Project: Extending the functionality of PintOS

<u>Report</u>

<u>Group Name- Team Rocket</u>

Neelesh Bhakt - 2018201022
Tarpit Sahu - 2018201089

==

<u>Initial Setup</u>

Following steps were performed in the installation of PintOS.

1. The source code of pintos was downloaded.

2. Set $path in .tcshrc till pintos/src/utils

3. Export path in .bashrc

4. In pintos/src/utils/pintos-gdb file, updated GDBMACROS variable as pintos/src/misc/gdb-macros, which has path to GDB macros file.

5. In pintos/src/utils/Makefile file, replaced LDFLAGS with LDLIBS.

6. Run command "make" at pintos/src/utils

7. In pintos/src/threads/Make.vars file,
   updated variable SIMULATOR = --qemu

8. Run command "make" at pintos/src/threads

10. In pintos/src/utils/pintos file,

a) In subroutine parse_command_line, updated $sim = "qemu"

b) In subroutine find_disks, updated my $name = find_file(<path of kernel.bin>)

This subroutine is used to locate the files used to back each of the virtual disks and creates temporary disks.

c) In subroutine run_qemu, updated my (@cmd) = ('qemu-system-x86_64');

This subroutine is used to run QEMU.

Also comment this line :
push (@cmd, '-device', 'isa-debug-exit');

11. In pintos/src/utils/Pintos.pm file,

a) In subroutine read_loader, updated $name = find_file (<path of loader.bin>)

12. cd into src/threads/build directory, run command, "make check"

Problem: Qemu gets stuck when PintOS powers off.

A special power-off sequence which is supported by QEMU is added.The sequence is,
outw(0x604,0x0 | 0x2000);

==

<u>Part 1: Create a file hello.c to print “Hello Pintos” message.</u>

A new file hello.c was added in the location "src/tests/threads/hello.c".

```
/*Code Snippet*/

#include<stdio.h>
#include"tests/threads/tests.h"
#include"threads/malloc.h"
#include"threads/synch.h"
#include"threads/thread.h"
#include"devices/timer.h"

void
test_hello (void)
{
  printf("Hello Pintos\n");

 }
```

We also performed changes to Make.tests,tests.c and tests.h in the src/tests/threads directory to ensure that hello.c is built with the kernel.

==

Part 2 : Pre-emption of threads

Objective of this part was to understand the thread sleep mechanism of pintos and provide an efficient solution to circumvent busy waiting. (Reimplement timer_sleep(), defined in devices/timer.c.)

----

void timer_sleep(int64_t ticks) was re-implemented.

In the original implementation, the function was repeatedly calling no. of ticks time yield function which was stopping the currently running thread and adding it to the ready-queue and scheduling the next thread. In this way, the above function was doing busy waiting up to ticks time.

```
void
timer_sleep (int64_t ticks)
{
  int64_t current_time = timer_ticks ();
  ASSERT (intr_get_level () == INTR_ON);
//*****************************************************************
  struct thread *sleep_thread = thread_current();
  sleep_thread->wk_time = current_time+ticks;
  enum intr_level old_level = intr_disable();
  list_insert_ordered (&sleeping_queue, &sleep_thread->elem, comparator_wake, NULL);
  thread_block();
  intr_set_level(old_level);
//*****************************************************************
}
```

In our implementation timer_sleep() function a separate sorted sleeping queue is maintained on the basis of wake-up time (current_ticks+ticks). The sleeping queue keeps track of all the sleeping thread, therefore every time whenever a new thread has to sleep for certain duration we add it into the sleeping queue and block that thread.

----

void thread_tick()

This function is invoked at every thread tick, it checks whether current thread tick is greater than the topmost thread waking up time in sleeping queue and while there is any such thread it removes that thread from the sleeping queue and add it to the ready queue with the help of thread_unblock function.

```
void
thread_tick (void)
{
  struct thread *t = thread_current ();
  int64_t current_time = timer_ticks();
  struct list_elem *q_front;
  struct thread *sleeping_queue_top_thread;
//*******************************************
  if(!list_empty(&sleeping_queue))
  {
  q_front = list_front(&sleeping_queue);
  sleeping_queue_top_thread = list_entry(q_front,struct thread,elem);

  while((!list_empty(&sleeping_queue)) && current_time >= sleeping_queue_top_thread-
>wk_time)
  {
    list_pop_front(&sleeping_queue);
    thread_unblock(sleeping_queue_top_thread);

    if(!list_empty(&sleeping_queue))
    {
      q_front = list_front(&sleeping_queue);
      sleeping_queue_top_thread = list_entry(q_front,struct thread,elem);
    }
 }

}
//*******************************************


/* Update statistics. */
if (t == idle_thread)
 idle_ticks++;
#ifdef USERPROG
else if (t->pagedir != NULL)
user_ticks++;
#endif
else
  kernel_ticks++;

/* Enforce preemption. */
 if (++thread_ticks >= TIME_SLICE)
   intr_yield_on_return ();
}
                    -----------------------------------------------------------
void
thread_unblock (struct thread *t1)
{
    enum intr_level old_level;
    ASSERT (is_thread (t1));
    old_level = intr_disable ();
    ASSERT (t1->status == THREAD_BLOCKED);
   list_insert_ordered (&ready_list, &t1->elem, priority_compare, NULL);
   t1->status = THREAD_READY;
    intr_set_level (old_level);
  }
```
==

<u>Part 3 : Implementation of priority scheduling.</u>

This part focusses on the implementation of a basic priority scheduling scheme. The main motive is a higher priority thread should preempt the lower priority thread.
Threads can take priority value anywhere between 0-63 (both exclusive). 0 means least and 63 means max priority.

===

Changes in thread.c

----
thread_priority_assign()

Functionality :
Checks the ready list, if the front thread has greater priority than the current thread, then thread_yield() function is called.

```
void thread_priority_assign()
{

        if(!list_empty(&ready_list))
        {
            struct list_elem *front = list_front(&ready_list);
            struct thread *t1 = list_entry(front, struct thread, elem);
            if(t1->priority > thread_current()->priority)
                        thread_yield();
        }
}
```

----
thread_create()

Modifications made according to the problem statement. The problem statement reads
"When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread."

In the thread_create() function we have called a function that checks the priority of the newly created thread and executes accordingly.

```
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;
  enum intr_level old_level;

  ASSERT (function != NULL);
  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

  /* Prepare thread for first run by initializing its stack.
     Do this atomically so intermediate values for the 'stack'
     member cannot be observed. */
  old_level = intr_disable ();

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;

  /* Stack frame for switch_entry(). */
  ef = alloc_frame (t, sizeof *ef);
  ef->eip = (void (*) (void)) kernel_thread;

  /* Stack frame for switch_threads(). */
  sf = alloc_frame (t, sizeof *sf);
  sf->eip = switch_entry;
  sf->ebp = 0;

  intr_set_level (old_level);

  /* Add to run queue. */
  thread_unblock (t);

  old_level = intr_disable();
  //*******************************************
  thread_priority_assign();
  //*******************************************
  intr_set_level(old_level);

  return tid;
}
```

----

priority_compare()

priority_compare is a comparator function which is used by list_insert_ordered() function to insert in list in decreasing (non-increasing, to be more precise) order of priority.

```
priority_compare (const struct list_elem *first, const struct list_elem *second, void *aux UNUSED)
```

```
          {
            struct thread *t1 = list_entry (first, struct thread, elem);
            struct thread *t2 = list_entry (second, struct thread, elem);
            if(t1->priority > t2->priority)
                return true;
            else
                return false;
          }
```

----

thread_unblock()

this function is modified inorder to accomodate the unblocked thread in the ready list as per the thread priority using list_insert_ordered() function.

```
thread_unblock (struct thread *t1)
{
  enum intr_level old_level;
  ASSERT (is_thread (t1));
  old_level = intr_disable ();
  ASSERT (t1->status == THREAD_BLOCKED);
  //*******************************************
  list_insert_ordered (&ready_list, &t1->elem, priority_compare, NULL);
  //*******************************************
  t1->status = THREAD_READY;
  intr_set_level (old_level);
}
```

----

thread_set_priority()

this function sets the new priority of currently running thread. thread_priority_assign() is called inorder to schedule thread according to the priority.

```
thread_set_priority (int new_priority)
{
  thread_current()->priority = new_priority;
  //********************************************
  thread_priority_assign();
  //********************************************
}
```

----

thread_yield()

this function is slightly modified. The modification is that now we insert the outgoing thread into the ready list as per the priority instead of just putting it in the end of the list.

```
thread_yield (void)
{
  struct thread *cur_thread = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  //********************************************
  if (cur_thread != idle_thread)
    list_insert_ordered(&ready_list, &cur_thread->elem, priority_compare, NULL);
  //********************************************
  cur_thread->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

==

Changes in Synch.c

----

sema_down()
Modification made : earlier the function was inserting in the Semaphore waiting list in arbitrary order. Now after modifications, the function is inserting on the basis of priority using the list_insert_ordered() function.

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());
  old_level = intr_disable ();
  while (sema->value == 0)
    {
          //******************************************
      list_insert_ordered(&sema->waiters, &thread_current ()->elem, priority_compare, NULL);
      thread_block ();
          //******************************************
    }
  sema->value--;
  intr_set_level (old_level);
}
```

----

sema_up()
Modification made : Earlier the function was putting the unblocked thread at the end of ready queue. Now after unblocking the thread we are making a call to thread_priority_assign() function that checks whether the newly unblocked thread has high priority than the current thread.

```
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));

  sema->value++;

  //***************************************
  thread_priority_assign();
  //***************************************

  intr_set_level (old_level);
}
```

----

comparator()
This function is a comparator that is passed as an argument to list_insert_ordered() function.

```
static bool comparator (const struct list_elem *first, const struct list_elem *second, void *aux
UNUSED)
{
  struct semaphore_elem *s1 = list_entry(first,struct semaphore_elem, elem);
  struct semaphore_elem *s2 = list_entry(second,struct semaphore_elem, elem);

        if(s1->semaphore_priority > s2->semaphore_priority)
                return true;
        else
                return false;
}
```

----

cond_wait()

```
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem sema_wait;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&sema_wait.semaphore, 0);

  //****************************************************
  sema_wait.semaphore_priority = thread_current()->priority;
  list_insert_ordered (&cond->waiters, &sema_wait.elem, comparator, NULL);
  //****************************************************

  lock_release (lock);
  sema_down (&sema_wait.semaphore);
  lock_acquire (lock);
}
```

----

cond_signal()

```
void cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters))
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);

 //****************************************************
  thread_priority_assign();
 //****************************************************
}
```

Description of Test Cases

Test Cases

1.alarm-single

This test case creates 5 threads, and each thread sleeps once.

2.alarm-zero

In this test case, the argument ticks in thread_ticks() function is passed as zero.

3.alarm-negative

In this test case, the argument ticks in thread_ticks() function is passed a negative value (-100). The only requirement it demands is that it should not crash.

4.alarm-simultaneous

This test case creates N threads, each of which sleeps a different, fixed duration, M times. Records the wake-up order and verifies that it is valid.

5.alarm-multiple

This test case creates 5 threads, and each thread sleeps seven times.

6.alarm-priority

In this testcase we check that when the alarm clock wakes up threads, the higher-priority threads run first.

7.priority-change

In this test case we verify that lowering a thread's priority so that it is no longer the highest-priority thread in the system causes it to yield immediately.

8.priority-fifo

In this test case creates several threads all at the same priority and ensures that they consistently run in the same round-robin order.

9.priority-sema

This tests that the highest-priority thread waiting on a semaphore is the first to wake up.

10.priority-condvar

This tests that cond_signal() wakes up the highest-priority thread waiting in cond_wait().