Project c2: Shell (SHELL)

Project's summary

The purpose of this project is to support running multiple processes at once from actual compiled programs stored on disk. These programs will be loaded into OS161 and executed in user mode, under the control of your kernel and the command shell in `bin/sh` (menu command: `p bin/sh`). The project is highly based on the availability of the `execv` and `dup2` system calls. The project can be limited to the EMUFS emulated file system

Required Background

Lab 2 to Lab 5 and basic knowledge of process management and file system are needed (see lessons os161-overview, os161-memory, os161-userprocess).

Working Environment

Your current OS161 system has minimal support for running executables -- nothing that could be considered a true process. Lab 2 to Lab 5 have provided limited support for process memory management, exit and file system.

The key files that are responsible for the loading and running of user-level programs are loadelf.c, runprogram.c, and uio.c, Understanding these files is the key to getting started with the implementation of multiprogramming.

kern/syscall/loadelf.c: This file contains the functions responsible for loading an ELF executable from the filesystem and into virtual memory space. (ELF is the name of the executable format produced by os161-gcc.) Of course, at this point this virtual memory space does not provide what is normally meant by virtual memory -- although there is translation between the addresses that executables "believe" they are using and physical addresses, there is no mechanism for providing more memory than exists physically.

kern/syscall/runprogram.c: This file contains only one function, runprogram(), which is responsible for running a program from the kernel menu. It is a good base for writing the execv() system call, but only a base -- when writing your design doc, you should determine what more is required for execv() that runprogram() does not concern itself with. Additionally, once you have designed your process system, runprogram() should be altered to start processes properly within this framework; for example, a program started by runprogram() should have the standard file descriptors available while it's running.

kern/lib/uio.c: This file contains functions for moving data between kernel and user space. Knowing when and how to cross this boundary is critical to properly implementing userlevel programs, so this is a good file to read very carefully. You should also examine the code in kern/vm/copyinout.c.

kern/arch/mips: traps and syscalls

Exceptions are the key to operating systems; they are the mechanism that enables the OS to regain control of execution and therefore do its job. You can think of exceptions as the interface between the processor and the operating system. When the OS boots, it installs an "exception handler" (carefully crafted assembly code) at a specific address in memory. When the processor raises an exception, it invokes this exception handler, which sets up a "trap frame" and calls into the operating system. Since "exception" is such an overloaded term in computer science, operating system lingo for an exception is a "trap". Interrupts are exceptions, and more significantly for this assignment, so are system calls. Specifically, syscall/syscall.c handles traps that happen to be syscalls. Understanding at least the C code in this directory is key to being a real operating systems junkie, so we highly recommend reading through it carefully.

locore/trap.c: mips_trap() is the key function for returning control to the operating system. This is the C function that gets called by the assembly exception handler. enter_new_process() is the key function for returning control to user programs. kill_curthread() is the function for handling broken user programs; when the processor is in usermode and hits something it can't handle (say, a bad instruction), it raises an exception. There's no way to recover from this, so the OS needs to kill off the process. Part of this assignment will be to write a useful version of this function.

syscall/syscall.c: syscall() is the function that delegates the actual work of a system call to the kernel function that implements it. Notice that reboot() is the only case currently handled. You will also find a function, enter_forked_process(), which is a stub where you will place your code to implement the fork()system call. It should get called from sys_fork().

user/lib/crt0/mips: This is the user program startup code. There's only one file in here, mips-crt0.S, which contains the MIPS assembly code that receives control first when a user-level program is started. It calls the user program's main(). This is the code that your execv() implementation will be interfacing to, so be sure to check what values it expects to appear in what registers and so forth.

user/lib/libc: This is the user-level C library. There's obviously a lot of code here. We don't expect you to read it all, although it may be instructive in the long run to do so. Job interviewers have an uncanny habit of asking people to implement standard C library functions on the whiteboard. For present purposes you need only look at the code that implements the user-level side of system calls, which we detail below.

user/lib/libc/unix/errno.c: This is where the global variable errno is defined.

user/lib/libc/arch/mips/syscalls-mips.S: This file contains the machine-dependent code necessary for implementing the user-level side of MIPS system calls.

syscalls.S: This file is created from syscalls-mips.S at compile time and is the actual file assembled into the C library. The actual names of the system calls are placed in this file using a script called syscalls/gensyscalls.sh that reads them from the kernel's header files. This avoids having to make a second list of the system calls. In a real system, typically each system call stub is placed in its own source file, to allow selectively linking them in. OS/161 puts them all together to simplify the makefiles.

Problem Definition

System calls and exceptions

Implement system calls (for file and process management) and exception handling. The full range of already defined (not implemented) system calls is listed in kern/include/kern/syscall.h. For this project, you should complete and/or implement:

- open, read, write, lseek, close, dup2, chdir, getcwd
- getpid
- fork, execv, waitpid, _exit

It's crucial that your syscalls handle all error conditions gracefully (i.e., without crashing OS/161.) You should consult the OS/161 man pages included in the distribution and understand fully the system calls that you must implement. You must return the error codes as described in the man pages.

Additionally, your syscalls must return the correct value (in case of success) or error code (in case of failure) as specified in the man pages. Some of the grading scripts rely on the return of appropriate error codes; adherence to the guidelines is as important as the correctness of the implementation.

The file user/include/unistd.h contains the user-level interface definition of the system calls that you will be writing for OS/161 (including ones you will implement in later assignments). This interface is different from that of the kernel functions that you will define to implement these calls. You need to design this interface and put it in kern/include/syscall.h. The integer codes for the calls are defined in kern/include/kern/syscall.h.

You need to think about a variety of issues associated with implementing system calls. Perhaps the most obvious one is: can two different user-level processes find themselves running a system call at the same time? Be sure to argue for or against this, and explain your final decision in the design document.

open(), read(), write(), lseek(), close(), dup2(), chdir(), and __getcwd()

For any given process, the first file descriptors (0, 1, and 2) are considered to be standard input (stdin), standard output (stdout), and standard error (stderr). These file descriptors

should start out attached to the console device ("con:"), but your implementation must allow programs to use dup2() to change them to point elsewhere.

Although these system calls may seem to be tied to the filesystem, in fact, these system calls are really about manipulation of file descriptors, or process-specific filesystem state. A large part of this assignment is designing and implementing a system to track this state. Some of this information (such as the current working directory) is specific only to the process, but others (such as file offset) is specific to the process and file descriptor. Don't rush this design. Think carefully about the state you need to maintain, how to organize it, and when and how it has to change.

Note that there is a system call __getcwd() and then a library routine getcwd(). Once you've written the system call, the library routine should function correctly.

getpid()

A pid, or process ID, is a unique number that identifies a process. The implementation of getpid() is not terribly challenging, but pid allocation and reclamation are the important concepts that you must implement. It is not OK for your system to crash because over the lifetime of its execution you've used up all the pids. Design your pid system; implement all the tasks associated with pid maintenance, and only then implement getpid().

fork(), execv(), waitpid(), _exit()

These system calls are probably the most difficult part of the assignment, but also the most rewarding. They enable multiprogramming and make OS/161 a much more useful entity.

fork() is the mechanism for creating new processes. It should make a copy of the invoking process and make sure that the parent and child processes each observe the correct return value (that is, 0 for the child and the newly created pid for the parent). You will want to think carefully through the design of fork() and consider it together with execv() to make sure that each system call is performing the correct functionality.

execv(), although "only" a system call, is really the heart of this assignment. It is responsible for taking newly created processes and make them execute something useful (i.e., something different than what the parent is executing). Essentially, it must replace the existing address space with a brand new one for the new executable (created by calling as_create in the current dumbvm system) and then run it. While this is similar to starting a process straight out of the kernel (as runprogram() does), it's not quite that simple. Remember that this call is coming out of userspace, into the kernel, and then returning back to userspace. You must manage the memory that travels across these boundaries very carefully. (Also, notice that runprogram() doesn't take an argument vector -- but this must of course be handled correctly in execv()).

A note on errors and error handling of system calls:

The man pages in the OS/161 distribution contain a description of the error return values that you must return. If there are conditions that can happen that are not listed in the man page, return the most appropriate error code from kern/include/kern/errno.h. If none seem particularly appropriate, consider adding a new one. If you're adding an error code for a condition for which Unix has a standard error code symbol, use the same symbol if possible. If not, feel free to make up your own, but note that error codes should always begin with E, should not be EOF, etc. Consult Unix man pages to learn about Unix error codes; on Linux systems man errno will do the trick.

Note that if you add an error code to src/kern/include/kern/errno.h, you need to add a corresponding error message to the filesrc/kern/include/kern/errmsg.h.

kill_curthread()

Feel free to write kill_curthread() in as simple a manner as possible. Just keep in mind that essentially nothing about the current thread's userspace state can be trusted if it has suffered a fatal exception -- it must be taken off the processor in as judicious a manner as possible, but without returning execution to the user level.

Testing using the shell

In user/bin/sh you will find a simple shell that will allow you to test your new system call interfaces. When executed, the shell prints a prompt, and allows you to type simple commands to run other programs. Each command and its argument list (an array of character pointers) is passed to the execv() system call, after calling fork() to get a new thread for its execution. The shell also allows you to run a job in the background using &. You can exit the shell by typing "exit".

Under OS/161, once you have the system calls for this assignment, you should be able to use the shell to execute the following user programs from the bin directory: cat, cp, false, pwd, and true. You will also find several of the programs in the testbin directory helpful.