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:

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    int64_t wake_tick;  /* Amount of system ticks for when the thread should wake up */
    struct list_elem allelem;           /* List element for all threads list. */
    ...

  };
```

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.

```
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);

  struct thread *t = thread_current();
  t->wake_tick = ticks + start;

  enum intr_level old_level = intr_disable();
  thread_block();
  intr_set_level(old_level);
}
```

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.

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

```
    thread_unblock(t);
  }
}

/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  thread_foreach(check_sleeping_thread, NULL);
}
```

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:

```
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else{
    // Traverses ready_list, stores the thread with the highest priority, removes it from the
        list and return it.

    struct list_elem* start_iter = list_begin (&ready_list);

    // Variable to store the list element of the thread with highest priority to remove it at
        the end of the function
    struct list_elem* max_priority_iter = start_iter;

    struct thread* max_priority_thread = list_entry (start_iter, struct thread, elem);
    int max_priority = max_priority_thread->priority;

    struct list_elem* second_iter = list_next(start_iter);

    for(struct list_elem* iter = second_iter;
    iter != list_end(&ready_list);
    iter = list_next(iter))
    {
      struct thread* t = list_entry(iter, struct thread, elem);
      if (t->priority > max_priority) {
        max_priority_thread = t;
        max_priority_iter = iter;
        max_priority = t->priority;
      }
    }

    list_remove(max_priority_iter);
    return max_priority_thread;
  }
}
```

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:

```c
void thread_unblock (struct thread *t) {
  list_insert_ordered (&ready_list, &t->elem, thread_priority_less, NULL);
  t->status = THREAD_READY;
}
```

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

```c
void thread_donate_priority (struct thread *t) {
  struct lock *lock = t->waiting_lock;
  struct thread *holder = lock->holder;

  if (t->priority > lock->max_priority)
    lock->max_priority = t->priority;

  if (t->priority > holder->priority)
    holder->priority = t->priority;  //doar prioridade
}
```

Aquisição de Lock (threads/synch.c)

```c
void lock_acquire (struct lock *lock) {
  if (lock->holder != NULL) {
    cur->waiting_lock = lock;
    thread_donate_priority (cur);  //doar antes de bloquear
  }
  sema_down (&lock->semaphore);
  lock->holder = cur;
  list_push_back (&cur->locks_held, &lock->elem);
}
```

Liberação de lock (threads/synch.c)

```c
void lock_release (struct lock *lock) {
  list_remove (&lock->elem);
  thread_update_priority (cur);  //restaurar prioridade
  lock->max_priority = PRI_MIN;
  sema_up (&lock->semaphore);
  thread_yield();
}
```

Atualização de prioridade (threads/thread.c)

```c
void thread_update_priority (struct thread *t) {
  int max = t->base_priority;

  // Percorre todos os locks que possui
  for (e = list_begin (&t->locks_held); ...) {
    struct lock *lock = list_entry (e, struct lock, elem);
    if (lock->max_priority > max)
      max = lock->max_priority;
  }

  t->priority = max;  //usar a maior prioridade
}
```