Assignment 1

Ameet Deshpande (CS15B001)

August 14 2017

1 Forking and Killing

- Using the function *fork()* creates a child process which starts executing with *Program Counter* pointing to the same value
- *fork()* function returns the *process_id* of the child process to the parent process and 0 to the child process. This value thus gives us a key to execute some code in parent/child process only
- A print statement after forking is printed twice as it is executed by both the parent and child process.
- Killing the child process using the function *kill(pid,SIGTERM)* and having a print statement after that ensures that the print statement is executed only once (Code in file *kill_child.c*).
- Killing the parent process explicitly using *kill(getpid,SIGTERM)* kills the parent process, but the child process independently continues to execute
- Printing the process IDS using the *getpid()* functions shows that the process ID returned by the fork call and that returned by *getpid()* in the child process are the same (Code in file *obtain_pid.c*)
- Even after the execution of the parent process is complete, using the keys *Ctrl+C* only stops the parent process and child process keeps running. This can be seen in the file *ctrlc.c*. Equivalently, if there is an infinite loop in both the parent and child process, using the key combination *Ctrl+C* stops only the parent process.

2 Working with Malloc and Variables

- When variables are created before or after the fork call, the whole segment is just copied. Thus the program works with different images of the variables. Causing a change to the variables in one of the programs doesn't affect the variables in the other program.

- This holds true even for variables created through *malloc*. This clearly shows that both the child and parent processes are using different heap and stack space and there is no interference.
- One very interesting thing that happens is that the address of the pointers is the same in both the parent and child processes (Code in *checking_malloc.c*). This is however only the virtual address and the actual physical address of the variables and pointers are still different after the fork fucntion call has been made.

3 Working with File Pointers

- File pointers pose an interesting problem compared to normal variables. The position of defining the File Pointer dictates the behavior
- When the file pointer is declared before the fork call (as in the case of *reading_from_file_1.c*), when the file is read character by character and then printed to *stdout*, the whole text in the file is printed only once. The inference that can be drawn is that there is only one file pointer for both the processes and thus once *EOF* is reached, the printing stops. Thus there is only one set of printing happening.
- When the file pointer is declared after the fork call, the text inside the file is printed in order twice. This shows that, since the processes have created file pointers after the fork call, they are using different file pointers and thus the whole output is printed twice. It is worth noting that the output is in correct order and is not jumbled up in any way. This means that each file pointer is maintaining its own buffer and printing to *stdout* just before termination.
- When a call is made to *fflush(stdout)*, the content of the file is printed twice. But this time the output is jumbled. This happens because each time there is a context switch from the parent to child or vice-versa, they work with independent file pointers. But the *fflush()* call forces the function to print the output. Thus the output becomes jumbled (Code in *reading_from_file_2.c*)
- The main inference drawn is that having the file pointer before the fork call forces the processes to use the same **File Descriptor/Pointer** and buffer, however having it after the fork call forces different buffers and descriptors
- When writing to a file, if the file is opened in append mode, declaring the file pointer after the fork call prints the same text to the file twice. If however the write function is used, it is printed only once (because calling the write clears the file and starts from scratch). But in both cases different file pointers are being used, as in the read case. (Code in *writing_to_file_1.c*)

- Contrary to the previous case, when the File Pointer is declared before the fork statement and opened in write mode the same output is printed twice. In this case it doesnt matter if the file is opened in append mode or write mode, as the same file pointer is being used. (Code in *writing_to_file_2.c*)