CMPT300 Project 0 Solution

1) What module (file) and what function is function ThreadTest started from?
Answer: From the main() function in code/threads/main.cc

2) What does the following statement do?
t->fork(procedure, parameter);
What are the parameters to fork and how are they used?
Answer: It generates a new thread which starts execution at the entry point “procedure” with a single argument “parameter” passed into the procedure. The parameters are pushed onto the stack, and the thread is put into the ready queue. NOTE: although our definition allows only a single integer argument to be passed to the procedure, it is possible to pass multiple arguments by making them fields of a structure, and passing a pointer to the structure.

3) What does the following statement do?
DEBUG('t', "Entering SimpleTest");
How are the two parameters for DEBUG used?
Answer: DEBUG prints a debug message, if flag is enabled. Like printf, only with an extra argument on the front. The statement will print "Entering SimpleTest". And "t" is the flag. If we want to print this debug message, we will have to run the executable by -d t option.

4) What does yield() do? What happens if yield() is executed and there is no other thread to run?
Answer: Relinquish the CPU if any other thread is ready to run. If so, put the thread on the end of the ready list, so that it will eventually be re-scheduled. NOTE: returns immediately if no other thread on the ready queue. Otherwise returns when the thread eventually works its way to the front of the ready list and gets re-scheduled.

5) When does a thread execute Finish()? Can a thread execute return() instead? Can it execute exit() instead?
Answer: A thread calls Finish() when it has finished execution. return() and exit() should not be used, since Finish() lets the system do some cleanup before destroying the thread. We do not immediately de-allocate the thread data structure or the execution stack, because the thread is still running and still on the stack. Instead, we set "threadToBeDestroyed", put the current thread to sleep. Then the scheduler runs, and Scheduler::Run() will call the destructor of this thread, once we're running in the context of a different thread.

6) What does scheduler::ReadyToRun() do?
Answer: Mark a thread as ready, but not running. Put it on the ready queue, for later scheduling onto the CPU.

7) What does scheduler::Run() do and how is it different from scheduler::ReadyToRun()?
Answer: Dispatch the CPU to nextThread. Save the state of the old thread, and load the state of the new thread, by calling the machine dependent context switch routine, SWITCH.

ReadyToRun() puts the thread into the ready queue, while Run() actually runs the thread (based on the CPU scheduling algorithm).

8) What data structure is defined in list.h and list.cc and how is it used in scheduler?
Answer: A list can contain any type of data structure as an item on the list: thread control blocks, pending interrupts, etc. That is why each item is a "void *", or in other words, a "pointers to anything". In the scheduler, the “ready list” or “ready queue” is used to manage ready threads. Scheduler::ReadyToRun(Thread *thread) appends the thread to the end of the ready list; Scheduler::Run(Thread *nextThread) removes the first thread in the ready list and runs it.

9) What module and what portion actually removes data structures of a finished thread and why the thread itself cannot do the cleanup?
Answer: Scheduler::Run (Thread *nextThread) in /threads/scheduler.cc:

```
if(threadToBeDestroyed!=NULL){
        delete threadToBeDestroyed;
        threadToBeDestroyed=NULL;
}
```

The thread that is finishing cannot do the cleanup by itself, since the thread is still running and still on the stack. It must pass control to the scheduler to do the cleanup AFTER it has finished.

10) What module keeps track of time used by nachos programs? Why cannot standard Unix time be used for this purpose?
Answer: Timer.h and Timer.cc contain routines to emulate a hardware timer device. A hardware timer generates a CPU interrupt every X milliseconds. This means it can be used for implementing time-slicing. We emulate a hardware timer by scheduling an interrupt to occur every time stats->totalTicks has increased by TimerTicks. Unix time cannot be used, since Nachos is a simulated OS that runs within a single UNIX/Linux process; it keeps track of time with its own unit Ticks While UNIX time from hardware clock is real-time (wall-clock time) that is actually divided among multiple concurrent processes (one of which runs Nachos).