Sistemas operativos
Proyecto 1 - Threads

Iván Emilius Mamani Arisaca
Willian Berrocan Alvarado

15 de octubre del 2020

Docente
Jorge Luis Gonzales Reaño

1. Implementación del Alarm Clock

En este ejercicio se nos pide optimizar la forma en la que pint-os cambia de thread, debido a que la forma actual en que lo hace es mediante "Bussy waiting", forma la cual es muy ineficiente pues gasta recursos innecesariamente.

```
void timer_sleep(int64_t ticks){
    int64_t start = timer_ticks();
    while(timer_elapsed(start) < ticks)
        thread_yield();
}
```

La nueva implementacion que realizaremos para resolver este problema se denomina "Sleep/Wake up", en el que mandamos la thread a "dormir" un determinado periodo de tiempo (n ticks de clock), para "despertarlo" más tarde cuando le toque ser ejecutada.

1.1. Pasos de implementación

Para implementar el mecanismo "sleep/wake up" se necesita tener noción de cuando una thread va a despertar, para ello le añadimos la variable *blocked_ticks* a la estructura threads

```
/* pintos/src/thread/thread.h */
struct thread{
...
int64_t blocked_ticks;
...
}
```

También se necesita una nueva lista para guardar las threads que están dormidas, por eso creamos *sleep_list*. Además, agregamos la variable *next_tick_to_awake* que guarda el tiempo donde la siguiente thread debe despertar, para así no gastar recursos entrando a la sleep list a revisar cuándo será que despierte la siguiente thread.
Y se incializa la lista *sleep_list* en la función thread_init

```
static struct list sleep_list;
static int64_t next_tick_to_awake;
...

void thread_init(void){
...
list_init (&sleep_list);
...
}
```

Ahora tenemos que crear nuevas funciones que sirvan para usar *next_tick_to_awake*: update y get
Update servirá para tener el tiempo más pequeño (el más próximo) de entre las threads para despertar; y get servirá como *getter*

```
/* pintos/src/thread/thread.c */

void update_next_tick_to_awake(int64_t ticks){

next_tick_to_awake = (next_tick_to_awake > ticks) ?
                     ticks : next_tick_to_awake;
}

int64_t get_next_tick_to_awake(void){
return next_tick_to_awake;
}
```

Para poner a dormir a las threads, debemos implementar una función que lo haga: *thread_sleep*, la cual añade la thread a la *sleep_list* y la bloquea.
Nota: Se debe evitar que la idle_thread se duerma al ser la que siempre se ejecuta cuando el CPU no hace nada

```
/* pintos/src/thread/thread.c */

void thread_sleep(int64_t ticks){
struct thread *currrent_thread;

enum intr_level old_level;
old_level = intr_disable();

currrent_thread = thread_current();
currrent_thread->blocked_ticks = ticks;
ASSERT(cur != idle_thread);
update_next_tick_to_awake(currrent_thread->blocked_ticks);


list_push_back(&sleep_list,&currrent_thread->elem);
thread_block();
intr_set_level(old_level);
}
```

Ahora que ya hay threads durmiendo, necesitamos una función para hacerlas "despertar" (sacarlas del *sleep_list*). Para esto, implementamos la función *thread_awake*, que revisa la *sleep_list* y verifica si el tick actual es menor al tick para despertar, de ser así, remueve la thread que debe despertar y la desbloquea, y procede a llamar a update_next_tick_to_awake que actualiza al nuevo tick para despertar.

```
/* pintos/src/thread/thread.c */
void thread_awake(int64_t wakeup_tick){
    next_tick_to_awake = INT64_MAX;
    struct list_elem *e;
    e = list_begin(&sleep_list);
    while(e != list_end(&sleep_list)){
        struct thread * t =
        list_entry(e, struct thread, elem);

        if(wakeup_tick >= t->wakeup_tick){
            e = list_remove(&t->elem);
            thread_unblock(t);
        }else{
            e = list_next(e);
            update_next_tick_to_awake(t->wakeup_tick);
        }
    }
}
```

Por último, cambiamos la parte que hacía busy waiting en timer.c por la función que pone la thread a dormir:

```
/* pintos/src/device/timer.c */
void timer_sleep (int64_t ticks){
    int64_t start = timer_ticks ();

    ASSERT (intr_get_level () == INTR_ON);

    thread_sleep(start + ticks);
}
```

Y también le añadimos una condición que revise, en cada tick, si hay una thread que deba despertar, y en ese caso llama a la función para despertarla:

```
/* pintos/src/device/timer.c */
static void timer_interrupt (struct intr_frame *args){
    ...
    if(get_next_tick_to_awake() <= ticks){
        thread_awake(ticks);
    }
}
```

1.2. Resultados

En la imagen se puede apreciar que a diferencia de cuando pint-os funcionaba con busy waiting, ahora los idle ticks (ticks de la thread que corre cuando no se hace nada) ya no son 0.
Esto debido a que anteriormente, cuando se hacia busy waiting, en realidad el CPU estaba siempre ocupado, incluso mientras dormía; pero ahora con la implementación del sleep/wakeup, el CPU ya no malgasta recursos y puede descansar verdaderamente.

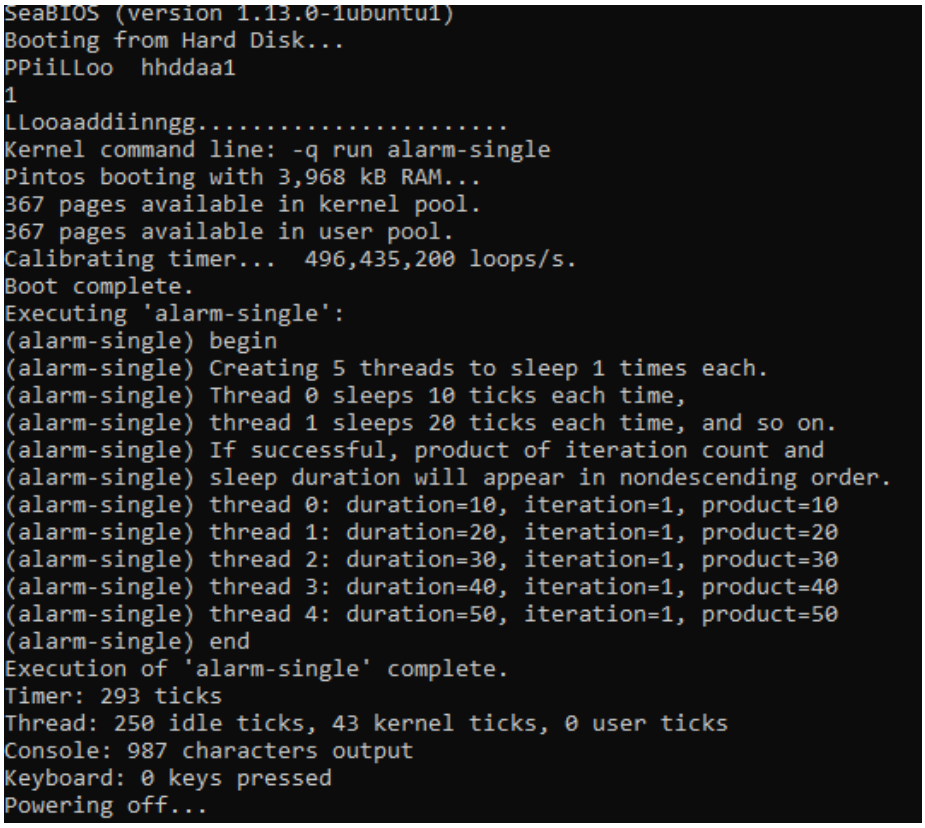

Figura 1: resultado de alarm-single

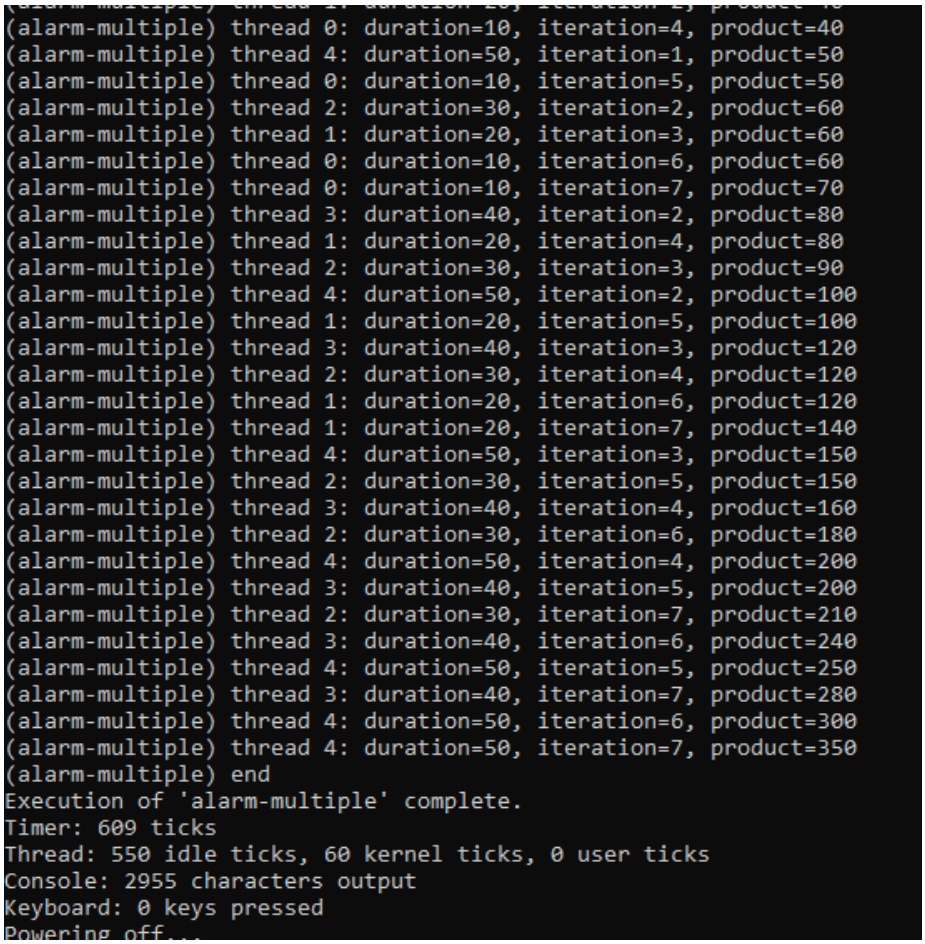

Figura 2: resultado de alarm-multiple

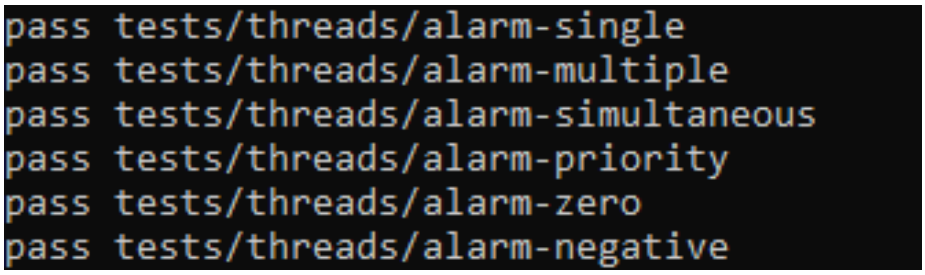

Figura 3: resultado de los tests

Además, podemos observar que la implementación ha pasado exitosamente los tests correspondientes al alarm.

2. Implementación del Priority Scheduling

Actualmente, pint-os ejecuta las threads simplemente en orden de llegada, esto debido a que todas las nuevas threads se colocan siempre al final de la ready_list sin importar su prioridad.
El objetivo de este ejercicio es modificar el orden en que pint-os ejecuta las threads, para que las que tengan mayor prioridad sean las que se ejecuten primero, y luego las demás en orden descendente de prioridad.

2.1. Pasos de implementación

Lo primero que haremos será crear una nueva función: thread_compare_priority que nos servirá para comparar prioridades entre threads:

```
/* pintos/src/threads/thread.c */
bool thread_compare_priority (const struct list_elem *a,
                              const struct list_elem *b,
                              void *aux UNUSED){
    return list_entry(a, struct thread, elem)->priority
         > list_entry(b, struct thread, elem)->priority;
}
```

Ahora vamos a la función thread_unblock y procedemos a reemplazar la función push_back, que se usaba antes para agregar todas las nuevas threads al final de la ready_list, por la función list_insert_ordered que nos permite agregar threads de forma que estén ordenadas por prioridad:

```
/* pintos/src/threads/thread.c */
void thread_unblock (struct thread *t){
    enum intr_level old_level;

    ASSERT (is_thread (t));

    old_level = intr_disable ();
    ASSERT (t->status == THREAD_BLOCKED);

    list_insert_ordered (&ready_list, &t->elem,
                         thread_compare_priority, 0);
    t->status = THREAD_READY;
    intr_set_level (old_level);
}
```

De igual forma, reemplazamos push_back por list_insert_ordered en la función thread_yield para que ya no inserte las threads al final del ready_list, sino que

lo haga de forma que las thread sque tienen mayor prioridad se puedan ejecutar antes:

```
/* pintos/src/threads/thread.c */
void thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    ASSERT (!intr_context ());

    old_level = intr_disable ();
    if (cur != idle_thread)
    list_insert_ordered (&ready_list, &cur->elem,
                         thread_compare_priority, 0);

    cur->status = THREAD_READY;
    schedule ();
    intr_set_level (old_level);
}
```

También se agrega la función test_max_priority que nos permite saber si la thread actual se cambiara por la siguiente. Esto comparando sus prioridades, y si la thread actual tiene más prioridad que la que le sigue en la ready_list, seguirá ejecutádose un rato más, hasta que acabe o hasta que aparezca otra con mayor prioridad.

```
/* pintos/src/threads/thread.c */
void test_max_priority (void){
    if (!list_empty (&ready_list) &&
        thread_current ()->priority <
        list_entry (list_front (&ready_list),
        struct thread, elem)->priority){
        thread_yield ();
    }
}
```

2.2. Resultados

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
FAIL tests/threads/priority-change
FAIL tests/threads/priority-donate-one
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
FAIL tests/threads/priority-fifo
FAIL tests/threads/priority-preempt
FAIL tests/threads/priority-sema
FAIL tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
19 of 27 tests failed.
```

Figura 4: tests del alarm y priority