x86 Assembly Writeup

Name: Joshua Abraham

Date: 9 November 2017

Current Module: x86 Assembly

Project Name: "Fibonacci"

Project Goals:

The relay project aims to create a program written in x86 assembly that calculates the n^{th} Fibonacci number, where $0 \leq n \leq 500$.

Considerations:

- The program must take one command line parameter, n.
- The program should produce an error message if no parameter is passed.
- The program must be able to calculate Fibonacci numbers from $0 \leq n \leq 500$.

Initial Design:

The project is composed of the following files:

· *Makefile*: The main makefile for the project.

· *fibonacci.s*: The source code for fibonacci.

· *test.sh*: The test runner for the project.

Data Flow:

My fibonacci program begins execution by performing command-line argument checks and string to integer conversion. The first check is on the number of arguments. The second check occurs during the string conversion routine. If the user input begins with a '-', the program exits and prints a usage statement. Next, it finishes converting the input into an integer. The program then enters the Fibonacci algorithm shown in Figure 1:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases} \quad (1)$$

If the input is zero or one, the program handles these "Edge cases" in Edge_fib and exits cleanly. Otherwise the program continues into a loop where it uses three six-QWORD chunk numbers to calculate a desired Fibonacci number.

```
long num, one = 0, two = 1, fib_num;
for (int i = 1; i < num; i++) {
            fib_num = one + two;
            one = two;
            two = fib_num;
}
```

Listing 1: Fibonacci in C

sleeps for 5000 nanoseconds. This loop is performed until the user gives a control-D to the dispatcher. In order to prevent a control-C from prematurely ending the dispatcher (which would result in the shared memory segment not being deleted), a tiny SIGINT handler is used at the beginning of the program. The listeners test for the existence of the shared memory segment and if it does not exist, the listener exits cleanly. Once the user sends a control-D to the dispatcher, the dispatcher sets the shared memory to EOT which is the magical number that closes the listeners.

Communications Protocol:

The dispatcher and listener performed IPC via System V shared memory. This communication was one-way and used exactly one byte of memory.

Potential Pitfalls:

- Listeners re-printing data in shared memory before it is deleted.
- Listeners closing when user gives a control-D to the dispatcher.
- Dispatcher not prematurely closing before sending EOT to listeners.

Test Plan:

User Tests:

· Performed test specified in the project supplement manual.

· Sent dispatcher a control-C in the middle of execution.

· Checked system IPC mechanisms via the "ipcs" command to ensure that all shared memory is deleted after successful execution.

Test Cases:

· Tested SIGINT handler and shared memory deletion.

Conclusion:

The relay project was a learning experience for me. I learned about the various IPC mechanisms that UNIX provides such as message queues, semaphores, shared memory, etc. I decided to utilize shared memory after reading the textbook and deciding that this method was best for one way, multicast communication. I initially planned on using semaphores for the dispatcher to keep track of all listeners, however, I decided that it added unneeded complexity and simply used the EOT character to terminate the listeners. Throughout the development of the project, I chose to prioritize efficiency and simplicity in order to produce a fast, accurate program. This is why I chose to operate character by character and to only use one byte of shared memory. I am very happy with the outcome. The only possible change for the future would be to use a faster mechanism for getting data from the shared memory.