Project 1: User Programs

Due: March 1, 2022

Contents

1 Introduction

Welcome to Project User Programs! After completing Project Pregame, you are probably left with a lot of questions on how Pintos works in general and the validity of the "fix" you implemented. Don't worry. This project will have you implement some core functionalities such as argument passing as well as essential syscalls and floating point operations that will fill in some holes from Project Pregame.

1.1 Setup

First, log into your VM and check if your `group` folder is initialized correctly.

```
> cd ~/code/group
> git remote -v
staff https://github.com/Berkeley-CS162/group0.git (fetch)
staff https://github.com/Berkeley-CS162/group0.git (push)
```

If your output doesn't match the above, reinitialize your `group` folder.

```
> rm -rf ~/code/group
> git clone -o staff https://github.com/Berkeley-CS162/group0.git ~/code/group
> cd ~/code/group
```

This should allow you to pull the starter code with `git pull staff master`.

Next, add a pre-commit hook that will format your code according to our linting standards.

```
ln -s -f ~/code/group/.pre-commit.sh ~/code/group/.git/hooks/pre-commit
```

This will ensure that before you make a commit, your code will be nicely formatted. Part of your project grade will be determined on your code quality, it is required that you install this formatter. See Code for more information.

Finally, head over to the group dashboard on the autograder[1]. If you are not registered as a group, make sure to do that first. Your teammates will all need to head to the group dashboard to accept your invite as well. Click on the GitHub logo which will take you to your group's repo. If you are unable to access the repo (i.e. 404 error), then make sure to check your email for an invitation. The invite may have expired, in which case you should follow up in the Piazza thread, and a staff member will resend the invitation.

Once you're at your repo page on GitHub, copy the SSH URI which should look like

```
git@github.com:Berkeley-CS162/groupX
```

where `X` is your group number. Head back over to your VM and add this as a remote, making sure to replace the `X`.

```
> git remote add group git@github.com:Berkeley-CS162/groupX.git
> git remote -v
group git@github.com:Berkeley-CS162/groupX.git (fetch)
group git@github.com:Berkeley-CS162/groupX.git (push)
staff https://github.com/Berkeley-CS162/group0.git (fetch)
staff https://github.com/Berkeley-CS162/group0.git (push)
> git remote show group
* remote group
  Fetch URL: git@github.com:Berkeley-CS162/groupX.git
  Push  URL: git@github.com:Berkeley-CS162/groupX.git
  HEAD branch: (unknown)
```

[1] https://cs162.org/autograder/dashboard/group/

2 Tasks

2.1 Argument Passing

The `process_execute` function is used to create new user processes in Pintos. Currently, it does not support command-line arguments. You must implement argument passing such that the `main` function of the user process will receive the appropriate `argc` and `argv`. For instance, if you called `process_execute("ls -ahl")`, the user program would receive 2 as `argc` and `["ls", "-ahl"]` as `argv`.

Many of our Pintos test programs start by printing out their own name (i.e. `argv[0]`). Since argument passing has not yet been implemented, all of these programs will crash when they access `argv[0]`. **Until you implement argument passing, most test programs will not work**.

2.2 Process Control Syscalls

Pintos currently only supports one syscall, `exit`, which terminates the calling process. You will add support for the following new syscalls: `practice`, `halt`, `exec`, `wait`. Make sure to read through Process Control for a full description on each of them.

To implement syscalls, you first need a way to safely read and write memory that's in a user process's virtual address space. The syscall arguments are located on the user process's stack, right above the user process's stack pointer. You are not allowed to have the kernel crash while trying to dereference an invalid or null pointer. For example, if the stack pointer is invalid when a user program makes a syscall, the kernel ought not crash when trying to read syscall arguments from the stack. Additionally, some syscall arguments are pointers to buffers inside the user process's address space. Those buffer pointers could be invalid as well.

You will need to gracefully handle cases where a syscall cannot be completed due to invalid memory access including null pointers, invalid pointers (e.g. pointing to unmapped memory), and illegal pointers (e.g. pointing to kernel memory). Beware: a 4-byte memory region (e.g. a 32-bit integer) may consist of 2 bytes of valid memory and 2 bytes of invalid memory, if the memory lies on a page boundary. You should handle these cases by terminating the user process. **We recommend testing this part of your code before implementing any other system call functionality**. See Accessing User Memory for more information.

2.3 File Operation Syscalls

In addition to the process control syscalls, you will also need to implement the following file operation syscalls: `create`, `remove`, `open`, `filesize`, `read`, `write`, `seek`, `tell`, and `close`. Pintos already contains a basic file system which implements a lot of these functionalities, meaning you will not need to implement any file operations yourself. Your implementations will simply call the appropriate functions in the file system library.

Keep in mind that most testing programs use the `write` syscall for output. **Until you implement `write` syscall, most test programs will not work**.

Pintos' file system is not thread-safe, so you must make sure that your file operation syscalls do not call multiple file system functions concurrently. In Project File Systems, you will add more sophisticated synchronization to the Pintos file system, but for this project, you are permitted to use a global lock on file operation syscalls (i.e. treat it as a single critical section to ensure thread safety). **We recommend that you avoid modifying the `filesys/` directory in this project**.

While a user process is running, you must ensure that nobody can modify its executable on disk. The `rox-*` tests check that this has been implemented correctly. The functions `file_deny_write` and `file_allow_write` can assist with this feature. Denying writes to executables backing live processes is important because an operating system may load code pages from the file lazily, or may page out some code pages and reload them from the file later. In Pintos, this is technically not a concern because the file is loaded into memory in its entirety before execution begins, and Pintos does not implement demand paging of any sort. However, you are still required to implement this, as it is good practice.

Your final code for Project User Programs will be used as a starting point for Project File Systems. The tests for Project File Systems depend on some of the same syscalls that you are implementing for this project, and you may have to modify your implementations of some of these syscalls to support additional features required for Project File Systems. While you certainly will not be able to plan too far in advance for Project File Systems, we recommend you write good code that can be easily modified by keeping good documentation.

2.4 Floating Point Operations

Pintos currently does not support floating point operations. You must implement such functionality so that both user programs and the kernel can use floating point instructions.

This may seem like a daunting task, but rest assurred, the operating system doesn't have to do much when it comes to implementing floating point instructions. The compiler does the hard work of generating floating point instructions, while the hardware does the hard work of executing floating point instructions. However, because there is only one floating-point unit[2] (FPU) on the CPU, meaning all threads must share it. As a result, the operating system must save and restore floating-point registers on the stack during context switches, interrupts, and syscalls like it does for general purpose registers (GPR). However, contrary to GPRs, The FPU registers need to be initialized correctly when a new thread or process is created as well, which is different from the GPRs. Additionally, the operating system needs to initialize the FPU during startup. See Floating Point for more details.

The learning goal for this class is not for you to learn the ins and outs of floating point operations but rather understand the details of thread/context switching. If you find yourself reading through specialized floating operations dealing with the actual arithmetic itself, take a step back and reread through the task and Floating Point appendix.

Since this task deals with some of the OS initialization and context switching code, an error in these files may result in some gnarly and indecipherable errors that result from corrupting a thread. **We recommend you work on floating point operations after all the other tasks**.

2.5 Concept Check

The following are conceptual questions meant to be answered in your design document. You won't need to write any code for these questions, but you may need to read and reference some.

1. Take a look at the Project User Programs test suite in `src/tests/userprog`. Some of the test cases will intentionally provide invalid pointers as syscall arguments, in order to test whether your implementation safely handles the reading and writing of user process memory. Identify a test case that uses an **invalid** stack pointer (`%esp`) when making a syscall. Provide the name of the test and explain how the test works. Your explanation should be very specific (i.e. use line numbers and the actual names of variables when explaining the test case).
2. Please identify a test case that uses a **valid** stack pointer when making a syscall, but the stack pointer is too close to a page boundary leading to some of the syscall arguments being located in invalid memory. Provide the name of the test and explain how the test works. Your explanation should be very specific (i.e. use line numbers and the actual names of variables when explaining the test case).
3. Identify **one** part of the project requirements which is **not** fully tested by the existing test suite. Provide the name of the test and explain how the test works. Explain what kind of test needs to be added to the test suite, in order to provide coverage for that part of the project.

2.6 Testing

Pintos already contains a test suite for Project User Programs, but not all of the parts of this project have complete test coverage. You must submit **two new test cases** which exercise functionality that is not

[2] https://en.wikipedia.org/wiki/Floating-point_unit

covered by existing tests. We will not tell you what features to write tests for, so you will be responsible for identifying which features of this project would benefit most from additional tests. Make sure your own project implementation passes the tests that you write. You can pick any appropriate name for your test, but beware that **test names should be no longer than 14 characters**.

You can either write tests in user space or kernel space. User space tests interact with the kernel through system calls, whereas kernel space tests have direct access to the kernel. We encourage you to write user space tests, though it may be more convenient to write kernel-based tests if you want to test the FPU. All user space tests and kernel space tests are respectively located in `src/tests/userprog` and `src/tests/userprog/kernel/`.

3 Deliverables

3.1 Design

Before you start writing any code for your project, you need to create an design plan for each feature and convince yourself that your design is correct. You must **submit a design document** and **attend a design review** with your TA. This will help you solidify your understanding of the project and have a solid attack plan before tackling a large codebase.

3.1.1 Document

Like any techinical writing, your design document needs to be clean and well formatted. We've provided you with a template linked on the website that you must use. The template can be found on the website. We use Dropbox Paper[3] which supports real-time collaboration like Google Docs with the added benefit of techincal writing support (e.g. code blocks, LaTeX). **Not using this template or failure to use code formatting will result in losing points**. The main goal of this is not to punish you for this but rather make it easy for TAs to read.

To get started, navigate to the template and click the "Create doc" button on the top right hand corner. You can share this doc with other group members to collaborate on it. **Make sure to click the blue "Share" button in *your* document, not the template**.

For each task except for Concept Check, you must explain the following aspects of your proposed design. We suggest you create a section for each task which has subsections for each of the following aspects.

Data Structures and Functions

List any `struct` definitions, global or static variables, `typedef`s, or enumerations that you will be adding or modifying (if it already exists). These should be **written with C not pseudocode**. Include a **brief explanation** (i.e. a one line comment) of the purpose of each modification. A more in depth explanation should be left for the following sections.

Algorithms

Tell us how you plan on writing the necessary code. Your description should be at a level below the high level description of requirements given in the assignment. **Do not repeat anything that is already stated on the spec**.

On the other hand, your description should be at a level above the code itself. Don't give a line-by-line run-down of what code you plan to write. You may use *small snippets* of pseudocode or C in places you deem appropriate. Instead, you need to convince us that your design satisfies all the requirements, **especially any edge cases**. We expect you to have read through the Pintos source code when preparing your design document, and your design document should refer to the appropriate parts of the Pintos source code when necessary to clarify your implementation.

Synchronization

List all resources that are shared across threads and processes. For each resource, explain how the it is accessed (e.g. from an interrupt context) and describe your strategy to ensure it is shared and modified safely (i.e. no race conditions, deadlocks).

In general, the best synchronization strategies are simple and easily verifiable. If your synchronization strategy is difficult to explain, this is a good indication that you should simplify your strategy. Discuss the time and memory costs of your synchronization approach, and whether your strategy will significantly limit the concurrency of the kernel and/or user processes/threads. When discussing the concurrency allowed by your approach, explain how frequently threads will contend on the shared resources, and any limits on the number of threads that can enter independent critical sections at a single time. You should aim to avoid locking strategies that are overly coarse.

[3] http://paper.dropbox.com/

Rationale Tell us why your design is better than the alternatives that you considered, or point out any shortcomings it may have. You should think about whether your design is easy to conceptualize, how much coding it will require, the time/space complexity of your algorithms, and how easy/difficult it would be to extend your design to accommodate additional features.

3.1.2 Review

After you submit your design doc, you will schedule a design review with your TA. A calendar signup link will be posted sometime before the design doc due date. During the design review, your TA will ask you questions about your design for the project. You should be prepared to defend your design and answer any clarifying questions your TA may have about your design document. The design review is also a good opportunity to get to know your TA for participation points.

3.1.3 Grading

The design document and design review will be graded together. Your score will reflect how convincing your design is, based on your explanation in your design document and your answers during the design review. If you cannot make a design review, please contact your TA to work out an arrangement. **An unexcused absence from a design review will result in a 0 for the design portion**.

3.2 Code

Code will be submitted to GitHub via your groupX repo. Pintos comes with a test suite that you can run locally on your VM. We will run the same tests on the autograder, meaning there are **no hidden tests**. As a result, we recommend you test locally as much as possible since the autograder's bandwidth is limited.

3.2.1 Checkpoints

On the autograder, we will provide checkpoints for you to stay on pace with the project. **Checkpoints will not be graded and have no effect on your grade**. However, we still encourage students to keep up with the checkpoints See Plan for more details on the specific checkpoints for this project.

Each checkpoint will have a corresponding autograder. Checkpoint 1 autograder will automatically run until its deadline, then Checkpoint 2 will automatically run until its deadline. The final autograder will not automatically run until after the Checkpoint 2 autograder. If you'd like, you can manually trigger the autograders as they'll be available throughout the entire project duration.

3.2.2 Testing

Your testing code needs to be included in your repo as well under the appropriate folder.

3.2.3 Quality

The score of your code will be mainly determined by your autograder score. However, you will also be graded on the quality of your code on some factors including but not limited to

- Does your code exhibit any major memory safety problems (especially regarding strings), memory leaks, poor error handling, or race conditions?
- Is your code simple and easy to understand? Does it follow a consistent naming convention?
- Did you add sufficient comments to explain complex portions of code?
- Did you leave commented-out code in your final submission?
- Did you copy and paste code instead of creating reusable functions?
- Did you re-implement linked list algorithms instead of using the provided list manipulation functions?
- Is your Git commit history full of binary files?

Note that you don't have to worry about manually enforcing code formatting (e.g. indentation, spacing, wrapping long lines, consistent placement of braces.) as long as you setup your precommit hook correctly in Project User Programs. You may also find it helpful to format code every once in a while by running `make format`.

3.3 Report

After you complete the code for your project, your group will write a report reflecting on the project. While you're not expected to write a massive report, we are asking for the same level of detail as expected for the design document. Your report should include the following sections.

Changes

Discuss any changes you made since your initial design document. Explain why you made those changes. Feel free to reiterate what you discussed with your TA during the design review if necessary.

Reflection

Discuss the contribution of each member. Make sure to be specific about the parts of each task each member worked on. Reflect on the overall working environment and discuss what went well and areas of improvement.

Testing

For each of the 2 test cases you write, provide

- Description of the feature your test case is supposed to test.
- Overview of how the mechanics of your test case work, as well as a *qualitative* description of the expected output.
- Output and results of your own Pintos kernel when you run the test case. These files will have the extensions `.output` and `.result`.
- Two non-trivial potential kernel bugs and how they would have affected the output of this test case. Express these in the form "If my kernel did X instead of Y, then the test case would output Z instead.". You should identify two different bugs per test case, but you can use the same bug for both of your two test cases. These bugs should be related to your test case (e.g. syntax errors don't

In addition, tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system? What did you learn from writing test cases?

3.4 Evaluations

After you finish all the components above, you must submit an evaluation of your group members. You are will be given $20(n-1)$ points to distribute amongst your group members *not including yourself*, where n is the number of people in your group. For instance, if you have four team members, you will get 60 points to distribute amongst the rest of your group members. If you believe all group members contributed equally, you would give each member 20 points each.

You will also fill out the details of what each member worked on. While this is similar to Report portion of the report, this will serve as a space to truthfully speak about the contributions since the report is a collaborative document.

To submit evaluations, navigate to your `personal` repo and pull from the staff repo. You should see a folder for evaluations (e.g. `proj-userprog-eval` for Project User Programs) which contains a `evals.txt`. Fill out the `evals.txt` file. Each line should pertain to one group member, so make your comments for each group member remain in that line. The following is an example of what your csv should look like.

```
Name|Autograder ID (XXX)|Score|Comment
Jieun Lee|516|23|Worked on all tasks finished the testing report
Taeyeon Kim|309|20|Worked on tasks 2, 3, 4, helped come up with a lot of design ideas.
```

```
Sean Kim|327|17|Helped with task 1 and testing but didn't work on design doc.
```

Your comments should be longer and more descriptive than the ones given above. Ideally, aim to write between 100 and 200 words for each group member.

This evaluation will remain anonymous from the rest of your group members. If we notice some extreme weightings of scores, we will reach out and arrange a meeting to discuss any group issues. **These evaluations are important and hold substantial weight, so fill them out honestly and thoroughly**. See Grading for more information on how evaluations will be used.

3.5 Submission

Design documents and final reports should be submitted to Gradescope to their respective locations. You can export your document from Dropbox Paper by clicking on the three dots in the top right hand corner and clicking "Export".

Make sure you've pushed your code and have an autograder build. This build **must include the testing code** for you to receive credit on testing.

Make sure to push your evals to your individual repo to trigger the autograder. If you get errors from the autograder, make sure to check that you've done the following.

- Included exactly $n - 1$ members (i.e. no evals of yourself).
- Autograder IDs are correct and *not prefixed with student* (i.e. 162 instead of student162).
- The points sum to exactly $20(n - 1)$.
- No negative points for any member.
- Header row is present.
- Each line contains one evaluation. Your comment section should not consist of multiple lines.
- No extra lines at the end of the file.

Make sure to not change the header row.

3.6 Grading

The components above will be weighed as follows: 15% Design, 70% Code, 15% Report. While evaluations are not explicitly part of your grade, your project score will be impacted based on them. To keep evaluations fair and anonymous, we will not be releasing how evaluations are factored in nor the final scores calculated after factoring in evaluations.

Slip days or extensions cannot be applied on design documents. TAs need enough time to read these design documents to be able to hold design reviews in a timely manner. Slip days will be applied as a maximum across all the other parts. **Evaluation submission will be factored into slip day usage**. For instance, if code is submitted on time, report is submitted one hour late, three members submit their evaluations on time, and one member submits their evaluation two days late, the number of slip days used will be 2 days.

4 Plan

We provide you a suggested order of implementation as well as the specifications for each checkpoint based on our and past students' experiences. However, this is merely a suggestion and you may elect to approach the project entirely differently.

Keep in mind that checkpoints are not graded.

4.1 Checkpoint 1

Start the project by implementing the `write` syscall for the `STDOUT` file descriptor. Once you've done this, the `stack-align-1` test should pass, assuming you build on the code you have at the end of Project Pregame.

Next, implement `practice` syscall and the Argument Passing task.

Finally, make sure that the `exit(-1)` message is printed even if a process exits due to a fault. Currently the exit code is printed [4] when the `exit` syscall is made from userspace, but not if it the process exits due to an invalid memory access.

After this checkpoint, you should be passing the `args-*`, `bad-*`, `stack-align-*`, `do-nothing`, `iloveos`, `practice`.

4.2 Checkpoint 2

Building off of Checkpoint 1, complete the remaining Process Control Syscalls and File Operation Syscalls.

After this checkpoint, you should be passing all tests except for `fp-*` and `kernel/fp-*`. If you're not passing the `no-vm/multi-oom` test, feel free to skip it for now. It is a harder test to debug that can often only be done by examining your code, so you should return to it after you've implemented all the other functionalities.

4.3 Final

Finish the project with Floating Point Operations. Make sure to fix the `multi-oom` test if you skipped that during Checkpoint 2. You should be passing all tests.

[4] see line 32 of `/pintos/src/userprog/syscall.c`

5 FAQ

How much code will I need to write?

Here's a summary of our reference solution. The reference solution represents just one possible solution. Many other solutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the reference solution.

```
lib/float.h          |    6
threads/init.c       |    7
threads/interrupt.h  |    1
threads/intr-stubs.S |    6
threads/start.S      |    5
threads/switch.S     |    4
threads/switch.h     |    5
threads/thread.c     |    4
threads/thread.h     |    1
userprog/exception.c |    7
userprog/process.c   |  268 +++++++++++++++++++++++++++++-----
userprog/process.h   |   37 ++++
userprog/syscall.c   |  392 ++++++++++++++++++++++++++++++++++++++++++++++++++--
userprog/syscall.h   |   20 ++
14 files changed, 697 insertions(+), 66 deletions(-)
```

The kernel always panics when I run a custom test case.

Is your file name too long? The file system limits file names to 14 characters. Is the file system full? Does the file system already contain 16 files? The base Pintos file system has a 16-file limit.

The kernel always panics with `assertion is_thread(t)` failed.

This happens when you overflow your kernel stack. If you're allocating large structures or buffers on the stack, try moving them to static memory or the heap instead. It's also possible that you've made your `struct thread` too large. See the comment underneath the kernel stack diagram in `threads/thread.h` about the importance of keeping your `struct thread` small.

All my user programs die with page faults.

This will happen if you haven't implemented argument passing (or haven't done so correctly). The basic C library for user programs tries to read argc and argv off the stack. If the stack isn't properly set up, this causes a page fault.

All my user programs die upon making a syscall.

You'll have to implement syscall before you see anything else. Every reasonable program tries to make at least one syscall (`exit`) and most programs make more than that. Notably, `printf` invokes the `write` syscall. The default syscall handler just handles `exit()`. Until you have implemented syscalls sufficiently, you can use `hex_dump` to check your argument passing implementation (see Startup).

How can I disassemble user programs?

The `objdump (80x86)` or `i386-elf-objdump` (SPARC) utility can disassemble entire user programs or object files. Invoke it as `objdump -d <file>`. You can use GDB's codedisassemble command to disassemble individual functions.

Why do many C include files not work in Pintos programs? Can I use libfoo in my Pintos programs?

The C library we provide is very limited. It does not include many of the features that are expected of a real operating system's C library. The C library must be built specifically for the operating system (and architecture), since it must make syscalls for I/O and memory allocation. Not all functions do, of course, but usually the library is compiled as a unit.

If the library makes syscalls (e.g, parts of the C standard library), then they almost certainly will not work with Pintos. Pintos does not support as rich a syscall interfaces as real operating systems (e.g., Linux, FreeBSD), and furthermore, uses a different interrupt number (0x30) for syscalls than is used in Linux (0x80).

The chances are good that the library you want uses parts of the C library that Pintos doesn't implement. It will probably take at least some porting effort to make it work under Pintos. Notably, the Pintos user program C library does not have a `malloc` implementation.

How do I compile new user programs?
Modify `examples/Makefile`, then run `make`.

Can I run user programs under a debugger?
Yes, with some limitations. See Project Pregame appendix.

What's the difference between `tid_t` and `pid_t`?
A `tid_t` identifies a kernel thread, which may have a user process running in it (if created with `process_execute`) or not (if created with `thread_create`). It is a data type used only in the kernel.

A `pid_t` identifies a user process. It is used by user processes and the kernel in the `exec` and `wait` syscalls.

You can choose whatever suitable types you like for `tid_t` and `pid_t`. By default, they're both `int`. You can make them a one-to-one mapping, so that the same values in both identify the same process, or you can use a more complex mapping. It's up to you.

5.1 Argument Passing

Isn't the top of stack in kernel virtual memory?
The top of stack is at `PHYS_BASE`, typically `0xc0000000`, which is also where kernel virtual memory starts. However, before the processor pushes data on the stack, it decrements the stack pointer. Thus, the first (4-byte) value pushed on the stack will be at address `0xbffffffc`.

Is `PHYS_BASE` fixed?
No. You should be able to support `PHYS_BASE` values that are any multiple of `0x10000000` from `0x80000000` to `0xf0000000`, simply via recompilation.

How do I handle multiple spaces in an argument list?
Multiple spaces should be treated as one space. You do not need to support quotes or any special characters other than space.

Can I enforce a maximum size on the arguments list?
You can set a reasonable limit on the size of the arguments.

5.2 Syscalls FAQ

Can I cast a `struct file *` to get a file descriptor? Can I cast a `struct thread *` to a `pid_t`?
You will have to make these design decisions yourself. Most operating systems do distinguish between file descriptors (or pids) and the addresses of their kernel data structures. You might want to give some thought as to why they do so before committing yourself.

Can I set a maximum number of open files per process?
It is better not to set an arbitrary limit. You may impose a limit of 128 open files per process, if necessary.

What happens when an open file is removed?
You should implement the standard Unix semantics for files. That is, when a file is removed any process which has a file descriptor for that file may continue to use that descriptor. This means that they can read and write from the file. The file will not have a name, and no other processes will be

able to open it, but it will continue to exist until all file descriptors referring to the file are closed or the machine shuts down.

How can I run user programs that need more than 4KB of stack space?

You may modify the stack setup code to allocate more than one page of stack space for each process. This is not required in this project.

What should happen if an `exec` fails midway through loading?

`exec` should return -1 if the child process fails to load for any reason. This includes the case where the load fails part of the way through the process (e.g. where it runs out of memory in the `multi-oom` test). Therefore, the parent process cannot return from the `exec` system call until it is established whether the load was successful or not. The child must communicate this information to its parent using appropriate synchronization, such as a semaphore, to ensure that the information is communicated without race conditions.

A User Programs

User programs are written under the illusion that they have the entire machine, which means that the operating system must manage/protect machine resources correctly to maintain this illusion for multiple processes. In Pintos, more than one process can run at a time, and as in other POSIX-like systems, each process can have multiple threads. Remember that it is individual threads, not the processes which own them, which actually execute on the CPU.

Pintos can run normal C programs, as long as they fit into memory and use only the system calls you implement. Notably, `malloc` cannot be implemented because none of the system calls required for memory allocation (e.g. `brk` and `sbrk` for memory allocation) are required for this project. Note that until you complete Project User Programs in its entirety, many programs will fail due to depending on unimplemented components. For example, until you implement Floating Point Operations, programs that use floating point operations will intermittently fail, as you won't have implemented the functionality necessary for the kernel to save and restore the floating-point registers between context switches.

The `src/examples` directory contains a few sample user programs. The `Makefile` in this directory compiles the provided examples, and you can edit it to compile your own programs as well. Pintos can load ELF executables with the loader provided for you in `userprog/process.c` — it can be informative to try and see which Linux userland utils are able to run after you've implemented Project User Programs.

Until you copy a test program to the simulated file system, Pintos will be unable to do useful work. You should create a clean reference file system disk and copy that over whenever you trash your filesys.dsk beyond a useful state, which may happen occasionally while debugging.

A.1 Source Files

`threads/thread.h`
: Contains the `struct thread` definition, which is the Pintos thread control block (TCB). The comments in this file may prove useful if you experience a kernel panic in `thread_current`.

`userprog/process.h`
: Contains the `struct process` definition, and thus that of the Pintos process control block (PCB).

`userprog/process.c`
: Contains implemetnations for the definitions in process.h. Also handles the loading of ELF binaries, starts processes, and switches page tables on context switch.

`userprog/pagedir.c`
: Manages the page tables. Pintos uses a single level paging scheme, so page directories and page tables are used interchangeably.

`userprog/syscall.c`
: Handles system calls. Currently, it only supports the `exit` syscall.

`lib/user/syscall.c`
: Provides library functions for user programs to invoke system calls from a C program. Each function uses inline assembly code to prepare the syscall arguments and invoke the system call. We do expect you to understand the calling conventions used for syscalls (also in Reference).

`lib/syscall-nr.h`
: Defines the numbers for each syscall.

`lib/float.h`
: Manages the hardware FPU.

`userprog/exception.c`
: Handle exceptions. Currently all exceptions simply print a message and terminate the process.

`userprog/gdt.c`
: The Global Descriptor Table (GDT) describes the segments in use since 80x86 is a segmented architecture. These files set up the GDT.

`userprog/tss.c`
: This file manages one particular segment, the Task-State Segment (TSS), which is used for 80x86 architectural task switching. Pintos uses the TSS only for switching stacks when a user process enters an interrupt handler, as does Linux.

A.2 Virtual Memory Layout

Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address `0` up to `PHYS_BASE`, which is defined in `threads/vaddr.h` and defaults to `0xc0000000` (3 GB). Kernel virtual memory occupies the rest of the virtual address space, from `PHYS_BASE` up to 4 GB.

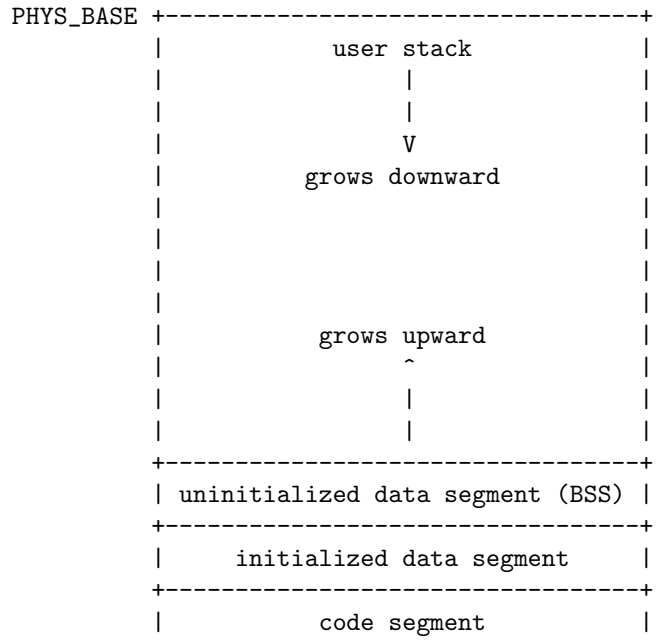
User virtual memory is per-process. When the kernel switches from one process to another, it also switches user virtual address spaces by changing the processor's page directory base register (see `pagedir_activate` in `userprog/pagedir.c`). `struct process` in `userprog/process.c` contains a pointer to a process's page table.

Kernel virtual memory is global. It is always mapped the same way, regardless of what user process or kernel thread is running. In Pintos, kernel virtual memory is mapped one-to-one to physical memory, starting at `PHYS_BASE`. That is, virtual address `PHYS_BASE` accesses physical address `0`, virtual address `PHYS_BASE + 0x1234` accesses physical address `0x1234`, and so on up to the size of the machine's physical memory.

A user program can only access its own user virtual memory. An attempt to access kernel virtual memory causes a page fault, handled by `page_fault` in `userprog/exception.c`, and the process will be terminated. Kernel threads can access both kernel virtual memory and, if a user process is running, the user virtual memory of the running process. However, even in the kernel, an attempt to access memory at an unmapped user virtual address will cause a page fault.

Typical Memory Layout

Conceptually, each process is free to lay out its own user virtual memory however it chooses. In practice, user virtual memory is laid out like this:



A.3 Accessing User Memory

As part of a system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above `PHYS_BASE`). All of these types of invalid pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

There are at least two reasonable ways to do this correctly:

- Verify the validity of a user-provided pointer, then dereference it. If you choose this route, you'll want to look at the functions in `userprog/pagedir.c` and in `threads/vaddr.h`. **This is the simplest way to handle user memory access.**
- Check only that a user pointer points below `PHYS_BASE`, then dereference it. An invalid user pointer will cause a "page fault" that you can handle by modifying the code for `page_fault` in `userprog/exception.c`. This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

In either case, you need to make sure not to "leak" resources. For example, suppose that your system call has acquired a lock or allocated memory with `malloc`. If you encounter an invalid user pointer afterward, you must still be sure to release the lock or free the page of memory. If you choose to verify user pointers before dereferencing them, this should be straightforward. It's more difficult to handle if an invalid pointer causes a page fault, because there's no way to return an error code from a memory access. Therefore, for those who want to try the latter technique, we'll provide a little bit of helpful code:

```
/* Reads a byte at user virtual address UADDR. UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault occurred. */
static int get_user (const uint8_t *uaddr) {
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:" : "=&a" (result) : "m" (*uaddr));
  return result;
}

/* Writes BYTE to user address UDST. UDST must be below PHYS_BASE. Returns
   true if successful, false if a segfault occurred. */
static bool put_user (uint8_t *udst, uint8_t byte) {
  int error_code;
  asm ("movl $1f, %0; movb %b2, %1; 1:" : "=&a" (error_code), "=m" (*udst) : "q" (byte));
  return error_code != -1;
}
```

Each of these functions assumes that the user address has already been verified to be below `PHYS_BASE`. They also assume that you've modified `page_fault` in `userprog/exception.c` so that a page fault in the kernel merely sets `eax` to `0xffffffff` and copies its former value into `eip`.

If you do choose to use the second option (i.e. rely on the processor's MMU to detect bad user pointers), do not feel pressured to use the `get_user` and `put_user` functions from above. There are other ways to modify

the page fault handler to identify and terminate processes that pass bad pointers as arguments to system calls, some of which are simpler and faster than using `get_user` and `put_user` to handle each byte.

A.4 80x86 Calling Convention

This section summarizes important points of the convention used for normal function calls on 32-bit 80x86 implementations of Unix. Some details are omitted for brevity.

1. The caller pushes each of the function's arguments on the stack one by one, normally using the `push` x86 instruction. Arguments are pushed in right-to-left order.

   The stack grows downward: each push decrements the stack pointer, then stores into the location it now points to, like the C expression `*(--sp) = value`.

2. The caller pushes the address of its next instruction (i.e. returna ddress) on the stack and jumps to the first instruction of the callee. A single 80x86 instruction, `call`, does both.

3. The callee executes. When it takes control, the stack pointer points to the return address, the first argument is just above it, the second argument is just above the first argument, and so on.

4. If the callee has a return value, it stores it into register `eax`.

5. The callee returns by popping the return address from the stack and jumping to the location it specifies, using the 80x86 `ret` instruction.

6. The caller pops the arguments off the stack.

Consider a function `f` that takes three int arguments. This diagram shows a sample stack frame as seen by the callee at the beginning of step 3 above, supposing that `f` is invoked as `f(1, 2, 3)`. The initial stack address is arbitrary.

```
                          +----------------+
                0xbffffe7c |        3       |
                0xbffffe78 |        2       |
                0xbffffe74 |        1       |
stack pointer --> 0xbffffe70 | return address |
                          +----------------+
```

A.5 Startup

The Pintos C library for user programs designates `_start`, in `lib/user/entry.c`, as the entry point for user programs. This function is a wrapper around `main` that calls `exit` if `main` returns.

```
void _start (int argc, char *argv[]) {
  exit (main (argc, argv));
}
```

The kernel must put the arguments for the initial function on the stack before it allows the user program to begin executing. The arguments are passed in the same way as the normal calling convention (see 80x86 Calling Convention).

Consider how to handle arguments for the following example command: `/bin/ls -l foo bar`. First, break the command into words: `/bin/ls`, `-l`, `foo`, `bar`. Place the words at the top of the stack. Order doesn't matter, because they will be referenced through pointers.

Then, push the address of each string plus a null pointer sentinel, on the stack, in right-to-left order. These are the elements of `argv`. The null pointer sentinel ensures that `argv[argc]` is a null pointer, as required by the C standard. The order ensures that `argv[0]` is at the lowest virtual address. The x86 ABI requires that `esp` be aligned to a 16-byte boundary at the time the `call` instruction is executed (e.g., at the point where

all arguments are pushed to the stack), so make sure to leave enough empty space on the stack so that this is achieved.

Then, push `argv` (the address of `argv[0]`) and `argc`, in that order. Finally, push a fake "return address": although the entry function will never return, its stack frame must have the same structure as any other.

The table below shows the state of the stack and the relevant registers right before the beginning of the user program, assuming `PHYS_BASE` is `0xc0000000`.

Address	Name	Data	Type
0xbffffffc	argv[3][...]	bar\0	char[4]
0xbffffff8	argv[2][...]	foo\0	char[4]
0xbffffff5	argv[1][...]	-l\0	char[3]
0xbfffffed	argv[0][...]	/bin/ls\0	char[8]
0xbfffffec	stack-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

In this example, the stack pointer would be initialized to `0xbfffffcc`.

As shown above, your code should start the stack at the very top of the user virtual address space, in the page just below virtual address `PHYS_BASE`.

You may find the non-standard `hex_dump()` function, declared in `<stdio.h>`, useful for debugging your argument passing code. Here's what it would show in the above example.

```
bffffc0                                      00 00 00 00 |            ....|
bffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 00 2f 62 69 |............./bi|
bfffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

B Floating Point

When you first try to issue floating point instructions on the processor, you may run into a "Device Not Found" exception. This is because the Pintos starter code intializes the CR0[5] register to indicate that there is no FPU by setting the `CR0_EM` bit. You must update this in `threads/start.S` to remove the flag and indicate to the processor that the FPU is present.

```
- orl $CR0_PE | CR0_PG | CR0_WP | CR0_EM, %eax
+ orl $CR0_PE | CR0_PG | CR0_WP, %eax
```

B.1 Initialization

Unlike GPRs, the x86 FPU is different since it doesn't have specific floating point registers that are independently addressable. Instead, it has a stack of registers, and the programmer interacts with the FPU by pushing and popping from the stack. For example, to compute the square root of π, you would first push π onto the stack, and then run the `fsqrt` instruction. This instruction will pop the top of the stack, compute the square root of that value, and push it onto the top of the stack. Now, $\sqrt{\pi}$ is at the top of the stack.

Newly-created threads and processes generally cannot assume anything about the contents of GPRs, meaning GPRs do not have to be initialized to any particular values when a new thread or process is created. On the contrary, newly-created threads and processes in general should be able to assume that the FPU is clean (i.e. initialized/reset properly), meaning the FPU needs to be initialized at the operating system startup, thread/process creation, and thread/context switches (e.g. interrupts, syscalls).

The idea of saving FPU registers is very similar to how the GPRs are saved during a thread switch or context switches. As a result, it will be helpful to understand interrupt handling code in `threads/interrupt.h` and `threads/intr-stubs.S` for context switches, `threads/switch.h` and `threads/switch.S` for thread switching. In particular, focus on the data structures `struct intr_frame` and `struct switch_threads_frame` and methods `switch_threads`, `intr_entry`, and `intr_exit`.

Implementation wise, saving GPRs and floating-point registers will differ since you will **not be interacting with each FPU register individually**. Instead, you will be working with the 108-byte FPU save state as a whole which looks like this.

Size	Data
2	Control Word
2	(unused)
2	Status Word
2	(unusued)
2	Tag Word
2	(unused)
4	Instruction Pointer
2	Code Segment
2	(unused)
4	Operand address
2	Data Segment
2	(unused)
10	ST(0)
10	ST(1)
10	ST(2)
10	ST(3)
10	ST(4)
10	ST(5)
10	ST(6)
10	ST(7)

[5] https://en.wikipedia.org/wiki/Control_register#CR0

This diagram is mainly for reference purposes, meaning it's not necessary to understand each individual component of the FPU save state. Instead, you should **figure out ways to interact with this FPU save state as a whole**.

During the operating system startup, the existing set of FPU registers don't need to be saved since no other process or thread is using them. However, when initializing the FPU for another thread/process, you must make sure the current thread (i.e. the thread that is creating the new thread/process) does not lose it's own floating point data. A simple way to accomplish this is by saving the current FPU registers into a temporary location (e.g. local variable), initializing a clean state of FPU registers, save these FPU registers into the desired destination, and restore the registers from the aforementioned temporary location.

B.2 Assembly

Just like when using GPRs, you must use assembly code to interact with the FPU. We recommend you read through floating-point control instructions (table 3-18)[6] that you can perform on an FPU.

As mentioned before, **you will not be working with the individual FPU registers but rather the entire FPU save state as a whole**. Therefore, if you find yourself trying to work with each `st` register individually, take a step back and reread through the aforementioned floating-point control instructions.

When you are writing assembly, you may either do so directly in `.S` files or in-line among C code using `asm`. Most likely, you will have to do both. We recommend you read through a guide on `asm`[7] to familiarize yourself. You may also find it helpful to reference some other parts of the code that use `asm` (e.g. `start_process` in `userprog/process.c`). Using `asm` can seem daunting, but keep in mind, at its core, it's just writing assembly instructions instead of C!

B.3 Syscall

As a chance for you to see floating point operations in practice and for us to test your code, you will also need to implement the `compute_e` syscall. This system call computes an approximation of e as $e \approx \sum_{i=0}^{n} \frac{1}{n!}$ using the Taylor series for the constant e. Most of the mathematical logic for implementing this system call has been done for you in `sys_sum_to_e` in `lib/float.c`. Once you validate the argument `int n` from the system call, you can simply store the result of `sys_sum_to_e` in the `eax` register for return. This system call is not intended to be difficult to implement correctly.

B.4 Testing

Our testing code uses an extremely simplified version of `stdio.h` to print floating point values. You can see how it is implemented in `lib/stdio.c`. As a result, printing very small or very large floating point numbers will cause errors. Our code only allows floating point values of the form `<whole>.<decimal>`, where `<whole>` and `<decimal>` each fit into a standard 32-bit integer. If you intend to write your tests for floating-point operations, please keep this in mind.

[6] https://docs.oracle.com/cd/E18752_01/html/817-5477/eoizy.html
[7] https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

C System Calls

One way that the operating system can regain control from a user program is **external interrupts** from timers and I/O devices. These are "external" interrupts, because they are caused by entities outside the CPU. The operating system also deals with **software exceptions**, which are events that occur in program code. These can be errors such as a page fault or division by zero. Exceptions are also the means by which a user program can request **system calls (syscalls)** (i.e. services) from the operating system.

In the 80x86 architecture, the `int` instruction is the most commonly used means for invoking system calls. This instruction is handled in the same way as other software exceptions. In Pintos, user programs invoke `int $0x30` to make a system call. The system call number and any additional arguments are expected to be pushed on the stack in the normal fashion before invoking the interrupt (see 80x86 Calling Convention).

Thus, when the system call handler `syscall_handler` gets control, the system call number is in the 32-bit word at the caller's stack pointer, the first argument is in the 32-bit word at the next higher address, and so on. The caller's stack pointer is accessible to `syscall_handler` as the `esp` member of the `struct intr_frame` passed to it. `struct intr_frame` is on the kernel stack.

The 80x86 convention for function return values is to place them in the |eax— register. System calls that return a value can do so by modifying the `eax` member of `struct intr_frame`.

Try to avoid writing large amounts of repetitive code for implementing system calls. Each system call argument, whether an integer or a pointer, takes up 4 bytes on the stack. You should be able to take advantage of this to avoid writing much near-identical code for retrieving each system call's arguments from the stack.

C.1 Process Control

`int practice (int i)`
: A "fake" syscall designed to get you familiar with the syscall interface This syscall increments the passed in integer argument by 1 and returns it to the user.

`void halt (void)`
: Terminates Pintos by calling `shutdown\_power\_off` function in `devices/shutdown.h`. This should be seldom used, because you lose some information about possible deadlock situations, etc.

`void exit (int status)`
: Terminates the current user program, returning status to the kernel. If the process's parent waits for it (see below), this is the status that will be returned. Conventionally, a status of 0 indicates success and nonzero values indicate errors. Every user program that finishes in the normal way calls `exit` —even a program that returns from `main` calls `exit` indirectly (see Startup). In order to make the test suite pass, you need to print out the exit status of each user program when it exits. The format should be `%s: exit(%d)` **followed by a newline** where the process name and exit code respectively subsitute `%s` and `%d` .

`pid_t exec (const char *cmd_line)`
: Runs the executable whose name is given in `cmd_line`, passing any given arguments, and returns the new process's program id (`pid`). If the program cannot load or run *for any reason*, return -1. Thus, the parent process cannot return from the exec until it knows whether the child process successfully loaded its executable. You must use appropriate synchronization to ensure this.

  Keep in mind `exec` is different from Unix `exec`. It can be thought of as a Unix `fork` + Unix `exec`.

`int wait (pid_t pid)`
: Waits for a child process `pid` and retrieves the child's exit status. If `pid` is still alive, waits until it terminates. Then, returns the status that `pid` passed to exit. If `pid` did not call `exit` but was terminated by the kernel (e.g. killed due to an exception), `wait` must return -1. It is perfectly legal for a parent process to wait for child processes that have already terminated by the time the parent

calls wait, but the kernel must still allow the parent to retrieve its child's exit status, or learn that the child was terminated by the kernel.

`wait` must fail and return -1 immediately if any of the following conditions are true.

- `pid` does not refer to a direct child of the calling process. `pid` is a direct child of the calling process if and only if the calling process received `pid` as a return value from a successful call to `exec`. Note that children are not inherited: if A spawns child B and B spawns child process C, then A cannot wait for C, even if B is dead. A call to `wait(C)` by process A must fail. Similarly, orphaned processes are not assigned to a new parent if their parent process exits before they do.
- The process that calls `wait` has already called `wait` on `pid`. That is, a process may wait for any given child at most once.

Processes may spawn any number of children, wait for them in any order, and may even exit without having waited for some or all of their children. Your design should consider all the ways in which waits can occur. All of a process's resources, including its `struct thread`, must be freed whether its parent ever waits for it or not, and regardless of whether the child exits before or after its parent.

You must ensure that Pintos does not terminate until the initial process exits. The supplied Pintos code tries to do this by calling the `process_wait` function in `userprog/process.c` from `main` function in `threads/init.c`. We suggest that you implement the `process_wait` funciton in `userprog/process.c` according to the docstring and then implement `wait` in terms of `process_wait`.

Implementing wait requires considerably more work than any of the rest.

C.2 File Operation

`bool create (const char *file, unsigned initial_size)`
Creates a new file called `file` initially `initial_size` bytes in size. Returns true if successful, false otherwise. Creating a new file does not open it: opening the new file is a separate operation which would require a open system call.

`bool remove (const char *file)`
Deletes the file called `file`. Returns true if successful, false otherwise. A file may be removed regardless of whether it is open or closed, and removing an open file does not close it. See this section of the FAQ for more details.

`int open (const char *file)`
Opens the file called `file`. Returns a nonnegative integer handle called a "file descriptor" (fd), or -1 if the file could not be opened.

File descriptors numbered 0 and 1 are reserved for the console: 0 (`STDIN_FILENO`) is standard input and 1 (`STDOUT_FILENO`) is standard output. `open` will never return either of these file descriptors, which are valid as system call arguments only as explicitly described below.

Each process has an independent set of file descriptors. File descriptors in Pintos are not inherited by child processes.

When a single file is opened more than once, whether by a single process or different processes, each open returns a new file descriptor. Different file descriptors for a single file are closed independently in separate calls to `close` and they do not share a file position.

`int filesize (int fd)`
Returns the size, in bytes, of the file open as `fd`.

`int read (int fd, void *buffer, unsigned size)`
Reads `size` bytes from the file open as `fd` into `buffer`. Returns the number of bytes actually read (0 at end of file), or -1 if the file could not be read (due to a condition other than end of file). `STDIN_FILENO` reads from the keyboard using the `input_getc` function in `devices/input.c`.

`int write (int fd, const void *buffer, unsigned size)`

Writes `size` bytes from `buffer` to the open file `fd`. Returns the number of bytes actually written, which may be less than `size` if some bytes could not be written.

Writing past end-of-file would normally extend the file, but file growth is not implemented by the basic file system. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written, or 0 if no bytes could be written at all.

Fd 1 writes to the console. Your code to write to the console should write all of `buffer` in one call to the `putbuf` function `lib/kernel/console.c`, at least as long as `size` is not bigger than a few hundred bytes. It is reasonable to break up larger buffers. Otherwise, lines of text output by different processes may end up interleaved on the console, confusing both human readers and our autograder.

`void seek (int fd, unsigned position)`

Changes the next byte to be read or written in open file `fd` to position, expressed in bytes from the beginning of the file. Thus, a position of 0 is the file's start.

A seek past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. However, in Pintos files have a fixed length until Project File Systems is complete, so writes past end-of-file will return an error. These semantics are implemented in the file system and do not require any special effort in the syscall implementation.

`unsigned tell(int fd)`

Returns the position of the next byte to be read or written in open file `fd`, expressed in bytes from the beginning of the file.

`void close (int fd)`

Closes file descriptor `fd`. Exiting or terminating a process implicitly closes all its open file descriptors, as if by calling this function for each one.

C.3 Floating Point

`double compute_e(int n)`

Similar to the `practice` syscall in that it's a "fake" system call that doesn't exist in any modern OS. You will implement this to see an example of floating point instructions in use. This system call simply computes n terms of the Taylor summation of e. Most of the work of this system call has been implemented for you.

D Advice

You should read through and understand as much of the Pintos source code that you mean to modify before starting work on project. In a sense, this is why we have you write a design document; it should be obvious that you have a good understanding, at the very least at a high level, of files such as `userprog/process.c`. We see groups in office hours who are really struggling due to a conceptual misunderstanding that has informed the way they designed their implementations and thus has caused bugs when trying to actually implement them in code.

You should learn to use the advanced features of GDB. Often times, debugging your code usually takes longer than writing it. However, a good understanding of the code you are modifying can help you pinpoint where the error might be; hence, again, we strongly recommend you to read through and understand at least the files you will be modifying in this project (with the caveat that it is a large codebase, so don't overwhelm yourself).

These projects are designed to be difficult and even push you to your limits as a systems programmer, so plan to be busy and have fun!

D.1 Group Work

In the past, many groups divided each assignment into pieces. Then, each group member worked on his or her piece until just before the deadline, at which time the group reconvened to combine their code and submit. This is a bad idea. We do not recommend this approach. Groups that do this often find that two changes conflict with each other, requiring lots of last-minute debugging. Some groups who have done this have turned in code that did not even compile or boot, much less pass any tests.

Instead, we recommend integrating your team's changes early and often, using git. This is less likely to produce surprises, because everyone can see everyone else's code as it is written, instead of just when it is finished. These systems also make it possible to review changes and, when a change introduces a bug, drop back to working versions of code.

We also encourage you to program in pairs, or even as a group. Having multiple sets of eyes looking at the same code can help avoid subtle bugs that would've otherwise been very difficult to debug.

D.1.1 Meetings

We encourage each group to have regular meetings (e.g. twice a week) to make sure everyone is on the same page. In-person meetings are generally much more productive, since people tend to be more attentive.

If you're meeting through a call (e.g. Zoom), we recommend you enable live transcription[8]. Moreover, you should take detailed notes during each meeting on a central document to reference back to.

D.2 Development

D.2.1 Compiler Warnings

Compiler warnings are your friend! When compiling your code, we have configured GCC to emit a variety of helpful warnings when it detects suspicious or problematic conditions in your code (e.g. using the value of an uninitialized variable, comparing two values of different types, etc). When you run `make` to compile your code, by default it echoes each command it is executing, which creates a huge amount of output that you usually don't care about, obscuring compiler warnings. To hide this output and show only compiler warnings (in addition to anything else printed to standard error by the commands `make` is running), you can run `make -s`. The `-s` flag tells `make` to be silent instead of echoing every command. **Do NOT pass the `-s` flag when running `make check`** or you won't see your test results! Only use the `-s` flag when compiling your code.

[8] https://support.zoom.us/hc/en-us/articles/207279736-Managing-closed-captioning-and-live-transcription

If your code is buggy, the first thing you should do is check to see if the compiler is emitting any warnings. While sometimes warnings might be emitted for code that is perfectly fine, in general it's best to remedy your code to fix warnings whenever you see them.

D.2.2 Faster Compilation

Depending on the machine you're using, compiling the Pintos code may take a while to complete. You can speed this up by using `make`'s `-j` flag to compile several files in parallel. For maximum effectiveness, the value provided for `-j` (which effectively specifies how many things to compile in parallel) should be equal to the number of (logical) CPUs on your machine which can be found by running the `nproc` command in your shell. You can combine all of this into one command by running `make -j $(nproc)` instead of running just `make` whenever you want to compile your code.

Please be warned however that **you should only pass the `-j` flag to `make` when compiling your code**. You should *not* use it when running your tests with `make check`. While it is actually safe to do so for Project User Programs, this is not the case for other projects, so it's best not to get into the habit of it.

D.2.3 Repeated Commands

You'll often find yourself having to type in the same/similar long commands (e.g. `PINTOS_DEBUG`, `loadusersymbols`). Instead of retyping these every time or copying and pasting, you can use reverse-i-search using `Ctrl-R`. This will alow you to quickly search through your command history. If there are multiple matches, you can cycle through them by pressing `Ctrl-R` repeatedly.

Within GDB, you can specifically shorten your commands which GDB will automatically match as long as there are no ambiguities. For instance, you can type `deb` instead of `debugpintos` and `n` instead of `next`. Moreover, pressing enter without typing any command will repeat the command, which is useful for stepping through the code.

D.2.4 Hail Mary

Rarely you may find yourself in a bizarre situation where the behavior of your kernel isn't changing even though you're certain you've changed your code in a way that should produce an obvious effect. If this occurs, you can destroy all compiled objects and caches, and restore your current terminal and shell parameters to sane values by running the following.

```
hash -r
stty sane
cd ~/code/group/pintos/src
make clean
```

Once you've done the above, recompile your code and try whatever it was you were doing again.