

NETWORK PROGRAMMING LAB

EXPERIMENT NO 1

Day 1: **23/03/21**

Aim:

- 1) To familiarize process related system calls fork (), wait (), exec (), exit ().
- 2) Explore Zombie process and Orphan process

FORK()

Fork system call is used for creating a new process, which is called **child process**, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

WAIT()

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent **continues** its execution after wait system call instruction. Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main

- It receives a signal (from the OS or another process) whose default action is to terminate.

Syntax in C language:

```
#include
#include
// take one argument status and returns
// a process ID of dead children.
pid_t wait(int *stat_loc);
```

If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates. If only one child process is terminated, then return a wait() returns process ID of the terminated child process. If more than one child processes are terminated than wait() reap any **arbitrarily child** and return a process ID of that child process. When wait() returns they also define **exit status** (which tells our, a process why terminated) via pointer, If status are not **NULL**. If any process has no child process then wait() returns immediately “-1”.

EXEC()

The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.

More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program.

The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec(). The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is

because we are not creating a new process we are just replacing a process with another process in `exec`.

PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.

`Exec` system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

1. `execl`
2. `execle`
3. `execlp`
4. `execv`
5. `execve`
6. `execvp`

It should be noted here that these functions have the same base `exec` followed by one or more letters. These are explained below:

e: It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.

l: l is for the command line arguments passed a list to the function

p: p is the path environment variable which helps to find the file passed as an argument to be loaded into process.

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

Syntaxes of `exec` family functions:

The following are the syntaxes for each function of `exec`:

```
int execl(const char* path, const char* arg, ...)
```

```
int execlp(const char* file, const char* arg, ...)
```

```
int execle(const char* path, const char* arg, ..., char* const envp[])
```

```
int execv(const char* path, const char* argv[])
```

```
int execvp(const char* file, const char* argv[])
int execvpe(const char* file, const char* argv[], char *const envp[])
```

Description:

The return type of these functions is `int`. When the process image is successfully replaced nothing is returned to calling function because the process that called it is no longer running. But if there is any error `-1` will be returned. If any error is occurred an *errno* is set.

In the syntax:

path is used to specify the full path name of the file which is to be executes.

arg is the argument passed. It is actually the name of the file which will be executed in the process. Most of the times the value of *arg* and *path* is same.

const char* arg in functions `execl()`, `execvp()` and `execle()` is considered as *arg0*, *arg1*, *arg2*, ..., *argn*. It is basically a list of pointers to null terminated strings. Here the first argument points to the filename which will be executed as described in point 2.

envp is an array which contains pointers that point to the environment variables.

file is used to specify the path name which will identify the path of new process image file.

The functions of `exec` call that end with ***e*** are used to change the environment for the new process image. These functions pass list of environment setting by using the argument ***envp***. This argument is an array of characters which points to null terminated String and defines environment variable.

To use the `exec` family functions, you need to include the following header file in your C program:

```
#include <unistd.h>
```

EXIT()

The function **_exit()** terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, *init*, and the process's parent is sent a **SIGCHLD** signal.

The value *status* is returned to the parent process as the process's exit status, and can be collected using one of the **wait()** family of calls

SYNTAX

void exit (int status);

exit() terminates the process normally.
status: Status value returned to the parent process. Generally, a status value of 0 or **EXIT_SUCCESS** indicates success, and any other value or the constant **EXIT_FAILURE** is used to indicate an error. **exit()** performs following operations.

- * Flushes unwritten buffered data.
- * Closes all open files.
- * Removes temporary files.
- * Returns an integer exit status to the operating system.

Programs to familiarize process related system calls fork (), wait (), exec (), exit ()

1)

```
#include<stdio.h>
```

```
#include<sys/types.h>
```

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
fork();
```

```
printf("Hello World!\n");
```

```
return 0;
```

```
}
```

```
[austinphilippaul@localhost r21]$ gedit helloworld.c
[austinphilippaul@localhost r21]$ gcc helloworld.c
[austinphilippaul@localhost r21]$ ./a.out
Hello World!
Hello World!
```

2)

```
#include<stdio.h>
```

```
#include<sys/types.h>
```

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
fork();
```

```
fork();
```

```
printf("Hello");
```

```
return 0;
```

```
}
```

```
[austinphilippaul@localhost r21]$ gcc helloworld3.c
[austinphilippaul@localhost r21]$ ./a.out
HelloHelloHelloHello[austinphilippaul@localhost r21]$ S
```

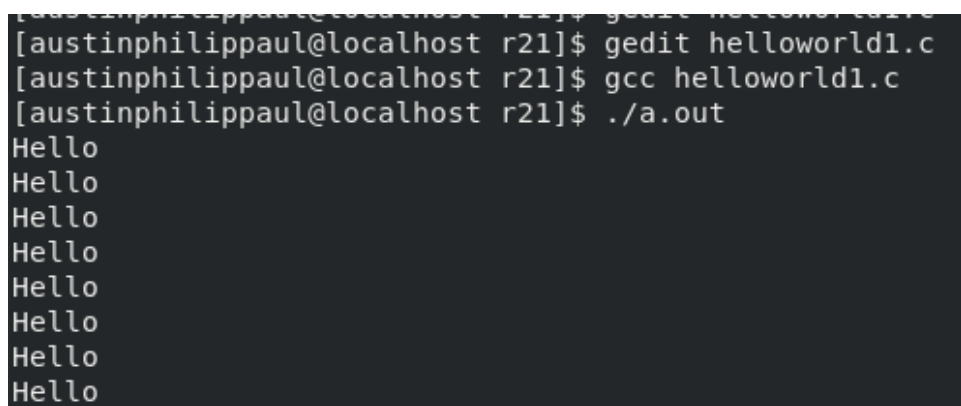
3)

```
#include<stdio.h>
```

```
#include<sys/types.h>
```

```
#include<unistd.h>
```

```
int main()
{
fork();
fork();
fork();
printf("Hello\n");
return 0;
}
```

A terminal window with a dark background. The prompt is [austinphilippaul@localhost r21]. The user enters 'gedit helloworld1.c', then 'gcc helloworld1.c', and finally './a.out'. The output of the program is eight lines of 'Hello' printed one after another.

```
[austinphilippaul@localhost r21]$ gedit helloworld1.c
[austinphilippaul@localhost r21]$ gcc helloworld1.c
[austinphilippaul@localhost r21]$ ./a.out
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

4)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
```

```
void forkexample()
{
int x=1;
if(fork() == 0)
```

```

printf("Child has x = %d\n",++x);
else
printf("Parent has x = %d\n",--x);
}
int main()
{
forkexample();

return 0;
}

```

```

[austinphilippaul@localhost r21]$ gcc helloworld2.c
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2
[austinphilippaul@localhost r21]$ gcc helloworld2.c
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2
[austinphilippaul@localhost r21]$ ./a.out
Parent has x = 0
Child has x = 2

```

5)


```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

void forkexample()
{

if(fork() == 0)
printf("Hello from Child!\n");
else
printf("Hello from Parent!\n");
}

int main()
{
forkexample();

return 0;
}
```

```
[austinphilippaul@localhost r21]$ gcc helloworld4.c
[austinphilippaul@localhost r21]$ ./a.out
Hello from Parent!
Hello from Child!
[austinphilippaul@localhost r21]$ ss
```

6)

```
#include<stdio.h>
#include<unistd.h>
```

```
int main()
{
if(fork() && fork())
printf("forked\n");
}
```

```
[austinphilippaul@localhost r21]$ gcc helloworld5.c
[austinphilippaul@localhost r21]$ ./a.out
forked
[austinphilippaul@localhost r21]$
```

7)

```
#include<stdio.h>
#include<unistd.h>
```

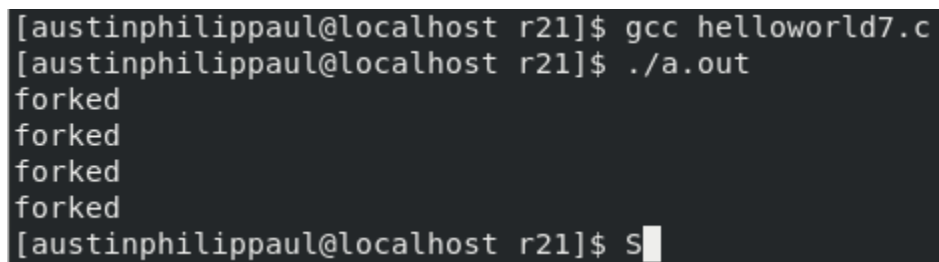
```
int main()
{
if(fork() || fork())
printf("forked\n");
}
```

```
[austinphilippaul@localhost r21]$ gcc helloworld6.c
[austinphilippaul@localhost r21]$ ./a.out
forked
forked
[austinphilippaul@localhost r21]$ S
```

8)

```
#include<stdio.h>
#include<unistd.h>
```

```
int main()
{
if(fork() || fork())
{
fork();
printf("forked\n");
}
}
```



```
[austinphilippaul@localhost r21]$ gcc helloworld7.c
[austinphilippaul@localhost r21]$ ./a.out
forked
forked
forked
forked
[austinphilippaul@localhost r21]$ S
```

9)

```
#include<stdio.h>
#include<unistd.h>
```

```
int main()
{
if(fork() && fork())
{
fork();
printf("forked\n");
}
```

```
}
```

```
}
```

```
[austinphilippaul@localhost r21]$ touch helloworld8.c  
[austinphilippaul@localhost r21]$ gcc helloworld8.c  
[austinphilippaul@localhost r21]$ ./a.out  
forked  
forked
```

10)

```
#include<stdio.h>
```

```
#include<unistd.h>
```

```
#include<stdlib.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
printf("PID of ex1.c=%d\n",getpid());
```

```
char *args[]={"Hello","Neso","Academy",NULL};
```

```
execv("./ex2",args);
```

```
printf("Back to ex1.c");
```

```
return 0;
```

```
}
```

```
#include<stdio.h>
```

```
#include<unistd.h>
```

```
#include<stdlib.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```

printf("We are in ex2.c\n");
printf("PID of ex2.c=%d\n",getpid());
return 0;
}

```

```

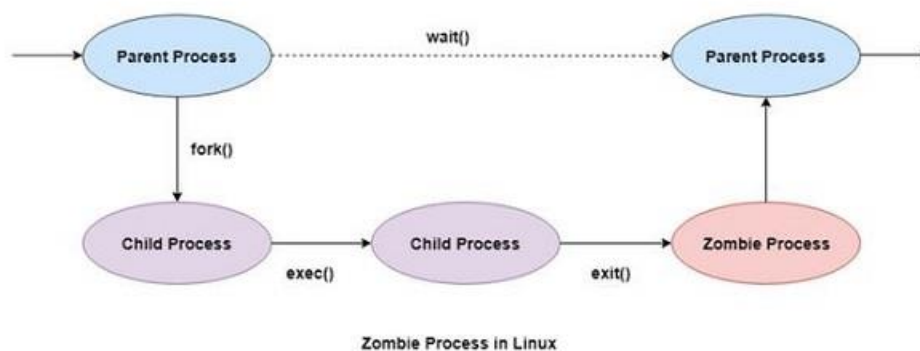
[austinphilippaul@localhost r21]$ gcc helloworld9.c -o ex1
[austinphilippaul@localhost r21]$ gcc helloworld10.c -o ex2
[austinphilippaul@localhost r21]$ ./ex1
PID of ex1.c=3646
We are in ex2.c
PID of ex2.c=3646
[austinphilippaul@localhost r21]$ ./ex2
We are in ex2.c
PID of ex2.c=3653

```

ZOMBIE PROCESS:

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

In the following code, the child finishes its execution using `exit()` system call while the parent sleeps for 50 seconds, hence doesn't call `wait()` and the child process's entry still exists in the process table.



// A C program to demonstrate Zombie Process.

```
// Child becomes Zombie as parent is sleeping
// when child process exits.
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```

ORPHAN PROCESS:

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

In the following code, parent finishes execution and exits while the child process is still executing and is called an orphan process now.

However, the orphan process is soon adopted by init process, once its parent process dies.

An intentionally orphaned process runs in the background without any manual support. This is usually done to start an indefinitely running service or to complete a long-running job without user attention.

An unintentionally orphaned process is created when its parent process crashes or terminates. Unintentional orphan processes can be avoided using the process group mechanism.

```
// A C program to demonstrate Orphan Process.
```

```
// Parent process finishes execution while the
```

```
// child process is running. The child process
```

```
// becomes orphan.
```

```
#include<stdio.h>
```

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
int main()
```

```
{
```

```
// Create a child process
```

```
int pid = fork();
```

```
if (pid > 0)
```

```
printf("in parent process");
```

```

// Note that pid is 0 in child process
// and negative if fork() fails
else if (pid == 0)
{
sleep(30);
printf("in child process");
}

return 0;
}

```

Write a Unix C Program using fork() system call that generates the factorial and gives a sequence of series like 1, 2, 6, 24, 120... in the child process. The number of sequences is given in command line.

```

#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>

int main(int argc,char *argv[])
{
int fact=1,i=2,n;
n=*argv[1];
n-=48;

```



```

while((i-2)<n)
{

if(fork()==0)
{    printf("%d, ",fact);
exit(0);

}
else
    {    wait(NULL);
fact*=i++;
}
}

printf("\b\b");
return 0;
}

```

```

[austinphilippaul@localhost r21]$ gcc helloworld11.c
[austinphilippaul@localhost r21]$ ./a.out 5
1, 2, 6, 24, 120[austinphilippaul@localhost r21]$ gcc

```

Without Wait()

```

[austinphilippaul@localhost r21]$ gcc helloworld11.c
[austinphilippaul@localhost r21]$ ./a.out 5
120, 1, 2, 6, 24, [austinphilippaul@localhost r21]$ S

```

Program to create four processes (1 parent and 3 children) where they terminates in a sequence as follows :

(a) Parent process terminates at last

(b) First child terminates before parent and after second child.

(c) Second child terminates after last and before first child.

(d) Third child terminates first.

```
#include<stdio.h>
```

```
#include<unistd.h>
```

```
#include<stdlib.h>
```

```
int main()
```

```
{
```

```
int pid, pid1, pid2;
```

```
    pid = fork();
```

```
    if (pid == 0)
```

```
    {
```

```
        sleep(5);
```

```
        printf("child[1] --> pid = %d and ppid = %d\n",getpid(),getppid());
```

```
    }
```

```
else {
```

```
pid1 = fork();
```

```
if (pid1 == 0) {
```

```
sleep(2);
```

```
printf("child[2] --> pid = %d and ppid = %d\n",getpid(),getppid());
```

```

}
else {
pid2 = fork();
if (pid2 == 0) {

printf("child[3] --> pid = %d and ppid = %d\n",getpid(), getppid());
}

else {

sleep(8);//Cannot use wait as parent will continue when any one child
terminates
printf("parent --> pid = %d\n", getpid());
}
}
}

return 0;
}

```

```

[austinphilippaul@localhost r21]$ gcc helloworld12.c
[austinphilippaul@localhost r21]$ ./a.out
child[3] --> pid = 29128 and ppid = 29125
child[2] --> pid = 29127 and ppid = 29125
child[1] --> pid = 29126 and ppid = 29125
parent --> pid = 29125

```

