

# Relatório PintOS

## Projeto 1: Threads

Alunos Romero Cartaxo e Severino Murilo da Silva

Professor Eduardo Antonio Guimaraes Tavares

### Alarm Clock

Para implementar o Alarm Clock sem a necessidade de escalonar a thread múltiplas vezes para checar se ela ainda está dormindo, decidimos registrar o tempo em que a thread deve acordar na struct da thread:

```
1 struct thread
2 {
3     /* Owned by thread.c. */
4     tid_t tid; /* Thread identifier. */
5     enum thread_status status; /* Thread state. */
6     char name[16]; /* Name (for debugging purposes). */
7     uint8_t *stack; /* Saved stack pointer. */
8     int priority; /* Priority. */
9     int64_t wake_tick; /* Amount of system ticks for when the thread should wake up */
10    struct list_elem allelem; /* List element for all threads list. */
11    ...
12
13 };
```

Com isso, ao chamarmos `timer_sleep`, devemos somar o valor de ticks já passados com o valor de ticks que a thread deverá dormir e atribuir esse valor ao atributo `wake_tick` da thread atual. Em seguida, desabilitamos a interrupção, guardamos o valor atual da interrupção em uma variável, bloqueamos a thread chamando a função (`thread_block`) e logo após, restauramos o antigo valor da interrupção. É necessário desabilitar a interrupção pois não queremos que a thread seja interrompida enquanto ela está no processo de ser bloqueada pois isso pode causar comportamentos inesperados. Quando restauramos o valor da interrupção após ela ser bloqueada, isso só irá acontecer quando a thread acordar novamente e retornar a execução. Porém, quando a próxima thread for escalonada, a interrupção será habilitada novamente, o que vai de acordo com o comportamento esperado.

```
1 timer_sleep (int64_t ticks)
2 {
3     int64_t start = timer_ticks ();
4
5     ASSERT (intr_get_level () == INTR_ON);
6
7     struct thread *t = thread_current();
8     t->wake_tick = ticks + start;
9
10    enum intr_level old_level = intr_disable();
11    thread_block();
12    intr_set_level(old_level);
13 }
```

Para acordamos a thread, precisamos modificar a função responsável por lidar com as interrupções do timer. Para auxiliar esta tarefa, criamos uma função `check_sleeping_thread` que, com o auxílio da função `thread_foreach`, irá verificar em cada thread, se a thread está bloqueada e se já está no momento de ela ser acordada. Caso sim, ela é colocada na lista de threads prontas para execução.

```
1 // Used by thread_foreach to wake sleeping threads on the right time
2 void check_sleeping_thread(struct thread *t, void *aux) {
3     int64_t start = ticks;
4     if (t->wake_tick && start > t->wake_tick && t->status == THREAD_BLOCKED) {
5         t->wake_tick = 0;
```

```

6     thread_unblock(t);
7 }
8 }
9
10 /* Timer interrupt handler. */
11 static void
12 timer_interrupt (struct intr_frame *args UNUSED)
13 {
14     ticks++;
15     thread_tick ();
16     thread_foreach(check_sleeping_thread, NULL);
17 }

```

## Priority Scheduler

Para implementarmos escalonamento por prioridade nesse projeto, precisamos modificar a função `next_thread_to_run` que é chamada toda vez que o escalonador é invocado. Como o nome da função indica, o seu objetivo é retornar a próxima thread que deve ser executada. Nela, nós iteramos sobre a lista de threads prontas para execução, guardamos a thread que possui o valor de prioridade mais alto, removemos ela da lista de threads prontas para execução e retornamos ela:

```

1 static struct thread *
2 next_thread_to_run (void)
3 {
4     if (list_empty (&ready_list))
5         return idle_thread;
6     else{
7         // Traverses ready_list, stores the thread with the highest priority, removes it from the
8         // list and return it.
9
10        struct list_elem* start_iter = list_begin (&ready_list);
11
12        // Variable to store the list element of the thread with highest priority to remove it at
13        // the end of the function
14        struct list_elem* max_priority_iter = start_iter;
15
16        struct thread* max_priority_thread = list_entry (start_iter, struct thread, elem);
17        int max_priority = max_priority_thread->priority;
18
19        struct list_elem* second_iter = list_next(start_iter);
20
21        for(struct list_elem* iter = second_iter;
22            iter != list_end(&ready_list);
23            iter = list_next(iter))
24        {
25            struct thread* t = list_entry(iter, struct thread, elem);
26            if (t->priority > max_priority) {
27                max_priority_thread = t;
28                max_priority_iter = iter;
29                max_priority = t->priority;
30            }
31        }
32
33        list_remove(max_priority_iter);
34        return max_priority_thread;
35    }
36 }

```

## Priority Donation

Para implementar a doação de prioridade foi necessário adaptarmos os seguintes arquivos:

Ordenação da ready List (threads/thread.c) -> pensei em algo como insertion sort pra inserir as threads já ordenadas por prioridade:

```

1 void thread_unblock (struct thread *t) {
2     list_insert_ordered (&ready_list, &t->elem, thread_priority_less, NULL);
3     t->status = THREAD_READY;
4 }

```

Doação de prioridade (threads/thread.c) - a thread que está segurando o lock recebe prioridade de quem está esperando.

```

1 void thread_donate_priority (struct thread *t) {
2     struct lock *lock = t->waiting_lock;
3     struct thread *holder = lock->holder;
4
5     if (t->priority > lock->max_priority)
6         lock->max_priority = t->priority;
7
8     if (t->priority > holder->priority)
9         holder->priority = t->priority; //doar prioridade
10 }

```

### Aquisição de Lock (threads/synch.c)

```

1 void lock_acquire (struct lock *lock) {
2     if (lock->holder != NULL) {
3         cur->waiting_lock = lock;
4         thread_donate_priority (cur); //doar antes de bloquear
5     }
6     sema_down (&lock->semaphore);
7     lock->holder = cur;
8     list_push_back (&cur->locks_held, &lock->elem);
9 }

```

### Liberação de lock (threads/synch.c)

```

1 void lock_release (struct lock *lock) {
2     list_remove (&lock->elem);
3     thread_update_priority (cur); //restaurar prioridade
4     lock->max_priority = PRI_MIN;
5     sema_up (&lock->semaphore);
6     thread_yield();
7 }

```

### Atualização de prioridade (threads/thread.c)

```

1 void thread_update_priority (struct thread *t) {
2     int max = t->base_priority;
3
4     // Percorre todos os locks que possui
5     for (e = list_begin (&t->locks_held); ...) {
6         struct lock *lock = list_entry (e, struct lock, elem);
7         if (lock->max_priority > max)
8             max = lock->max_priority;
9     }
10
11     t->priority = max; //usar a maior prioridade
12 }

```