# Laboratory: Fork, Exec and Process control

Download codes:

## fork():

fork - create a child process

The fork() system call will spawn a new child process which is an identical process to the parent except that has a new system process ID. The process is copied in memory from the parent and a new process structure is assigned by the kernel. The return value of the function is which discriminates the two threads of execution. A zero is returned by the fork function in the child's process.

The environment, resource limits, umask, controlling terminal, current working directory, root directory, signal masks and other process resources are also duplicated from the parent in the forked child process.

**Example 1**

**test_fork.c**

**Compile**: gcc -o test_fork.c test_fork
**Run**: ./test_fork

Parent Process:
Global variable: 2
Stack variable: 20
Child Process:
Global variable: 3
Stack variable: 21

**Note on exit()** vs **_exit():** The C library function exit() calls the kernel system call _exit() internally. The kernel system call _exit() will cause the kernel to close descriptors, free memory, and perform the kernel terminating process clean-up. The C library function exit() call will flush I/O buffers and perform aditional clean-up before calling _exit() internally. The function exit(status) causes the executable to return "status" as the return code for main(). When exit(status) is called by a child process, it allows the parent process to examine the terminating status of the child (if it terminates first). Without this call (or a call from main() to return()) and specifying the status argument, the process will not return a value.

**[Potential Pitfall]:** Some memory duplicated by a forked process such as file pointers, will cause intermixed output from both processes. Use the wait() function so that the processes do not access the file at the same time or open unique file descriptors. Some like stdout or stderr will be shared unless synchronized using wait() or some other mechanism. The file close on exit is another gotcha. A terminating process will close files before exiting. File locks set by the parent process are not inherited by the child process.
**[Potential Pitfall]:** Race conditions can be created due to the unpredictability of when the kernel scheduler runs portions or time slices of the process. One can use wait(). the use of sleep() does not guarentee reliability of execution on a heavily loaded system as the scheduler behavior is not predictable by the application.

## Zombies process

This program creates a child process that terminates its execution immediately. Meanwhile, the parent process goes into sleep mode for 60 seconds. Because the child process ends before

the parent process, it becomes a zombie process.

**Example 2**

**test_zombies.c**

**Compile**: gcc -o test_zombies.c test_zombies

It is necessary open two other terminals separately and type the commands:

- watch -n 1 'ps -o pid,uname,comm -C test_zombies'
- **Run:** ./test_zombies
- ps xl  Obs. After the code is executed

The first command will monitor the zombie_process process and its children and refresh every 1 second. The second command will display a complete list of all processes plus the number of existing zombie processes. First, run the fisrt command, and then on a second terminal, run this program. And on a third terminal execute ps xl. Notice that there will immediately be 1 zombie process.

In the first terminal will appear two zombie_process commands, one being the father and the other the son that will have his front the tag <defunct>, indicating that he is a zombie. After the 60 seconds pass, both processes disappear and there will be no more zombie process.

## vfork():

vfork - create a child process and block parent
_exit - - terminate the current process

The *vfork()* function is the same as *fork()* except that it does not make a copy of the address space. The memory is shared reducing the overhead of spawning a new process with a unique copy of all the memory. This is typically used when using fork() to exec() a process and terminate. The *vfork()* function also executes the child process first and resumes the parent process when the child terminates.

**Example 3**

**test_vfork.c**

**Compile:** gcc -o test_vfork.c test_vfork
**Run:** ./test_vfork

Parent Process:
Global variable: 2
Stack variable: 20
Child Process:
Global variable: 3
Stack variable: 21

**Note:** The child process executed first, updated the variables which are shared between the processes and **NOT** unique, and then the parent process executes using variables which the child has updated.

**[Potential Pitfall]:** A deadlock condition may occur if the child process does not terminate, the parent process will not proceed.

## wait():

The system call *wait*() is easy. This function blocks the calling process until one of its child processes exits or a signal is received.

**wait():** Blocks calling process until the child process terminates. If child process has already teminated, the *wait()* call returns immediately. if the calling process has multiple child processes, the function returns when one returns. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer.

**waitpid():** Options available to block calling process for a particular child process not the first one.

The execution of wait() could have two possible situations.

If there are at least one child processes running when the call to *wait()* is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
If there is no child process running when the call to *wait()* is made, then this *wait()* has no effect at all. That is, it is as if no wait() is there.

**Example 4**

This program creates a child process through the fork () call and waits for its execution to end through the wait () call. There is a global variable and a local variable and their respective values are changed within the child process to verify that this change has no effect on the parent process.

**test_fork_wait.c**

**Compile:** gcc -o test_fork_wait.c test_wait
**Run**: ./test_fork_wait

**Example 5**

The parent process will often want to wait until all child processes have been completed. This can be implemented with the wait() function call.

**test_wait.c**

**Compile:** gcc -o test_wait.c test_wait
**Run**: ./test_wait

**Example 6**

The program test_wait_fork.c shows some typical process programming techniques. The main program creates two child processes to execute the same printing loop and display a message before exit. For the parent process (i.e., the main program), after creating two child processes, it enters the wait state by executing the system call *wait()*. Once a child exits, the parent starts execution and the ID of the terminated child process is returned in pid so that it can be printed. There are two child processes and thus two wait()s, one for each child process. In this example, we do not use the returned information in variable status.

**test_wait_fork.c**

**Compile**: gcc -o test_wait_fork.c test_wait_fork
**Run:** test_wait_fork

## clone():

The function clone() creates a new child process which shares memory, file descriptors and signal handlers with the parent. It implements threads and thus launches a function as a child. The child terminates when the parent terminates.

**clone():** function is used to create a child process from calling process. Calling process will be the parent process of the child process.

*clone()* function is used to run a particular function in a separate thread other than the calling process. Unlike *fork()* function where rest of the code after *fork()* function call will execute both parent and child processes. But with *clone()*, a particular function will be executed in separate thread. Once the function is returned, the thread will be closed; means child process will terminates.

The syntax of the function likes below:

int clone(int (*pfn)(void *), void *child_stack, int flags, void *arg, ...);

**pfn**: is a pointer to a function, which takes void * as an argument and returns an int value. This is the function going to execute in the child process; if the child process successfully created. Unlike fork(), the process created through clone() function; can share some resources with calling process (the parent).

**child_stack:** Calling process's responsibility to create a stack for child process. Usually stacks grow downward on most of the processors; child_stack should points to the topmost address of the stack memory created by calling process for child process.

**flags:** indicates what is shared between the calling process and child process. We can pass multiple flags with bitwise-OR.

**arg:** is the argument for the function which points to pfn. We can pass multiple arguments through clone() function; because of ellipse (three dots).

On function call success, clone() will return the thread ID of the process. If the function fails, it returns "-1".

**Example 7**

The Example 5 will display "Nth" mathematical table. This table will form in child process and display the mathematical table on console window. We have to pass argument "N" when running the program from console; if it is "5", this program will display "5th" mathematical table.

In the code was created a function fn and place our mathematical table generation logic. This function fn is taking a single argument declared as per the guideline defined by clone(). So, we can pass this function to clone(). We have to create a stack for child process. So, we have allotted a memory using malloc function and release the memory using free once child process terminates. Observe that we pointed the pointer to point to topmost address of the stack memory. We are calling clone() with SIGCHLD flag. So, we can keep calling process in wait state using wait() until the child process finishes.

We are taking an argument through main() and passing it to child function through clone(). Inside child function we are converting it to an integer; as our "Nth" mathematical table is based on int value. Once you compile and run our program; it will display the below result.

**test_clone.c**

**Compile**: gcc -o test_clone.c test_clone
**Run:** ./test_clone 5 (Remember to pass an argument while running the application)

## Sending a Signal to Another Process: System Call kill()

To send a signal to another process, we need to use the Unix system **kill()**. The following is the prototype of **kill()**:

int kill(pid_t pid, int sig)

- **pid**, is the process ID you want to send a signal to, and the second,
- **sig**, is the signal you want to send. Therefore, you have to find some way to know the process ID of the other party.
- If the call to **kill()** is successful, it returns 0; otherwise, the returned value is negative.
- Because of this capability, **kill()** can also be considered as a communication mechanism among processes with signals **SIGUSR1** and **SIGUSR2**.
- The **pid** argument can also be zero or negative to indicate that the signal should be sent to a group of processes. But, for simplicity, we will not discuss this case.

**Example 8**

This example consists of two processes, one of which sends **SIGINT** and **SIGQUIT** to interrupt and kill the other's computation.

**test_signal.c**

**Compile**: gcc -o test_signal.c test_signal
**Run:** ./test_signal

This program installs two signal handlers for SIGINT and SIGQUIT. Then, it creates a shared memory, attaches this shared memory to its address space, and saves its process ID there for another process to retrieve. After this is done, this program enters an infinite loop.

The SIGINT handler prints out a message indicating that a SIGINT signal was received.
The SIGQUIT handler prints out a message indicating a SIGQUIT was receives. Then, it detaches and removes the shared memory, and exits!

Then, the program terminates. This is not very interesting !!!!!.

**Example 9**

This example consists of two processes, the first, major process that performs the computation.

**test_kill.c**

**Compile**: gcc -o test_kill.c test_killl
**Run:** ./test_killl

This program requests the shared memory segment created by the previous program. Since both programs use **ftok()** to create the key, they should be in the same directory.
Then, it retrieves the process ID stored in the shared memory segment and enters an infinite loop asking for a single character input.
If the input is **i**, a **SIGINT** is sent to the other process with **kill()**.
If the input is **k**, a **SIGQUIT** is sent to the other process with **kill()**. After this is done, this program exits.

## system()

system - execute a shell command

The system() call will execute an OS shell command as described by a character command string. This function is implemented using fork(), exec() and waitpid(). The command string is executed by calling /bin/sh -c command-string. During execution of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored. The call "blocks" and waits for the task to be performed before continuing.

**Example 10**

**test_kill.c**

**Compile**: gcc -o test_kill.c test_killl
**Run:** ./popen

The statement "Command done!" will not print untill the "ls -l" command has completed.

## popen()

popen - process I/O

The `popen()` call opens a process by creating a pipe, forking, and invoking the shell (bourne shell on Linux). The advantage to using `popen()` is that it will allow one to interrogate the results of the command issued.
This example opens a pipe which executes the shell command "`ls -l`". The results are read and printed out.
**Example 11**

**popen.c**

**Compile**: gcc -o popen.c popen
**Run:** ./popen

## exec() functions and execve():

The exec() family of functions will initiate a program from within a program. They are also various front-end functions to `execve()`.
The functions return an integer error code. (0=Ok/-1=Fail).

### execl() and execlp():

The function call "`execl()`" initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. `/bin/ls`) and arguments are passed to the function. Note that "arg0" is the command/file name to execute.

```
int execl(const char *path, const char *arg0, const char *arg1, const
char *arg2, ... const char *argn, (char *) 0);
```

The routine `execlp()` will perform the same purpose except that it will use environment variable `PATH` to determine which executable to process. Thus a fully qualified path name would not have to be used. The first argument to the function could instead be "ls". The function `execlp()` can also take the fully qualified name as it also resolves explicitly.
**Example 12**

**execl.c**

**Compile**: gcc -o execl.c execl
**Run:** ./execl

### execv() and execvp():

This is the same as `execl()` except that the arguments are passed as null terminated array of pointers to `char`. The first element "argv[0]" is the command name.

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

The routine `execvp()` will perform the same purpose except that it will use environment variable `PATH` to determine which executable to process. Thus a fully qualified path name would not have to be used. The first argument to the function could instead be "ls". The function `execvp()` can also take the fully qualified name as it also resolves explicitly.

**Example 12**

**execv.c**

**Compile**: gcc -o execv.c execv
**Run:** ./execv

References:

CS4411 Introduction to Operating Systems. Available in: http://www.csl.mtu.edu/cs4411.ck/www/. Acess: September, 2017.

YoLinux Tutorial: Fork, Exec and Process control. Available in:<http://www.yolinux.com/TUTORIALS/ForkExecProcesses.html>. Acess: September, 2017.