OS

Assignment on processes: #1

Copy the *simplechain.c* program into your directory, check it for errors, modify if necessary and compile it in Linux. Proceed then to the questions and experiment section following the program text.

simplechain.c

```
/*
*      Simple C program for generating a chain of processes. Invoke this
*      program with a command-line arguments including name of the executable
*      file and the number of processes in the chain.
*      After the chain is created, each process identifies itself with
*      its process ID, the process ID of its parent, and
*      the process ID of its child. Each process then exits */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){    /* check for valid number of command-line arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid = fork())
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

1. Where is the critical section in this program? Justify your answer.
2. Run this program on in Linux and observe the results for different number of processes. Make necessary changes, if needed.
3. Draw a diagram of the generated process chain with boxes representing processes and connecting lines representing the parent-child dependencies for a run with command-line argument value of 5 and enter the actual PIDs into the boxes.
4. Put a loop around the final *fprintf*. Have the loop execute *k* times. Put a sleep(m) inside this loop after the *fprintf*. Pass *k* and *m* on the command line. Run the program for several values of *n*, *k*, and *m* and observe the results.
5. Modify the original program by putting a *wait(NULL)* system call before the final *fprintf* statement. How does this affect the output of the program ?
6. Modify the original program by replacing the final *fprintf* statement with four *fprintf* statements, one each for the four integers displayed. Only the last one should output a newline. What happens when you run this program? Can you tell which process generated each part of the output? Run this program several times and see if there is a difference in the output.

Write a report answering all of the questions and describe your observations for each OS using a word processor. Submit it together with listings of all the modified versions of the original program.

Assignment on processes: #2

1. In the original program of assignment #1 replace the test
   `if (childpid = fork())`
   with test
   `if (childpid = fork()) <= 0`
   Run the program with n=5, describe what happens and explain it.
   Draw a diagram illustrating the relationships between processes

2. Modify the program of p.1 in such a way that you can see in another window the list of processes running by using the command *ps*.

3. In the modified program of p.1 insert a statement
   `while(wait(NULL) > 0);`
   after *break* and before *fprintf*.
   Describe what happens and explain it.

4. What happens if in the program of p.3 you interchange the *while* loop and fprintf statements? Explain it.

5. What happens if in the program of p.3 you replace the *while* loop with the statement
   `wait(NULL);`? Explain it.

6. In the modified program of p.1 insert the following statements;

```
for( ; ;) {
childpid = wait(NULL);
if ((childpid == -1)&&(errno != EINTR))
break;}
```

   after the first break and before *fprintf*.
   Run the program several times for n=10.
   Describe what happens and why. How output messages are ordered?

7. In the modified program of p.1 replace the test
   `if (childpid = fork()) <= 0`
   with
   `if (childpid = fork()) == -1`

   Run the program with n=5 and describe what happens. Explain it.
   Draw a diagram illustrating the relationships between processes.
   Write a report containing all of your observations and diagrams using a word processor. Submit it together with listings and outputs

Assignment on processes: #3

Write a programme to create Zombie processes. Show which part of your programme is responsible for Zombie processes. Can you create orphan processes? Add a separate module for Orphan process or write another programme. Display different types of processes running in your Linux system.

OPERATING SYSTEM (III B Tech CSE)-Assignment Series-2

1.Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report:

The average value is 82
The average value is 72
The average value is 95

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. (Expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

2. Write a multithreaded program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

3.The Fibonacci sequence is the series of numbers {0, 1, 1, 2, 3, 5, 8, ...}

Formally, it is expressed as:

$fib_0 = 0$
$fib_1 = 1$
$fib_n = fib_{n-1} + fib_{n-2}$

Write a multithreaded program that generates the Fibonacci sequence. This program works as follows. On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the *Fibonacci sequence* until the child thread finishes, the parent thread will have to wait for the child thread to finish.

4. Write a producer-consumer problem that uses threads and shares a common buffer. Just let each thread access the shared buffer when it wants to. Use sleep and wakeup to handle the full and empty conditions. See how long it takes for a fatal race condition to occur. For example, you might have the producer print a number once in a while. Do not print more than one number every minute because the I/O could affect the race conditions.

5. A process can be put into a round-robin queue more than once to give it a higher priority. Running multiple instances of a program each working on a different part of a data pool can have the same effect. First write a program that tests a list of numbers for primality. Then devise a method to allow multiple instances of the program to run at once in such a way that no two instances of the program will work on the same number. Can you in fact get through the list faster by running multiple copies of the program? Note that your results will depend upon what else your computer is doing; on a personal computer running only instances of this program you would not expect an improvement, but on a system with other processes, you should be able to grab a bigger share of the CPU this way.

6. The objective of this assignment is to implement a multithreaded solution to find if a given number is a perfect number. N is a perfect number if the sum of all its factors, excluding itself, is N; examples are 6 and 28. The input is an integer, N. The output is true if the number is a perfect number and false otherwise. The main program will read the numbers N and P from the command line. The main process will spawn a set of P threads. The numbers from 1 to N will be partitioned among these threads so that two threads do not work on the name number. For each number in this set, the thread will determine if the number is a factor of N. If it is, it adds the number to a shared buffer that stores factors of N. The parent process waits till all the threads complete. Use the appropriate synchronization primitive here. The parent will then determine if the input number is perfect, that is, if N is a sum of all its factors and then report accordingly.

7. Implement a program to count the frequency of words in a text file. The text file is partitioned into N segments. Each segment is processed by a separate thread that outputs the intermediate frequency count for its segment. The main process waits until all the threads complete; then it computes the consolidated word-frequency data based on the individual threads output.

Chapter 6
- Peterson
- CS
- Synch

P-C Semaphore
R-W mutex
DP