Akshita Mittel
CS13B1040

Assignment 5: CS3030

Task 1:

1. The memory was shared between both the children processes and the parent process using shm and mmap, which creates a map of specific size and returns a pointer to that location. The size is allocated using the ftruncanate function. In both ftruncanate and mmap, the shared memory is recognized by the shm_fd, which was created using shm_open. The memory was created and opened in both read and write mode, since the child needs to both write to and read from the memory. The argument MAP_SHARED in mmap, specifies that the memory is a shared memory.

2. The size allocated to the shared memory was based on the num provided by the user, i.e. the number of random numbers that need to be generated. This number, along with the maximum value of any random number were placed in the shared memory, allowing it to be used by the child processes.

3. Two child process pid's were defined, and the process was forked twice. The child process were executed selected using (!(child= fork())). This will generate a valid child process if the pid we get after forking does not equal to 0.
   The parent process waited for both the child processes to be terminated, till it carried on with it's execution. Else, since all the processes are running simultaneously the parent may achieve completion before its children and exit. The two forked processes were carried out simultaneously as follows:

   - **setRand(int n):** This function takes in the maximum value that a random number can have from the shared memory. It keeps on generating random numbers and finding the modulus between 0 and max – 1, till it gets a non-zero number. It then returns this value.

   - **Memory management:**

0	1	2	3	4	...	3 + num	4 + num	...	3 + 2*num
max	num	sum	prod	sum_x1	...	sum_xn	prod_x1	…	prod_xn

   The size of memory allocated was = (4 + 2*num)*sizeof(int). Here the space after the 4th element was to store the random numbers generated. **Since both the child processes are running simultaneously, it is difficult to print the data in order**.

   - **Child1:** A seed was used to generate random numbers. After that the number of random numbers generated was obtained by mem[1], which holds num as shown above. Each number was stored in memory as shown in the table, and it was multiplied to the product, which was again stored in the shared memory.

   - **Child2:** It followed a similar process to child1, but the random number seed was different, to ensure that numbers aren't repeated.

- **Parent:** The parent waited for both the child processes to terminate after-which, it computed the difference between the prod and sum, obtained from each child process. The shared memory was accessed to print out all the data that was required for the output format.

Task 2:

1. **Checking for valid PID:** After inputting the pid from the user we check if the pid is valid. The method getpid, returns the Pid of the called process. If the process does not exist, it returns -1.

2. **Using the pstree command:** If the pid is valid, we want to send the command to the commandline and get it executed by the system. The command to get the ancestors of any branch of the pstree is pstree -s <pid>
   The option 'p' specifies that we wish to get the pid's of all the ancestral processes. Therefor the command that we wish to execute is : pstree -s -p <pid>.

3. **Sending the command to the processor:** We can send the command to the processor by using the syscall “system”, This syscall passes the command specified by the char* argument to the host environment where it is executed by the command processor. After execution it returns to the program. The command is assembled using sprintf.