OS Assignment 5: Processes in Linux

Written by Akilesh B, CS13B1042

September 6, 2015

Task 1:

The main task here is to create a concurrent program in which a parent creates two child processes.

To do this, I first fork a child (child1) and check if its pid is zero, perform the multiplication operation here. If pid of child1 is not zero, then fork a new child (child2). If pid of child2 is zero, perform the addition operation. The parent waits for both its children to finish, then computes the difference of product and sum, prints it.

```
pid_t child_a, child_b;
child_a = fork();
if (child_a == 0)
{
/* first child => performs multiplication.
Write code for multiplication here.
*/
}
else {
child_b = fork();
if (child_b == 0) {
/* second child => performs addition.
Write code for addition here.
*/
 } else {
/* Parent Code
This computes difference between product and sum. Prints it
*/
 }
}
```

Child 1 and Child 2 execute concurrently. In order to share memory between processes, I'm using a global variable and then use mmap which is a method of memory-mapped file I/O.

This can be done by:

static unsigned long *glob_var;

glob_var = mmap(NULL, sizeof *glob_var, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

So output of child 1 and child 2 are stored in *global_var1* and *global_var2* respectively. The parent process prints the difference *global_var1 - global_var2*.

In child1 and child2, I call function *generate_random*. *generate_random* generates random numbers , stores it in an array and returns it. These values can be accessed in the child through the pointer returned by the generate_random function.

The random numbers are generated as *rand()%MAX + 1*, since each element is between 1 and MAX. rand() is initialized by seed, srand((unsigned)time(&t)). I also check if the random number generated is already present in the array, if so, I generate another. *already_exists* function serves this purpose.

The parent process waits for both the children to terminate, this is accomplished using WAIT(NULL).

The shared memory can be deallocated using : munmap(glob_var, sizeof *glob_var);

Since child1 and child2 are executing concurrently, the order of output of these processes can be different.

Task 2:

The task is to take a PID as input from the user and display list of all of its ancestors. If the pid is not valid, it should output pid is invalid.

After taking input PID, check if PID is valid or not. All process IDs are stored in */proc/<pid-no>*. So, I open the directory and check if it exists. If it exists, I then check if it is accessible.

```
sprintf(p, "/proc/%d/", pid); //checking if the path of the pid folder exits or not
DIR* dir = opendir(p); // opening a directory
bool flag;
if(dir) { // PID exists
flag=true;
closedir(dir);
} else if(ENOENT == errno) {
flag=false;
printf("Invalid PID\n"); //PID doesn't exist.
 } else {
flag=false;
printf("Permission Denied\n"); //PID exists but we cannot access it.
 }
```

If flag is true, then given PID exists and is accessible.

pstree -p -s -A <pid> command gives the process tree of the process with given pid. For example, pstree -p -s -A 428 gives the output *init(1)—rsyslogd(426)—{rsyslogd}(428)* .

-p => It shows PIDs
-s => It shows parent process of specified process.
-A => It uses ASCII characters to draw the tree.

I store the output of the pstree command in a file d.txt

```
strcpy(command, "pstree -p -s -A "); //pstree command
```

```
strcat(command, str); // executing command
strcat(command , " 1>d.txt 2>&1");
FILE *fp = fopen("d.txt", "ab+");
int out = system(command); // system command executes pstree command
```

The pstree command prints the entire process tree of the process with given PID, but in my program I want to print only the ancestors. So, after storing the output of pstree command in a text file, I traverse the file as long as I encounter the process with given PID and stop it there. I print this.

Output:

The output is the set of all ancestors of the process with given PID.