# COMP 310 A2 Grading Rubric

October 22, 2020

## 1 Required Functions

The following functions must be implemented

### 1.1 `sut_init()` **APPLICATION**

This function is called by the user of your library before calling any other function.

You can use this function to perform any initialization (such as creating your kernel level threads).

### 1.2 `sut_create(sut_task_f fn)` **APPLICATION**

This function will be called by the user of your library in order to add a new task which should be scheduled to run on your `C-Exec` thread.

The parameter `fn` is a function the user would like to run in the task. You can see the provided tests for examples.

### 1.3 `sut_shutdown()` **APPLICATION**

This function will be called by the user when they are done adding tasks and would like to wait for the currently running tasks to finish and then clean up any internal library state.

You can use this function to join your threads and wait for their work to complete. Make sure you don't interrupt tasks that haven't yet finished!

### 1.4 `sut_yield()` **TASK**

This function will be called within a user task. When called, the state/context of the currently running task should be saved and rescheduled to be resumed later. The `C-Exec` thread should be instructed to schedule the next task in its queue of ready task.

## 1.5 `sut_exit()` **TASK**

This function will be called within a user task. When called, the state/context of the currently running task should be destroyed – you should not resume this task again later. The `C-Exec` thread should be instructed to schedule the next task in its queue of ready tasks.

## 1.6 `sut_open(char *dest, int port)` **TASK**

This function will be called within a user task. When called, the `I-Exec` thread should be instructed to open a TCP socket connection to the address specified by `dest` on port `port`. You can refer to the client portion of your first assignment if you don't remember how to do this, it should be very similar. The socket should be associated to the current task.

After signaling the `I-Exec` thread, the currently running task (the one that called `sut_open()`) should have its state/context saved. The task should not be resumed until the `I-Exec` thread has completed the operation. In the meanwhile, the `C-Exec` thread should be instructed to schedule the next task in its queue of ready tasks.

Remember **not to perform any IO on the `C-Exec` thread**. The IO should be performed on the `I-Exec` thread. Also remember **not to block the `C-Exec` thread**. After requesting IO, the `C-Exec` thread should not wait for the request to be serviced. Other available tasks should be scheduled to run until the `I-Exec` thread has completed the operation.

## 1.7 `sut_read()` **TASK**

This function will be called within a user task. It is an error to call this function from a task that has not already successfully called `sut_open()`. When called, the `I-Exec` thread should be instructed to read from the task's associated socket until there is no more data to read.

After signaling the `I-Exec` thread to perform the read, the currently running task (the one that called `sut_read()`) should have its state/context saved. The task should not be resumed until the `I-Exec` thread has completed the read operation. In the meanwhile, the `C-Exec` thread should be instructed to schedule the next task in its queue of ready tasks.

Remember **not to perform any IO on the `C-Exec` thread**. The IO should be performed on the `I-Exec` thread. Also remember **not to block**

**the `C-Exec` thread**. After requesting IO, the `C-Exec` thread should not wait for the request to be serviced. Other available tasks should be scheduled to run until the `I-Exec` thread has completed the operation.

### 1.8 `sut_write(char *buf, int size)` **TASK**

This function will be called within a user task. It is an error to call this function from a task that has not already successfully called `sut_open()`. When called, the `I-Exec` thread should be instructed to write `size` bytes from `buf` to the socket associated with the current task.

Note that this is a *non-blocking write*. After the `I-Exec` thread is signaled to perform the write, the calling task **should continue on the `C-Exec` thread** and not be interrupted. The write should be performed by the `I-Exec` thread concurrently with the still-running calling task on the `C-Exec` thread.

Remember **not to perform any IO on the `C-Exec` thread**. The IO should be performed on the `I-Exec` thread.

### 1.9 `sut_close()` **TASK**

This function will be called within a user task. It is an error to call this function from a task that has not already successfully called `sut_open()`. When called, the `I-Exec` thread should close the socket associated with the current task.

Note that this is a *non-blocking close*. After the `I-Exec` thread is signaled to close the socket, the calling task **should continue on the `C-Exec` thread** and not be interrupted. The close should be performed by the `I-Exec` thread concurrently with the still-running calling task on the `C-Exec` thread.

Remember **not to perform any IO on the `C-Exec` thread**. The IO should be performed on the `I-Exec` thread.

## 2 Implementation

You should use `pthreads` for your `C-Exec` and `I-Exec` kernel threads. You should use the `ucontext` API (`swapcontext()`, `getcontext()`, `makecontext()`) for user tasks. You should use `pthread_mutex` for synchronization. Examples of the use of these tools were given in the 9 October and 16 October advanced tutorials.

Note that **only one user task will be requesting IO at any given time**. This *should* make your implementation simpler, because you do not need to create as many queues to handle IO as you would in the general case. The `I-Exec` thread will only have one request to process at a time, which means it should be easy to determine which task should be rescheduled when the request is processed. We will not evaluate you on how many queues you use for IO.

## 3 Grading Scheme

The grades will be distributed in the following way.

Each of the following subsections are worth 10% of the total grade. The relative worth of a requirement within a subsection is according to its point value.

### 3.1 `sut_init()`

| Requirement | Points |
|---|---|
| A `pthread` is created for each of `C-Exec` and `I-Exec` | 1 |
| The `C-Exec` and `I-Exec` threads outlive the function. | 1 |
| The function does not block its caller. | 1 |

### 3.2 `sut_create(sut_task_f fn)`

| Requirement | Points |
|---|---|
| `fn` runs on `C-Exec`, not the caller's thread. | 2 |
| A new context is properly created for the new task. | 1 |
| There is no data race. | 1 |
| The function does not block its caller. | 1 |

### 3.3 `sut_shutdown()`

| Requirement | Points |
|---|---|
| The function eventually returns. | 2 |
| Currently running tasks are not interrupted. | 1 |
| There is no data race. | 1 |
| Any currently running `pthread` is joined | 1 |

### 3.4 `sut_yield()`

| Requirement | Points |
|---|---:|
| The current task context is properly saved. | 2 |
| The `C-Exec` thread schedules the next available task. | 2 |
| The task is eventually rescheduled. | 2 |
| There is no data race. | 1 |

### 3.5 `sut_exit()`

| Requirement | Points |
|---|---:|
| The `C-Exec` thread schedules the next available task. | 2 |
| The task is never rescheduled. | 2 |
| There is no data race. | 1 |
| The current task context is removed from any state. | 1 |

### 3.6 `sut_open(char *dest, int port)`

| Requirement | Points |
|---|---:|
| **Only** the `I-Exec` thread performs IO. | 5 |
| The `C-Exec` thread is not blocked. | 2 |
| The `C-Exec` thread schedules the next available task. | 2 |
| The task is only resumed when the IO has been performed. | 2 |
| The task is eventually rescheduled. | 2 |
| There is no data race. | 1 |

### 3.7 `sut_read()`

| Requirement | Points |
|---|---:|
| **Only** the `I-Exec` thread performs IO. | 5 |
| The `C-Exec` thread is not blocked. | 2 |
| The `C-Exec` thread schedules the next available task. | 2 |
| The task is only resumed when the IO has been performed. | 2 |
| The task is eventually rescheduled. | 2 |
| There is no data race. | 1 |

### 3.8 `sut_write(char *buf, int size)`

| Requirement | Points |
|---|---|
| **Only** the `I-Exec` thread performs IO. | 5 |
| The `C-Exec` thread is not blocked. | 2 |
| The `C-Exec` thread continues the current task. | 2 |
| There is no data race. | 1 |

### 3.9 `sut_close()`

| Requirement | Points |
|---|---|
| **Only** the `I-Exec` thread performs IO. | 5 |
| The `C-Exec` thread is not blocked. | 2 |
| The `C-Exec` thread continues the current task. | 2 |
| There is no data race. | 1 |

### 3.10 Code Quality

The submitted code should be organized, legible, and demonstrate the programmer's soundness of mind and understanding of the material.

The program should compile as-submitted, free from any errors or warnings. There should not be memory leaks or concurrency issues.

## 4 Submission

Submit all your source files and a `Makefile` in a zip file following the naming pattern `StudentName_ID.zip`, as with the previous assignment. You may optionally submit a `README` containing anything you would like to communicate to the grader. **Do not submit compiled binaries.**

A sample directory structure looks like this

```
sut.c
sut.h
queue.h
Makefile
```

## 5 Getting Help

If you need help, post to the discussion boards and a TA will try to answer your question as quickly as possible. You can also come to office hours if you need to talk through your code.