ASST2: System calls and processes

Spec v0.9: Initial release of the spec. Implementation task is specified in full. Two documentation questions remain to be specified.

Spec v0.95: FAQ entry about append mode.

Spec v1.0: Final documentation questions and some clarification of the advanced part submission rules. Edits to the spec for `copy_splice()`.

Table of Contents

- Due Dates and Mark Distribution
  - Advanced Components and Marking Rules for Extended OS Students
- Introduction
  - User-level programs
  - Design
- Existing Code Walkthrough
- Basic Assignment
  - Setup
  - Building and Testing the Provided Code
    - Configure OS/161 for Assignment 2
    - Building for ASST2
    - Command Line Arguments to OS/161
    - Running "asst2"
  - The Assignment Task: File System Calls
    - Notes on the file system system calls
    - Notes on standard file descriptors
    - Some Design Questions
  - Part 1: Implement the common system calls
  - Part 2: Implement the specialised system calls
  - Documenting your solution
  - Basic Assignment Submission
- Advanced Assignment
  - Part A-1: Forking additional processes
    - `fork()`
    - `getpid()`
  - Part A-2: Copying Data within the Kernel
  - Part A-3: Ending Processes
    - `waitpid()`, `_exit()`
    - `kill_curthread()`
  - Part A-4: Running new binaries
    - `execv()`
  - Design Questions
  - Advanced Assignment Submission

- Hints
  - Copying Memory
  - Arguments in Memory
  - The VOP macros
- FAQ
  - On Specification and Error Codes
  - On Team-Work and Contributions
  - On the Append Mode

Due Dates and Mark Distribution

Due Date & Time: 4pm, Tue week 8, 21st July

Marks: Worth 20% of the marks for the course

Parts 1 and 2 are of equal value. For COMP3231/9201 students, they are worth 50% of the assignment each.

Advanced Components and Marking Rules for Extended OS Students

Assignments 2 and 3 have *advanced components*. Students are welcome to attempt any advanced components they find interesting. These are worth a small number of marks compared to their difficulty.

These advanced components are worth "bonus marks". They can be used to make up for any shortfall in other assessment components of the course (other than the exam). **The maximum amount of bonus marks that can be earned on this assignment is equivalent to 15% of the value of this assignment.**

Extended OS students will be marked on a slightly different scheme, essentially treating some advanced components as a mandatory component. In this assignment, parts 1 and 2 will be worth 40% each for Extended OS students, and component A-2 will be worth the remaining 20% of the marks.

You may attempt the assignment in groups of 2, also known as pairs. Pairs may be formed which mix the course codes, in which case the students will each be marked according to the scheme for their course code. To be fair to their partners, students who pair up with a COMP3891/9283 student should plan on doing some work on the mandatory advanced components, for which they will receive bonus marks rather than regular marks.

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

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. In assignment 1 we worked with some invented system calls that were globally available to all user code. Assignment 2 starts the transformation of OS/161 into something closer to a true operating system. Running processes will have a context of their own and an I/O interface to perform meaningful actions. Unlike in assignment 1, user-level processes will be able to use `printf` to produce output. 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 provided 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. You are given 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. In assignment 1, we did this for you to produce the starting state of test programs such as `left_wave` and `async_scale`.

Design

In the beginning, you should tackle this assignment by producing a DESIGN. The 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 to comprehend what is already provided.

To get a feel for what problems you need to solve, review the design questions and design document section later in this specification.

Existing Code Walkthrough

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

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.

You will soon be able to clone the ASST2 source repository from the CSE's gitlab instance, replacing the XXX with your 3 digit group number:

```
https://gitlab.cse.unsw.edu.au/coursework/COMP3231/26T2/assigns/grpXXX-asst2.git
```

The above requires a group number! You probably won't have been issued a group yet. See the group nomination site here.

You should receive an email from the gitlab instance once your group repository has been set up. This will typically happen overnight once your group nomination is finished.

There is also a repository which all students have read-only access to, from which you can clone the code prior to your group being formed. This repository can be found here:

```
% https://gitlab.cse.unsw.edu.au/coursework/comp3231/26t2/assigns/asst2-rel
```

Notes:

- The nominations to group ID and group ID to gitlab project scripts run from time to time. There might be a modest delay between nominating your partner and receiving an email about a gitlab repository. If you have been waiting for much more than 24h, ask on the forum.
- The gitlab repository above 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 the Provided Code

Configure OS/161 for Assignment 2

Before proceeding further, configure your new sources:

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

Build and install the user-level programs:

```
% 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.

Alternative, install it is as follows:

```
% cd ~/cs3231/root
% wget http://cgi.cse.unsw.edu.au/~cs3231/26T2/assignments/asst2/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
exit() was called, but it's unimplemented.
This is expected if your user-level program has finished.
panic: Can't continue further until sys_exit() is implemented
```

`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
```

```
* 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
exit() was called, but it's unimplemented.
This is expected if your user-level program has finished.
panic: Can't continue further until sys_exit() is implemented
```

Note that the final panic is expected, and is due to `exit()` (system call 3) not being implemented completely by OS/161. Implementing `exit()` is part of the advanced assignment.

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`, and **document your design**.

Note: You will be 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 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.) no matter what an application requests. Your code should also be memory leak free. You should consult the OS/161 man pages (also included in the distribution) and understand the system calls that you must implement. Your system calls must return the correct value (in case of success) or an appropriate 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 the specific code in the case of potential ambiguity as to which error code would be most appropriate. It's also not necessary to generate all error codes listed in the man pages.

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`.

Notes on the file system system calls

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

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.**

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 filesystem state. A large part of this assignment is designing and implementing a system to track this state.

Some of this state is specific to a process and file descriptor (i.e. one can expect to extend OS/161 with a per-process file descriptor data structure), and some information is shared across multiple processes (e.g. an open file table). 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.

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 in a complete system).

Note that the basic assignment does not involve implementing `fork()` (that's part of the advanced assignment). So in regard to concurrency, you can assume only a single process runs at a time. **You should NOT synchronise any data structures you add for the basic assignment.** Synchronised code with `fork()` unimplemented will likely attract attention for suspected plagiarism.

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, and then only synchronise your existing design to handle the concurrency.

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` **.**

Boot time initilisation code, if you need any, can be called from the end of `boot()` **in** `kern/main/main.c` **.**

Notes on standard file descriptors

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.

Some 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? 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. `exit()`), or a user-level program error. It is not okay for the kernel to crash due to user-program invoking your system calls with erroneous arguments. 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

Part 1: Implement the common system calls

Marks: Worth 50% of the marks for this assignment, 35% for auto-tests and 15% for documentation. Worth 40% (25% + 15% documentation) for COMP3891/9283 Extended-OS students, see the mark distribution section at the start.

Implement `open()`, `close()`, `read()` and `write()` as described above.

Once these are working, OS/161 can run various simple programs. The `asst2` test we have provided you with should succeed in producing output and testing read and write operations, up until the point at which it attempts `lseek`.

Once this simple test passes, you have finished Part 1.

Once you have finished Part 1, commit your work using `git`. You will need to refer to this commit in the documentation part, so it would be a good idea if this commit message mentioned that you have finished Part 1.

To make it even easier to find this revision number again in the future, you could give it a branch or tag name using `git`, for instance like this:

```
% # add a branch "part1_done" at the current revision.
% git branch part1_done
% # list your branches. you should still be on the "main" branch.
% git branch -v
```

Part 2: Implement the specialised system calls

Marks: Worth 50% of the marks for this assignment, 35% for auto-tests and 15% for documentation. Worth 40% (25% + 15% documentation) for COMP3891/9283 Extended-OS students, see the mark distribution section at the start.

This task is to implement `dup2()` and `lseek()` as described above.

These system calls are a little more specialised. Almost every UNIX application will open, close, read and write files, and almost every UNIX file can be read or written. Far fewer UNIX applications will directly manage their file descriptors with `dup2()`, and many UNIX files do not support seeking or have a data format which does not make sense to seek.

The `asst2` test will do some simple tests of `lseek()`. We do not provide a test for `dup2()`, and you might want to add one.

Once you are happy that these system calls are working as well, you have finished Part 2. Just like Part 1, make sure that your work is committed using `git`. If you plan to make further changes, e.g. in the advanced parts, make sure that you take a note of this revision number in the same way.

Documenting your solution

Like assignment 1, we want you to answer some simple questions about your code and save those answers in a documentation file `doc.txt`.

We will provide an example file you can start from . Unlike assignment 1, this is not part of the repository. If you are working with a partner, you should complete the implementation part of the assignment (in the repository) together with your partner, but each do the documentation part separately.

Here are the questions:

Part 1:

- What is the "magic number" associated with OS/161 vnode objects? What is it used for? Why do you think the kernel implementation is so careful about the relevant objects?
- Did you add an open file table to your implementation? What issues did you consider in making this design decision?
- New files can be created when `open()` is called with the `O_CREAT` flag. How does the OS/161 VFS layer go about delegating the job of creating a new file to the relevant file system implementation?

Part 2:

- What is an example of a file in OS/161 that does not permit `lseek()` system calls? What OS/161 functions are involved in that decision?

- How much code did you have to modify to implement Part 1, and how much to implement Part 2 once you had finished Part 1? The `git log --stat` and `git diff --stat` commands may be able to help you (look up their respective manual pages for more info), especially if you followed the intructions and committed at the end of Part 1.
- The OS/161 vnode structure includes a "reference count" value. Why does the system count references? Why is a reference-counting implementation a good fit for the VFS design?

Like with assignment 1, we hope that your answers will be clear and concise. We think that each of these questions can be answered in under 100 words, and therefore your overall set of answers will be 600 words or less.

Basic Assignment Submission

The standard 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. For the basic submission, you will also include a `part3.md` document separately.

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

Submit your bundle and your document via `give`:

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

The repository also contains the same python submission-helper as assignment 1. It may help you assemble the bundle and submit on CSE. It is run like this:

```
% python3 submission/submit main doc.txt
```

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

These are the advanced assignment components.

- `fork()` and `getpid()`

- `copy_splice()`
- `waitpid()`, `_exit()` and `kill_curthread()`
- `execv()`

The advanced assignment is to complete the basic assignment, plus provide support for the additional features below.

Some additional "bonus marks" are available for the advanced part of this assignment. They can be used to make up for any shortfall in other assessment components of the course (other than the exam). The components below list their value. **The maximum bonus mark for this assignment is equivalent to 15% of the value of this assignment.**

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
```

Part A-1: Forking additional processes

Marks: worth the equivalent of 10% of this assignment

Implement the `fork()` and `getpid()` system calls.

This system call (together with `execv()` below) is 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. If you plan on doing Part A-4, consider it together with `execv()` to make sure that each system call is performing the correct functionality. More on that below.

`fork()`

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.

Part A-2: Copying Data within the Kernel

For COMP3231/9201 students: bonus marks worth the equivalent of 5% of this assignment

For Extended OS COMP3891/9283 students: worth 20% of the regular marks for this assignment

Part A-2 is unrelated to Part A-1.

Your task is to implement the `copy_splice()` system call, which is a hybrid of Linux's `splice()` and `copy_file_range()` system calls. The Linux variants specify which file objects they can be used on, `splice()` only for pipes and `copy_file_range()` only for regular files. Your `copy_splice()` should support both regular files and devices such as the console device "con:".

The intended signature is:

```
size_t copy_splice(int fd_in, int fd_out, size_t len);
```

This signature omits the offsets and flags in the Linux signatures for `splice()` etc, but has

otherwise the same intention, and the arguments and return values have the same meaning as their `splice()` equivalents. The user program wants to copy up to `len` bytes from the input file to the output file without those bytes needing to be copied to and from user memory.

This is not a complex concept, and requires *less* of the complex kernel to user copies than the `read()` and `write()` system calls from the main part. However, there are some potential pitfalls, and the Linux man pages mention buggy implementations. Think through all the possible cases before you start implementing.

Like the assignment 1 "aux-I/O" system calls, it's up to you to add this system call so that it can be called by user programs.

Part A-3: Ending Processes

Marks: worth the equivalent of 5% of this assignment

This part depends on Part A-1.

`waitpid()`, `_exit()`

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()`

The `kill_curthread` function handles cases where the current user-level task encounters a fatal exception. It is currently mostly unimplemented; your job is to find it and get it working.

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.

Part A-4: Running new binaries

Marks: worth the equivalent of 5% of this assignment

This part depends on Part A-1.

`execv()`

This system call, together with `fork()`, is probably the most difficult part of the whole assignment, but also the most rewarding.

You will want to think carefully through the design of `fork()` and `execv()` together. Although `execv()` is "only" a system call in one process, it is really the heart of this assignment. It is responsible for taking newly created processes and making 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()`).

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 Parts A-3 and A-4, 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 advanced component is given to a distinguished assignment name:** `asst2_adv`. Again, you need to generate a bundle based on your repository.

We recommend you create a bundle with only a single branch in it when submitting the advanced part. Make sure that the branch you are submitting is the correct one and not your basic assignment!

We will (soon) push a variant of the submission helper python script to the bugfix branch, and it will clarify the history of the branch it is creating a bundle from.

This year you don't need to do any extra documentation for your advanced part. This paragraph replaces a previous paragraph that was left in from last year.

Submit your solution by doing:

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

Hints

Copying Memory

In assignment 1, we already thought about how to move values between user programs and the kernel. However we only addressed the case where arguments and return values are passed in registers. In assignment 2, we need to move larger quantities of data in and out of user memory.

As we will soon cover in lectures, the kernel and user programs have a different view of memory. OS/161 is a `copyin()`/`copyout()` kernel. Any time that user memory is to be read, it is first copied into a kernel buffer via `copyin()`, and vice versa, all writes to user memory are performed via `copyout()`.

It is important that the kernel doesn't use a user-level address as a pointer directly. For instance, the `char *` and `void *` arguments to user-level `open()`, `write()` etc should be treated as `userptr_t` within the OS/161 kernel.

If you haven't done it already, look for the `copyin()` and `copyout()` functions and their close relatives. There is a longer guided walkthrough available here that discusses them in

relatives. There is a longer guided walkthrough available here that discusses them in detail.

The related UIO mechanism is used for copying data to or from an I/O subsystem (e.g. a file system or other device). It can copy directly to or from user memory, in which case it will perform the `copyin()` and `copyout()` calls required.

Arguments in Memory

Like in assignment 1, most system call arguments will be passed in via trapframe registers `a0` through `a3`.

The exception is `lseek(int fd, off_t pos, int whence)`. Its offset argument `pos` is 64-bit, so requires two argument registers. By convention, the register pairs available are `a0/a1` and `a2/a3`, so `pos` is placed in the second of these, `a1` is not used, and `whence` is left to be passed on the stack. There is further discussion of the implications of the calling convention in the OS/161 sources, and I recommend you look for it.

To convert the two 32-bit argument registers into a 64-bit value and vice versa, you can use the `join32to64` and `split64to32` helper functions.

The VOP macros

In the VFS lecture, we described how VFS vnode objects have a "method table" of function pointers. This allows a write operation on an "emufs" file and a write operation on the console device to be implemented by completed different functions.

The `vnode.h` header provides macros that capture the common pattern of fetching an operation from a vnode and passing the vnode back to it. These look like `VOP_FOO`, and, as the header explains, `VOP_FOO(vnode, arg1, ...)` expands to `vnode->vn_ops->vop_foo(vnode, arg1, ...)`.

FAQ

On Specification and Error Codes

When the manual page for one of these syscalls suggests possible error codes (`EINVAL`, `EMFILE`, `EIO` etc), what does that mean? Is it a problem if your implementation won't generate one of those errors?

For instance, does your `open` implementation **need** to be able to generate both `EMFILE` and `ENFILE` errors?

Not necessarily. Often a manual entry needs to give a specification of an interface that might have many implementations. One example is UNIX. A UNIX programmer might assume that they can read the manual entries for `open` `read` and `close` and create a C program that works on many UNIX implementations (Linux, MacOS X, BSD, SunOS etc).

Another example is OS/161. The manual pages specify the interface. Literally hundreds of students are producing different implementations, which might have slightly different behaviour.

When the manual lists a collection of error codes, it's not a problem if your code doesn't generate all of them. It would be a problem if your code generated error codes that aren't on the list; that might be surprising to valid user programs. It's unlikely we'll test exhaustively for that kind of error though, we'd have to intentionally drive your code into rare error cases.

On Team-Work and Contributions

Most students are working in a pair, and this is the recommended approach.

Multiple students have, in the past, asked how closely the pair have to work together. How much do they have to contribute to each part, and to what extent does it matter who commits what?

It is natural to divide up the task somewhat, especially in the early phases. The parts of the job are fairly interlinked, though, so to get the assignment working you will probably have to review your partner's work, and likely make changes or request changes. If that somehow isn't the case, you should review your partner's work anyway. Consider it your professional duty to take some responsibility for whatever you submit.

We have mentioned already that you are not allowed to simply "trade" and have one partner do all of assignment 2 and the other do all of assignment 3.

The expectation is that the partners will make roughly even contributions. If that is far from the case, you can request we intervene, in which case we will interrogate both partners and whatever documentary evidence exists. Hopefully we don't have to do that.

On the Append Mode

Some students have found the discussion of `O_APPEND` semantics in the supplied manual pages.

In short, the "optional" concurrent aspect of `O_APPEND` can be ignored for this assignment. It might not be possible to implement it without making a lot of changes to the OS/161 version you were provided.

Longer answer: There are two ways to think about opening with `O_APPEND`. The simple design (which you might have assumed) ensures that the file pointer/offset is at the end of the file once `open()` is complete, and subsequent writes extend the file. This may be tested. The more thorough implementation allows two independent processes to append (e.g. for logging) to the same file by ensuring that their offset/pointer is always at the end of the file. Doing this correctly in a concurrent environment requires some way to lock or atomically update the file contents and file pointer. The "emufs" file system we are using doesn't provide that anyway. This thorough/concurrent aspect will not be tested

using doesn't provide that anyway. This thorough/concurrent aspect will not be tested.