OPERATING SYSTEM
ASSIGNMENT 3
PintOS (Scheduling)

Part 1: Getting Started

- **Understand Pintos basics-**

Pintos is computer software, a simple instructional operating systemframework for the x86 instruction set architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way. It was created at Stanford University by Ben Pfaff in 2004.

- **Build the Pintos executable from source code**

Installing Pintos development environment on our machine.The Pintos development environment is targeted at Unix-like systems.
Prerequisites for installing a Pintos development environment include the following, on top of standard Unix utilities: GCC, GNU binutils, Perl, GNU make, QEMU (version 0.8.0 or later) , GDB(not necessary but recommended).

Instructions to install Pintos:

- Download Pintos.Extract it in some directory say $HOME/os-pg/
- open the script 'pintos-gdb' (in $HOME/os-pg/pintos/src/utils) in any text editor. Find the variable GDBMACROS and set it to point to '$HOME/os-pg/pintos/src/misc/gdb-macros'.
- Compile the Utilities.
- Open the file "$HOME/os-pg/pintos/src/threads/Make.vars" and change the last line to:SIMULATOR = --qemu
- Compile Pintos Kernel.
- In "$HOME/os-pg/pintos/src/utils/pintos":On line no. 257, replace "kernel.bin" to path pointing to kernel.bin file at "$HOME/os-pg/pintos/src/threads/build/kernel.bin".

- Open “$HOME/os-pg/pintos/src/utils/Pintos.pm” and replace “loader.bin” at line no. 362 to path till loader.bin located at “$HOME/os-pg/pintos/src/threads/build/loader.bin”.
- Run Pintos

➢ **Getting familiar with the source code (files and data structures)-**

Pintos starts from *init.c main* function which is in build under threads folder. It first takes the command line , breaks it into arguments and parse the options. It calls other init functions consisted in the booting process and then runs the specified acton of kernel command line by calling the *run_actions* function and then finishing up by calling shutdown function and exiting thread. The *run_actions* function extracts the action name to be performed (say run) and then calls the function to be performed according to the task, provided in command line, in the test.c file of tests/threads folder from the already defined list of function names.

➢ **Creating a hello.c file which consists of main function and prints “Hello World” message -**

command- pintos run hello-world

- First create file hellopintos.c file in tests/threads folder containing the functions we need to perform (here the function was to just print “Hello World”).
- Add entry as the function name in the array of struct test of tests.c file and tests.h

Now, when the run_actions function checks for the function to be called corresponding to the argument hellopintos in the tests[] array it will call the function defined in *hello_world.c* and hence desired action will take place.

Part 2: Pre-emption of threads

➢ **Understanding the thread sleep mechanism of pintos-**

The original version of the Pintos operating system uses a simple round-robin scheduler(with time quantum=4). A thread switch occurs

when a thread calls *thread_yield*, when a thread blocks, or during a timer interrupt when the current time slice has expired. The round-robin scheduler simply switches to the thread that is at the front of the list of ready threads. When a thread is created or unblocked, or its time-slice expires, that thread is added to the back of the ready list.

Problems with the basic pintos:

Busy Waiting -
In original pintos ,the timer_sleep function used to call thread_yield function which used to simply pushback the running thread to the back of the ready queue .On every call of thread_sleep function, the same procedure was followed. This process used to go on till the program get completely executed , hence threads were busy waiting for their turn in the ready queue without actually waiting aside for the time they were actually supposed to wait.

- **Reimplementing timer_sleep()**

Originally the function *timer_sleep* was calling the *thread_yield* function while the *timer_elapsed* function returned a value less than the value of ticks. But this caused busy waiting as explained above.
To remove Busy waiting , we created a new list and inserted the threads in it according to their wait time and hence removed busy waiting.
The procedure to be followed is-

- remove the while and *thread_yield* function call part from the *timer_sleep*
- change the structure of thread already defined in thread.h by adding a variable named *wait_time* which will store the time that thread needs to wait.
- Create a new list named as *wait_list* which will store the list of all the waiting threads.
- Create a function namely *sleep_thread* ,which takes the sum of current time and the time that thread needs to sleep for as argument and does the following-
  - disables interrupt ,so that timer interrupt does not occur.
  - Now, assign *wait_time* of thread as the argument it recieved since this will the time it needs to wait for which it will have to sleep.

- Inserting this thread to the *wait_list* according to the waiting time.
- Enable the interrupt

```
//=================================================================== //
//===when thread comes to sleep insert them in wait_list_according to its sleep time====//
void sleep_thread(int64_t ticks){
  struct thread * t = thread_current();
  ASSERT(!intr_context());
  enum intr_level level=intr_disable();
  if(t!=idle_thread){
  t->wait_time=ticks;
  list_insert_ordered(&wait_list,&t->elem,(list_less_func *)compare_wake_up_time
    ,NULL);
  thread_block();
  }
  intr_set_level(level);
}
//=============================================================================//
```

- *thread_sleep* function will return immediately if the ticks value is less than 0.
- call *sleep_thread* function from timer_sleep.
- Create a function named *wake_check* which basically just checks whether the wait_time of the front thread of wait_list greater than the current time or not ,if yes then pop it from the wait_list and unblock the thread so that it can be added to the ready queue.
- Call this *wake_check* function from timer_interrupt after it calls *thread_tick* function.

```
//=======check whether a thread has woke up ======================================//
void wake_check(){
   int64_t tk = timer_ticks();
  while (!list_empty(&wait_list))
  {
    struct list_elem * front = list_front (&wait_list);
    struct thread * front_entry = list_entry (front, struct thread, elem);
    if (front_entry->wait_time > tk){
        break;
    }
    list_pop_front(&wait_list);
    thread_unblock (front_entry);
  }

}
//=============================================================================//
```

Part 3: Implementation of priority scheduling

In the basic version of priority scheduling, a thread is assigned a priority when it is created.It is common for operating systems to assign "priorities" to threads. A priority is just a number between 0 and some maximum (63 in Pintos). The running thread should always be the thread that has the largest priority among all runnable threads.When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread immediately yields the processor to the new thread.

- **Implementation of a basic Priority Scheduling scheme-**

To implement the basic priority based schedulling , procedure we followed-

- Created a new comparator function named *compare_priority*, which compares the priority of two given threads and returns true if first thread has higher priority than the second one and false otherwise.

```
//===============compare priority of thread============================//
bool compare_priority(struct list_elem *thread1 ,
  struct list_elem *thread2,
  void *aux UNUSED){
  const struct thread *t1,*t2;
  t1=list_entry(thread1,struct thread,elem);
  t2=list_entry(thread2,struct thread,elem);
  return t1->priority > t2->priority;
}
//====================================================================//
```

- When the thread are being created, initially the priority of these threads are compared to the priority of current running thread.If it is greater than the current thread priority then *thread_yeild* is called and thread with higher priority takes over .

```
//================ if new thread has higher then current thread set it as current===========//
  old_level=intr_disable();
  if(t->priority>thread_current()->priority)
    thread_yield();
  intr_set_level(old_level);
//=========================================================================================//
```

- In the *thread_yeild* function, the insertion of thread in the ready queue is done according to its priority using the *compare_priority* comparator.
- Whenever *thread_unblock* function is called , it inserts the given thread to the ready queue according to its priority again using the *compare_priority* comparator.
- Whenever priority of currently running thread needs to be changed , the *thread_set_priority* function will change the priority of this thread and compares this new priority with the priority of the thread in front the ready queue and hence runs the one which has greater priority.

```
void
thread_set_priority (int new_priority)
{  ASSERT(!intr_context());
  enum intr_level level=intr_disable();
  int old_priority = thread_current()->priority;
  thread_current()->priority = new_priority;
//====check whether after changing priority current thread has highest priority or not ==//
  if(!list_empty(&ready_list)){

    struct list_elem * front = list_front (&ready_list);

    struct thread * front_entry = list_entry (front, struct thread, elem);

    if(new_priority<front_entry->priority){
      thread_yield();
    }
  }
//=============================================================================//
  intr_set_level(level);
```

➢ **Priority in semaphores**

To assign the semaphores to the thread in accordance to their priorities ,we did the following changes-

- In the *sema_down* function in synch.c file instead of simply pushing back the thread in waiters list of semaphores, inserting

them in order of their priority using the *compare_priority* comparator.

- In sema_up function ,simply call *thread_yield* function to get the next thread with the highest priority to work on the semaphore.

➢ Priority in Conditional variable

To assign the semaphores to the thread in accordance to their priorities whenever a given condition occurs(cond_signal is called), we did the folllowing changes-

- change in semaphore structure in synch.h – added another integer variable *thread_priority* ,which will store thread priority of the thread in the waiters list as the waiters list only contains one thread at a given instance.
- *Cond_wait* function declares a variable waiter of type struct semaphore_elem. This semaphore_elem has semaphore struct in it and list_elem.
- Created another comparator functon *compare_priority_semaphore* which compares semaphore's thread priority .
- In cond_wait function ,assign current threads priority to the waiter.semaphore.thread_priority and insert waiter_elem in waiters list (struct condition pointer) according to the comparator *compare_priority_semaphore.*

```
//================compare the priority of threads conditions waiters list ========//
bool compare_priority_semaphore(struct list_elem * l1 ,struct list_elem * l2, void *aux UNUSED){

    struct semaphore_elem * s1 =list_entry (l1, struct semaphore_elem, elem);
    struct semaphore_elem * s2 =list_entry (l2, struct semaphore_elem, elem);
    //printf("%d",list_size(&s1->semaphore.waiters));
    return s1->semaphore.thread_priority > s2->semaphore.thread_priority;

}
//=============================================================================//

void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));
  //=========================set semaphore priortiy to current thread priority =========//
  waiter.semaphore.thread_priority= thread_current()->priority;
  //=============================================================================//
  sema_init (&waiter.semaphore, 0);
  //list_push_back (&cond->waiters, &waiter.elem);
  //printf("%d\n",list_size(&cond->waiters));
  //=========================insert semaphore according to thread priority==============//
  list_insert_ordered(&cond->waiters,&waiter.elem,compare_priority_semaphore,NULL);
  //printf("%d",list_size(&s1->semaphore.waiters));
  //=============================================================================//
  lock_release (lock);
  sema_down (&waiter.semaphore);
  //printf("%d",list_size(&s1->semaphore.waiters));

  lock_acquire (lock);
}
```