Vidhya Lakshmi Venkatarama
CS533: Concepts of Operating Systems
Course Project: Assignment 2: Write Up

a) This assignment is an extension of the previous context switching assignment where we created 2 threads and switched between those. Here, we create more threads (4) and manage these using a scheduler. This scheduler follows the round-robin scheduling principle. This project has been implemented according to the guidelines in the specification. For detection of context switching & threads scheduling, an extra field *thread_id* is added to the *struct thread*. The given test file *main.c* has been used to test the working of implementation with slight modifications. The main function calls the scheduler to begin and forks 3 threads to print the nth prime number, where n = n1, n2 & n3. The *print_nth_prime* function yields on every number that is checked to be a prime.

On running the unmodified *main.c*, it is observed that the 10000^{th} prime number is printed before the 20000^{th}, which is printed before the 30000^{th} prime number, though the order of forking threads to execute the *print_nth_prime* is different. This is because the thread with smallest n yields less number of time and is done before the others. Hence, the output is in increasing order of prime numbers.

Few modifications were made to the *main.c* file to make this observation on the order of output.

1. n1 = 23, n2 = 18, n1 =20 and the *print_nth_prime* function yields whenever n%2 is 0. The order of output was 23^{rd} prime number, 18^{th} prime number, 20^{th} prime number. This is because when n is 23, the function doesn't yield and it does in the other two cases.
2. With n1 =20000, n2 =10000, n3 = 30000, a blocking (*scanf*) statement was added just after forking the first thread for n1. This blocked the entire process until completion of input and then resumed and output order was same as before.
3. An extra call to *scheduler_end* was made just after forking the first thread. This obviously let the first thread complete before forking the other two.

A thread, when done, returns to *thread_wrap* and this is where its status is set to DONE so that it is not put on the scheduler's ready list again. An infinitely yielding loop is added to the end of *thread_wrap* to avoid segmentation fault as discussed in assignment 1.

b)

1. The threads' memory can be reclaimed once we are sure that it is no more of use. This normally happens when the thread exits (except in cases when the thread is to return a result or is associated with a join). In our set up, since there is no returning of results or any implementation of 'join', we can reclaim the threads stack and thread control block as soon as they finish their task. Though a thread's written to the *thread_wrap* function implies that the thread finished the task assigned to it, it is still running and its memory can't be freed yet. On the call to yield, we check the running thread's state and do not put it in scheduler's ready list if it is DONE. It can be thought of freeing the finished thread's memory here, but it is still the same thread running until the task of switching is complete. It is not the best approach to reclaim the memory in *scheduler_end* because this function would usually be called at the end of the application and freed memory won't be of much use here. A suitable way would be to collect all the exiting threads in a separate data structure, in the yield function and to recheck & reclaim the memory in the next call to yield. In a real-life application, such a strategy would lead to too many allocations and deallocations. To avoid this, we could create and manage a thread pool and use it instead of forking and destroying threads.

2. A scheduler's procedure is called by the running thread and during the execution, we are running the calling thread. *thread_fork* is called by the parent of created thread and in the test code we have used, it is always the main thread that forks a new thread and hence, it is the one running in the *thread_fork* method. The call to yield is made by almost all of the created threads & the main thread and they are running during the execution. *scheduler_begin* and *scheduler_end* methods are called and executed by the main thread. Whenever there is a call to yield, there is a possibility of context switch (except when there is no thread on the scheduler's ready list). In this function, the calling thread executes until *thread_switch* (in the assembly code) is called and the control is passed over to the next thread in the ready list. This is where one thread stops running and the other starts.
3. From the observations of testing the implemented functionalities, a blocking call (eg. *scanf*) at one thread pauses the entire program from running, which doesn't resume until the call completes. In order to let other threads run behind the wait, there is need for implementation of asynchronous blocking functionality to our threading interface. Also, if we want to add shared variables in our program, we would need synchronization primitives to protect them and yet achieve concurrency. These would be two essential functionalities to improve our threading API.