CENG328 Operating Systems
Laboratory Chapter 2

1 Processes

Nowadays, a regular computer allows multi-programming. Few of them are: ls, cp, mkdir, Firefox, Kate, GCC, etc... When a program is executed, by clicking on its icon or its name typed in a terminal window, the operating system creates an executable copy in the RAM and starts its execution. A **running program** is known as **process**. There may be more than processes active in a system at the same time. A process can start another process during the execution.

1.1 Process Creation

More than one process can be created by either running the same program again and again, or internally by another process. Try the following program (fork.c):

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

int main() {
    printf("test 1\n");
    fork();
    printf("test 2\n");
    return 0;
}
```

When executed, you should observe that this program produces one line of "test 1" string and two lines of "test 2" strings. Why?

At the moment of first printf(), there is only one running process (the one you have executed). When the program code reaches the **fork()** system call line, the operating system creates a duplicate of this process that shares the same code. Therefore all lines of code following the fork() system call will be executed by two seperate processes. This is the reason you get two "test 2" strings. The original process is called as **parent process** and the new process is called as **child process**.

The following figure visually depicts this procedure:

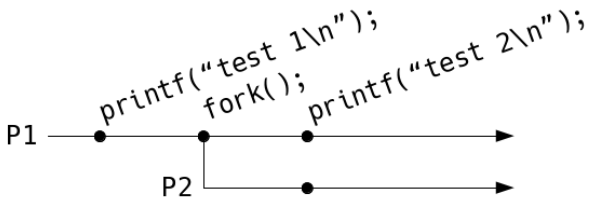

Figure 1: fork() system call

1.2 Controlling Processes

Any running process is assigned a unique **process ID (pid)** by the operating system. You can use these id's to check if a process is successfully created or not and also assign different programs to them (pid.c):

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

int main() {
  pid_t child_pid;
  printf("Original pid is %d\n", getpid());

  child_pid = fork();
  if (child_pid == -1)
    printf("Can't create child process\n");
  else if (child_pid == 0)
    printf("This is child process, pid = %d, parent pid = %d\n", getpid(), getppid());
  else /* child_pid > 0 */
    printf("This is parent process, pid = %d\n", getpid());
  return 0;
}
```

getpid() returns the process id of the calling process and **getppid()** returns the pid of its parent process. fork() returns -1 on error. If succeeded, it returns 0 to the newly created child process and pid of child process to parent process (a positive integer).

Do note that since we have seperate processes after a fork() call, variables in the program will be duplicated too, but because two processes have their variables stored in different addresses in memory, they may have different runtime values (fork2.c):

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

int main() {
    pid_t child_pid;
    int a = 5;
    printf("a before fork: %d\n", a);
    child_pid = fork();
    if (child_pid == 0)
        a = 7;
    printf("a after fork: %d\n", a);
    return 0;
}
```

What is the output of this program? Why?

1.3 Zombie Processes

If a child process terminates before its parent process, it is still kept in the memory until its parent process terminates too. A process in this state is called as a **zombie** process or **defunct** process. Study the following program (zombie.c):

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

int main() {
    pid_t child_pid;
    int i;
    char* msg;

    child_pid = fork();
    if (child_pid == -1) {
        perror("fork");
        return -1;
    } else if (child_pid == 0) {
        msg = "CHILD";
        i = 3;
    } else if (child_pid > 0) {
        msg = "PARENT";
        i = 10;
    }
    for (; i >= 0; i--) {
        printf("[%s] i = %d\n", msg, i);
        sleep(1);
    }
    return 0;
}
```

While running this program, open a second terminal window and run "**ps xfau**" several times. After three seconds, you should see that the child process will have some changes:

```
5636  0.0  0.0   4352   780 pts/4    S+   12:12   0:00  |   \_ ./zombie
5637  0.0  0.0   4352    72 pts/4    S+   12:12   0:00  |       \_ ./zombie
...
[3 SECONDS]
...
5636  0.0  0.0   4352   780 pts/4    S+   12:12   0:00  |   \_ ./zombie
5637  0.0  0.0      0     0 pts/4    Z+   12:12   0:00  |       \_ [zombie] <defunct>
```

With the help of this behaviour, you may let the parent process wait for a child process to end its job before continuing (wait.c):

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

int main() {
    pid_t child_pid;
    FILE* fp;
    int status;
    char text[30], input[30];

    child_pid = fork();
    if (child_pid == -1) {
        perror("fork");
        return -1;
    }
    if (child_pid > 0) {
        waitpid(child_pid, &status, 0);
        fp = fopen("hello.txt", "r");
        fgets(text, 30, fp);
        printf("Read from file: %s\n", text);
        fclose(fp);
    }
    if (child_pid == 0) {
        fp = fopen("hello.txt", "w");
        printf("Enter something: ");
        scanf("%s", input);
        fprintf(fp, "%s", input);
        fclose(fp);
    }
    return 0;
}
```

1.4 Starting Programs from Other Programs

We have used fork() in order to create duplicate processes of the same programs. But sometimes it may be necessary to execute different programs. In such cases it is possible to start a new program directly from instructions written in source code. Study the following code (execl.c):

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

int main() {
    printf("Displaying directory contents:\n");
    execl("/bin/ls", "ls", NULL);
    printf("Displayed directory contents.\n");
    return 0;
}
```

What is the output of this program?

There are 6 functions (execl, execlp, execle, execv, execvp, execvpe) that can be used to call other programs. They all do their jobs by replacing the current program with the new one, therefore it is

impossible to return to the original code unless these functions fail. There are minor differences between these six functions; you may learn all about them by executing "**man execl**".

Parameters of execl function can be explained in three parts:

1. The first parameter must be the path of the program to be executed. You may learn the program paths with the help of **which** command (e.g. which ls).
2. The following parameters are all passed to the new process as command line arguments. Study the following examples:
   - `execl(‘‘/bin/ls’’, ‘‘ls’’, NULL);` - The /bin/ls program is executed and its command line arguments are "ls".
   - `execl(‘‘/bin/ls’’, ‘‘ls’’, ‘‘-l’’, ‘‘-a’’- NULL);` - The /bin/ls program is executed and its command line arguments are "ls -l -a".
3. Finally, execl parameters must always end with NULL, so that the system knows when to stop parsing arguments.

Since these functions replace the program of the calling process, it is a good exercise to create a dedicated child process for such purposes; so that the main process still continues running. Study the following code (execl2.c):

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

int main() {
     int status;
     pid_t child_pid;
     child_pid = fork();

     if (child_pid == -1) {
          perror("fork");
          return -1;
     } else if (child_pid == 0)
          execl("/bin/ls", "ls", NULL);
     else if (child_pid > 0) {
          printf("Displaying directory contents:\n");
          waitpid(child_pid, &status, 0);
          printf("Displayed directory contents.\n");
     }
     return 0;
}
```

2 Exercises

1. Read man pages for the following system calls: fork, getpid, getppid, wait.
2. Execute the given programs at least once.
3. Modify fork.c such that it contains three consecutive fork() calls. How many processes will be created? What will the output be?
4. Modify pid.c such that the parent process also calls getppid(). You should see the pid of an another process. Which process is it? Does it have a parent process too? How far can you go?
5. Modify execl.c such that you use execl, execlp, execv and execvp correctly one-by-one. You may learn the differences between them by executing "man execl".