OPERATING SYSTEMS – ASSIGNMENT 1

Introduction

Throughout this course we will be using a simple, UNIX like teaching operating system called xv6: https://pdos.csail.mit.edu/6.828/2016/xv6.html

The xv6 OS is simple enough to cover and understand within a few weeks yet it still contains the important concepts and organizational structure of UNIX. To run it, you will have to compile the source files and use the QEMU processor emulator (installed on all CS lab computers).

- xv6 was (and still is) developed as part of MIT's 6.828 Operating Systems Engineering course.
- You can find a lot of useful information and getting started tips there: https://pdos.csail.mit.edu/6.828/2016/overview.html
- xv6 has a very useful guide. It will greatly assist you throughout the course assignments: https://pdos.csail.mit.edu/6.828/2016/xv6/book-rev9.pdf

In this assignment we will extend xv6 to support the PATH environment variable, an implicit call to exit and understand some basic system calls such as fork, exec and pipe.

You can download xv6 sources form the GitHub: https://github.com/mit-pdos/xv6-public

Task 1: Warm up ("HelloXV6"):

This part of the assignment is aimed at getting you started. It includes a small change in xv6 shell. Note that in terms of writing code, the current xv6 implementation is limited: it does not support system calls you may use when writing on Linux and its standard library is quite limited.

Task 2: Support the PATH environment variable :

When a program is executed, the shell seeks the appropriate binary file in the current working directory and executes it. If the desired file does not exist in the working directory, an error message is printed.

A ***"PATH"*** is an environment variable which specifies the list of directories where commonly used executables reside. If, upon typing a command, the required file is not found in the current working directory, the **shell** attempts to execute the file from one of the directories specified by the **PATH**. An error message is printed only if the required file was not found in the working directory or any of the directories listed in the ***PATH***.

Your first task is to add support for the ***PATH*** environment variable. This environment variable should be controlled by the **shell** (i.e., by a user space program called **sh.c**). First you should add a global PATH variable to the shell which should contain a list of directories where the shell should search for executables (for the sake of simplicity the list of directories should be limited to 10 entries). Next, "**set *PATH***" command should be added to the shell. It will change the value of PATH variable. Each directory name listed should be delimited by a colon **(':')**.

For example, if we wanted to add the root directory and the bin directory to the PATH variable one could use following command:

set PATH *:/bin/:*

Every time set ***PATH*** command is called, the new paths will replace the existing values of ***PATH***. Finally, the **shell** must be aware of ***PATH*** environment variable when executing a program.

The first place to seek the binary of the executed command should be the current working directory, and only if the binary does not exist, the shell must search it in directories defined by the ***PATH***. The list of directories can be traversed in random order and must either execute the binary (if it is found in one of the directories) or print an error message in case the program is not found.

Note that the user can execute a program by providing to the shell absolute path (i.e., the path which has '/' as its first character) or relative path. The search for the binary should be executed only on **absolute paths**

Test your implementation by executing a binary which does not reside in current working directory but pointed by **PATH**. The tests must include commands which uses I/O redirection (i.e., file input/output and pipes).

Task 3 : Extend Functionality of Xv6:

Task 3.1 : Warm up before system call

In order to enrich the functionality of the xv6 you are required to add a user space program called ***tee***. This user space program receives a single argument or two arguments.

If it received **single arguments** a filename. It should read from standard input and write to both standard output and the file provided as argument. Consider following command executed by shell:

tee tmp.txt

It must read the standard input and write it to standard output and file tmp.txt both of them

If it received 2 arguments, it should read from filename and write to the second file that provided as second argument. Consider following command executed by shell:

tee tmp1.txt tmp.txt

It will read from ***tmp1.txt*** and write to ***tmp*.txt** only .

Task 3.2 : getpinfo system call

getpinfo is system call the print a list of currently process, it's displays information about the process

<Number of Row>	<Process ID>
0	28
1	23

Pay attention, when you add/change a system call, you must update both kernel sources and user space programs sources.

Task 4: Wait and exit system calls

In most of the operation systems the termination of a process is performed by calling an exit system call. The exit system call receives a single argument called **"status",** which can be collected by a parent process using the wait system call , If a process ends without an explicit call to exit, an implicit call to exit is performed with the status obtained from the return value of the main function.

This is not the case in xv6 – the exit system call does not receive a status and the wait system call does not return it, In addition, no implicit call to exit is performed.

The following task will modify xv6 in order to support this common behaviour. In this part you are required to extend the current kernel functionality so as to maintain an **exit status** of a process and to endow the kernel with an ability to make an **implicit system call exit** when the process is done.

First, you must **add a field** to the process control block PCB ***(see proc.h – the proc structure)*** in order to save an exit status of the terminated process. Next, you have to change all system calls affected by this change ***(i.e., exit and wait).*** Finally, you must endow the current implementation of the exec system call with the ability to make an implicit system call exit at the time the process is exiting its main function without explicitly calling the exit system call.

4.1. Updating the exit system call:

Change the exit system call signature to void exit(int status). The exit system call must act as previously defined (i.e., terminate the current process) but it must also store the exit status of the terminated process in the proc structure.

- In order to make the changes in the exit system call you must update the following files: user.h, defs.h, sysproc.c, proc.c and all the user space programs that use the exit system call.

- Note, you must change all the previously existing user space programs so that each call to exit will be called with a status equal to 0 (otherwise they will fail to compile).

4.2. Updating the wait system call:

Update the wait system call signature to int wait(int *status). The wait system call must block the current process execution until any of its child processes is terminated (if any exists) and return the terminated child exit status through the status argument.

- The system call must return the process id of the child that was terminated or -1 if no child exists (or unexpected error occurred).

- Note that the wait system call can receive NULL as an argument. In this case the child's exit status must be discarded.

- Note that like in **task 4.1** (exit system call) you must change all the previously existing user space programs so that each call to wait will be called with a status equal to 0 (NULL) (otherwise they will fail to compile).

4.3. Updating the wait system call:

In the current implementation of xv6 each user program should explicitly perform an exit system call in order to correctly terminate its execution. If no such call is made, the process crashes. In this task you must change the exec system call **(see exec.c)** so that if a user program **exit** the main function, the **exit** system call will be implicitly performed, and the return value of the main function must be the status argument for the implicit exit system call. The simplest way to perform this task is to inject the exit code to the process memory (in order to know its address) and push its address to the stack so that the **ret** command of the main function will jump to this code.

In order to better understand how process termination works (and eventually change it) implement a user space program that does not perform an explicit call to exit upon its termination (you should add the user space program to the UPROGS variable at Makefile). If such a user space program is executed, it will "crash" with the following error message:

pid 3 test: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff—kill proc

The reason for such a message comes from the fact that the process (CPU) tried to perform a restricted operation – jumps to the return address of the main function, which is invalid and therefore failed. As a result the **kernel "kills"** this process.

Your task is to find the code **(in exec.c**) that initializes the process's **user space stack**. First, inject a code that calls exit directly into the user space stack. Next, the main function arguments must be placed on the stack (as in the unmodified version) and finally, change the return address of the main function to point to the injected code that calls exit.