Sistemas Operativos. Práctica: "Priority Scheduling"

Ricchy Alain Pérez Chevanier
Angel Renato Zamudio Malagón

1. Objetivo

El objetivo de esta práctica es que el estudiante modifique la implementación del *scheduler* de *pintos*, de tal forma que funcione como un *scheduler de prioridades* con *preemption*. Dicha implementación requiere que el estudiante maneje adecuadamente varios escenarios, del tal forma que el sistema operativo garantice en todo momento que el *thread* de más alta prioridad sea el que se encuentra en ejecución(`RUNNING`).

2. Introducción

2.1. El Calendarizador (Scheduler)

Después de tener la capacidad de crear nuevos hilos, el siguiente paso es poder administrar el tiempo que cada hilo utiliza el procesador. A esta funcionalidad del sistema operativo se le conoce como calendarizador (*scheduler*). El calendarizador es la parte del sistema operativo encargada de administrar el uso CPU y por ello tiene un lugar fundamental en el funcionamiento del sistema operativo.

Es claro que para poder administrar los hilos del sistema, es necesario tener acceso a ellos. En el caso de Pintos se utilizan dos listas doblemente ligadas, ambas definidas en el archivo *src/threads/thread.c* como variables globales. La primer lista se llama *ready_list* y en ella se almacenan los hilos del sistema que están listos para ejecutarse. La segunda se llama *all_list* y en ella se almacenan todos lo hilos de ejecución del sistema, no importa si están bloqueados o listos para ejecutarse. Cuando un hilo no se encuentra en la *ready_list* se considera que el hilo está bloqueado, por lo tanto basta con agregar un hilo a la *ready_list* para desbloquearlo. Pintos utiliza las funciones *thread_block* y *thread_unblock* para meter y sacar hilos de la *ready_list*:

```
1  void
2  thread_block (void)
3  {
4    ASSERT (!intr_context ());
5    ASSERT (intr_get_level () == INTR_OFF);
6
7    thread_current ()->status = THREAD_BLOCKED;
8    schedule ();
9  }
10
11 void
12 thread_unblock (struct thread *t)
13 {
14   enum intr_level old_level;
15
16   ASSERT (is_thread (t));
17
18   old_level = intr_disable ();
19   ASSERT (t->status == THREAD_BLOCKED);
20   list_push_back (&ready_list, &t->elem);
21   t->status = THREAD_READY;
22   intr_set_level (old_level);
23 }
```

Existen algunas carácteristicas que deben estar presente en la mayoría de los calendarizadores, pues dichas características permiten al calendarizador tener un mayor control sobre los hilos. En términos generales podemos decir que un calendarizado debe ser capaz de:

1. Cambiar un hilo en ejecución por otro.
2. Decidir en que momento debe ocurrir el cambio de hilos.
3. Determinar el siguiente hilo a ser ejecutado.

2.2. Cambio de hilos

Cambiar un hilo de ejecución por otro significa que el hilo que tiene control del procesador debe cederlo y colocar a otro hilo en su lugar. El cambio puede ser provocado esencialmente por dos cosas: Un hilo cede voluntariamente el procesador o el tiempo asignado al hilo se ha terminado. En el caso de Pintos ambos casos utilizan la función *thread_yield* para iniciar el proceso de cambio de hilo:

```
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 hilo en ejecución al final de la lista de hilos en espera de ejecución, en caso de que no haya otro hilo esperando ejecución, el mismo hilo se volverá a ejecutar. El segundo punto importante ocurre en la línea 12, donde se invoca a la función *schedule*, la cual es es propiamente la acción del calendarizador.

```
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 hilo en ejecución y el hilo siguiente a ejecutar respectivamente. En la línea 12 es donde ocurre el cambio de ejecución entre hilo actual y el siguiente a ejecutar, a esta tarea se le conoce como cambio de contexto. Según Silberschatz[1]:

> Cambiar el procesador de un proceso a otro requiere guardar el estado del proceso actual y restablecer el estado de un proceso diferente. Esta tarea es conocida como *cambio de contexto*.

Para poder realizar el cambio de contexto necesitamos definir concretamente que es el contexto de un hilo. El contexto del hilo debe representar el entorno de ejecución de un hilo en términos del procesador, por ello, podemos decir que el contexto de un hilo es el valor de los registros del procesador cuando el hilo está en ejecución. Por lo tanto lo que necesitamos hacer para realizar un cambio de contexto es

guardar los registros del procesador del hilo actual y reemplazarlos por los registros del hilo siguiente a ejecutar. Como necesitamos tratar directamente con los registros del procesador, el cambio de contexto tiene que implementarse en lenguaje de ensamblador, en el caso de Pintos dicha tarea la realiza la función *switch_threads* definida en el archivo *src/threads/switch.S*.

2.3. Tiempo de ejecución

Una vez que el calendarizador tiene la capacidad de cambiar un hilo de ejecución por otro, el siguiente paso de determinar cuanto tiempo tiene cada uno de los hilos una vez que se encuentran en ejecución. Según Tanenbaum[2] existen dos tipos de calendarizador:

1. **Nonpreemptive**: Un calendarizador *nonpreemptive* selecciona un proceso para ejecución y lo deja hasta que se bloqueé o hasta que voluntariamente ceda el procesador.
2. **Preemptive**: Un calendarizador *preemptive* selecciona un proceso y lo deja en ejecución por un tiempo fijo. Si el proceso sigue en ejecución al final del intervalo el procesos se suspende y se selecciona otro para ser ejecutado.

Ambos tipos de calendarizador tienen ventajas y desventajas, sin embargo en los sistemas modernos se utilizan calendarizadores *preemptive* pues son más flexibles y permiten implementar sistemas interactivos (en los que se requiere un tiempo de respuesta alto). Un calendarizador de tipo *preemptive* se debe ejecutar de forma periódica para poder ceder el control del procesador de un hilo a otro.

Cuando se describe al calendarizador de forma abstracta, no se toma en cuenta que el sistema operativo podría no tener el control de algún procesador en el momento que se necesita ejecutar el calendarizador. Lo anterior es un detalle a considerar sobretodo cuando el sistema solo tiene un procesador, por ello, para garantizar la ejecución del calendarizador a intervalos de tiempo regulares es necesario utilizar el soporte del hardware. En particular se utiliza el dispositivo *timer*, el cual se puede programar para que cada determinado tiempo genere una interrupción 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 e invoca al sistema operativo para que atienda la interrupción por medio del manejador de interrupciones. En Pintos el manejador de interrupciones se encuentra en el archivo *src/threads/interrupt.c*, particularmente en la función *intr_handler*. A continuación se muestra la función.

```
1  void
2  intr_handler (struct intr_frame *frame)
3  {
4    bool external;
5    intr_handler_func *handler;
6
7    external = frame->vec_no >= 0x20 && frame->vec_no < 0x30;
8    if (external)
9      {
10       ASSERT (intr_get_level () == INTR_OFF);
11       ASSERT (!intr_context ());
12
13       in_external_intr = true;
14       yield_on_return = false;
15     }
16
17   handler = intr_handlers[frame->vec_no];
18   if (handler != NULL)
19     handler (frame);
20   else if (frame->vec_no == 0x27 || frame->vec_no == 0x2f)
21     {
22     }
23   else
24     unexpected_interrupt (frame);
25
26   if (external)
```

```
27      {
28        ASSERT (intr_get_level () == INTR_OFF);
29        ASSERT (intr_context ());
30
31        in_external_intr = false;
32        pic_end_of_interrupt (frame->vec_no);
33
34        if (yield_on_return)
35          thread_yield ();
36      }
37  }
```

En la línea 19 se invoca al manejador particular de cada interrupción. Si la interrupción es externa (línea 26), es decir es una interrupción por hardware, y la variable global *yield_on_return* es *true* (línea 34), entonces se invoca a la función *thread_yield*, la cual realiza el cambio de hilos. El valor de la variable *yield_on_return* se determina por el manejador del dispositivo específico. En este caso el manejador del *timer* determina el valor de dicha variable, o en otras palabras si se realiza un cambio de hilos. El manejador del *timer* se encuentra en el archivo *src/devices/timer.c* en la función *timer_interrupt*, dicha función solamente invoca a la función *thread_tick* que se encuentra en el archivo *src/devices/thread.c*. A continuación se muestran ambas funciones:

```
1  static void
2  timer_interrupt (struct intr_frame *args UNUSED)
3  {
4    ticks++;
5    thread_tick ();
6  }
7
8  void
9  thread_tick (void)
10 {
11   struct thread *t = thread_current ();
12
13   /* Update statistics. */
14   if (t == idle_thread)
15     idle_ticks++;
16 #ifdef USERPROG
17   else if (t->pagedir != NULL)
18     user_ticks++;
19 #endif
20   else
21     kernel_ticks++;
22
23   /* Enforce preemption. */
24   if (++thread_ticks >= TIME_SLICE)
25     intr_yield_on_return ();
26 }
```

La parte que nos concierne ocurre en las líneas 24 y 25. En la línea 24 revisamos si la variable *thread_ticks* ya rebasó a *TIME_SLICE*, si es así, se invoca la función *intr_yield_on_return* la cual asigna a la variable *yield_on_return* el valor *true*. La variable *thread_ticks* guarda el número de ticks del hilo actual desde que éste está en ejecución, mientras que *TIME_SLICE* es una constante del número de *ticks* que debe ocurrir entre cada cambio de hilo.

En Pintos el *TIME_SLICE* es 4, lo cual significa que cada 4 *ticks* se realiza un cambio de hilo. Por medio del valor *TIME_SLICE*, podemos asignar mayor o menor tiempo de procesador a cada hilo en su turno. Para obtener una interactividad alta el valor de *TIME_SLICE* debe ser bajo, sin embargo debe haber un límite inferior, pues asignar un tiempo muy bajo de ejecución a un hilo puede provocar que se ocupe más tiempo en hacer cambios de contexto que en realizar computo real.

Otro factor a considerar es el intervalo de tiempo en el que el dispositivo *timer* genera una interrupción. Al igual que con el valor del *TIME_SLICE*, se debe tener cuidado con el intervalo de interrupción, pues un intervalo muy grande puede provocar una apariencia de baja interactividad, pero un intervalo

muy bajo implica un mal uso del procesador. En el caso de Pintos el intervalo de interrupción es de 100 veces por segundo (100 HZ), en el caso de Linux el valor por defecto es de 250 HZ a partir de las versiones 2.6.13.

2.4. Algoritmos de calendarización

El aspecto más importante en cuanto al comportamiento del calendarizador es el criterio con el que se selecciona el siguiente hilo por ejecutar. A dicho criterio se le conoce como algoritmo de calendarización. Para poder diseñar un algoritmo de calendarización es necesario conocer el escenario en el que se va a utilizar el sistema operativo.

2.4.1. Calendarizador FIFO

FIFO son las siglas en inglés para *First In First Out*, lo cual significa que el hilo que se elige para ser ejecutado es el que se insertó primero en la lista de hilos por ejecutar. Este algoritmo de calendarización es el más sencillo, su implementación se puede realizar con una lista en la que los hilos se agregan en algún extremo de la lista, mientras que los hilos que se ejecutan se obtienen del otro extremo. El costo de agregar y quitar a un hilo de la lista es de orden constante.

Pintos cuenta por defecto con un calendarizador FIFO, para ello se utiliza una lista doblemente ligada (*ready_list*) en la cual se insertan los hilos (*struct thread*) al final de la lista y se obtienen del inicio de la misma. La extracción del hilo de la lista la podemos observar en la función *next_thread_to_run*, que se muestra a continuación:

```
1 static struct thread *
2 next_thread_to_run (void)
3 {
4   if (list_empty (&ready_list))
5     return idle_thread;
6   else
7     return list_entry (list_pop_front (&ready_list), struct thread, elem);
8 }
```

Aunque un calendarizador de tipo FIFO es muy fácil de implementar, tiene el inconveniente de que se comporta de de manera aceptable solo para muy pocos escenarios. Por ejemplo en un sistema que necesita un tiempo de respuesta alto y hay muchos procesos en ejecución, un calendarizador de tipo FIFO no se comporta de adecuadamente.

2.4.2. Calendarizador de prioridades

El principal inconveniente de un calendarizador de tipo FIFO es que no hay forma de categorizar los procesos, es decir, no hay forma de tener procesos más importantes que otros. En un sistema de tiempo real, lo anterior es un gran inconveniente. La forma más sencilla, en términos de implementación, de corregir el problema de categorizar los procesos es utilizar un calendarizador de prioridades.

En un calendarizador de prioridades cada procesos tiene asociada una prioridad (usualmente de utiliza un número entero). Al momento de que el calendarizador selecciona el siguiente proceso a ser ejecutado, se selecciona al proceso de prioridad más alta. Lo anterior implica que los procesos con mayor prioridad se ejecutarán primero que los procesos con prioridades bajas.

Aunque Pintos cuenta con un calendarizador de tipo FIFO, también tiene la infraestructura básica para que con unos ligeros ajustes se pueda implementar un calendarizador de prioridades. Si observamos el PCB de los hilos en Pintos, podemos notar que hay un campo llamado *priority*:

```
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  };
```

El campo *priority* es un entero el cual representa la prioridad del hilo. Pintos también tiene un rango de prioridades previamente definido en el archivo *src/threads/thread.h*:

```
1  #define PRI_MIN 0                       /* Lowest priority. */
2  #define PRI_DEFAULT 31                  /* Default priority. */
3  #define PRI_MAX 63                      /* Highest priority. */
```

Las prioridades en Pintos van de 0 (prioridad mínima) a 63 (prioridad máxima). Cuando se crea un hilo de ejecución, por medio de la función *thread_create*, se indica la prioridad del hilo. Si el hilo es de uso general se recomienda que tenga la prioridad por defecto (31). La asignación de prioridades de los hilos de ejecución se puede hacer de dos formas según Silberschatz[1]:

> Las prioridades pueden ser definidas internamente o externamente. Las prioridades internamente definidas utilizan algunas medidas para calcular la prioridad de un proceso. Las prioridades externas son asignadas por criterios externos al sistema operativo.

2.4.3. Prioridades con derecho preferente

Un calendarizador de prioridades puede tener derecho preferente (*preemtive*) o no (*non preemptive*). En un calendarizador con derecho preferente al menos un hilo de más alta prioridad debe estar siempre en ejecución, siempre que dicho hilo esté listo para ejecución. En el caso de Pintos los anterior significaría que no puede existir un hilo en la lista *ready_list* que tenga mayor prioridad que el hilo en ejecución. Mantener la lista ordenada u obtener el hilo de mayor prioridad no hacen que el calendarizador tenga derecho preferente, pues, existen algunos casos en los que puede existir un hilo en la *ready_list* de mayor prioridad que el hilo en ejecución:

1. Cuando se crea un nuevo hilo de mayor prioridad que el hilo en ejecución.
2. Cuando se desbloquea un hilo de mayor prioridad que el hilo en ejecución.
3. Cuando la prioridad del hilo en ejecución disminuye y deja de ser el de más alta prioridad en el sistema.

En estos casos existe un hilo de mayor prioridad en la *ready_list* que el hilo en ejecución, pero el hilo en ejecución cede el procesador hasta que el calendarizador se ejecuta, que usualmente es cuando el dispositivo *timer* genera una interrupción. Lo anterior significa que el hilo de mayor prioridad pasa tiempo en la lista, lo cual contradice la definición de calendarizador con derecho preferente.

3. Forma de Trabajo

Pasos a seguir para trabajar en esta práctica:

1. Debes de tener una solución para la práctica anterior en la rama `proyecto01/alarm-clock` o `main`.
2. A partir de la rama que contiene la solución de la práctica anterior (`proyecto01/alarm-clock` o `main`) crea una nueva rama llamada `proyecto01/priority-scheduling` y posiciónate en ella.
3. En el repositorio reemplaza el archivo `.github/workflows/run-tests.yml` con el que entregamos junto con este documento.
4. En el repositorio reemplaza el archivo `src/threads/Make.vars` con el que entregamos junto con este documento.

5. En el repositorio copia los archivos `execute-tests-assignment-01` y `execute-tests-assignment-02` en el directorio `src/threads` y el archivo `execute-tests` en el directorio `src`. Debe de quedar al final en `src/threads/execute-tests-assignment-01`, `src/threads/execute-tests-assignment-02` y `src/execute-tests`. Asegurate de que estos archivos tengan permiso de ejecución con el comando `ls -la`, debes de ver los permisos de la siguiente manera `rwxr-xr-x`, si le hace falta el permiso `x`, para agregárselos puedes ejecutar el comando
`chmod +x path/to/file`.

6. Agrega al repositorio en el directorio raíz el archivo `designdoc-priority-scheduling.md`.

7. Haz commit de estos cambios y sube (*push*) la rama al repositorio.

8. Crea un *Pull Request* para la rama que abas de crear hacia la rama que contiene la solución de la práctica anterior. Este *Pull Request* debe de contener 6 archivos nuevos o modificados en la sección de *Files changed*, que son exactamente los que introdujiste en los tres pasos anteriores.

9. En la sección de *checks*, esta vez solamente vas a seguir pasando las pruebas de *Alarm Clock*, pero tienes que modificar la implementación del *scheduler* para pasar el resto de las pruebas.

10. Una nota importante es que para pasar las pruebas vamos a utilizar la máquina virtual `qemu`, pues `bochs` puede tener algunos problemas.

En el archivo `readme.md` del repositorio se incluyen instrucciones de cómo compilar el código y de cómo ejecutar algunas pruebas. Para ejecutar todas las pruebas de esta práctica, puedes acceder al contenedor, posicionarte en el directorio `src/threads` y ejecutar la instrucción `./execute-tests-assignment-02`.

Tu solución tiene que ser incremental, es decir, debes de pasar las pruebas de la práctica anterior y también las pruebas específicas de esta práctica, de hecho en circleci se van a ejecutar como dos pasos independientes.

4. Actividades

Cambiar el calendarizador de Pintos por un calendarizador de prioridades con derecho preferente (preemptive).

4.1. Implementación (5 pts + 1 pto extra)

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

También puedes pasar las siguientes pruebas para ganar un punto extra:

1. priority-sema
2. priority-condvar

Es necesario que modifiques el script `execute-tests-assignment-02`, para que considere estas dos pruebas adicionales. En dicho archivo incluimos instrucciones de qué lineas hay que descomentar para realizar este cambio. Si no realizas esta modificación tu implementación no será evaluada y por consiguiente no podemos darte el punto extra correspondiente.

Algunos archivos que podrías modificar como parte de tu solución son los siguientes:

- El módulo de threads `src/threads/thread.h` y `src/threads/thread.c`
- El módulo del primitivas de sincronización `src/threads/sync.h` y `src/threads/sync.c`

4.2. Documento de Diseño (5 pts)

Contestar todas las sección del documento de diseño que se entrega junto con este documento.

4.3. Extra: Listas de prioridades (1 pto)

Debido a que las prioridades en Pintos solo pueden tener valores entre 0 y 63 prioridades, es posible representar la `ready_list` con una estructura de datos que se tarde tiempo `O(1)` en insertar hilos y en extraer el siguiente hilo que va a poner en ejecución el calendarizador. Si logras implementar esta solución tendrás este punto adicional.

5. 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 `ready_list`.

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_compare(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 }
```

6. Entrega

En Google Classroom agrega el enlace del `Pull Request` que contiene tu solución y en donde se deben de observar los resultados de Github Actions al correr las pruebas. También marca la tarea como entregada en Google Classroom.