Assignment 1: Operations on Processes

Process: A process is a program in execution. For execution of the program, it is loaded in computer‟s memory and it becomes a Process.A program is a passive entity and process is an active entity. A process performs all the tasks mentioned in the program. A process can be divided into foursections ─ stack, heap, text and data.

Stack: The process Stack contains the temporary data such as method/function parameters, return address and local variables.

Heap: This is dynamically allocated memory to a process during its run time.

Text: This includes all instructions specified in the program.

Data: This section contains the global and static variables.

In Linux operating system, new processes are created through fork() system call.

Fork() System Call:

To create a new process fork() system call is used. It takes no arguments and returns a process ID. The process that creates new process is called as **Parent Process** while newly created process is called as **Child Process**. Child process is exact copy of parent process. Syntax of this system call is

int fork()

This system call is available in library file <**unistd.h**>. This system call return following values.

1) returns the value 0 to the newly created child process

2) returns Process ID of child process to parent process.

3) returns -1 if fork fail to create child process or on error.

Both parent and child processes continue executing with the instruction that follows the call to fork. Fork system call is often followed by exec call.

The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an

integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to thisprocess.

If the call to fork() is executed successfully, Linux will

- Make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call.

Program 1 : Consider the following example:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        // make two process which run same
        // program after this instruction
        fork();
        printf("Hello \n");
        return 0;
}
```

Output of above program:

Hello

Hello

Here printf() statement after fork() system call executed by both parent and childprocess. Both processes start their execution right after the system call fork(). Since bothprocesses have identical but separate address spaces, those variables initialized before the fork()call have the same values in both address spaces. Since every process has its own address space,any modifications will be independent of the others. In other words, if the parent changes the

value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though theyhave identical variable names.

Program 2: Run the below given program and see the output(Use of getpid() function)

```
#include <stdio.h>
#include <unistd.h>
#include<stdlib.h>
int main()
{
intpid;
pid=fork();
if(pid<0)
```

```
    {
printf("\n Error in creation of Child Process ");
exit(1);
    }
else if(pid==0)
    {
printf("\n Hello, I am the child process ");
printf("\n My pid is %d ",getpid());
exit(0);
    }
else
    {
printf("\n Hello, I am the parent process ");
printf("\n My pid is %d \n ",getpid());
exit(1);
    }
}
```

exec() system call

The exec family of system calls replaces the program executed by a process. When a process calls exec, all code (text) and data in the process is replaced with the executable of the new program. It loads the program into the current process space and runs it from the entry point. Theprocess id PID is not changedbecause we are not creating a new process, we are just replacing a process with anotherprocess in exec.

The exec() family consists of following functions,

execl(): l is for the command line arguments passed a list to the function.

Syntax: intexecl(const char *path, const char *arg, ...);

execlp(): p is the path environment variable which helps to find the file passed as an argument

to be loaded into process.

Syntax: intexeclp(const char *file, const char *arg, ...);

execle(): It is an array of pointers that points to environment variables and is passed explicitly tothe newly loaded process.

Syntax: intexecle(const char *path, const char *arg, ..., char * constenvp[]);

execv(): v is for the command line arguments. These are passed as an array of pointers to the function.

Syntax: intexecv(const char *path, char *constargv[]);

execlp() System Call:

execlp() system call is used after a fork() call by one of the two processes to replace the processes memory space with a new program. This call loads a binary file into memory and starts its execution. So two processes can be easily communicates with each other.

Program 3:

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
intpid;
pid = fork(); /* fork a child process */
if (pid< 0) /* error occurred */
{
        fprintf(stderr, "Fork Failed");
        return 1;
}
else if (pid == 0) /* child process */
        execlp("/bin/wc","wc",NULL);
else /* parent process */
{
        wait(NULL); /* parent will wait for the child to complete    */
        printf("Child Complete");
}
return 0;
}
```

fork() vs exec()

- Fork() starts a new process which is a copy of the one that calls it, while exec() replacesthe current process image with another (different) one.
- Both parent and child processes are executed simultaneously in case of fork() while

Control never returns to the original program unless there is an exec() error.

Sleep() System Call:

This system call takes a time value as a parameter, specifying the minimum amount of time that the process is to sleep before resuming execution. The parameter typically specifies seconds, although some operating systems provide finer resolution, such as milliseconds or microseconds.

Wait() system call

The wait() system call suspends execution of the calling process until one of its children terminates.

wait(int&status);

waitpid() system call :

By using this system call it is possible for parent process to synchronize its execution with child

process. Kernel will block the execution of a Process that calls waitpid system call if a some child of that process is in execution. It returns immediately when child has terminated by return

termination status of a child.

Syntax :intwaitpid(intpid, int *status, int options);

nice() System Call:

Using nice() system call, we can change the priority of the process in multi-taskingsystem. The new priority number is added to the already existing value.

int nice(intinc);

nice() adds inc to the nice value. A higher nice value means a lower priority. The range of the nice value is +19 (low priority) to -20 (high priority).

Program 4:

```
#include<stdio.h>
main()
```

```
{
intpid, retnice;
printf("press DEL to stop process \n");
pid=fork();
for(;;)
{
if(pid == 0)
{
retnice = nice (−5);
print("child gets higher CPU priority %d \n", retnice);
sleep(1);
}
else
{
retnice=nice(4);
print("Parent gets lower CPU priority %d \n", retnice);
sleep(1);
}
}
}
```

Orphan process

Orphan processes are those processes that are still running even though their parent process has terminated or finished. A process can be orphaned intentionally or unintentionally. Usually, a parent process waits for its child to terminate or finish their joband report to it after execution but if parent fails to do so its child results in the Orphan process.

In most cases, the Orphan process is immediately adopted by the init process (a very first processof the system).

Program 5:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
```

```
{
intpid = fork();
if (pid> 0) {
        //getpid() returns process id and getppid() will return parent process id
printf("Parent process\n");
printf("ID : %d\n\n", getpid());
   }
else if (pid == 0) {
printf("Child process\n");
        // getpid() will return process id of child process
printf("ID: %d\n", getpid());
        // getppid() will return parent process id of child process
printf("Parent -ID: %d\n\n", getppid());
sleep(10);
        // As this time parent process has finished, it will show different parent process
id
printf("\nChild process \n");
printf("ID: %d\n", getpid());
printf("Parent -ID: %d\n", getppid());
   }
else {
printf("Failed to create child process");
   }

return 0;
}
```

Clock() function: The clock function is used to determine the processor time in executing a program or part of a program. The header file <time.h> should be included in the program for its application. The function prototype is given below.

clock_t clock(void);

The function returns a value of *type* c1ock_t which should be divided by the value of the macroCLOCK_PER_SEC to obtain the processor time in seconds.

Shell:

Shell is an interface between user and operating system. It is the command interpreter, which accept the command name (program name) from user and executes that command/program. Shell mostly accepts the commands given by user from keyboard. Shell gets started automatically when Operating system is successfully started.When shell is started successfully it generally display some prompt (such as #,$ etc) to accept and execute the command.Shell executes the commands either synchronously or asynchronously.

When shell accepts the command then it locates the program file for that command, start its execution, wait for the program associated with the command to complete its execution and thendisplay the prompt again to accept further command. This is called as Synchronous execution ofshell.

In asynchronous execution shell accept the command from user, start the execution of program

associated to the given command but does not wait for that program to finish its execution, display prompt to accept next command.

How Shell Execute the command?

1. Accept the command from user.
2. Tokenize the different parts of command.
3. First word on the given command line is always a command name.
4. Creates (Forks) a child process for executing the program associated with given command.
5. Once child process is created successfully then it loads (exec) the binary (executable) imageof given program in child process area.
6. Once the child process is loaded with given program it will start its execution while shell iswaiting (wait) for it (child) to complete the execution. Shell will wait until child finish itsexecution.
7. Once child finish the execution then Shell wakeup, display the command prompt again andaccept the command and continue.

Example

$ cat file1.dat file2.dat file3.dat

This command line has four tokens- cat, file1.dat, file2.dat and file3.dat. First token is the command name which is to be executed. In Linux operating system, some commands are internally coded and implemented by shell such as mkdir, rmdir, cd, pwd, ls, cat, grep,

who etc.

Objective of this practical assignment is to simulate the shell which interprets all internal or predefined Linux commands and additionally implement to interpret following extended commands.

(1) Count: To count and display the number of lines, words and characters in a given file.

(2) List: It will list the files in current directory with some details of files

Program Logic

1) Main function will execute and display the command prompt as $

2) Accept the command at $ prompt from user.

3) Separate or tokenize the different parts of command line.

4) Check that first part is one of the extended commands or not (count, typeline, list, search).

5) If the command is extended command, then call corresponding functions which is implementing that command

6) Otherwise fork a new process and then load (exec) a program in that newly created process

and execute it. Make the shell to wait until command finish its execution.

7) Display the prompt and continue until given command is “q” to Quit.

Practical Assignments:

Set A

1. Create a child process using fork(), display parent and child process id. Child process will display the message “Hello World” and the parent process should display “Hi”.

2. Create a child process using the command exec(). Note down process ids of the parent and the child processes, check whether the control is given back to the parent after the child process terminates. Write a similar program using execv() and execvp() and observe the differences in behaviours of the commands.

3. Creating a child process without terminating the parent process Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

Set B

1. Write a program to illustrate the concept of orphan process(Using fork() and sleep()) (Refer Program 5).

2. Write a program that demonstrates the use of nice() system call. After a child process is started using fork(), assign higher priority to the child using nice() system call.

3. Write a program to find the execution time taken for execution of a given set of instructions (Hint: use clock() function. This function clock() is called at the beginning of program and again at the end of the program and the difference between the values returned gives the time spent by processor on the program.)

Set C(Shell Command)

1. Implement the shell program that accepts the command at $ prompt displayed by your shell (myshell$). Implement the „count“ command by creating child process which works as follows:
   myshell$ count c filename: To display the number of characters in given file
   myshell$ count w filename: To display the number of words in given file
   myshell$ count l filename: To display the number of lines in given file

2. Extend the shell to implement the commands „list“ which works as follows:
   myshell$ list f dirname: It will display filenames in a given directory.
   myshell$ list n dirname: It will count the number of entries in a given directory.
   myshell$ list i dirname: It will display filenames and their inode number for the files in a given directory.

Assignment Evaluation

0: Not Done [] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of the Instructor **Date of Completion____/____/_____**