COMP3231/9201/3891/9283 Operating Systems 2020/T2

UNSW

Administration
- Notices
- Course Outline
- UNSW Timetable
- Consultations
- Group Nomination
- Survey Results!!

Work
- Lectures
- Tutorials

Support
- Piazza Forums
- Wiki

Assignments
- Submission Guide
- Assignment 0 Warm-up
- Assignment 1
- Assignment 2
- Assignment 3

Exam
- Sample Questions and Answers

Resources

OS/161
- General
- Man Pages
- Sys161 Pages

C coding
- Info Sheet

Debugging
- Learn Debugging

General
- "Hardware" Guide
- R3000 Reference Manual
- Intro. to Prog. Threads

Previous years
- 2020 T1
- 2019 T1
- 2018 S1
- 2017 S1
- 2016 S1
- 2015 S1
- 2014 S1
- 2013 S1
- 2012 S1
- 2011 S1
- 2010 S1
- 2009 S1
- 2008 S1

ASST2: System calls and processes

Table of Contents

Due Dates and Mark Distribution

Due Date & Time: 8am, Fri July 17th

Marks: Worth 30 marks (of the class mark component of the course)

The 2% per day bonus for each day early applies, capped at 10%, as per course outline.

Students can do the advanced part with the permission of the lecturer, and only if the basic assignment is completed one week prior to the deadline. Marks obtained are added to any shortfall in the class mark component up to a maximum of 10 bonus marks overall for all assignments.

Introduction

In this assignment you will be implementing a software bridge between a set of file-related system calls inside the OS/161 kernel and their implementation within the VFS (obviously also inside the kernel). Upon completion, your operating system will be able to run a single application at user-level and perform some basic file I/O.

A substantial part of this assignment is understanding how OS/161 works and determining what code is required to implement the required functionality. Expect to spend at least as long browsing and digesting OS/161 code as actually writing and debugging your own code.

If you attempt the advanced part, you will add process related system calls and the ability to run multiple applications.

Your current OS/161 system has minimal support for running executables, nothing that could be considered a true process. Assignment 2 starts the transformation of OS/161 into something closer to a true operating system. After this assignment, OS/161 will be capable of running a process from actual compiled programs stored in your account. The program will be loaded into OS/161 and executed in user mode by System/161; this will occur under the control of your kernel. First, however, you must implement part of 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.

The 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 might be adding in this and future assignments. 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 itself. 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". We give you two system call implementations: `sys_reboot()` in `main/main.c` and `sys___time()` in `syscall/time_syscalls.c`. In GDB, if you put a breakpoint on `sys_reboot()` and run the "reboot" program, you can use "backtrace" (or "where") to see how it got there.

User-level programs

Our System/161 simulator can run normal C programs if they are compiled with a cross-compiler, os161-gcc. A cross compiler runs on a host (e.g., a Linux x86 machine) and produces MIPS executables; it is the same compiler used to compile the OS/161 kernel. Various user-level programs already exist in `userland/bin`, `userland/testbin`, and `userland/sbin`. Note: that only a small subset these programs will execute successfully due to OS/161 only supporting a small subset of the system call interface.

To create new user programs (for testing purposes), you 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

In the beginning, you should tackle this assignment by producing a DESIGN. The design should clearly reflect the development of your solution. The design should not merely be what you programmed. If you try to code first and design later, or even if you design hastily and rush into coding, you will most certainly end up in a software "tar pit". Don't do it! Plan everything you will do. Don't even think about coding until you can precisely explain to your partner what problems you need to solve and how the pieces relate to each other. 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. The important thing is to go ahead and try. Always try to describe your ideas and designs to someone else. In order to reach an understanding, you may have to invent terminology and notation, this is fine. 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. Why do you think that CS is filled with so much jargon? To help you get started, we have provided the following questions as a guide for reading through the code. We recommend that you answer questions for the different modules and be prepared to discuss them in your tutorial. Once you have prepared the answers, you should be ready to develop a strategy for designing your code for this assignment.

Walkthrough

A guided walkthrough of the relevant code base is available here.

This walkthrough complements the existing ASST2 video . There are answers available on the course wiki . Additionally, if you have any further queries, use one of the week 6 consultations that are running during the normal tutorial timeslots.

Basic Assignment

Setup

We assume after ASST0 and ASST1 that you now have some familiarity with setting up for OS/161 development. If you need more detail, refer back to ASST0.

Clone the ASST2 source repository from gitlab.cse.unsw.edu.au, replacing the XXX with your 3 digit group number:

```
% cd ~/cs3231
% git clone https://zNNNNNNN@gitlab.cse.unsw.edu.au/COMP3231/20T
2/grpXXX-asst2.git asst2-src
```

Note: The gitlab repository is shared between you and your partner. You can both push and pull changes to and from the repository to cooperate on the assignment. If you are not familiar with cooperative software development and git you should consider spending a little time familiarising yourself with git.

Building and Testing Your Assignment

Configure OS/161 for Assignment 2

Before proceeding further, configure your new sources:

```
% cd ~/cs3231/asst2-src
% ./configure
```

Unlike previous the previous assignment, you will need to build and install the user-level programs that will be run by your kernel in this assignment:

```
% cd ~/cs3231/asst2-src
% bmake
% bmake install
```

For your kernel development, again we have provided you with a framework for you to run your solutions for ASST2. You have to reconfigure your kernel before you can use this framework. The procedure for configuring a kernel is the same as in ASST0 and ASST1, except you will use the ASST2 configuration file:

```
% cd ~/cs3231/asst2-src/kern/conf
% ./config ASST2
```

You should now see an ASST2 directory in the compile directory.

Building for ASST2

When you built OS/161 for ASST1, you ran make from `compile/ASST1`. In ASST2, you run make from (you guessed it) `compile/ASST2`:

```
% cd ../compile/ASST2
% bmake depend
% bmake
% bmake install
```

If you are told that the `compile/ASST2` directory does not exist, make sure you ran config for ASST2.

Command Line Arguments to OS/161

Your solutions to ASST2 will be tested by running OS/161 with command line arguments that correspond to the menu options in the OS/161 boot menu. **IMPORTANT: Please DO NOT change these menu option strings!**

Running "asst2"

For this assignment, we have supplied a user-level OS/161 program that you can use for testing. It is called `asst2`, and its sources live in `src/testbin/asst2`. You can test your assignment by typing `p /testbin/asst2` at the OS/161 menu prompt. As a shortcut, you can also specify menu arguments on the command line, example: `sys161 kernel "p /testbin/asst2"`.

Note: If you don't have a `sys161.conf` file, you can use the one from ASST1.

The simplest way to install it is as follows:

```
% cd ~/cs3231/root
% wget http://cgi.cse.unsw.edu.au/~cs3231/20T2/assignments/asst
2/sys161.conf
```

Running the program produces output similar to the following prior to starting the assignment:

```
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
:
:
Unknown syscall 55
Unknown syscall 55
Unknown syscall 3
Fatal user mode trap 4 sig 10 (Address error on load, epc 0x4008
14, vaddr 0xeeeee00f)
```

`asst2` produces the following output on a (maybe partially) working assignment:

```
OS/161 kernel [? for menu]: p /testbin/asst2
Operation took 0.000212160 seconds
OS/161 kernel [? for menu]:

**********
* File Tester
**********
* write() works for stdout
**********
* write() works for stderr
**********
* opening new file "test.file"
* open() got fd 3
* writing test string
* wrote 45 bytes
* writing test string again
* wrote 45 bytes
* closing file
**********
* opening old file "test.file"
* open() got fd 3
* reading entire file into buffer
* attempting read of 500 bytes
* read 90 bytes
* attempting read of 410 bytes
* read 0 bytes
* reading complete
* file content okay
**********
* testing lseek
* reading 10 bytes of file into buffer
* attempting read of 10 bytes
* read 10 bytes
* reading complete
* file lseek  okay
* closing file
Unknown syscall 3
Fatal user mode trap 4 sig 10 (Address error on load, epc 0x4008
14, vaddr 0xeeeee00f)
```

Note that the final fatal error is expected, and is due to `exit()` (system call 3) not being implemented by OS/161. If `exit()` returns to userland (which would not happen in a complete OS implementation), the userland exit library code simply accesses an illegal memory address in order to cause a fault, which subsequently causes the program (and system) to stop. You can distinguish this expected fault from other faults by the address accessed: `0xeeeee00f`.

The Assignment Task: File System Calls

Of the full range of system calls that is listed in `kern/include/kern/syscall.h`, **your task is to implement the following file-based system calls:** `open`, `read`, `write`, `lseek`, `close`, `dup2`. Note: You are writing the kernel code that implements part of the system call functionality **within the kernel**. You are not writing the C stubs that user-level applications call to invoke the system calls. The userland stubs are automatically generated from this file when you build OS/161 in *build/userland/lib/libc/syscalls.S* which you should not modify.

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 (also included in the distribution) and understand fully the system calls that you must implement. Your system calls 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 auto-marking scripts rely on the return of error codes, however, we are lenient as to which specific code in the case of potential ambiguity as to the most appropriate error code.

The file `userland/include/unistd.h` contains the user-level interface definition of the system calls. This interface is different from that of the kernel functions that you will define to implement these calls. You need to design the kernel side of this interface. The function prototype for your interface can be 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? If so, what are data structures are private to each process, what are shared (and thus have concurrency issues).

Note that the basic assignment does not involve implementing `fork()` (that's part of the advanced assignment). However, the design and implementation of your system calls should **not** assume only a single process will ever exist at a time. It should be possible to add a `fork()` implementation to your system call implementation without requiring your implementation to be redesigned to support more than one process.

That said, if you add a new data structure, you will need to think about its synchronisation. However, **you do NOT need to explicitly synchronise it for the basic assignment. Instead, you must clearly comment at the definition of any new data structure whether any concurrency issues exist in the presence of `fork()` , (including why or why not), and what is required to solve them.**

Notes on the file system system calls

`open()`, `read()`, `write()`, `lseek()`, `close()`, and `dup2()`

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) respectively. **For this basic assignment, the file descriptors 1 (stdout) and 2 (stderr) must start out attached to the console device ("con:"), 0 (stdin) can be left unattached.** You will probably modify `runprogram()` to achieve this. Your implementation must allow programs to use `dup2()` to change stdin, stdout, stderr 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 cwd) is specific only to the process, but others (such as 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 organise it, and when and how it has to change.

While this assignment requires you to implement file-system-related system calls, you actually have to write virtually no low-level file system code in this assignment. You will use the existing VFS layer to do most of the work. **Your job is to construct the subsystem that implements the interface expected by userland programs by invoking the appropriate VFS and vnode operations.**

While you are not restricted to only modifying these files, please place most of your implementation in the following files: function prototypes and data types for your file subsystem in `kern/include/syscall.h` **or** `kern/include/file.h` **, and the function implementations and variable instantiations in** `kern/syscall/file.c` **.**

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 return (see here for an online version). 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 error codes. 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 `user/lib/libc/string/strerror.c`.

Design Questions

Here are some additional questions and issues to aid you in developing your design. They are by no means comprehensive, but they are a reasonable place to start developing your solution. 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? 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 when invoking the file system calls. It is okay in the basic assignment for the kernel to perform a controlled panic for an unimplemented system call (e.g. `execv()`), or a user-level program error. It is not okay for the kernel to crash due to user-program behaviour. Decide which functions you need to change and which structures you may need to create to implement the system calls. How you will keep track of open files? For which system calls is this useful? For additional background, consult one or more of the following texts for details how similar existing operating systems structure their file system management:

- Section 10.6.3 and "NFS implementation" in Section 10.6.4, Tannenbaum, *Modern Operating Systems*.
- Section 6.4 and Section 6.5, McKusick *et al.*, *The Design and Implementation of the 4.4 BSD Operating System*.
- Chapter 8, Vahalia, *Unix Internals: the new frontiers*.
- The original VFS paper is available here

FAQ, Gotchas, and Video

See https://wiki.cse.unsw.edu.au/cs3231cgi/2020t2/Asst2 for an up to date list of potential issues you might encounter.

There is also an overview video on the assignment available on the lectures page in the course account http://cgi.cse.unsw.edu.au/~cs3231/lectures.php.

Basic Assignment Submission

The submission instructions are available on the Wiki . Like ASST0 and ASST1, you will be submitting the git repository bundle via CSE's `give` system. For ASST2, the submission system will do a test build and run a simple test to confirm your bundle at least compiles.

Warning

Don't ignore the submission system! If your submission fails the simple tests in the submission process, you may not receive any marks.

To submit your bundle:

```
% cd ~
% give cs3231 asst2 asst2.bundle
```

You're now done.

Even though the generated bundle should represent all the changes you have made to the supplied code, occasionally students do something "ingenious". So always push your changes back to gitlab (and keep your git repository) so that you may recover your assignment should something go wrong.

Advanced Assignment

The advanced assignment is available for bonus marks. Marks are awarded as follows:

- `fork()`, `getpid()`: 2 marks.
- `waitpid()`, `_exit()`, `kill_curthread()`: 2 marks.
- `exec()`: 1 mark.

The advanced assignment is to complete the basic assignment, plus the additional task below.

Given you're doing the advanced version of the assignment, I'm assuming you are competent with managing your git repository and don't need detailed directions. We expect you to work on a specific `asst2_adv` branch in your repository to both build upon your existing assignment, while keeping your advanced assignment separate at the same time.

Here are some git commands that will be helpful.

- One member of your group should create the branch and push it back to gitlab:

```
% git checkout -b asst2_adv
% git push --set-upstream origin asst2_adv
```

- To switch back to the basic assignment at some point:

```
% git checkout master
```

- To switch to the advanced assignment at another point:

```
% git checkout asst2_adv
```

User-level Process Management System Calls

`fork()`

Implement the `fork()` system call. Your implementation of fork should eventually be the same as that described in the man page, however for testing initially, you might consider always returning 1 for the child process id (pid) instead of implementing pid management. The amount of code to implement fork is quite small; the main challenge is to understand what needs to be done. Note: You will also need to revisit your existing file-related system calls and solve the concurrency issues you identified earlier.

Some hints:

- Read the comments above `mips_usermode()` in `kern/arch/mips/locore/trap.c`
- Read the comments in `kern/include/addrspace.h`, particularly `as_copy()`.
- You will need to copy the trapframe from the parent to the child. You should be careful how you do this, as there is a possible race condition (where?/why?).
- You may wish to base your implementation on the `thread_fork()` function in `kern/thread/thread.c`.

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()`. When your pid system is working correctly, change your `fork()` implementation to return the child's pid to the parent, rather than 1.

execv(), waitpid(), _exit()

These system calls are probably the most difficult part of the whole assignment, but also the most rewarding. They enable multiprogramming and make OS/161 a much more useful entity. `fork()` is your 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 from 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 these 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 carefully to understand the semantics, and consider these semantics from the ground up in your design. You may also wish to consult the 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()` relatively 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.

`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 Questions

Here are some additional questions and thoughts to aid in your design. They are not, by any means, meant to be a comprehensive list of all the issues you will want to consider. Your system must allow user programs to receive arguments from the command line.

By the end of Assignment 2, you should be capable of executing lines (in user programs) such as:

```
char *filename = "/bin/cp";
char *args[4];
pid_t pid;
args[0] = "cp";
args[1] = "file1";
args[2] = "file2";
args[3] = NULL;
pid = fork();
if (pid == 0)
execv(filename, argv);
```

which will load the executable file `cp`, install it as a new process, and execute it. The new process will then find `file1` on the disk and copy it to `file2`. You can test your implementation using OS/161's shell, `/bin/sh`.

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? This is rather tricky, and there are many ways to be led astray. You will probably find that very detailed pictures and several walk-throughs will be most helpful.
- 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?
- What new data structures will you need to manage multiple processes?
- What relationships do these new structures have with the rest of the system?
- How will you manage file accesses? When we invoke the `cat` command, and it starts to read `file1`, what will happen if the shell also tries to read `file1`? What would you like to happen?
- How will you keep track of running processes. For which system calls is this useful?
- How will you implement the execv system call. How is the argument passing in this function different from that of other system calls?

Advanced Assignment Submission

Submission for the advanced assignment is similar to the basic assignment, **except the advance component is given to a**

distinguished assignment name: `asst2_adv`. Again, you need to generate a bundle based on your repository. Note: Our marking scripts will switch to the `asst2_adv` branch prior to testing the advanced assignment.

Submit your solution by doing:

```
% cd ~
% give cs3231 asst2_adv asst2_adv.bundle
```

Page last modified: 5:47pm on Sunday, 5th of July, 2020

Print Version