** FOR INSTALLATION **

Part:1

Installation Requirements

binutils :
    sudo apt-get install binutils

Perl:
    perl -v

Qemu emulator:
    sudo apt-get install qemu

kvm --version

gdb --version

Installation Procedure

1)~/.bashrc

add executables files

2)in /utils, changed makefile to include -lm option for every line

change /utils/pintos-gdb file to point to correct path for macros

3)in /utils/Makefile

change to LDLIBS for -lm

4)in threads/Make.var , change --bochs to --qemu

5)in utils/pintos

in line 103, change bochs to qemu

in line 259, change the path to appropriate path for kernel.bin to threads/build

in line 623, change qemu to qemu-system-x86_64

6)in utils/Pintos.pm

at line 362, add the path of loader.bin

7)to run,

>>tcsh

>> pintos -- run alarm-multiple

Part 2: Preemption of Threads

Initially, Pintos was implementing a simple threading system.

The simple flow of execution for a sleeping thread is timer_sleep() continuously checks whether the current thread is sleeping or not, if yes then thread_yield() is called, which is releasing the processor voluntarily.

This initial implementation of timer_sleep() busy waits. Hence, it is wasteful.

The aim of this assignment is to remove busy waiting.

We added 'time' attribute in thread structure to denote wake up time for sleeping threads.

A trivial solution for this problem is to implement a new list sleep_queue, like ready_queue, which will hold the sleeping thread.

When a thread goes to sleep, it's thread element will be inserted into this sleep_queue.

On every tick in timer_tick(), check whether a sleeping thread needs to wake up or not.

If a thread needs to wake up, then it will be removed from sleep_list and inserted into ready_list.

There was an implementation choice for sleep_queue.One of them was to insert thread at rear of the list and while removing/checking find minimum using comparator which thread has to be woken up first.The other choice was to insert thread element in order of wake-up time in sleep_queue and remove/check front element of the list.

The second choice was preferred because the first choice will always require a complete list traversal to find the minimum, whereas first choice needs may or may not need to traverse entire list while inserting in order.

Also, the priority of threads was decided based on their wake-up time and not based on the amount of time it sleeps for.

Above solution was easy to understand and was successfully implemented.

Another solution which is actually implemented in this assignment is using Priority Queue using Min-Heap data structure

This will make the complexity of insert O(Log n), which is better than insert in ordered which takes O(n).
Checking for minimum takes O(1) , removal of minimum takes O(Log n) in priority queue.

End of Part2

Part 3: Implementing priority scheduling

In this part of the assignment, the aim was to implement get priority function, which would get priority of the current thread, and set priority function, which would set the priority of current thread and if possible preempt current thread for a higher priority thread.Also, queues of semaphore and condition variables to be maintained on the basis of priority.

The solution for this task is to maintain a list as per priorities. Custom comparator function was provided to list insertion function. For set priority, we can set the priority of the current thread by changing it's priority value and then check priority of the front thread element of the ready list. If current thread priority is less than ready queue front thread element priority then thread_yield is called. For get priority, we can get priority of the current thread by returning its priority value.

In case of semaphores, on sema_down operation, if semaphore already has a value 0 then thread will add itself in waiters list of that semaphore and block itself. For sema_up operation, thread with highest priority will be unblocked.

Difficulty arised in condition variable part of this assignment.In condition variable part, it became difficult to maintain the list as per the priorities of threads because a semaphore element was maintained in the waiter list of a condition variable.

Hence solution for this problem was to maintain a priority value of each semaphore element, which is assigned by thread priority.

End of Part3

Part 4: Advanced scheduler

This part of assignment involved implementation of fixed point arithmetic, using integers and also the implementation of formulae to calculate niceness value of a thread, load average, recent CPU usage and priority.

Logical implementation of this part was trivial as compared to earlier parts as this part was completely formulae based and already mentioned on the reference links. Key part while implementing Advanced scheduler was getting accurate output from fixed-point arithmetic functions to add, subtract, divide and multiply, which required some care.

Reference Links Used for mathematical functions :
http://math.hws.edu/eck/cs431/f16/assg5/fixed_arithmetic.h, http://www.scs.stanford.edu/07au-cs140/pintos/pintos_7.html#SEC128
refrence link used for logic to calculate niceness and thread priorities :
http://math.hws.edu/eck/cs431/f16/assg5/index.html

Also, it was important not destabilize the previously implemented parts, by checking mlfqs option.

End of Part4