Deliverable 3: Test Report

Ara Demmings

Joel Mercer

March 23, 2015

Professor Kent

CS 4405

In this document we have included details of our tests as well as their outcomes for all essential functions outlined in the given os.h header file. We also included a note section for any issues we discovered in the custom functions present in the operating system. Included in the zip file with this PDF are two files containing testing code.

**1 Operating System Testing**

1.1 OS Initialization

1.1 OS\_Init - We tested this by placing print statements to see if the assigning of values for the process table were occurring correctly and it seems that they are. It appears to work was intended as we never received a readme document outlining the intention of this function.

1.1.2 OS\_Start - When we tested this we found that OS\_Start does not use the scheduler, it just runs a while loop that does not end unless OS\_Abort is called.

1.1.3 OS\_Abort - We tested this by calling it manually. The OS did not abort, so we used a print statement to check the value of "is\_halting" and the value does change like it should, but the OS does not abort. However, when is\_halting is equal to 0, the timer does interrupt OS\_Start and it calls the scheduler. It does return to main and the timer never stops and main never returns.

1.2 Process Management Primitives

1.2.1 OS\_Create - We found that programs never seem to really run after they are created. We tried to run the led\_toggle\_program that was included with the OS and it looked like it was running but nothing else would run. We found that the function that the program calls, toggle\_led, was being called from inside do\_scheduling and when this was commented out, the LED lights no longer turned on. As it appears, programs do not actually run.

Another issue that we noticed is that processes start indexing from 1 to 15, resulting in an incorrect number of processes. This means that processes[0] does not hold the first process created, processes[1] does. This also limits the number of processes created since there are only 15 (1 through 15, inclusive) rather than 16 (0 through 15, inclusive).

*Test Case 1:* Create One Process - We created a process and printed the values of the process in the process table and it is being added to the table. The PID is correctly returned.

*Test Case 2:* Create 16 Processes - When we created the maximum number of processes, only 14 were added to the table. We found this by printing the process values for every entry in the process table. The 14th - 16th entries returned 0s.

*Test Case 3:* Create 17 Processes - When we created 17 processes, 14 were added to the table. Again, we found this by printing the table entries. The 14th - 17th processes returned 0s and we were able to create more than the maximum number of processes.

*Test Case 4:* Create and Run a Process - We created a very simple process:

void test(){

printf("Test Process");

}

When we created it from the main method and tried to run it we found that it never really ran.

1.2.2 OS\_Terminate - When we were looking through the code, we found that this function called a series of other functions, leading to the handle\_sys\_call function where it appears a switch statement should call \_OS\_Terminate but this section was not written in. We were unable to attempt to terminate a process since we could never run one.

In addition, the code for \_OS\_Terminate does not release any of the process’s resources (memory, FIFOs, semaphores that it was waiting on or held)

1.2.3 OS\_Yield - We were unable to test this from a running program but instead we directly called OS\_Yield. We found that it was unable to yield as it entered the\_exception and stored the registers but never called the interrupt\_handler funtion. We discovered this by placing print statements in both interrupt\_handler and the\_exeption. Only the ones placed before the call to interrupt\_handler printed and the ones within interrupt\_handler never printed.

1.2.4 OS\_GetParam - We were unable to test this from a running program but instead we created a process using OS\_Create then called OS\_GetParam. Using a print statement, we found that the value OS\_GetParam retrieved was equal to that of processes[0].arg. We were never able to completely test this since we never could make a program run, and nothing ever enters the position at processes[0], but it seems like it is functioning correctly.

1.3 Semaphore Primitives

1.3.1 OS\_InitSem - When looking at the code, it only aborts when a number less than 0 or greater than 16 is given as an identifier, meaning that it allows the creation of 17 semaphores. Additionally, in every case, after OS\_InitSem is called it will abort. We placed a print statement inside the if statement which aborts and it was always printed. When looking through the code, we noticed that semaphores[s][0] is never initialized to a value, and thus whatever was in that memory, typically not -1, so this condition was always met.

*Test Case 1:* Create One Semaphore - We tested this by calling OS\_InitSem and it seemed to function correctly.

*Test Case 2:* Create 16 Semaphores - We tested this by creating 16 semaphores with identifiers between 0 and 15, inclusive. This was successful.

*Test Case 3:* Create 17 Semaphores - We tested this by creating too many semaphores, which it allowed us to do successfully. It is supposed to abort when given an identifier larger than 16, but it does not.

1.3.2 OS\_Wait - By looking at the code, this function has some issues. The int variable "i" is declared but never initialized to anything so when OS\_Wait is called for the first time for a semaphore, the while loop that sets it does not get run since the two values in the semaphores array are the same. This means that, in this case, any time that it is used in this function it would have any random value in it.

*Test Case 1:* Call Wait Once - We tested this by creating a semaphore and calling wait, which seemed successful in that we could do it with no errors or crashing. It successfully decremented the number of available semaphores for that particular identifier. We were never able to run a process to see if interrupts were correctly disabled.

*Test Case 2:* Call Wait More than Once - We tested this by creating one semaphore with an n of 4 and calling wait four times on the same semaphore. It worked for the first wait call, but after that it did not work. Calling wait twice caused it to enter the while loop in OS\_Wait and then loop indefinitely. In the case where n is equal to 1, the first wait call is successful and then calling it again causes it to loop indefinitely.

*Test Case 3:* Check if it Yields - We tested this by placing a print statement in the while loop before OS\_Yield was called and it does not seem to ever enter this section (See 2.3 OS\_Yield). This also may be an issue caused by the looping described in the previous test case.

1.3.3 OS\_Signal - We were unable to test this function since, although present, it was completely empty.

1.4 FIFO Primitives

1.4.1 OS\_InitFiFo - This section of code was present and commented out, however, in an incomplete form. When we uncommented it, the function would not compile. By looking at the code, it has some issues such as the int variable "found" never being initialized before use. It will also not work since the FIFO struct has been commented out and is largely incomplete.

1.4.2 OS\_Write - We were unable to test this function as it was commented out and essentially empty.

1.4.3 OS\_Read - We were unable to test this function as it was missing entirely.

1.5 Memory Primitives

1.5.1 OS\_InitMemory - We were unable to test this function as it was missing entirely.

1.5.2 OS\_Malloc - This section of code was present and commented out. When we uncommented it, the function would not compile.

1.5.3 OS\_Free - We were unable to test this function as it was missing entirely.

1.6 Other Notes

- When OS\_Start is not called, the main method seems to infinitely loop, even with a return statement. This was found by removing the call to OS\_Start.

- OS\_Init starts the timer before OS\_Start is ever called. Perhaps it should wait until OS\_Start is called to begin the timer as it was affecting the creation of a large number of processes and print statements.

- process\_start is never called. We tested this by placing a print statement inside the function but it never printed anything.

- When the\_exception is called, it never gets to call interrupt\_handler. We found this when testing OS\_Yield.

2.0 Scheduling

We attempted to create processes of different level (device, periodic, and periodic) to test the scheduling but no processes ever were ever able to be run.

3.0 Testing Files

We have included three text files containing code to test components of the OS requiring creation of structures, specifically semaphores and processes. All other testing was performed largely using print statements. In order to use this code, simply copy it and then paste it into the main method in os.c.

3.1 oscreate.txt - This code creates 17 processes, more than should be allowed. This can be altered by adding or removing creation lines in order to match the test cases listed above. It also contains code to print out the process ID of each process that is created and the process table data or each one.

3.2 sch and create.txt - This code contains the test program used to test OS\_Create, it is also the one we used to try to run programs. It also contains the code used to test scheduling.

3.3 semaphore.txt - This code creates 17 semaphores, more than should be allowed. This can be altered by adding or removing creation lines in order to match the test cases listed above. It also contains code to utilize OS\_Wait, consistent with test case 2, listed under 1.3.2 (OS\_Wait).