Machine Problem 2: System Calls and Critical OS Functions

DUE DATE: Friday Feb 17, 2017 11:59pm

Introduction:

There are several operating system functions that are essential to the proper functioning of a computer. These functions may seem simple in name, but they involve many complex instructions that interact with the entire system.

For example in order to create a new process, memory needs to be allocated and organized, the program to be run must be copied from storage into memory, kernel data structures must be set up, and the process must be inserted into the system scheduler. Each of these actions is rife with potential for error or sabotage. Imagine if you (the programmer) had to perform each one of these actions manually. To run a simple hello world program, you would need an intricate understanding of interacting with secondary storage, knowledge of the memory allocation portion of the operating system, complete understanding of every single kernel data structure dealing with processes in execution, and considerable knowledge of the system in general. Much of this low level information may even be architecture specific. This means that you might need to learn all this information over again if you moved to a new computer.

Even something as simple as printing a string to the console requires complex interactions with the UNIX filesystem, kernel data structures, and the terminal.

Complexity aside, there is great potential for harm to be dealt to the system if these complex actions (i.e. dealing with memory, disks, data structures) was open for regular programmers to perform. A novice, unaware user could inadvertently corrupt his/her entire operating system by performing a single erroneous operation. A malicious user could use these low level operations to compromise the system and execute any unauthorized commands they so choose.

Instead of this, the Linux/UNIX operating system (and almost all operating systems for that matter) abstract all detailed and/or architecture specific operations into a series of predefined functions known as system calls. Kernel functions are restricted to normal users (i.e. you as a non-root programmer can't manipulate kernel data structures). Instead, the system calls are there to serve as entry points into the kernel so that they can perform all this work for you in a safe, protected way.

Each call performs a specific operation. For example, process creation is reduced from an intricate dealing in memory management to two simple calls: fork() and exec(). The fork() call duplicates the current process and ensures that the memory space is set up correctly, and the exec() call replaces the address space of the newly created process with that of a completely different process pertaining to the program called by exec.

Linux systems that you will work with today employ hundreds of system calls. In this assignment, you will explore a limited subset of some of the most important system calls,

specifically those involved in process creation and management along with basic system calls for reading and writing data to and from file descriptors.

Side note: If you wish to have a more complete reference to the system calls available on the UNIX/Linux system, please consult *Advanced Programming in the UNIX Environment* by Richard Stevens.

Assignment:

In this assignment, you are to implement a series of programs as standalone files that use fundamental system calls. In doing so, you will learn how these functions work and how to use them in future coding assignments. Specifically, this assignment concentrates on the following functions: fork, exec, wait/waitpid, pipe, and unbuffered I/O (read, write, close). The usages of these functions are incredibly well documented in sections 2 and 3 of the Linux Programmer's Manual (i.e. built in man pages).

Just as a note, this assignment will not be difficult to code and shouldn't take you too much time to complete. Most of the functions you will create are only a few lines. However, the understanding of these functions will help you considerably in future machine problem assignments. The programs here are even designed so that you can use the code semi-directly later. You must use the functions dictated in each of the parts below. NOTE: You may not use the corresponding c/c++ system() function. Doing so will result in a 0 for that part of the assignment.

To complete the assignment, you are required to write and turn in the following:

- **A total of 4 programs, mp2_part1.cpp through mp2_part4.cpp**: each implements one of the following exec functions in the following order (i.e. the first one in the list should be part1, the second one in the list should be part2, and so on...): execl, execlp, execv, execvp. Use each exec function to start the program **ls -la**. Note: There are more than 4 exec functions implemented on most Linux/UNIX machines. The others deal with passing a custom environment pointer which is beyond the scope of this assignment.

- **A program named mp2_part5.cpp** which calls fork to create a child. Have the child print out the string "Hello" and then the parent print out the string "World!" (Note: Don't print the quotes). Use either wait() or waitpid() to ensure that the program prints out the strings in the correct order. Note: You can use either of the two functions to synchronize the output of your parent child program.

- **A program named mp2_part6.cpp** which starts the command "ls -la" using a fork() followed by an exec() (any of the exec functions will work). Use a UNIX pipe system call to send the output of ls -la back to the parent, read it using the read() function, and then write it to the console using the write() function. Note: the pipe will not work if you do not close and/or redirect the correct file descriptors. It's up to you to figure out which ones those need to be.

- **A program named mp2_part7.cpp** which implements the command pipeline "ls -la | tr [a-zA-Z0-9] a". Use any combination of fork, exec, wait, read, and write necessary to create this functionality. **Note: the parent process must be the one to output the data to the console.**

Grading Rubric:

A total of 100 points are allocated for this assignment. Due to the simple nature of these assignments, each part of the assignment will be graded on a pass/fail basis. Full credit will be given for each function that runs as expected. No credit will be provided for functions that do not work as stated in the requirements.

Important Note: The grading for each of the files mp2_part1.cpp through mp2_part7.cpp will be done using a bash script. We understand that it is easy to make your program fake the output (even though actually doing the assignment would barely be harder than going through the trouble to fake the output...). However, the grading script will be checking very carefully to make sure that your submitted code performs the required functions. If any of those checks determines that you submitted fraudulent code (i.e. you're trying to trick the grading script), your entire assignment will be given a grade of zero.

Note: Make sure not to zip your files.

- mp2_part1.cpp: 5 points
- mp2_part2.cpp: 5 points
- mp2_part3.cpp: 5 points
- mp2_part4.cpp: 5 points
- mp2_part5.cpp: 15 points
- mp2_part6.cpp: 25 points
- mp2_part7.cpp: 40 points

Advanced Concepts:

fork();

When the operating system is initialized, three processes are created. The first, known as the swapper (process id 0), serves as the scheduler for jobs on the system. The second, known as init (process id 1), sets up many processes and services across the system. After it performs its initial tasks, init becomes a looping call to wait() which reclaims the state of orphaned processes. The third, known as the pagedaemon (process id 2), is responsible for memory management.

Other than these three processes started by the operating system, every other process in the system is brought to life through a call to the fork() system call. A process calling fork is copied by the kernel. At this point, the process that called fork is known as the parent, and the newly

created process is known as the child since the parent process caused the child to be created. The newly created process is essentially the same as the child, even having the exact same variables and open files. Fork is unique in that it is called once by the parent process and returns twice (to the parent and child separately). To the parent, fork returns the process id (PID) of the newly created child. To the child, fork returns 0. If the call fails entirely, -1 is returned to the parent, and no child is created. For more information about fork, see its manpage by calling `man 2 fork` .

Copying the entire context of a process could take quite a long time, especially if the parent process contains a large amount of data. Instead, modern kernels perform an action called **copy-on-write.** When the fork call returns, both processes point to the same regions in memory. As soon as either process wants to write data anywhere in the address space, the kernel creates a new page in memory for that process. Therefore, both processes use the same memory regions initially. As time progresses and both processes change data in their address spaces, the regions of memory that they refer to will no longer be the same. Some areas, such as the text segment which holds program code, never change. Those regions of memory can still be shared for as long as both processes are executing on the system, thus reducing the amount of extra memory required to start a new process.

A pseudocode description of the actual fork algorithm is provided below (Bach).

```
algorithm fork
input:  none
output: to parent proces, child PID number
                to child process. 0
{
        check for available kernel resources;
        get free proc table slot and unique PID number;
        check that user not running too many processes;
        mark child state as ”being created”;
        copy data from parent proc tabgle slot to new child slot;
        increment counts on current directory inode and changed root (if applicable);
        increment open file counts in file table;
        make copy of parent context (PCB, text, data, stack) in memory);
        push dummy system level context layer onto child system level context;
                dummy context contains data allowing child process to recognize
                itself, and start running from here when scheduled
        if(executing process is parent process)
        {
                change child state to ”ready to run”;
                return(child ID);
        }
        else
        {
                initialize PCB timing fields;
```

```
                return(0);
        }
}
```

The Six Exec Functions

After executing fork() , you are left with two identical processes. This isn't usually particularly useful unless you specifically need two exact copies of a process. Instead, UNIX provides a function, known as exec, which allows you to change a process's address space to run an entirely new program. Running an exec command deletes the existing text, data, and stack segments of the existing process and replaces them with those of a new program. The algorithm for exec is presented below (Bach 218).

```
algorithm exec
input:  (1) file name
                (2) parameter list
                (3) environment variables list
output:                 none
{
        get file inode (algorithm namei);
        very file is executable and user has permission to execute;
        read file headers, check that it is a load module;
        copy exec parameters from old address space to system space;
        for(every region attached to process)
        {
                detach all old regions (algorithm detach);
        }
        for(every region specified in load module)
        {
                allocate new regions (algorithm allocreg);
                attach the regions (algorithm attachreg);
                load region into memory if appropriate (algorithm loadreg);
        }
        copy exec parameters into new user stack region;
        special processing for setuid programs, tracing;
        initialize user register save area for return to user mode;
        release inode of file (algorithm iput)
}
```

In UNIX systems, there is only one algorithm for exec, but the system call interface provides a total of six different variations of the exec function which differ only on how they handle input arguments. This means that only one system call (usually execve()) is actually required to be implemented. The other functions are stubs with perform necessary preparations and then eventually call execve. Short function descriptions of each version are provided below (Stevens pp. 207-209).

```
#include<unistd.h>
1. int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );
2. int execv(const char *pathname, char *const argv[]);

The following exec commands take an environment array (envp) in addition to the
other arguments. This can be used to define a custom environment for the process.
Otherwise, uses parent's environment.
3. int execle(const char *pathname, const char *arg0, ... /* (char *) 0 */ , char
 * const envp[]);
4. int execve(const char *pathname, char *const argv[], char *const envp[]);

Exec commands ending in a p use the PATH environ variable in order to find the
executable file. An error is thrown if the file is not in a directory specified by PATH.
5. int execlp(const char* filename, const char *arg0, ... /* (char *) 0 */ );
6. int execvp(const char *filename, char *const argv[]);

All return -1 on error and do not return when successful
```

Example #1: Exec Functions

```
#include<unistd.h> // Contains fork and exec
#include<sys/types.h> // Contains pid t datatype
#include<sys/wait.h> // Contains wait & waitpid
#include<errno.h> // Contains errno
#include<string.h> // Contains strerror
#include<iostream> // Contains cout & endl
int main(int argc, char * * argv)
{
        pid t pid = fork();
        if(pid == 0)
        {
                execl("/bin/ls", "/bin/ls", "–la", NULL);
        }
        else if(pid == –1)
        {
                cout << "[ERROR]: Fork failed: " << strerror(errno) << endl;
        }
        else
        {
                waitpid(pid, NULL, 0);
        }
        exit (0);
}
```

Example #2: Use of fork & exec

```
akirfman@karibot:~/git/CSCE_313_PI_Project/Episode_3$ g++ report_ex2.cpp
akirfman@karibot:~/git/CSCE_313_PI_Project/Episode_3$ ./a.out
total 44
drwxr-xr-x  2 akirfman akirfman  4096 Jun 11 09:01 .
drwxr-xr-x 15 akirfman akirfman  4096 Jun 11 08:06 ..
-rwxr-xr-x  1 akirfman akirfman  9984 Jun 11 09:01 a.out
-rw-r--r--  1 akirfman akirfman 13126 Jun 11 08:06 examples.cpp
-rw-r--r--  1 akirfman akirfman   618 Jun 11 08:06 Makefile
-rw-r--r--  1 akirfman akirfman   935 Jun 11 09:00 report_ex2.cpp
-rw-r--r--  1 akirfman akirfman     0 Jun 11 08:06 temp
```

Figure #3: Results of Example #2

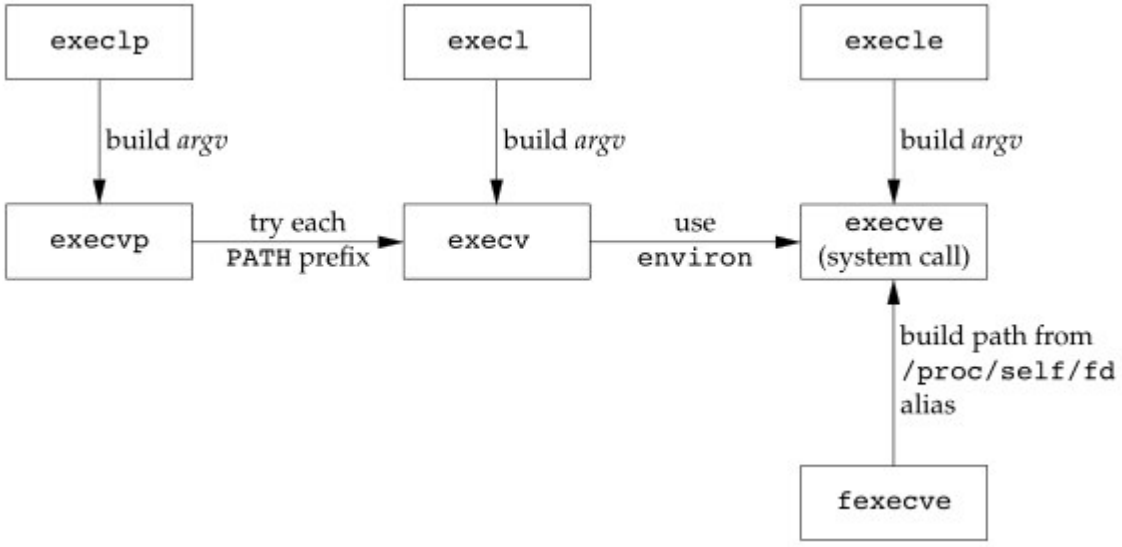

Figure #4: Relationship of exec functions

UNIX Pipes

The kernel serves three important roles. Firstly, it functions as a referee, managing resources between all entities on the system. Secondly, it functions as an illusionist, insulating entities from each other and ensuring security. Finally, it functions as a glue, providing a common software interface to all programs running on the system.

The operating system's behavior as an illusionist is critical to data integrity. It prevents separate processes from accessing data that they themselves do not own. However, there are a multitude of situations wherein processes must communicate with each other. Since, because of the kernel, processes cannot access each other's memory regions, a specific mechanism must be provided to facilitate communication. We will discuss this and other communication mechanisms in detail once we talk about Inter-Process Communication in detail in our class later in the semester.

Since the kernel works to insulate processes from each other, any communication between them, must be managed by the kernel. The kernel provides several mechanisms, known as

inter-process communication methods (For interested parties, named pipes and shared memory are the mechanisms not considered here).

The simplest method of inter-process communication is a UNIX pipe. Unix pipes are unidirectional data channels. Pipes are accessed through a set of file descriptors. In order to access them, one first must declare a integer array with two elements: `int pipe fds[2];` .

Afterwards, passing the array to the pipe system call (`pipe(pipe fds);`) causes the kernel to set up the channel.

After the channel is set up, the pipe has a read end and a write end. The read end is the 0th element in the array (`pipe fds[0];`), and the write end is the 1st element in the array (`pipe fds[1];`). Any process with access to the pipe can write data to the write end of the pipe and then read it back in the order that it was written from the read end of the pipe.

Read():

The read function is used to read *n* bytes of data from a file descriptor into a buffer. The function takes three arguments: the file descriptor to read from, a void* pointer to a buffer to store the data, and a size_t (basically just an unsigned integer) that denotes the number of bytes to try and read. The function should appear in code according to the following declaration:

```
#include <unistd.h>
ssize_t read(int fd, void * buf, size t count);
```

The function returns an ssize_t (basically a signed integer). Read will attempt to read up to 'count' bytes from the file descriptor denoted by 'fd' into the buffer whose starting address is 'buf'. Note that 'buf' must be greater than or equal to 'count' so that you have room to store all of the data that could possibly be read.

On success, the number of bytes read is returned. Read will try to read what data it can. If it reaches the end of the file stream before reading all 'count' bytes (i.e. it read all of the data that was currently available on 'fd', and it wasn't equal to 'count'), that is not an error. Read will just return the number that was read. If read returns 0, that usually means that it reached the end of file character immediately. When an error occurs, -1 is returned, and the variable errno will be set accordingly. For the different errno codes returnable by the read function, see read's man page.

Write():

The write function is used to write *n* bytes of data from a buffer to a file descriptor. It's essentially the inverse of the read function. The arguments to write() are the same as with read(), but their meaning is slightly different: 'fd' is the file descriptor to write to, 'buf' is where the data is currently stored, and 'count' is the number of bytes to write. Therefore, the function declaration is as follows.

```
ssize_t write(int fd, const void * buf, size t count);
```

Write returns the number of bytes that it returned. If the number of bytes written can be less than 'count' if, for example, there is insufficient space on the destination storage device or if a system call interrupted the write function.

As with read, write returns -1 and sets the errno variable in the event of an error. See the man page for write for documentation on the various values of errno and any other features not documented here.

UNIX man Pages:

One incredibly useful feature of UNIX operating systems that many new developers do not know about is the built in manual system. Using the command 'man', you can access information about most aspects of the operating system from general commands all the way to system call APIs.

The structure of man pages in UNIX are organized into sections by number follows (Wikimedia Foundation):

1. General Commands
2. System Calls
3. Library Functions (Specifically, the C standard library)
4. Special Files
5. File Formats
6. Games
7. Miscellanea
8. System Administration

You may (or may not, that's fine too) find the following manual pages useful when creating this assignment. Each one of these lines can be executed as a valid shell command to open a particular manual page. Note, the number indicates the manual section that that function resides.

- man 3 exec
- man 2 fork
- man 2 wait
- man 2 pipe
- man 2 read
- man 2 open
- man 2 close
- man 3 fopen
- man 3 fclose

Bibliography:

[1] Bovet, Daniel Pierce. Understanding the Linux Kernel: From I/O Ports to Process Management. U.S.A: O'Reilly, 2003. Print.

[2] Glass, Graham. Linux for Programmers and Users. Upper Saddle River, NJ: Pearson Prentice Hall, 2006. Print.

[3] Bach, Maurice J. The Design of the UNIX Operating System. Englewood Cliffs, NJ: Prentice-Hall, 1986. Print.

[4] Stevens, W. Richard. Advanced Programming in the UNIX Environment. Reading, MA: Addison-Wesley Pub., 1992. Print.

[5] "Man Page." Wikipedia. Wikimedia Foundation, n.d. Web. 18 May 2016.