February 13th, 2008

LIACS, Operating Systems course, Spring Semester 2008 by Nies Huijsmans
Url http://www.liacs.nl/~shenstra/os

Lab Assignment #1, original author is dr. A.H. Deutz

Student assistant, Fabrice Colas (LIACS, room 133, fcolas@liacs.nl, 071 527 7033)

Reports to be sent by February the 27th at fcolas@liacs.nl with the following deliverables

1) http://www.liacs.nl/~shenstra/os/documents/assignment1_report.doc with your answers to the questions
2) C program **with comments** of Part A iv), **compiling** and **running**
3) C program **with comments** of Part B, **compiling** and **running**
4) your answer to Part D
5) in assignment1_report.doc, explain what you have done and learned from this lab? Finally, tell what your lab partner and eventually your colleagues taught you.

Check that each file contains
the name of the students, lab assignment #1, due date, and date turned in.
Pack and compress everything into a ZIP archive

Grading of the lab assignment

1) pay a very special attention on the documentation of your source code
2) eventually, refer to webpages, books or colleagues that you got ideas from
3) your final source code should **run** and **compile**

4) **optionally** report the portability issues when you tried to compile and run your program under other OS like Windows, MacOSX, Minix, other Linux OS

Preliminary remarks

1. *replace the code on page 63 of the book of Gary Nutt with the code in the Errata list*
2. *in the code of page 64 of the book of Gary Nutt omit in the definition of WHITESPACE "." And "," ; leave in space, tab and newline. In other words it should read: #define WHITESPACE " \t\n" * with leading space! **

Goal: Learn how to create and how to work with/how to manipulate processes (or rather the implementation of processes) in Linux. Notably 1) the creation of processes (the system calls `fork` and `exec`) 2) the creation of clones (the system call `fork`), 3) how a process can change its own core image (the system call `exec`), 4) how processes can communicate in a very rudimentary way (the system calls `wait` and `waitpid`), and lastly 5) the `exit` system call.

Introduction

The most central concept in any operating system is the *process*. A very rough definition of this notion is: an abstraction of a running program. Later we shall give a more precise definition of this notion and also look how it is implemented in operating systems, in particular in the Linux operating system. Later on we shall also distinguish between the classical notion of the process and the modern one (which includes the notion of threads).

The interface between the operating system and the user programs is defined by the set of system calls that the operating system provides. To understand what operating systems do, we must study this interface closely. The system calls available in the interface vary from operating system to operating system (although the underlying concepts tend to be similar). In this assignment we will study the process management sys calls in a Unix like operating system (i.e., Linux).

Process Management

Call	Description
`pid = fork( )`	Create a child process identical to the parent
`pid = wait(&statloc)` `pid = waitpid(pid, &statloc, options)`	Wait for a child to terminate
`s = execve(name, argv, environp)`	Replace a process' core image
`exit(status)`	Terminate process execution and return status

Fork is the only way to create a new process in UNIX. It creates a duplicate of the original process, including all the file descriptors, registers – everything. After the *fork*, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the *fork*, but since the parent's data are copied to create the child, subsequent changes to one of them do not affect the other one. (The program text, which is unchangeable, is shared between parent and child.) The *fork* call returns a value, which is zero in the child and equal to the child's process identifier (`PID`) in the parent. Using the returned `PID`, the two processes can see which one is the parent process and which one is the child process.

In most cases, after a *fork* the child will need to execute different code from the parent. Consider the case of the UNIX shell (also called **command line interpreters**). It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a *wait* or *waitpid* system call, which just waits until the child terminates (any child if more than one exists). *Waitpid* can wait for a specific child, or for any old child by setting the first parameter to -1. When *waitpid* completes, the address pointed to by the second parameter, *statloc* will be set to the child's exit status (normal or abnormal termination and exit value). Various options are also provided, specified by the third parameter.

Now consider how *fork* is used by the a(the) UNIX shell. When a command is typed, the shell forks off a new process. This child process must execute the user command. It does this by using the *execve* system call, which causes its entire core image to be replaced by the file named in its first parameter.

A stripped down UNIX shell:

```
while (1) {
    echo_prompt();
    read_and_parse_command(command, parameters);
    if (fork()!=0) {
        /* code for parent */
        waitpid(-1, status, 0);
    } else {
        /* code for child */
        execve (command, parameters, 0);
    }
```

```
}
```

In the most general case, *execve* has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array. (The various library routines, including *execl*, *execle*, *execv* are provided to allow the parameters to be omitted or specified in various ways.)

If *execve* seems complicated, do not despair; it is semantically the most complex of all the POSIX system calls. All the other ones are much simpler.

As an example of simple one, consider *exit*, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent via *statloc* in the *waitpid* system call.

Assignment

In part A of the assignment we will experiment with the system calls *fork* and *execve* separately. Subsequently we will see what the effects are, if we use them in conjunction. As an aside we also learn a little bit more C programming: we will write a C program which uses command line parameters. In part B we will built a simple UNIX shell.

Part A

i) Take your favorite C program and embellish it with the system call *fork*. For instance:

```
/* version 0 ; sys name: parent0.c a simple parent
    child will be created;
    exact copy of parent;
    shares hardly anything with parent;
    child and parent share files though but no data/vars etc;
    child has its own address space.
    In  this version we don't  change the core image for the
child process. */
/*
#include <sys/wait.h>
#define NULL 0
*/
int main (void) {
    int pidValue = fork();
    printf("the value which is returned by fork: %d \n",
pidValue);
    printf ("pid: %d \n", getpid());
    printf("Process[%d]: Parent or Child! terminating ... \n",
getpid());
    exit(0);
}
```

Of course, compile and run this or your own program ... and explain the behavior of this or your own program.

ii) In this exercise we experiment with the sys call *execve* (in isolation): Take your favorite program and change into a program such that associated processes will change their core image. For instance:

```
/* file name: execUse.c  */
#include <unistd.h>
int main () {
    char * a[3];
    a[0] = "dont care what you put here\0";
    a[1] =".";
    a[2] = "..";
    a[3] = 0;
    execve("/bin/ls", a, 0);
    return 1;
}
```

Again compile and run this or your own program and explain its behavior. You get a slightly more interesting exercise by taking the core image of your own favorite program instead of the existing binary "ls" - do that!

Iii) In this exercise we experiment with the sys calls *fork* and *execve* in one program. Consider the following two programs:

```
/* a simple parent; version1; sys name: parent1.c
Now we better create a program, which will replace the replica
of the parent. We have named this replacing program child1.c
and have made an executable out of it carrying the name child1
(use the -o option). In parent process fork() returns the pid
of the child; in the child process fork returns 0. */

#include <sys/wait.h>
#define NULL 0

int main (void) {
    int pidValue = fork();

    if (pidValue > 0) {
        /* Parent code here */
        printf ("Process[%d]: Parent in execution ... \n",
getpid());
        sleep(2);
        if (wait(NULL) >0) /* Child terminating */ {
            printf("Process[%d]: Parent detects terminating
child \n", getpid());
        }
        printf("Process[%d]: Parent terminating ... \n",
getpid());
    } else {
        if (pidValue == 0) {
            /* this is the child's process */
            execve("child1", NULL, NULL);
            exit(0); /* should never get here, terminate */
        } else {
            /* should never get here; things such as not
enought mem */
            exit(0);
        }
    }
}
```

The child process will "ask" for execution of the *execve* sys call. So we better provide the first parameter: a core image named "child1" (the other parameters are set to 0 and thus not needed). We can accomplish this by compiling the following program (and for ease of use put it in the same directory as the core image of the previous program):

```
/* a simple child */
int main (void) {
    /* the child's process new program */
    /* This program replaces the parent's program */
    printf("Process[%d]: child in execution ... \n",
getpid());
    sleep(1);
    printf("Process[%d]: child, terminating .. \n", getpid());
}
```

iv) Write a simple C-program to which command line parameters can be passed. Write this program in such a way that the command line parameters will be printed

Part B

Starting with code in section 2.4 of the book design and implement a simple, interactive shell program that prompts the user for a command, parses the command, and then executes it with a child process. In your solution you are required to use the *execv* instead of *execp*, which means that you have to read the `PATH` environment variable, then search each

directory in the `PATH` for the command file name that appears on the command line.

Part C
Inspect the `PCB` in the Linux source code. What is the `PCB` called in Linux?

Part D
Why did the designers of a shell use the sys calls *`fork`* and *`exec`*?
What are the advantages of such a design?

Collaboration:
Part A: consult and discuss with your lab; then carry it out on your own.
Parts B-D: work in pairs on these parts of the assignment.