CS 471 Operating Systems
Fall 2020

Programming Assignment 2 (PA2):
Due date: Friday, December 4, 11:59 pm

Table of contents

1. Introduction

Your current OS/161 system has minimal support for running executables - nothing that could be considered a true process. This project involves the transformation of OS/161 into a true multi-tasking operating system. The project is left open--ended deliberately; there are several design options that you can consider and commit to. After this assignment, your OS/161 kernel will be capable of running multiple processes (as long as those processes use the system calls you implement) at once from actual compiled programs stored in your account. These programs will be loaded into OS/161 and executed in user mode by System/161; this will occur under the control of your kernel. In other words, in this project, you will implement some of the system calls we discussed in the beginning of the semester, including `fork, exec`, and `waitpid`.

Before starting, you will need to complete a few steps (as in OS/161 Asst 1) to “tag your Git repository”, “configure OS/161 for ASST 2”, and “build for ASST 2”. Please see the previous assignment’s (OS 161/Asst 1) hand out for instructions on how to do that. When you run your kernel configured for this assignment, you will again see a menu with various options (including test options).

As a first step, you must implement the interface between user-mode programs ("userland") and the kernel. As usual, we provide part of the code you will need. Your job is to design and build the missing pieces. You will also be implementing the subsystem that keeps track of the multiple tasks you will have in the future. You must decide what data structures you will need to hold the data pertinent to a "process" (hint: look at kernel include files of your favorite operating system for suggestions, specifically the `proc` structure.)

In order to do that you will have to read and understand the parts of the system that are written for you. This is an essential component of this assignment. The existing code can run one user-level C program at a time as long as it doesn't want to do anything but shut the system down. We have provided sample user programs that do this (`reboot, halt, poweroff`), as well as others that make use of features you will be adding in this assignment.

So far, all the code you have written for OS/161 has only been run within, and only been used by, the operating system kernel. In a real operating system, the kernel's main function is to provide support for user--level programs. Most such support is accessed via "system calls." The system gives you one system call, `reboot()`, which is implemented in the function `sys_reboot()` in `main.c`. In GDB, if you put a breakpoint on `sys_reboot` and run the `"reboot "` program, you can use `"backtrace"` to see how it got there.

User--level programs

The System/161 simulator can run normal programs compiled from C. The programs are compiled with a cross---compiler, `cs161-gcc`. This compiler runs on the host machine and produces MIPS executables; it is the same compiler used to compile the 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.

Design

Please note that your **design documents** become an important part of the work you submit in this project. The design documents should clearly reflect the development of your solution. They should not merely explain what you programmed.

Note that it can often be hard to write (or talk) about new software design - you are facing problems that you have not seen before, and therefore even finding terminology to describe your ideas can be difficult. There is no magic solution to this problem; but it gets easier with practice. Now is a really good time to work closely with your partner. As you design this assignment, you may have to ”invent” terminology and notation - this is fine (just be sure to explain it to your TA in your design document). If you do this, by the time you have completed your design, you will find that you have the ability to efficiently discuss problems that you have never seen before.

To help you get started, we have provided the following questions as a guide for reading through the code. We recommend that you divide up the code and have each partner answer questions for different modules. After reading the code and answering questions, get together and exchange summations of the code you each reviewed. Once you have done this, you should be ready to discuss strategy for designing your code for this assignment. As always, make sure to comply with the GMU and CS honor codes – the only person you can cooperate with is your partner (if you have one).

2. Code walk-through (20 points)

Include the answers to the code walk-through questions as the first part of your design document.

`kern/userprog`: This directory contains the files that are responsible for the loading and running of user-level programs. Currently, the only files in the directory are `loadelf.c`, `runprogram.c`, and `uio.c`, although you may want to add more of your own during this assignment. Understanding these files is the key to getting started with the implementation of multiprogramming. Note that to answer some of the questions, you will have to look in files outside this directory.

`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 `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 executables "believe" they are using and physical addresses, there is no mechanism for providing more memory than exists physically.

`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.

`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 user-level programs, so this is a good file to read very carefully. You should also examine the code in `lib/copyinout.c`

Questions

1. What are the ELF magic numbers?
2. What is the difference between UIO_USERISPACE and UIO_USERSPACE? When should one use UIO_SYSSPACE instead?
3. Why can the `struct uio` that is used to read in a segment be allocated on the stack in `load_segment()` (i.e., where does the memory read actually go)?
4. In `runprogram()`, why is it important to call `vfs_close()` before switching to usermode?
5. What function forces the processor to switch into usermode? Is this function machine dependent?
6. In what file are `copyin`, `copyout` and `memmove` are defined? Why can't `copyin` and `copyout` be implemented simply as `memmove`?
7. What is the purpose of `userptr_t`? Explain briefly.

kern/arch/mips/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, 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" -- when the OS traps execution. In OS 161 terminology, interrupts are exceptions, and more significantly for this assignment, so are system calls. (The terminology difference between OS/161 and many OS textbooks are unfortunate; but it is not uncommon at all – recall our class discussion about “interrupts” and “exceptions”). Specifically, `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 programmer, so we highly recommend reading through it carefully.

`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. `md_usermode()` 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.c: mips_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, `md_forkentry()`, which is a stub where you will place your code to implement the `fork()` system call. It should get called from `mips_syscall()`.

Questions

8. What is the numerical value of the exception code for a MIPS system call?
9. Why does `mips_trap()` set `curspl` to `SPL_HIGH` "manually", instead of using `splhigh()`?
10. How many bytes is an instruction in MIPS? (Answer this by reading `mips_syscall()` carefully, not by looking somewhere else.)
11. Why do you "probably want to change" the implementation of `kill_curthread()`?
12. What would be required to implement a system call that took more than 4 arguments?

`lib/crt0:` 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.
`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. 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.

`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 `callno-parse.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.

Questions

13. What is the purpose of the `SYSCALL` macro?
14. What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)

3. Design (20 points) and Implementation (60 points) - (80 points total)

Download necessary files

As in the previous ASST1, you need to download two files from Blackboard and put them in their appropriate directory. You need to update your source code with these two files before you begin your assignment:

- `callno.h` in 'os161-1.11/kern/include/kern/'
- `unistd.h` in 'os161-1.11/include/'

After replacing the files accurately, tag your repository as `asst2-begin`, if you have not already done so.

System calls and exceptions

Implement system calls and exception handling. The full range of system calls that we think you might want over the course of the semester is listed in kern `/include/kern/callno.h`. For this assignment you should implement:

- getpid
- getppid
- 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 (in `~os161/os161-1.11/man/syscall` directory) and understand fully the system calls that you must implement. **You must return the error codes as described in those 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 `include/unistd.h` contains the user-level interface definition of the system calls that you will be writing for OS/161 (including ones that are relevant for another potential assignment). 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`. As you may have discovered in OS 161/Asst0, the integer codes for the calls are defined in `kern/include/kern/callno.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 (or user-level threads, if you choose to implement them) 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.

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:"`).

An important design consideration is to decide where to put the code for these system calls. A good place would be inside the `kern/userprog` directory. For the 6 system calls in this assignment, create 6 files named `getpid.c`, `getppid.c`, `fork.c`, `execv.c`, `waitpid.c`, and `exit.c`, inside the `kern/userprog` directory. Make sure to add the file names to your `kern/conf/conf.kern` file in an appropriate place, as you have done in Assignment-0 for the hello.c file. By convention, the names of the systems call functions will be as follows: `sys_getpid, sys_getppid, sys_fork, sys_execv, sys_waitpid, sys_exit`. These names must also be added as declarations in the `kern/include/syscall.h` file.

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()`.

getppid()

This system call works exactly like `getpid()`, except that it returns the *parent* process' pid. Note that, the `getppid()` system call is not included in the OS/161 man pages, and you can assume the following spec:

`getppid()` will return the process ID of the parent of the calling process. This will be either the ID of the process that created this process using `fork()` (if the parent hasn't terminated yet), or -1 (if the parent has already terminated).

fork(), execv(), waitpid(), _exit()
These system calls are probably the most challenging 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()`).

Although it may seem simple at first, `waitpid()` requires a fair bit of design. Read the specification in the OS 161 `waitpid()` man page carefully to understand the semantics, and consider these semantics from the ground up in your design. You may also wish to consult the general UNIX man page, though keep in mind that you are not required to implement all the things UNIX `waitpid()` supports, nor is the UNIX parent/child model of waiting the only valid or viable possibility.

The implementation of `_exit()` is intimately connected to the implementation of `waitpid()`. They are essentially two halves of the same mechanism. Most of the time, the code for `_exit()` will be simple and the code for `waitpid()` more complicated, but it's perfectly viable to design it the other way around as well. If you find both are becoming extremely complicated, it may be a sign that you should rethink your design.

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/errno.h`. If none seems 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 `kern/include/kern/errno.h`, you need to add a corresponding error message to the file `lib/libc/strerror.c`

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.

Design Considerations

Here are some additional questions and thoughts to aid in writing your design document. They are not, by any means, meant to be a comprehensive list of all the issues you will want to consider. You do not need to explicitly answer or discuss these questions in your design document, but you should at least think about them.

Your system must allow user programs to receive arguments from the command line. In addition, it must also work with the "p" menu option (e.g., it must work for `p testbin/add 2 5`.) To accomplish that, you need to edit the file `menu.c`. See "Testing Your System Call Implementation" section below as well.

For instance, you should be able to run the following program:

```
char *filename = "/testbin/add";
char *args[4];
pid_t pid;
args[0] = "add";
args[1] = "3";
args[2] = "4";
args[3] = NULL;
pid = fork();
if (pid == 0) execv(filename, argv);
```

which will load the executable file `testbin/add`, install it as a new process, and execute it by computing the sum (7).

Some questions to think about:

- Passing arguments from one user program, through the kernel, into another user program, is a bit of a chore. What form does this take in C? You will probably need several iterations to get this right.
- What primitive operations exist to support the transfer of data to and from kernel space? Do you want to implement more on top of these?
- How will you determine: (a) the stack pointer initial value; (b) the initial register contents; (c) the return value; (d) whether you can exec the program at all?
- You will need to "bullet-proof" the OS/161 kernel from user program errors. There should be nothing a user program can do to crash the operating system (with the exception of explicitly asking the system to halt).
- What new data structures will you need to manage multiple processes?
- What relationships do these new structures have with the rest of the system?

Testing your system call implementations

To test your system call implementations, the best way would be to write small programs that make use of those system calls (fork, exec,waitpid, etc.), compile with `cs161-gcc` compiler, and run it under os161.

Specifically, when you build and invoke your kernel, on the sys161 menu, choose first the "Operations Menu", and then use the "[p]" option to invoke the executable. For example, typing:

```
p testbin/forktest
```

would invoke the `forktest` executable which is already provided in the "`testbin`" directory. The "`testbin`" directory has other executables; and some of them may be useful for debugging. By reading the first few lines of C programs there, you may be able to see whether the test program is suitable for this project or not.

But more importantly, do not restrict your tests with the programs in the `testbin` directory. In fact it may be wiser to first use very small programs (like the program with `fork()` and `exec()` whose code is given on the previous page). For example you can store that program in a separate directory under `testbin` and compile (by preparing and running a `Makefile` similar to that available for "`forktest`" in `testbin`). Those makefiles call `cs161-gcc` compiler with proper flags. And then you can again use `p testbin/myprogram` if the name of your new executable is `myprogram`. In your submission, include in your asst2 directory the scripts for the tests that you designed so that we can evaluate them as well.

In this project, it is NOT necessary to implement the filesystem related system calls (such as `read()` and `write()`). You can certainly do that if you wish; but it is not required. However, you will still need a "shortcut" in order to execute library calls like `putchar()`, `printf()` to print to the screen; because those calls issue a call to "`write()`". If you don't want to implement "`write()`" separately, you can do the following:

Go to the `kern/arch/mips/mips` directory, and open for editing the file `syscall.c`. Within the `mips_syscall()` function, you will see a `switch` statement. To handle the calls to "`write()`", add the following to the switch statement:

```
case SYS_write:
      for (i = 0; i < (size_t) tf->tf_a2; ++i) {
      kprintf("%c", ((char *) tf->tf_a1)[i]);
}
break;
```

You would also need to add a declaration for the variable `i` as an `unsigned int`, and put it in the beginning of the `mips_syscall()` function. Save `syscall.c`, build your kernel again, and this should be sufficient for simply printing to the screen. In fact, even the "`forktest`" program will need this "patch" in addition to the correct implementation of `fork()`, and `exec()`, for proper printing. But as we mentioned, it is best to test your implementations first with very short programs, before moving to "`forktest`" which is more complicated. You may also find some of the programs from the bin directory useful, such as `false`, and `true`. For `getppid()`, write your own (short) test program.

4. Coding Practices, Submission and Grading

In this project, you can continue to work with your partner from Project 1. If you worked alone in Project 1, you can pair up with another CS 471 student from your section who also worked alone in Project 1.

You will need to (again) put significant time in reading the existing OS 161 kernel code. Make sure to allocate sufficient time to debugging and extensive testing.

On the assignment due date:

1. Commit all your final changes to the system. Make sure your partner has committed everything as well. Make sure you have the most up-to-date version of everything. Re-run make on both the kernel and userland to make sure all the last-minute changes get incorporated.
2. Run the tests and generate scripts. If anything breaks, repeat step 1.
3. Tag your Git repository; prepare the diff and submission source tree.
4. Make your submission to the Blackboard by following the guidelines below.

Submissions

Your final submission should be a `.tar.gz` file. Please name this file `uid1_uid2-asst2.tar.gz`, where uid1 and uid2 are you and your partner's GMU email IDs. If you work alone, it should be named as `uid-asst2.tar.gz` where uid is your GMU e-mail id. It should contain the following:

- A copy of the complete current source code of your OS/161 version.
- All of the following in a newly-created asst2 directory
    - **Your design document. Please use only .pdf, or.txt file formats.**
    - A "diff" between asst2-begin and asst2-end.
    - A script of OS/161 running the various tt* tests successfully.
    - A script of the new test or tests you added for testing your system call implementations.

Note: Your design document is worth 20% of the grade for this assignment. It should contain:

- A high level description of how you are approaching the problem.
- A detailed description of the implementation (e.g., new structures, why they were created, what they are encapsulating, what problems they solve).
- A discussion of the pros and cons of your approach.
- Alternatives you considered and why you discarded them.

Include your answers to the code walk-through questions in the design document as well (worth another 20% of the grade).

On Blackboard, under the Projects/Project 2 folder you will find a link to submit your compressed file. All members of a group must submit separately the same compressed file, and before the deadline. Make sure to coordinate.
You can make as many submissions as you like; we will consider ONLY the last submission that you make. So in case you need to re-submit, you must make sure the compressed file contains all the components listed above.

Questions:
Programming – and system-related questions about the project should be directed to Fall 2020 CS 471 OS/161 Project Piazza Page, which is monitored by the GTAs from 10 AM to 6 PM on weekdays. Your posts to the Piazza are by default *private*, meaning that it is received only by the instructors and GTAs. You should NOT post public inquiries by changing those settings (occasionally, the GTAs can decide to make some questions and answers *public,* if they think they are relevant for many students).
But keep in mind that code-reading and exploring the OS/161 kernel files is a component of the assignment, so you should not expect detailed explanations of how the OS/161 kernel functions work.

Grading

This programming assignment accounts for **12%** of your final grade. Late penalty for this assignment will be 15% for each day. Submissions that are late by 3 days or more will not be accepted.