# CS 332/532 – 1G- Systems Programming

# Lab 7

## Objectives

The objective of this lab is to introduce you to create processes and manage them.

1. Creating child processes using fork.
2. Replacing child process using exec.

## Description

### Example 1:

We can use the fork() system call to create a new process (referred to as the child process) which is an identical image of the calling process (referred to as the parent process). Here is the C interface for the fork() system call:

```
#include <unistd.h>

pid_t fork(void);
```

If the fork() call is successful, then it returns the process ID of the child process to the parent process and returns 0 in the child process. fork() returns a negative value in the parent process and sets the corresponding errno variable (external variable defined in *errno.h*) if there is any error in process creation and the child process is not created. We can use perror() function (defined in *stdio.h*) to print the corresponding system error message. Look at the man page for perror to find out more about the perror() function.

Once the parent process creates the child process, the parent process continues with its normal execution. If the parent process exits before the child process completes its execution and terminates, the child process will become a zombie process (*i.e.*, a process without a parent process). Alternatively, the parent process could wait for the child process to terminate using the wait() function. The wait() system call will suspend the execution of the calling process until one of the child process terminates and if there are no child processes available the wait() function returns immediately. The wait() call returns the PID of the child process that terminated when successful, otherwise, it returns -1. The wait() call also sets an integer value

that is passed as an argument to the function which can be inspected with various macros provided in <sys/wait.h> to determine how the child process completed (e.g., terminated normally, terminated by a signal).

If the calling process created more than one child process, we can use the waitpid() system call to wait on a specific child process to change state. A state change could be any one of the following events: the child was terminated; the child was stopped by a signal; or the child was resumed by a signal. Similar to wait(), waitpid() returns the PID of the child process that changed state when successful, otherwise, it returns -1.

Here are the C APIs for the wait() and waitpid() system calls:

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

The following sample program illustrate how to use fork() to create a child process, wait for the child process to terminate, and display the parent and child process ID in both processes.

Source file: `fork.c`

```
/*
 * Simple program to illustrate the use of fork() system call
 * to create a new process.
 * To Compile: gcc -Wall fork.c
 * To Run: ./a.out
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv) {
   pid_t pid;
   int status;

   pid = fork();
   if (pid == 0) { /* this is child process */
      printf("This is the child process, my PID is %ld and my parent PID is %ld\n",
         (long)getpid(), (long)getppid());
   } else if (pid > 0) { /* this is the parent process */
      printf("This is the parent process, my PID is %ld and the child PID is %ld\n",
         (long)getpid(), (long)pid);

      printf("Wait for the child process to terminate\n");
```

```
    wait(&status);  /* wait for the child process to terminate */
    if (WIFEXITED(status))  { /* child process terminated normally */
       printf("Child process exited with status = %d\n", WEXITSTATUS(status));
    } else { /* child process did not terminate normally */
       printf("ERROR: Child process did not terminate normally!\n");
       /* look at the man page for wait (man 2 wait) to
          determine how the child process was terminated */
    }
  } else { /* we have an error in process creation */
    perror("fork");
    exit(EXIT_FAILURE);
  }

  printf("[%ld]: Exiting program .....\n", (long)getpid());
  return 0;
}
```

You should see the following output if the program executes correctly (the PID values will be different):

```
This is the parent process, my PID is 13129 and the child PID is 13130
Wait for the child process to terminate
This is the child process, my PID is 13130 and my parent PID is 13129
[13130]: Exiting program .....
Child process exited with status = 0
[13129]: Exiting program .....
```

## Example 2:

Note that the child process is a copy of the parent process and control is spilt at the invocation of the fork() call between the parent and the child process. If we like the child process to execute a different program other than making a copy of the parent process, we can use the exec family of system calls to replace the current process image with a new one. Here is the C APIs for the exec family of system calls:

```
#include <unistd.h>

int execl(const char *pathname, const char *arg, ...);
int execlp(const char *filename, const char *arg, ...);
int execle(const char *pathname, const char *arg, ..., char * const envp[]);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *filename, char *const argv[]);
int execvpe(const char *filename, char *const argv[], char *const envp[]);
```

We will use the execl() to replace the child process created by fork(). The execl() function takes as arguments the full pathname of the executable along with a pointer to an array of characters for each argument. Since we can have a variable number of arguments, the last

argument is a null pointer. The updated program is shown below with the new code highlighted.

Source file: `forkexecl.c`

```
/* Simple program to illustrate the use of fork-exec-wait pattern.
 * To Compile: gcc -Wall forkexecl.c
 * To Run: ./a.out
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv) {
   pid_t pid;
   int status;

   pid = fork();
   if (pid == 0) { /* this is child process */
      execl("/bin/uname", "uname", "-a", (char *)NULL);
      printf("If you see this statement then execl failed ;-(\n");
      perror("execl");
      exit(-1);
   } else if (pid > 0) { /* this is the parent process */
      printf("Wait for the child process to terminate\n");
      wait(&status);  /* wait for the child process to terminate */
      if (WIFEXITED(status))  { /* child process terminated normally */
         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
      } else { /* child process did not terminate normally */
         printf("Child process did not terminate normally!\n");
         /* look at the man page for wait (man 2 wait) to determine
            how the child process was terminated */
      }
   } else { /* we have an error */
      perror("fork"); /* use perror to print the system error message */
      exit(EXIT_FAILURE);
   }

   printf("[%ld]: Exiting program .....\n", (long)getpid());
   return 0;
}
```

The above program will display something like this (the hostname vulcan9 will be different based on the system where you are executing this program):

```
Wait for the child process to terminate
Linux vulcan16.cis.uab.edu 3.10.0-957.12.1.el7.x86_64 #1 SMP Mon Apr 29 14:59:59 UTC 2019 x86_64 x86_64 x86
```

```
_64 GNU/Linux
Child process exited with status = 0
[13439]: Exiting program .....
```

Let us look at the different versions of the exec functions. There are two classes of exec functions based on whether the argument is a list of separate values (l versions) or the argument is a vector (v versions):

1. functions that take a variable number of command-lines arguments each as an array of characters terminated with a null character and the last argument is a null pointer – *(char *)NULL (execl, execlp, and execle)*
2. functions that take the command-line arguments as a pointer to an array of pointers to the arguments, similar to argv parameter used by the main method (execv, execvp, and execvpe)

Functions that have *p* in the name use *filename* as the first argument while functions without *p* use the *pathname* as the first argument. If the filename contains a slash character (*/*), it is considered as a pathname, otherwise, all directories specified by the PATH environment variable are searched for the executable.

Functions that end in *e* have an additional argument – a pointer to an array of pointers to the environment strings.

Additional examples: `forkexecv.c forkexecvp.c forkexecvp2.c`

## Lab 7 Assignment (Not REQUIRED / Bonus Points)

Write a program that takes a filename as a command-line argument and performs the following steps:

1. Open the file specified as the command-line argument.
2. Read contents of the file one-line at a time.
3. Use fork-exec to create a new process that executes the program specified in the input file along with the arguments provided.
4. The parent process will make note of the time the program was started (you can use a timer such as the *time* function defined in <*time.h*> to capture the time before the fork method is invoked, you can find out more about *time* function by typing *man time*).
5. Then the parent process will wait for the child process to complete and when the child process terminates successfully, the parent process will capture the time the child process completed (you can again use a timer function to capture the time when the *wait* function returns).
6. Open a log file (say, output.log) and write the command executed along with arguments, start time of the process, and the end time of the process separated by tabs.
   Use *ctime* function to write the time in a human readable form.
7. Go to step 2 and repeat the above process for very command in the input file.

Please use standard I/O library function calls for all I/O operations in this assignment. Make sure you open the file in the appropriate modes for reading and writing and also make sure to close the file, especially when you are writing the file.

Here is a sample input file:

```
uname -a
/sbin/ifconfig
/home/UAB/puri/CS332/lab7/hw1 500
/home/UAB/puri/CS332/lab7/hw1 1000
```

Here is the corresponding sample output log file:

```
uname -a    Thu Oct 10 17:43:44 2019    Thu Oct 10 17:43:44 2019
/sbin/ifconfig    Thu Oct 10 17:43:44 2019    Thu Oct 10 17:43:44 2019
/home/UAB/puri/CS332/lab7/hw1 500    Thu Oct 10 17:43:45 2019    Thu Oct 10 17:43:46 2019
/home/UAB/puri/CS332/lab7/hw1 1000    Thu Oct 10 17:43:46 2019    Thu Oct 10 17:43:57 2019
```

**Hints:** You can download the sample C program `hw1.c`. This program has some similar functions, feel free to use them. Compile the program using the following command:

```
$ gcc -Wall -O -o hw1 hw1.c
```

## Optional Submission

You are required to submit the lab to Canvas by the deadline. No late submissions will be accepted.

Submission Checklist:

- ❑ Upload the C source file (.c file) to Canvas as part of this lab submission.
- ❑ Test your program on GitHub codespaces and include a link to your lab assignment in your shared repository on GitHub in the comments.
- ❑ Upload your README.md file which should include instructions on compiling through your Makefile, running the program and documentation.
- ❑ Screenshots of your output.
- ❑ Upload a Makefile for compiling your code.

Please do not upload executables or object files. Independent Completion Forms are not required for lab assignments.