Vidhya Lakshmi Venkatarama
CS533: Concepts of Operating Systems
Course Project: Assignment 1: Write-up

a) In this assignment, a simple context switching between two threads has been implemented using the guidelines given in the specification. The given "fun_with_threads" function has been re-used with little modification. In order to detect instances of context switching and currently running thread, an extra field " thread_id" has been added to the thread_t data structure. Various sets of maximum value for loop variables for yielding loops in the test function and the 'main' function were used for testing the cases when the subordinate thread finishes first and those where the main thread.
One such instance could be:

1. The main thread finishes first when:
   a. the maximum value of loop variable i in main is 4 and the thread yields when i%4 is 1
   b. the loop variable in fun_with_threads, i<8 and thread yields when i%3 ==1

   In such a case, no segmentation fault occurred.
2. The secondary thread finishes first when:
   a. in main: i<13, thread yields when i%4==1
   b. in fun_with_threads: i<8 and thread yields when i%3 ==1

   This case threw a segmentation fault.

The reason for segmentation fault as explained in the guidelines is that when the subordinate thread finishes first, it returns to its caller, the 'thread_wrap' which has nowhere to return to. As a solution to this, a call to 'yield()' was added at the end of the 'thread_wrap()' method (to execute when the subordinate thread returns). However, this was unable to fix the segmentation fault as in the case when secondary thread finishes first and calls an 'yield()' from'thread_wrap()', and the control returns to the 'main()' function which is still yielding and wants to switch to secondary thread which again results in segmentation fault. As a fix to this, an infinite loop for yielding has been added at the end of 'thread_wrap()' method. In addition, a number of print statements have been included in order to ease the debugging of code.
Due to the presence of an infinitely yielding loop in the 'thread_wrap' function, we cannot deallocate the memory associated with the subordinate thread in the scope of this function. One possible way could be to do this after the yielding loop in the end of the main() function, since there is no possible way of a secondary thread running after the main one exits.

b)

1. The thread_start method we have implemented saves the callee-save registers & switches the control to the created thread which in turn starts executing its initial function. Another way of creating multiple threads could be through a function, which saves the callee-save registers, allocates memory for the new thread's stack, initializes its stack pointer and the program counter (or if we use initial function and arguments, switch it) and return back to what it was doing. The created threads must be scheduled in order to run and hence we would require a

scheduling algorithm. This scheduling would also manage the switching between threads that are ready to run.

2. The simple threading that we have implemented here will be useful when:
   i. threads need to wait without blocking, i.e. when they yield to other threads.
   ii. for concurrency in multiprocessors.

   In order to make it more useful, we can
   iii. add a function to create more threads.
   iv. implement a scheduler for managing these threads.
   v. add blocking code for threads.
   vi. include synchronisation primitives.

3. It is safe to free a thread's stack once we are sure that thread or its stack content is no longer required. Normally, this can be done once the thread exits(in the thread library in the function where the thread communicates that it exited). The thread control block can also be freed as soon as the thread exits when its context is of no further use .

   However, this will be dangerous if we want to return results to other threads or implement a join. In cases where results are required from the exiting thread, we will have to save the results in the heap and make the thread return that address (probably by saving it in the Thread Control Block in some variable). If the exiting thread is associated with a join, the thread's ID and status are required to prevent the parent from blocking forever. We could keep this thread in what is called as 'zombie state' and free its stack, preserving only the Thread Control Block,or empty all it's content from the thread control block except for the ID. It depends on how we implement our thread structure and if the thread structure would be still valid on freeing the TCB.