*, ésta ultima acción se realiza utilizando la función *thread_unblock*.

La función propia del calendarizador se realiza cuando un thread ha utilizado su tiempo de ejecución y es necesario quitarle el control del CPU para que otro *thread* lo utilice. Si suponemos que en el peor de los casos nuestra máquina cuenta únicamente con un solo procesador, la única forma que tenemos para quitar a un *thread* es que él mismo ceda el control del procesador a otro thread. Para ello necesitamos un mecanismo que permita a un thread determinar cuando se ha agotado su tiempo de ejecución, dicho mecanismo son las interrupciones por hardware.

Una interrupción por hardware es una señal que un dispositivo envía al procesador para notificarle que algo sucedió, el procesador detiene la ejecución y atiende la interrupción. En Pintos el manejador de interrupciones se encuentra en el archivo *src/threads/interrupt.c*. Para solucionar el problema de limitar el tiempo de ejecución de un thread, se utiliza un dispositivo llamado *timer*, este dispositivo se configura para que cada determinado tiempo envíe una señal al procesador, de esta manera un thread puede determinar si ya se cumplió su tiempo de ejecución. El código que atiende la interrupción del *timer* se encuentra el el archivo *src/devices/timer.c* en la función *timer_interrupt*, la cual se muestra a continuación.

```
1 static void
2 timer_interrupt (struct intr_frame *args UNUSED)
3 {
4   ticks++;
5   thread_tick ();
6 }
```

La función únicamente lleva la cuenta del número de *ticks* e invoca a la función *thread_tick*. A continuación se muestra la función *thread_tick*.

```
1  void thread_tick (void)
2  {
3    struct thread *t = thread_current ();
4
5    /* Update statistics. */
6    if (t == idle_thread)
7      idle_ticks++;
8  #ifdef USERPROG
9    else if (t->pagedir != NULL)
10     user_ticks++;
11 #endif
12   else
13     kernel_ticks++;
14
15   /* Enforce preemption. */
16   if (++thread_ticks >= TIME_SLICE)
17     intr_yield_on_return ();
18 }
```

La función *thread_tick* no invoca propiamente el cambio de thread, lo que hace es invocar la función *intr_yield_on_return* la cual indica al manejador de interrupciones que cuando haya terminado de manejar la interrupción invoque la función *thread_yield*, la cual es la encargada de hacer el cambio de threads.

```
1  void thread_yield (void)
2  {
3    struct thread *cur = thread_current ();
4    enum intr_level old_level;
5
6    ASSERT (!intr_context ());
7
8    old_level = intr_disable ();
9    if (cur != idle_thread)
10     list_push_back (&ready_list, &cur->elem);
11   cur->status = THREAD_READY;
12   schedule ();
13   intr_set_level (old_level);
14 }
```

El primer punto importante ocurre en la línea 10, lo que se hace es colocar el thread actual al final de la lista de threads listos para ejecutar, esto se hace con el objetivo de que sino hay mas threads esperando ejecución, entonces el mismo thread se volverá a ejecutar. El segundo punto importante ocurre en la línea 12, donde se invoca a la función *schedule*, la cual se muestra a continucación.

```
1  static void schedule (void)
2  {
3    struct thread *cur = running_thread ();
4    struct thread *next = next_thread_to_run ();
5    struct thread *prev = NULL;
6
7    ASSERT (intr_get_level () == INTR_OFF);
8    ASSERT (cur->status != THREAD_RUNNING);
9    ASSERT (is_thread (next));
10
11   if (cur != next)
12     prev = switch_threads (cur, next);
```

```
13   thread_schedule_tail (prev);
14 }
```

En la línea 3 y 4 se obtiene el thread actual y el thread siguiente a ejecutar respectivamente. En la línea 12 es donde ocurre el cambio de ejecución del thread actual por el nuevo, es decir, después de la línea 12 el thread actual ya no está en ejecución y comienza a ejecutarse el thread *next*. La rutina *switch_threads* es la encargada de cambiar los threads, está programada en ensamblador y se encuentra en el archivo *src/threads/switch.S*.

Es necesario señalar que la función *thread_yield* no solamente se usa cuando el dispositivo timer interrumpe al procesador, también se utiliza para forzar el cambio de un thread en casos específicos, por ejemplo cuando el calendarizador es de prioridades.

1.1. Calendarizador de prioridades

Una característica casi indispensable en un sistema operativo moderno es la capacidad de manipular de alguna forma el orden de ejecución de los threads. Un calendarizador de prioridades es una buena forma de implementar dicha característica, en un calendarizador de prioridades cada thread tiene asignada una prioridad (normalmente un número entero), cuando es momento de elegir al siguiente thread que se debe ejecutar, la elección se realiza en función de la prioridad del thread.

El calendarizar de Pintos es de tipo FIFO (First In First Out), lo cual significa que los threads se ejecutan conforme se entran a la lista *ready_list*. Este tipo de calendarizador es el más simple pero no tiene la posibilidad de definir un orden de ejecución. Pintos ya cuenta con la infraestructura básica para poder soportar un calendarizador de prioridades, para ello se utiliza un campo en el bloque de control de proceso el cual se llama *priority*. Las prioridades en Pintos van de 0 (PRI_MIN) a 63 (PRI_MAX), donde los números bajos corresponden a prioridades bajas.

2. Requerimientos

Cambiar el calendarizador de Pintos por un calendarizador de prioridades. Lo que debe cumplir el calendarizador es que cuando un thread de alta prioridad (es decir de mayor prioridad que todos los threads en la *ready_list*) se agregue a la *ready_list*, dicho thread debe ejecutarse inmediatamente. La pruebas que verifican el requerimiento son las siguientes:

2.1. Pruebas

1. priority-fifo
2. priority-preempt
3. priority-change

2.2. Consejos

Hay dos estrategias principales para implementar el calendarizador de prioridades. La primera consiste en que al momento de seleccionar un thread de la *ready_list*, se busque el thread con máxima prioridad. La segunda estrategia consiste en mantener la lista ordenada, esto se consigue al insertar ordenadamente los threads en la lista. En términos de complejidad asintótica ambas estrategias son iguales, sin embargo, desde el punto de vista estadístico es mejor mantener ordenada la lista.

Para poder insertar de forma ordenada en una lista es posible usar la función *list_insert_ordered* la cual recibe como tercer parámetro una función de comparación, para una lista donde todos los elementos son threads se puede utilizar la siguiente función como función de comparación:

```
1 bool
2 thread_less(const struct list_elem* e1, const struct list_elem *e2, void*
    aux UNUSED){
3   struct thread* d1 = list_entry(e1, struct thread, elem);
4   struct thread* d2 = list_entry(e2, struct thread, elem);
5
6   return d1->priority > d2->priority;
7 }
```

2.3. Extra (2pts)

Debido a que las prioridades en Pintos son contantes, es decir, únicamente tenemos 64 prioridades, es posible utilizar un arreglo con 64 listas. Cada una de las listas debe contener a todos los threads que tengan la misma prioridad, por ejemplo, la lista ubicada en el indice 12 del arreglo debe tener a todos lo threads con prioridad 12. Con esta pequeña modificación hacemos que el calendarizador utilice tiempo constante.

2.4. Preguntas

1. En el PCB de Pintos (*struct thread*) no hay una referencia directa al código del thread, ¿De qué forma sabemos en que instrucción se bloqueó el thread por la acción del calendarizador?
2. ¿De qué forma podemos obtener el PCB del thread que está en ejecución?
3. ¿Qué se tiene que hacer para cambiar un thread en ejecución por otro?
4. En Pintos, ¿Qué diferencias existen entre el thread *idle* y el thread *main*?

2.5. Entregables

La práctica se debe enviar al correo **renatiux@gmail.com** con el asunto **PRÁCTICA 2**, 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.