CSE 4300 Assignment 3
Assigned: 4/5/2019
Due: 4/19/2019, 11:59 p.m.
Total points: 120

In this project, you will learn – (1) How to change OS/161 kernel and compile, (2) How to add system calls to operating system kernel, and (3) How to make the system level functions available to the user program.

This Project requires you to implement new system calls in OS/161. You will also develop user programs (one for each new system call or just one program that tests all new system calls) to test the new system calls. You can use the "ASST0" code base to test your programs.

(10 points) Part A
This part is to get yourself familiarized with the OS/161 source code. Your job is to identify and revise the proper module(s) to customize your OS/161 greeting message that is displayed on the menu screen. Print your name during the boot process.

Hints:
1. Look inside os161-1.11/kern/main directory
2. You will need to compile kernel and execute ASST0. You should already know how to compile kernel (assuming you have already installed OS/161).

(30 point) Part B

Implement a simplified system call *void _exit(int exitCode)*

In OS161, you can end user programs by calling the "_exit( )" system call. **Without an implementation of _exit( ), the threads created to handle user programs will hang around forever, executing an infinite loop in the user space and taking up a lot of CPU time.**

(HINT: You should find an existing function that is almost an ideal handler for the _exit( ) system call – then modify that function to receive the exitCode as a parameter (you will also have to modify all the other uses of that function in order to pass the exitCode).

In this part, you have to print out the exitCode that is passed.

Once you implement the _exit() system call, you will find that useful for the next part to prevent infinite loop.

(30 point) Part C

You need to implement a system call "*int printint(int c)*"
This system call should accept an integer and print it using the internal kprintf( ) function. The return value should be 0 if the integer is a multiple of 2 or 1 otherwise.

(30 point) Part D

You need to implement a system call *int reversestring(const char *str, int len)*
This system call should accept a string and length of the string as input and print the reverse of the string using the internal kprintf( ) function. The return value should be 0 if the length of the string is multiple of 2 or 1 otherwise.

Test Functions for the System Calls

Write **user level programs** to test Part B, C, and D respectively. You can write one program to test all three or write individual program as well.

(10 point) Part E. Create a test program, named ***testprint***, to test the **printint( )** system call developed in Part C. Pass 5 integers (-1, 0, 1, 2, 3) to printint in a for loop and print the return values use printint as well.

(10 point) Part F. Create another test program, named ***testreverse*** to use the **reversestring**( ) system call developed in Part D. Pass string "HelloWorld" as argument and print the return value.

Please note that you will need to use the _exit system call in the above programs to avoid infinite loop.

Note that the system calls given here are user-level functions, i.e. these are the functions that user programs will call to invoke the system calls. You will need to use different names for the handlers of these functions in the kernel (e.g., sys_reboot( ) is called to handle the reboot( ) system call.)

As a reference, the Appendix contains a list of OS/161 system calls, along with some literature to help make understanding these systems calls a little easier. You are highly encouraged to study and discuss the Appendix among yourselves while working on this assignment.

Source code: (1/2)
1. Include any source files you modified or created. If possible, provide a *Patch* file that include all modifications for each part (+10).
Report: (1/2)
1. Add a cover page describing what parts are completed.
2. Include the list of the files you changed or added for each part (including test programs). Explain the reasons/purpose behind your changes briefly.
3. Include the sample output of your test programs (if applicable).

You may include above items in a word file (report) and a zip file (source files or patch files) and upload them to HuskyCT.

Important Points:

- The system calls given here are user-level function names (e.g., printchar), that the user programs will call to invoke these system calls. You will need to use different names for the handlers of these functions in the kernel (e.g., sys_printchar( )), which will be called to handle the printchar( ) system call.
- You should look at the following files:
  - unistd.h
  - syscall.h
  - syscall.c
  - callno.h
  - conf.kern

  - **Caution 1:** One or more files in this list (e.g., unistd.h) may have multiple copies in separate directories. You need to change the right file.

  - **Caution 2:** Read the comments in syscall.c to figure out how to pass multiple parameters.

- You may add a new file (e.g., simple_syscalls.c) that will implement your newly added system calls.
- After you finish adding each system call, you will need to compile the new kernel and user programs to see the effect of your change. Please refer to the first assignment to see how to compile a kernel.

Writing, compiling and running Test Programs

- Your test programs need to be saved under **os161-1.11/testbin** directory. Create a directory called /testA and save your test program in the directory. Other directories under testbin include other test programs.
- You will need to modify the makefile that you can copy from another directory within testbin(e.g., /palin).
- You will also need to edit the makefile under testbin directory. **Hint: look at the content of the makefile and see how other test programs are included. You will need to add one line per test program.**
- To compile your user program, just type "make" from /os161-1.11
- You can run your user program as follows:
  - First, execute the following from command prompt:
    - >sys161 kernel-ASST0
  - Next, execute the following from your new kernel command prompt
    - p testbin/testA (assuming testA.c is your test program name for Part A and it is saved under testbin/testA directory)

You will need to modify the code in **syscall.c** to detect your new system calls and dispatch appropriate

system call handlers. Although these system calls are simple enough to implement fully within syscall.c, it is suggested that you place the handlers in a separate function in a file in the **/userprog** subdirectory. You should name this file simple_syscalls.c. You will also need to add an entry for this new file in **conf.kern** file, and reconfigure your kernel so that it is included in the build. Because you will call functions from syscall.c, which are defined elsewhere, you should add prototypes for these functions to the **syscall.h** header file, similar to what is done for the sys_reboot( ) function.

Understanding System Calls

In this document, we walk through the functions and steps that are involved in (1) starting a user-level program, (2) handling system calls in the OS, and (3) invoking system calls at the user-level. You should read this document together with the OS/161 source code files that it refers to.

User-Level Programs

The System/161 simulator can run normal programs compiled from C. These programs are compiled with the cross-compiler cs161-gcc. This compiler runs on the host machine and produces MIPS executables. It is the same compiler used to compile your OS/161 kernel. To create new user programs, you will need to edit the Makefile in bin, sbin, or testbin (depending on where you put your programs), and then create a directory similar to those that already exist. Use an existing program and its Makefile as a template. You should create new user-level test programs that use the system calls you are adding.

Getting to User-Mode (i.e. Starting a User-Level Program)

Examine menu.c to see how the "p" menu command is handled. This command allows us to load and execute a single user-level program. We will describe this in great detail, most of which is not of critical importance for this Project. But you will need to understand this eventually, however, and the sooner the better.

The menu( ) function loops forever, printing the menu prompt, getting a string from the console, and calling menu_execute( ) to handle the input. Looking at menu_execute( ), we see that it separates the input into individual commands (indicated by a semi-colon) and then calls cmd_dispatch( ) for each command. In cmd_dispatch( ), the name of the command is separated from its arguments, and the name is looked up in the cmdtable data structure. This table stores the string name of each menu command and a function pointer to the function that should handle that command.

Looking at the cmdtable declaration, we find that the "p" command is handled by calling the cmd_prog( ) function. This function simply strips off the "p" part and passes the rest of the input to the common_prog( ) function. Looking at common_prog( ) we see that it calls thread_fork( ), creating a new thread to run the specified user-level program. Following this call to thread_fork( ), our system has 2 threads: the initial boot thread that runs the menu( ) loop, and the new thread that runs the requested user-level program.

You can look thread.c to see what thread_fork( ) does, but the important thing right now is that the fourth argument to thread_fork( ) specifies the function that the new thread should start executing (in this case, cmd_progthread( )), and the second and third arguments to thread_fork( ) are the arguments to pass to that function (in this case, the name of the program to load). The cmd_progthread( ) function calls runprogram( ), passing it the name of the program to load and execute. Note that if runprogram( ) is successful, the thread will continue with the execution of the user-level program and will never return to cmd_progthread( ).

Now let's consider how runprogram( ) operates. The source files that are responsible for the loading and running of user-level programs include loadelf.c, uio.c, and runprogram.c.

runprogram.c: This file contains only one function, runprogram( ), which is responsible for running a program from the kernel menu. It uses the virtual file system operations to open the file containing the program we want to load, creates an address space for the thread, and loads the program into that address space, using the load_elf( ) function. If loading the program is successful, runprogram( ) then sets up the user stack area in the address space, and switches to user mode and start running the user program.

loadelf.c: This file contains the functions responsible for loading an ELF executable from the file system into the virtual memory space. (ELF is the name of the executable format produced by cs161-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 the executables "believe" they are using and physical addresses, there is no mechanism for providing more memory than exists physically.

uio.c: This file contains functions for moving data between kernel and user spaces. Knowing when and how to cross this boundary is critical to properly implementing user level programs, so this is a good file to carefully study.

Once a user program starts running, it requests service from the OS by way of system calls. We now take a look at these.

Getting to System Mode: 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 OS boots, it installs an "exception handler" (essentially carefully crafted assembly code) at a specific address in memory. When the processor raises an exception, it invokes this, which sets up a "trap frame" and calls into the OS. Initializing the trap frame and returning from an exception is all done in assembly code, and is found in exception.S.

You need not be able to read MIPS assembly code, but the comments in this .S file are reasonably good. You can see how all the registers are saved ("sw" == "store word"), followed by a "jal" call to the mips_trap function ("jal" == "jump and link"). On return from mips_trap, all of the saved state is restored ("lw" == "load word") and execution resumes at the point where the exception occurred.

Looking at struct trapframe in trapframe.h, we can see that a trap frame includes space to save all the processor registers that the user program might have been using, as well as some additional state that identifies the cause of the exception (the "tf_cause" field), and the

instruction that was being executed when the exception occurred (the ”tf_epc” field, where epc == ”Exception PC” == contents of the program counter register when the exception occurred). Note that since ”exception” is such an overloaded term in computer science, OS lingo for an exception is a ”trap” – when the OS traps execution. Interrupts are exceptions, and more significantly for this Project, so are system calls. Specifically, syscall.c handles traps that happen to be system calls.

mips_trap( ) in trap.c is the key function for returning control to the operating system. This is the C function that gets called by the assembly language exception handler. It includes code to determine what type of exception occurred, and to dispatch an appropriate handler. If the exception code is EX_SYS, then mips_syscall( ) is called to handle the system call, passing it the trap frame. mips_syscall( ) in syscall.c is the function that delegates the actual work of a system call to the kernel function that implements it. Read the comments at the top of this file (second block of comments) carefully! In mips_syscall( ), we begin by extracting the system call number from the trapframe's tf_v0 field. This means that the system call number was stored in register v0 prior to executing the syscall( ) instruction. Then we simply switch on the system call number, with a separate ”case” to handle each possible system call. Notice that reboot() is the only case currently handled. You will need to add your staffs here to handle your newly added system calls.

Following the switch statement, we prepare to return from the system call. The user-level side of the system call expects to find the result of the system call in register v0, with register a3 indicating whether or not an error occurred. We accomplish this by setting the appropriate fields in the trap frame, which will be loaded into the machine registers before returning control to the user program. Finally, we have to advance the program counter to the next instruction, so that the mips_syscall( ) instruction will not be repeated. This is done by incrementing the tf_epc field of the trap frame.

User Side of a System Call

That is what happens on the OS side when a system call occurs. Now let’s look at the user-level interface to system calls. Most of this is encapsulated in C library functions. /lib/libc/: This is where the user-level C library is. 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.

errno.c: This is where the global variable “errno” is defined. Note that this variable is a global within a user-level C program. Therefore, you **cannot** set errno for a user-level program by setting a variable named ”errno” in the kernel.

syscalls-mips.S: This file contains the machine-dependent code necessary for implementing the user-level side of MIPS system calls. It consists of a C pre-processor (i.e. #define) that declares the body of each system call. The body of each system call is identical, except that a different system call number “num” is used. This body simply loads the system call number (as defined in syscall.h) into register v0 and then jumps to the code common to all system calls. (Compare this with what’s on the OS side, where the system call number is extracted from the tf_v0 field of the trap frame).

You may notice that it looks like the code jumps to the __syscall label *before* setting the system call number. This is just a peculiarity of the MIPS architecture – the instruction immediately following a branch is called a ”delay slot”. This means that an instruction can be scheduled and executed *during* a branch. Such instructions appear immediately after the branch instruction itself. Therefore, the ”addiu v0, $0, SYS_##sym” instruction happens before we get to the common syscall code at the label __syscall.

Now, let’s take a look at the assembly code at the __syscall label. The common system call code begins with the syscall instruction, which causes a ”system call exception”, and transfers control to the OS as outlined above. The tf_epc field in the trap frame (on the OS side) points to this instruction, and we finish our system call handler by setting tf_epc to the next instruction. Thus, on return from the system call, we execute the beq instruction. This instruction tests if the system call failed or succeeded (recall that the OS side sets tf_a3 to 1 on error, and 0 on success). If an error occurred, we take the error code from the v0 register and store it into the global variable errno, and set the return value of the system call (the v0 register) to -1. If no error occurred, then the OS puts the result of the system call into tf_v0, and we can just return.

syscalls.S: This file is created from syscalls-mips.S at compile time, and is the actual file assembled into the C library. Adding new entries to syscall.h automatically causes new user-level system call procedures to be defined when you rebuild the user-level code. Every “SYSCALL(sym, num)” macro statement in syscalls.S is expanded by the C preprocessor into a declaration of the appropriate system call function.

../os161-1.11/include/unistd.h: This file contains the user-level interface definition of the system calls for OS/161 (including the ones you will implement). The only thing you need to do to complete the user-level system call interface is to declare prototypes for your new system calls in unistd.h. Everything else on the user side happens automatically when you rebuild the kernel after updating syscall.h. Note that the user-level interface defined in unistd.h differs from that of the kernel functions you will define to implement the calls. You will need to declare these kernel functions in syscall.h (note the name will be different from your user side definition).

APPENDIX

A List (Not Exhaustive) of OS/161 Systems Calls

errno - error code reporting
_exit - terminate process
chdir - change current directory
close - close file
dup2 - clone file handles
execv - execute a program
fork - copy the current process
fstat - get file state information
fsync - flush file system data for a specific file to disk
ftruncate - set size of a file
__getcwd - get name of current working directory (backend)
getdirentry - read filename from directory
getpid - get process id
ioctl - miscellaneous device I/O operations
link - create hard link to a file
lseek - change current position in file
lstat - get file state information
mkdir - create directory
open - open a file
pipe - create pipe object
read - read data from file
readlink - fetch symbolic link contents
reboot - reboot or halt system
remove - delete (unlink) a file
rename - rename or move a file
rmdir - remove directory
sbrk - set process break (allocate memory)
stat - get file state information
symlink - create symbolic link
sync - flush file system data to disk
__time - get time of day
waitpid - wait for a process to exit
write - write data to file