CS 367 Project 1 - Fall 2024:
Egul Scheduler - Testing Strategies

1 Introduction

You will finish writing functions for the **Egul CPU Scheduler**. As you will need to finish most of the 11 functions **before** StrawHat will work, you may want to test your functions outside of that environment to begin with.

Since your code is all within a single file (**egul_sched.c**), you can use a custom **tester** program that will just call your functions directly with any input you like and allow you to see if the output is what you expect. This is a type of testing known as **unit testing**, where you can have a small program that just consists of main and a few testing functions.

These functions need to set up the environment the way you want it for a test case, then you can call your op functions directly and see if they did the right things.

For example, if you want to see if your **egul_insert** function is working right, you may need to create the Schedule Header and at least one queue (Ready Queue) and you may want to create a few processes to test inserting. Once you have all that set-up, then you can call your **egul_insert** function to see if it's putting the new processes in properly.

Since this would be a separate main, you don't need to have everything running to begin testing!

To help with this process, we have provided a Testing main file for you with a few examples inside of it on how you can use it to test your code separately and we set it up to work using the Makefile. The code provided in **src/test_egul_sched.c** is just sample code and you can add/remove/modify it as you like. It's just a sample to show you how to use this type of tester.

When you're ready to test, you can use a special make command: **make tester**. This will make the **tester** program and then you can run **tester** to run your tests. This also makes it a lot easier to debug your code with **gdb** than it would be if you were debugging through StrawHat TM.

Again, this is not a full unit tester for your program. This is a framework that you can use to write your own.

We added one unit test as an example so you can see how you can write your own.

2 Testing your Code

This is not necessary to use at all, however, if you want to test your code without running StrawHat, we do have a special **src/test_egul_sched.c** source file that has a main you can use to call your functions with whatever arguments you want and you can then look at the outputs to test your functions in isolation first.

To build: `make tester`

To run: `./tester`

The source file we provide also shows you how to use a few helper print macros we made for testing.

These two will work just like printf, but they will always print, regardless of how **g_debug_mode** is set.

- **PRINT_INFO** This prints out a nice info message.
- **MARK** This also prints the filename, line number, and function that you called it from.

The rest of the macros all work exactly like printf as well! So, you can use format codes (eg. %d) and values like printf. These will only print if **g_debug_mode** is set to 1.

Note: You can use these for debugging and then when you are ready, you can set **g_debug_mode** to 0 and recompile the program and they'll no longer print anything without having to delete these lines.

- **PRINT_STATUS** Prints a nice status message.
- **PRINT_DEBUG** Prints a nice debug message.
- **PRINT_WARNING** Prints a nice warning message.
- **PRINT_ERROR** Prints a nice error message.

3 Sample Outputs of tester

The tester calls a demo function to print out demo outputs of all the macros. We're going to ignore these here.

Since our sample unit test case tests **egul_initialize**, here are two outputs:

3.1 Sample Output with a completely unwritten `egul_initialize` function…

```
[Info ] Test 1: Testing egul_initialize()
[ERROR] ...egul_initialize returned NULL!
[Status] ...egul_initialize test complete.

[Status] All tests complete!
```

This sample test case can not go any farther when **egul_initialize** output NULL, so we used the PRINT_ERROR macro to print out an error message. This lets us very quickly see how the test run went. Now we tested the code, we have identified a bug! **egul_initialize** is returning NULL. This is expected, of course, since we haven't written it yet. So, let's look at an output with that function completely written.

3.2 Sample Output with a properly working `egul_initialize` function…

```
[Info ] Test 1: Testing egul_initialize()
[Status] ...Checking the Ready Queue
[Status] ...Checking the Suspended Queue
[Status] ...Checking the Terminated Queue
[Status] ...egul_initialize test complete.

[Status] All tests complete!
```

The function being called here is **`test_egul_initialize`**, which we wrote to just call our **`egul_initialize`** function for testing and then check to see if the pointer we get back meets the standards in the documentation.

If you look through the function code in **`test_egul_sched.c`**, you will see that for each test we do, we use PRINT_STATUS first to tell us what we're about to test and then if the value isn't correct, we call PRINT_ERROR.

So, if you check all of the different requirements of what egul_initialize was supposed to do, then when you run the tester, you will only see the status messages telling you what is being tested. This means we passed all of our tests.

Of course, there's nothing special about any of this. We wrote a single test case in that tester program as a demonstration of how you can write your own test cases. If you want to, you can follow the same format and use PRINT_STATUS to tell you what you're testing and then PRINT_ERROR or PRINT_WARNING to tell you something isn't working correctly. As long as you have a check for all of the different requirements for each function, then this will give you a very thorough test for bugs.

The best part is that if any of your errors or warnings are printed, your status messages will tell you exactly what you were doing, and your error or warning messages should tell you what went wrong. Now you know you have a bug and a very good idea about where it is. This makes debugging much easier!

4 Notes on Output

There isn't any **expected** output here because the code is entirely up to you! We just included a very simple starting function to show you how you **could** use this to test your functions outside of StrawHat. You don't have to use this at all, but if you want to, it's easier to debug your code than with the full system running.

5 Notes on GDB and the Tester

For GDB while running the tester you can just run **gdb** ./**tester** since it's a separate program from strawhat.

Now you don't have to worry about any of the threads, processes, or other junk that the StrawHat is doing that might distract you from testing your code.

It is MUCH easier to call your functions from this tester code and to use gdb on this tester program than it is to run gdb on the full strawhat. This tester should only call your own functions, so it leaves our code out of it.

6 Document Changelog

- v1.0 (build 4a99cc46)
    - Release Version