CIS 5480 - Spring 2023 - Project 2

PennOS: A User-level UNIX-like Operating System

"UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity."
– Dennis Ritchie

Prof. Boon Thau Loo

Milestone 0	:	March 20
Milestone 1	:	March 27
Due	:	April 14 @ 10:00 PM

Directions

This is a group project. You must work in groups of four or five. You are expected to follow along to Ed as updates and clarifications made there will be reflected in the final grading. You may reuse code from previous projects, **but only code you wrote**.

> **You are forbidden to reuse any code obtained from the internet or from previous semesters. Violators will be sent to the Office of Student Conduct and will receive a failing grade for the course.**

Overview

In this assignment you will implement *PennOS*, your own UNIX-like operating system. PennOS is designed around subsystems that model those of standard UNIX. This will include programming a basic priority scheduler, FAT file system, and user shell interactions. Unlike a *real* operating system, you will not be required to actually boot on hardware; rather, your PennOS will run as a guest OS within a single process running on a host OS.

How to use this document

The following specification provides a road map for completing this project; however, as you develop your code, you may find it necessary to expand upon the specification. **Feel free to add additional shell/user level functions to support debugging, as long as you adhere to the provided interfaces in this document.** Not only is this encouraged, but it is expected. If you are ever in doubt about a design decision, ask your friendly TAs, and be sure to document such changes in your `README` and companion document.

1 Specification

There are two states in which an operating system exists: (1) *kernel land* and (2) *user land*. During execution, an operating system switches between these two states continuously. As an example, consider what happens when a program issues a system call. First, the system call is executed in user land which hits a hook; that is, the running process actually calls the system call. Next, the operating system must handle the system call, switching from user to kernel land. Once the system call completes, the operating system returns control to the calling process, returning back to user land. Unlike in a real operating system, however, this separation between user land and kernel land is simply taken on faith. You are not required to implement a mechanism to enforce this separation.

In the previous projects, you have interacted with the operating system at the user land level, and in this project, you will take a peek behind the curtain at kernel land. Well, not exactly, but you will simulate the basic functionality of an operating system by programming your own operating system simulator *PennOS*. Using the `ucontext` library, you will implement a basic priority scheduler; additionally, you will implement a FAT file system for your operating system to mount, and a basic shell and programming API for a user to interact with your operating system.

Unlike a *real* operating system, your PennOS is not required to boot on hardware (or handle devices in general); instead, it will run as a single process on a host OS. The `ucontext` library is similar to a threading API in that it allows one process to split its resources across multiple instances. `ucontexts` do not provide a scheduler like you may be used to with traditional threading, and your first task will be implementing a `SIGALRM`-based scheduler for context switching.

Another critical part of an operating system is handling file reads and writes. Your operating system will mount a single file system, PennFAT: a simple file system implementation based on FAT. Your PennFAT implementation will be stored within a single file on the host file system, and will be mounted by PennOS in a *loopback* like manner. Additionally, unlike traditional file systems, PennFAT is only required to handle files within a single top level directory. You are required to allow the creation, modification, and removal of files under the top level directory.

The last part of your operating system is providing user land interaction via a simple shell. You will program this shell using the user land system calls providing by PennOS. Your shell will provide job control, `stdin`/`stdout` redirection, and a functional set of built-in commands for testing and exploring your operating system.

One last note: **this is a long and complex project that will take you many, many hours to complete.** This document can only provide you with a primer of the innumerable challenges you will face; read it carefully, but be sure to use other resources as well. In particular, read the manual pages, and ask questions of your friendly TAs. Likely, this will be the largest program you have yet written as a student, so divide and conquer and plan ahead so that the pieces fit together. Remember, a lot of small, easy programs equal one large, complex program. Don't try to conquer the entire project in one go: it is organized such that splitting up tasks into discrete components and writing modular code are strategies that will be rewarded.

1.1 Function Terminology

Your operating system will provide a number of different functions and interfaces. The following symbols indicate how the functions are to be used:

- **(K)**: indicates a kernel level function that may only be called from the kernel side of the operating system.

- **(U)**: indicates a user level function that may only be called from the user side of the operating system. These are your operating system's system calls.
- **(S)**: indicates a built in program that can be called directly from the shell.

The function definitions provided are suggestions for you to build upon. You may find it useful to add additional kernel/system/user level functions as you see fit as long as they are in the spirit of the assignment.

1.2 PennOS Processes/Threads

Your PennOS operating system will run as a single process on the host OS (e.g., Linux on your Docker Container). That is, each of the "processes" in PennOS is really the same process as PennOS according to the host operating system, but within PennOS, each process will be separated into context threads that are independently scheduled by PennOS. To accomplish this task you will be using the `ucontext` library: **If you are issuing calls to `fork(2)`, `exec(2)`, or `wait(2)`, you are doing something very wrong**. Despite being context threads[1], your operating system will treat them like processes and will organizing them into **p**rocess **c**ontrol **b**locks (or *PCB*'s).

A PCB is a structure that describes all the needed information about a running process (or thread). One of the entries will clearly be the `ucontext` information, but additionally, you will need to store the thread's process id, parent process id, children process ids, open file descriptors, priority level, etc. Refer to Steven's *Advanced Programming in the UNIX Environment* and Tanenbaum's *Modern Operating Systems* for more information about data normally stored in a PCB. **You must describe your PCB structure in your `companion document`**.

1.2.1 Process Related Required Functions

Your operating system must provide the following user level functions for interacting with PennOS process creation:

- `pid_t p_spawn(void (*func)(), char *argv[], int fd0, int fd1)` (U) forks a new thread that retains most of the attributes of the parent thread (see `k_process_create`). Once the thread is spawned, it executes the function referenced by `func` with its argument array `argv`. `fd0` is the file descriptor for the input file, and `fd1` is the file descriptor for the output file. It returns the pid of the child thread on success, or -1 on error.

- `pid_t p_waitpid(pid_t pid, int *wstatus, bool nohang)` (U) sets the calling thread as blocked (if `nohang` is false) until a child of the calling thread changes state. **It is similar to Linux `waitpid(2)`**. If `nohang` is true, `p_waitpid` does not block but returns immediately. `p_waitpid` returns the pid of the child which has changed state on success, or -1 on error.

- `int p_kill(pid_t pid, int sig)` (U) sends the signal `sig` to the thread referenced by `pid`. It returns 0 on success, or -1 on error.

- `void p_exit(void)` (U) exits the current thread unconditionally.

Additionally, your operating system will have the following kernel level functions:

[1] We use "context" and "thread" interchangeably in this document to describe a PennOS process.

- `k_process_create(Pcb *parent)` (K) create a new child thread and associated PCB. The new thread should retain much of the properties of the parent. The function should return a reference to the new PCB.
- `k_process_kill(Pcb *process, int signal)` (K) kill the process referenced by `process` with the signal `signal`.
- `k_process_cleanup(Pcb *process)` (K) called when a terminated/finished thread's resources needs to be cleaned up. Such clean-up may include freeing memory, setting the status of the child, *etc.*

1.2.2 Zombies and Waiting

As processes complete, it may not be the case that their parent threads can wait on them immediately. If that is the case, you must queue up these threads so that the parent may wait on them in the future. These threads are *Zombies*. You will likely have a zombie queue for each thread, referenced in the PCB. If at any time the parent thread exits without waiting on zombie children, the zombie children should immediately die, as well as non-zombie children threads. Note, this is similar to how `INIT` inherits orphan child processes and kills them off.

Additionally, as noted above, other child process state changes can cause a `p_waitpid()` to return. In UNIX, a child process that transitions from running to stopped would issue a `SIGCHLD` signal. Your operating system also should have functionality for parent process to learn of similar state changes. In your PennOS kernel land implementation of `p_waitpid()` you will find it useful to maintain other queues of "waitable" children, not just zombie children.

1.2.3 Signals in PennOS

Your operating system will not have traditional signals and signal handlers. (However, the host operating system may deliver signals that you must handle.) Instead, signaling a PennOS process indicates to PennOS that it should take some action related to the signaled thread, such as change the state of a thread to stopped or running. Your operating system will minimally define the following signals:

- `S_SIGSTOP`: a thread receiving this signal should be stopped
- `S_SIGCONT`: a thread receiving this signal should be continued
- `S_SIGTERM`: a thread receiving this signal should be terminated

If you would like to add additional signals, be sure to document them and their functionality in your companion document file.

1.2.4 Process Statuses

PennOS will provide *at least* the following user level functions/macros that will return booleans based on the status returned from `p_waitpid`.

- `W_WIFEXITED(status)`: return true if the child terminated normally, that is, by a call to `p_exit` or by returning.
- `W_WIFSTOPPED(status)`: return true if the child was stopped by a signal.
- `W_WIFSIGNALED(status)`: return true if the child was terminated by a signal, that is, by a call to `p_kill` with the `S_SIGTERM` signal.

1.3 Programming with User Contexts

`ucontext` is a basic thread-like library provided on most UNIX systems. Essentially, it allows a user to isolate some portion of code execution within a *context* and switch between them. On the course website, we have provided a sample program that demonstrates how to switch between contexts in a round robin fashion. A high-level description of context creation and execution is provided below, and more information can be found in the manual.

First, a `ucontext` structure must be initialized with a call to `getcontext(2)`. The structure will have the following fields (and more not shown):

```
typedef struct ucontext {
  struct ucontext *uc_link;
  sigset_t         uc_sigmask;
  stack_t          uc_stack;
  ...
} ucontext_t;
```

You still need to set the structure values above: `uc_link` is the next context to run when this context completes[2]; `uc_sigmask` is a signal mask for blocking signals in this context; and, `uc_stack` is the execution stack used by this context. For a description of the `uc_stack` structure reference the manual for `sigaltstack(2)`. Setting these values is still insufficient to execute the context, and you still need to set up the function that will be called when the context is set or swapped. This done by a call to `makecontext(2)`, and it is well described in the manual. A context is switched by using either `setcontext(2)` or `swapcontext(2)`, which either directly sets the context, or sets and also saves the state of the running context, respectively.

We are leaving much of the details for you to learn on your own, but a good place to start is with a *Hello World* program for `ucontext`. We have provided one in the Appendix (item A). Try editing that programming and adding new features. Here are some mini-exercise you might want to try: What happens if you want to switch back to the main program after printing "Hello World"? Can you write a program that alternates between two functions indefinitely? What happens when a signal is delivered? How do signals affect the execution of a context? How do you track the current context?

1.4 Priority Scheduler

Perhaps the most critical part of any operating system is the scheduler. This is the part of your operating system that chooses which program to run next. As described in class, most operating systems, including Linux, use a priority queue based scheduler. In your PennOS, you will also implement a priority scheduler, although it will be a much more simplified version with a fixed quanta.

1.4.1 Clock Ticks

You will schedule a `SIGALRM` signal to be delivered to your operating system every 100 milliseconds. We refer to this event as a *clock tick*, and on every clock tick the operating system will switch in and execute the scheduler, which will then choose which process to run next. This may be any process that is *runnable* (i.e., not zombied, blocked, nor stopped) to execute, including the shell. To set an alarm timer at millisecond granularity, refer to `setitimer(2)`. Note that since your operating system is relying on `SIGALRM`, **non-kernel functions may not block `SIGALRM`**.

[2] What might you want to set `uc_link` to?

1.4.2 Priority Queues

Your operating system will have three priority levels: -1, 0, 1. As in standard UNIX, the lower a priority level, the more heavily the "process" should be scheduled. Child threads created by `k_process_create` should have a default priority level of 0, unless otherwise specified. For example, the shell should execute with priority level -1 because it is interactive.

Your priority queues will be relative. Threads scheduled with level -1 should run 1.5 times more often as threads scheduled with priority level 0, which run 1.5 times more often as threads scheduled with priority level 1. Within each priority queue, the threads are selected in a round robin format, and **each queue must be implemented as a linked list**. You may reuse your linked list implementation from previous projects. **You must also ensure that no thread is starved**.

As an example, consider the scheduling of the following threads with these priority levels: (1) `shell`, priority level -1; (2) `sleep`, priority level 0; and (3) `cat`, priority level 0. With a 100 millisecond quanta, after 10 seconds, what is the proportion of quanta for each process? First, there are 10 quanta per second, which means a total of 100 quanta in 10 seconds. Of the available quanta, 60 quanta will be used for priority level -1, (that is the `shell`), since it must be scheduled 1.5 times as often. Of the remaining 40 quanta, 20 quanta will be used for `sleep`, and 20 quanta for `cat`.

To verify the correctness of your scheduler, you will implement a detailed logging facility that will generate a log entry for every clock tick. The specification of the log format is described in Section 1.8.

1.4.3 The Idle Process

When all processes in the scheduler queues are blocked, the scheduler needs to decide what to do. It cannot just keep waiting in a while loop until a process is ready to be scheduled, since this would spike the CPU usage of PennOS up to 100%, when it should actually be 0% (since the OS isn't actually doing anything). To get past this, you can create an additional "idle" process.

The idle process does nothing and suspends itself. To be clear, **the idle process must not run an infinite while loop**, since this would spike CPU usage to 100%. You should use `sigsuspend(2)` instead, which will suspend the idle process until a signal is delivered to it, resulting in 0% CPU usage. The idle process must be scheduled **only when all other processes are blocked** and the scheduler has no real process to schedule.

Correct usage of the idle process is as follows: The shell spawns a new process `sleep 200`. The child process for `sleep` will be blocked for 200 clock ticks, and the shell will be blocked waiting for `sleep` to terminate. At this point, all the scheduler queues are empty and the idle process can be scheduled until `sleep` completes.

Incorrect usage of the idle process is as follows: The shell spawns a new process `busy`. The shell is blocked waiting for `busy` to complete, the `busy` process is being scheduled, and so is the idle process. So the resulting CPU usage is 60%. This is incorrect. Since the scheduler has the `busy` process to schedule, the idle process must not be scheduled, and the resulting CPU usage of PennOS should increase to 100%.

1.5 Running vs. Stopped vs. Blocked vs. Zombied

Threads can exists in four states: running, stopped, blocked, or zombied. A running thread may be scheduled; however, a stopped, blocked, or zombied thread should not be. A thread should only become stopped if it was appropriately signaled via `p_kill`. A thread should only be blocked if it made a call to `p_waitpid` or `p_sleep` (see below).

1.5.1 Required Scheduling Functions

Your operating system will provide *at least* the following user level functions for interacting with the scheduler:

- `int p_nice(pid_t pid, int priority)` (U) sets the priority of the thread `pid` to `priority`.
- `void p_sleep(unsigned int ticks)` (U) sets the calling process to blocked until `ticks` of the system clock elapse, and then sets the thread to running. Importantly, `p_sleep` should not return until the thread resumes running; however, it can be interrupted by a `S_SIGTERM` signal. Like `sleep(3)` in Linux, the clock keeps ticking even when `p_sleep` is interrupted.

1.6 PennFAT File System

Your operating system will mount its own file system, PennFAT, based on FAT16. (For references, see Chapter 4.3.2 in Tanenbaum 4th edition, pages 285-286.) **You are only required to implement the root directory of PennFAT.**

1.6.1 FAT File System Regions

A FAT file system has mainly two regions: the FAT region which hosts the File Allocation Table (FAT), and the data region which stores the files (including directory files). **The block numbering of PennFAT's data region begins at 1.**

1.6.2 FAT File Systems and File System Blocks

Conceptually, you can think of a file as a sequence of fixed-size blocks and a file system as a way to find the right blocks for a file in the right order. A FAT provides a simple way to do this by storing links between physical blocks of a file. Placed at the beginning of the file system before any block, the FAT has a predetermined size and can be easily mapped into memory. This is also its greatest disadvantage: being of fixed size, the FAT also limits the size of the file system.

The FAT of PennFAT consists of a predetermined number of entries. It is specified when the filesystem is first formatted using the `mkfs` command, which dictates how the FAT and your file system as a whole should be formatted. The least significant byte (LSB) of the first entry of the FAT specifies the size of each block with the mapping displayed below:

```
   LSB   | Size (bytes)
---------+-------------
    0    |  256
---------+-------------
    1    |  512
---------+-------------
    2    |  1024
---------+-------------
    3    |  2048
---------+-------------
    4    |  4096
```

The most significant byte (MSB) of the first entry of the FAT will be used to calculate the size of the FAT, with the MSB ranging from 1-32 (numbers outside of this range will be considered an error). The size of your FAT will be

```
FAT size = block size * number of blocks in FAT
```

and with each entry of the FAT consisting of 2 bytes, the number of entries in your FAT will be

```
# of FAT entries = block size * number of blocks in FAT / 2
```

In the figure below, each row (table entry) corresponds to a block in the file system and the value (`Link`) represents the next *block number* of the file (or `0xFFFF` to denote the last block of the file). In other words, each table entry is a link to the table entry for the next block of the file.

```
 Index   | Link
---------+-------
   0     | 0x2004 <--- MSB=0x20 (32 blocks in FAT), LSB=0x04 (4K-byte block size)
---------+-------
   1     | 0xFFFF <--- Block 1 is the only block in the root directory file
---------+-------
   2     |   5    <--- File A starts with Block 2 followed by Block 5
---------+-------
   3     |   4    <--- File B starts with Block 3 followed by Block 4
---------+-------
   4     | 0xFFFF <--- last block of File B
---------+-------
   5     |   6    <--- File A continues to Block 6
---------+-------
   6     | 0xFFFF <--- last block of File A
---------+-------
  ...    |  ...
```

Like other FAT16 variants, **0** denotes a free block, and 0xFFFF the last block of a file. (Thus, the maximum possible block number for PennFAT is $2^{16} - 2$).

With each FAT entry representing a file block, there will be a total of

```
Data region size = block size * (number of FAT entries - 1)
```

We subtract 1 here because the first entry (`fat[0]`) in the FAT is used to store the filesystem formatting information, so it does not have a corresponding block. (This is also why the block numbering of the data region begins at 1.) Consequently, `fat[1]` references Block 1 which will always be the first block of the root directory file (the directory file may encompass multiple blocks as it grows).

A directory file consists of directory entries, each of which stores metadata about another file (including a directory file). PennFAT has a 64-byte fixed directory entry size. The structure of the directory entry as stored in the filesystem is as follows:

```
  char name[32];
  uint32_t size;
  uint16_t firstBlock;
  uint8_t type;
  uint8_t perm;
  time_t mtime;
  // The remaining 16 bytes are reserved (e.g., for extra credits)
```

- `char name[32]`: null-terminated file name

  `name[0]` also serves as a special marker:

  - `0`: end of directory
  - `1`: deleted entry; the file is also deleted
  - `2`: deleted entry; the file is still being used

- `uint32_t size`: number of bytes in file
- `uint16_t firstBlock`: the first block number of the file (undefined if `size` is zero)
- `uint8_t type`: the type of the file, which will be one of the following:
  - `0`: unknown
  - `1`: a regular file
  - `2`: a directory file
  - `4`: a symbolic link
- `uint8_t perm`: file permissions, which will be one of the following:
  - `0`: none
  - `2`: write only
  - `4`: read only
  - `5`: read and executable (shell scripts)
  - `6`: read and write
  - `7`: read, write, and executable
- `time_t mtime`: creation/modification time as returned by `time(2)` in Linux

By iterating over the directory files, all files in the file system can be enumerated. To find a block in the file system using a FAT entry, you will use `lseek(2)`. For example, given a FAT entry for the value 5, you can find that block in the file containing your PennFAT by the following call to `lseek`:

```
lseek(fs_fd, fat_size + block_size * 4, SEEK_SET)
```

where `fat_size` is the size of the FAT and `block_size` is the size of a block.

1.6.3 Standalone PennFAT

Your file system itself will exist as a single binary file on the host OS. In order to interact with the file system from outside of your operating system implementation (i.e. the shell), you will provide a single executable that will interact with your file system accordingly.

You will provide a single standalone program, separate from the final PennOS executable, called `pennfat`, which doesn't have any arguments. Within `pennfat`, there exist a number of commands similar to their counterparts in bash:

- `mkfs FS_NAME BLOCKS_IN_FAT BLOCK_SIZE_CONFIG`

  Creates a PennFAT filesystem in the file named `FS_NAME`. The number of blocks in the FAT region is `BLOCKS_IN_FAT` (ranging from 1 through 32), and the block size is 512, 1024, 2048, or 4096 bytes corresponding to the value (1 through 4) of `BLOCK_SIZE_CONFIG`.

- `mount FS_NAME`

  Mounts the filesystem named `FS_NAME` by loading its FAT into memory.

- `umount`

  Unmounts the currently mounted filesystem.

- `touch FILE ...`

  Creates the files if they do not exist, or updates their timestamp to the current system time.

- `mv SOURCE DEST`

  Renames `SOURCE` to `DEST`.

- `rm FILE ...`

  Removes the files.

- `cat FILE ... [ -w OUTPUT_FILE ]`

  Concatenates the files and prints them to `stdout` by default, or overwrites `OUTPUT_FILE`. If `OUTPUT_FILE` does not exist, it will be created. (Same for `OUTPUT_FILE` in the commands below.)

- `cat FILE ... [ -a OUTPUT_FILE ]`

  Concatenates the files and prints them to `stdout` by default, or appends to `OUTPUT_FILE`.

- `cat -w OUTPUT_FILE`

  Reads from the terminal and overwrites `OUTPUT_FILE`.

- `cat -a OUTPUT_FILE`

  Reads from the terminal and appends to `OUTPUT_FILE`.

- `cp [ -h ] SOURCE DEST`

  Copies `SOURCE` to `DEST`. With `-h`, `SOURCE` is from the host OS.

- `cp SOURCE -h DEST`

  Copies `SOURCE` from the filesystem to `DEST` in the host OS.

- `ls`

  List all files in the directory, similar to `ls -il` in bash. For example:

  ```
    2 -rw 30000 Nov  7 01:00 up-to-thirty-one-byte-file-name
  120 -r-  1000 Oct 31 21:59 Posix-file_name.2
  ```

  The columns above are first block number, permissions, size, month, day, time, and name.

- `chmod`

  Similar to `chmod(1)` in the VM.

The purpose of the `pennfat` program is to test your file system's functionality, independent of your PennOS. This will be needed for the second milestone as well as the final demo, so make sure you update it as you make progress or change things.

1.6.4 Loading the FAT into Memory

When mounting your PennFAT filesystem, you will likely find it useful to `mmap(2)` the FAT region directly into memory. This way you will have copy-on-write for free. Here is a code snippet (minus error checking) to get you started on this process:

```
#include <stdint.h>
#include <sys/mman.h>
...
int fs_fd = open(fs_name, O_RDWR);
uint16_t *fat = mmap(NULL, fat_size, PROT_READ | PROT_WRITE, MAP_SHARED, fs_fd, 0);
```

Now, `fat` references an array.

1.6.5 File System and Operating System Interaction

The role of the operating system is to protect the file system from corruption as well as manipulate it by reading, writing, and deleting files. Internally, the operating system will store a reference to a file descriptor (an integer) for each open file, as well as a structure indicating the mode for the file and relevant file pointers indicating where subsequent reads and writes should take place. The user side will reference a file by its file descriptor and manipulate the file via the user level interface described below.

Note that the file system-related system calls are abstraction layers and should not care about the format of the underlying file system. That is, consider what must occur when an operating system has mounted multiple file systems of different types. When there is a call to `read(2)` for a particular file descriptor, the operating system will determine on which file system variant the file lives (e.g., `FAT32`, `ext3`, etc.) and then call the appropriate file system dependent function to perform the read operation (which usually is provided by module).

PennOS must work in a similar way, particularly when considering how to handle the `stdin` and `stdout` file descriptors. Essentially, PennOS must manipulate two types of file descriptors, those for `PennFAT` and those for `stdin` and `stdout`. A user level program should be able to write to `stdout` using the same interface as it would write to a `PennFAT` file. **If a user level program is calling `read(2)`, then you are doing something wrong.**

1.6.6 Required File System Related System Calls

Your operating system will provide *at least* the following functions for file manipulation.

- `f_open(const char *fname, int mode)` (U) open a file name `fname` with the mode `mode` and return a file descriptor. The allowed modes are as follows:
  - `F_WRITE` - writing and reading, truncates if the file exists, or creates it if it does not exist. Only one instance of a file can be opened in `F_WRITE` mode; error if PennOS attempts to open a file in `F_WRITE` mode more than once
  - `F_READ` - open the file for reading *only*, return an error if the file does not exist
  - `F_APPEND` - open the file for reading and writing but does not truncate the file if exists; additionally, the file pointer references the end of the file.

  `f_open` returns a file descriptor on success and a negative value on error.[3]

[3]What characters are allowed in filenames? You may follow the POSIX standard, linked here.

- `f_read(int fd, int n, char *buf)` (U) read `n` bytes from the file referenced by `fd`. On return, `f_read` returns the number of bytes read, 0 if `EOF` is reached, or a negative number on error.
- `f_write(int fd, const char *str, int n)` (U) write `n` bytes[4] of the string referenced by `str` to the file `fd` and increment the file pointer by `n`. On return, `f_write` returns the number of bytes written, or a negative value on error.
- `f_close(int fd)` (U) close the file `fd` and return 0 on success, or a negative value on failure.
- `f_unlink(const char *fname)` (U) remove the file. [5]
- `f_lseek(int fd, int offset, int whence)` (U) reposition the file pointer for `fd` to the `offset` relative to `whence`. You must also implement the constants `F_SEEK_SET`, `F_SEEK_CUR`, and `F_SEEK_END`, which reference similar file `whence`s as their similarly named counterparts in `lseek(2)`.
- `f_ls(const char *filename)` (U) list the file `filename` in the current directory. If `filename` is `NULL`, list all files in the current directory.

You may require kernel level functions as well. Be sure to document them in your code and companion document file.

1.7 Shell

Once PennOS is booted (i.e., executed from the host OS), it will execute a shell. You will program this shell using the PennOS user interfaces described above. Although there is not strong separation between user and kernel land, **you may only use the user level functions to program your shell and programs that run in it**. That is, you may only use functions indicated with a (U).

The shell is like any other `ucontext` thread running in PennOS and should be scheduled as such, except it will always have a priority level of -1. Unlike traditional shells, it is not capable of running arbitrary programs; instead you will provide built-in programs to execute within a user context. Below are the following features your shell should provide:

- *Synchronous Child Waiting*: PennOS does not provide a means to perform asynchronous signal handling; instead, you will use a synchronous signal handler. Before prompting for the next command, your shell will attempt to wait on all children using `p_waitpid`.
- *Redirection*: Your shell must handle >, <, and >> redirection; however, you are not required to have pipelines.
- *Parsing*: You must parse command line input, but you may use your previous implementation, or the parser provided in the previous project.
- *Terminal Signaling*: You should still be able to handle signals like `CTRL-Z` and `CTRL-C`, and neither should stop nor terminate PennOS. Instead, they must be properly relayed to the appropriate thread via the user land interface. PennOS should shutdown when the shell exits (e.g., when the shell reads `EOF`).

[4] *bytes* not *chars*, these can be anything, even `\0`.

[5] Be careful how you implement this, like Linux, you should not be able to delete a file that is in use by another process.

- *Terminal Control of `stdin`*: Just as before, you must provide protection of `stdin`; however, you cannot use `tcsetpgrp(2)` since PennOS executes as a single process on the host OS process. Instead, your OS should have a way to track the terminal-controlling process. Functions that read from `stdin` (e.g., `cat`), should be stopped (by them sending a `S_SIGSTOP`) signal if they do not have control of the terminal.

The shell prompt is `$`.

1.7.1 Shell Built-ins

The following shell built-ins should run as independently scheduled PennOS processes (indicated by (S*)).

- `cat` (S*) The usual `cat` from `bash`, etc.
- `sleep n` (S*) sleep for `n` seconds.
- `busy` (S*) busy wait indefinitely.
- `echo` (S*) similar to `echo(1)` in the VM.
- `ls` (S*) list all files in the working directory (similar to `ls -il` in bash), same formatting as `ls` in the standalone PennFAT.
- `touch file ...` (S*) create an empty file if it does not exist, or update its timestamp otherwise.
- `mv src dest` (S*) rename `src` to `dest`.
- `cp src dest` (S*) copy `src` to `dest`.
- `rm file ...` (S*) remove files.
- `chmod` (S*) similar to `chmod(1)` in the VM.
- `ps` (S*) list all processes on PennOS. Display pid, ppid, and priority.
- `kill [ -SIGNAL_NAME ] pid ...` (S*) send the specified signal to the specified processes, where `-SIGNAL_NAME` is either `term` (the default), `stop`, or `cont`, corresponding to `S_SIGTERM`, `S_SIGSTOP`, and `S_SIGCONT`, respectively. Similar to `/bin/kill` in the VM.

In order to test your zombie and orphan process handling in your kernel, you must also implement the following two programs, which should be independently scheduled in your OS.

- `zombify` (S*) execute the following code or its equivalent in your API using `p_spawn`:

```c
void zombie_child() {
    // MMMMM Brains...!
    return;
}

void zombify() {
    p_spawn(zombie_child);
    while (1) ;
    return;
}
```

- `orphanify` (S*) execute the following code or its equivalent in your API using `p_spawn`:

```
void orphan_child() {
    // Please sir,
    // I want some more
    while (1) ;
}

void orphanify() {
    p_spawn(orphan_child);
    return;
}
```

These are simple shell programs that need not do anything besides this — you will use these in conjunction with your scheduler logs to demonstrate that your scheduler properly handles zombies and orphans.

The following shell built-ins should be run as the shell; that is, they should each execute as a shell sub-routine rather than as an independent process.

- `nice priority command [arg]` (S) set the priority of the `command` to `priority` and execute the command.
- `nice_pid priority pid` (S) adjust the nice level of process `pid` to priority `priority`.
- `man` (S) list all available commands.
- `bg [job_id]` (S) continue the last stopped job, or the job specified by `job_id` [6].
- `fg [job_id]` (S) bring the last stopped or backgrounded job to the foreground, or the job specified by `job_id`.
- `jobs` (S) list all jobs.
- `logout` (S) exit the shell and shutdown PennOS.

1.7.2 Shell scripts

Your shell should be capable of executing scripts. For example:

```
$ echo echo line1 >script
$ echo echo line2 >>script
$ cat script
echo line1
echo line2
$ chmod +x script
$ script > out
$ cat out
line1
line2
```

[6] Note that this does mean you will need to implement the `&` operator in your shell.

1.8 PennOS Logging

You are required to provide logging facilities for your scheduler and kernel level functionality. The logs you produce will be used for grading, but you should also use the logs as part of your debugging and development. Any additional logging events you report should fit the formats below and should be documented in your README and companion document.

All logs will have the following set format:

```
[ticks] OPERATION ARG1 ARG1 ...
```

Where `ticks` is the number of clock ticks since boot, `OPERATION` is the current scheduling procedure (e.g., swapping in a process), and the `ARGS` are the items being acted upon in the procedure. Your log file should be **tab delimited**. The following events should be logged with the following formats.

- Schedule event: The scheduling of a process for this clock tick

  ```
  [ticks] SCHEDULE PID QUEUE PROCESS_NAME
  ```

- Process life-cycle events: The creation and completion stages possible for a process.

  ```
  [ticks] CREATE PID NICE_VALUE PROCESS_NAME
  [ticks] SIGNALED PID NICE_VALUE PROCESS_NAME
  [ticks] EXITED PID NICE_VALUE PROCESS_NAME
  [ticks] ZOMBIE PID NICE_VALUE PROCESS_NAME
  [ticks] ORPHAN PID NICE_VALUE PROCESS_NAME
  [ticks] WAITED PID NICE_VALUE PROCESS_NAME
  ```

  Regarding process termination states, if a process terminates because of a signal, then the `SIGNALED` log line should appear. If a process terminates normally, then the `EXITED` log line should appear.

- Nice event: The adjusting of the nice value for a process

  ```
  [ticks] NICE PID OLD_NICE_VALUE NEW_NICE_VALUE PROCESS_NAME
  ```

- Process blocked/unblocked: The (un)blocking of a running process

  ```
  [ticks] BLOCKED PID NICE_VALUE PROCESS_NAME
  [ticks] UNBLOCKED PID NICE_VALUE PROCESS_NAME
  ```

- Process stopped/continue: The stopping of a process

  ```
  [ticks] STOPPED PID NICE_VALUE PROCESS_NAME
  [ticks] CONTINUED PID NICE_VALUE PROCESS_NAME
  ```

1.9 Arguments to PennOS

```
 ./pennos fatfs [schedlog]
```

If schedlog is not provided, logs will be written to a default file named `log` that will be created if it does not exits. Note that this log file lives on the host OS and not within a `PennFAT`. The `fatfs` argument is required and refers to the host OS file containing your PennFAT file system.

2 Maintaining the Abstraction

As PennOS seeks to have you create your own implementation of an operating system, we will not be allowing you to simply push all of the functionality down to the host operating system you are currently running on. To this end, you may not bypass your own kernel/file system in the shell or user programs by calling things like `fork(2)`, `read` etc from the shell. These should be replaced by your own kernel/file system functions like `f_read` and `p_spawn`. If you use any functions that are not in the spirit of this assignment, you will receive a ZERO.

You should not expose any kernel level (K) functions to the user (i.e. the shell). This can be done through careful inclusion of header files. For example, say you have two header/code file pairs, one named `kernel.h` and `kernel.c` that contain your (K) functions and one named `user.h` and `user.c` that contain your user functions (which call your (K) functions). In order to maintain the abstraction, you would put the line `#include "kernel.h"` in the `user.c` file instead of `user.h` so that when you include `user.h` in your shell you have access to the user level functions but not kernel level functions. You should also be careful to not expose any file system functions to the kernel and vice versa in order to avoid breaking the abstractions in between them.

On the other hand, unless otherwise forbidden, you **may** use any userspace library functions you want (such as those found in `string.h`). Libraries that implement kernel/file system functionalities described in this write-up, e.g `<pthread.h>` (except `pthread_mutex`), are forbidden.

3 Error Handling

As before, you must check the error condition of all system calls for the host OS. Additionally, you must provide an error checking mechanism for the PennOS system calls.

You should implement a header file similar to `<errno.h>`, providing your own `ERRNO` variable that is set to different integer constants based on what error occurs from a PennOS system call. You should also define a function called `p_perror` (similar to `perror(3)` in Linux) that allows the shell to pass in a message describing what error occurred, which the function then concatenates to the default error message based on your `ERRNO`. Note that the file system and the kernel should set `ERRNO` and return -1 when a PennOS system call fails, and then the shell should call `p_perror` to report the error that occurred.

You should aim to have at least one `ERRNO` value defined for each system call in your OS. **In your companion document, you must describe the error values each PennOS system call could set.** Be mindful to not break the abstraction except for host OS system calls like `malloc`, for which you can just call `perror` to report the error - no need to call your own `p_perror`.

4 Companion Document

In addition to the standard `README` file, you are also required to provide a companion document for your PennOS. This document will contain all the documentation for all PennOS system calls, their error conditions, return values, in which files they belong, and a depth-indented listing of the files (refer to the Linux `tree` command). Also, you must describe your PCB, signals, and other data structures that are used. You can consider the man pages as a guide for the kind of information expected in your companion document (although there is no need to be as verbose as the man pages). **We will only accept submission of companion documents in PDF format**. If you submit in any other format, you will **receive a 0 on that part of the project**. You also have the option of using **Doxygen** for generating this document automatically from comments in the code.

5 Memory Errors

While `valgrind` and `ucontext` do not play nice together, it is possible to have a completely `valgrind` clean PennOS. We have included a scheduler demo with the necessary measures to make `valgrind` and `ucontext` play nice. However, due to the difficulty of this assignment, **no deductions will be made for `valgrind` errors or memory leaks**. Instead, for this year, having no leaks **will be extra credit!** (And having no errors will be worth even more extra credit!) This should be very doable for those of you that have been paying attention to good practice in C coding, so please make your best effort to not ignore `valgrind` unless necessary and feel free to come to office hours if you need help with any errors. Additionally, if you ignore `valgrind`, it tends to lead to many more unpredictable errors, so it is a good idea to at least keep leaks to a minimum.

6 Developing Your Code

We highly recommend you use the course-standardized Docker Container given by course staff because all grading will be done on the course Docker Container. Please do not develop on OSX as there are significant enough differences between OSX and Unix variants such as Ubuntu Linux. If you decide to develop in a different container/machine anyway, you must compile and test your PennOS on the course-standardized Docker Container to ensure your PennOS runs as you expect.

7 What to turn in

You must provide each of the following for your submission, *no matter what.* Failure to do so may reflect in your grade.

1. `README` file. In the `README` you will provide
   - Your name and PennKey
   - A list of submitted source files
   - Extra credit answers
   - Compilation Instructions
   - Overview of work accomplished
   - Description of code and code layout
   - General comments and anything that can help us grade your code
2. Companion Document describing your OS API and functionality. **Only PDF submissions will be accepted**.
3. `Makefile`. We should be able to compile your code by simply typing `make`. If you do not know what a `Makefile` is or how to write one, ask one of the TA's for help or consult one of many on-line tutorials.
4. Your code. You may think this is a joke, but at least once a year, someone forgets to include it.
5. Your submission for the PennFAT should include both the standalone program (for our testing) and an integrated version with your Shell implementation.

8 Submission

This project will be submitted on Canvas. To submit, place **all** relevant code and documents in a directory named *project-2-group-#* where # is your group number. **You must organize your code into directories from the top-level as follows:**

- **./bin** - all compiled binaries should be placed here
- **./src** - all source code should be placed here
- **./doc** - all documentation should be place here
- **./log** - all PennOS logs should be placed here

Your code should compile from the top level directory by issuing the `make` command.

Then, from the parent directory, compress the directory into a tarball **with the same name**. Finally, copy this file to your local machine so you can upload through Canvas's web interface. E.g.

```
local:~> tar --exclude-vcs -cvaf project-2-group-\#.tar.gz project-2-group-\#
```

Replace # with your group's number. You can then upload the file from your local home directory to the Canvas website.

9 Grading Guidelines

Each group will receive 4 grade entries for this project: A pass/fail grade for the 2 project milestones and 2 grades (one for documentation, and one for actual demo + implementation). Poor performance in the milestones or demo may affect your numeric grade. In particular, the first milestone will be a general discussion regarding the design of your project. The second milestone will be a demo of the progress your group has made. That is, you should be more than **60%** complete on each of the project parts and have PennOS code that runs and at least schedules dummy programs.

Each team will receive a group grade for the development; however, individual grades may differ. This can occur due to lack of individual effort, or other group dynamics. Below is an approximate grading guideline to give you some ideas on breakdown on code complexity:

- 5% Documentation
- 45% Kernel/Scheduler
- 35% File System
- 15% Shell

Closer to the deadline, we will release the actual demo plan that you have to stick to for the demo. A separate document on extra credits will also be released by the second milestone.

Please note that general deductions may occur for a variety of programming errors including memory violations, lack of error checking, poor code organization, etc. Also, do not take the documentation lightly, it provides us (the graders) a road-map of your code, and without it, it is quite difficult to grade the implementation.

Your programs will be graded on the course-standardized Docker Container, and must execute as specified there. Although you may develop and test on your local machine, you should always test that your program functions properly there.

9.1 Milestone 0 Expectations

Requirements for the first milestone are as follows. Your group should:

- Have decided how the components of the OS are divided up and created a rough timeline as to when each feature will be completed. This will also allow your TA to hold team members accountable for what they said they would do and by when.
- Have run the given scheduler sample code `sched-demo.c` provided on Canvas. The group should be able to demonstrate an understanding of user contexts.
- Prepare a list of questions about various components of the project.
- Have sketched out high-level implementation details related to the file system, PCB, etc.

9.2 Milestone 1 Expectations

For the second milestone, we'll be looking for the following:

File System

- Functional standalone `pennfat` program that can read, write/append, and remove files.
- A demonstration of the standalone with debugging tooling (e.g., **hd**) that allows inspection of blocks in the file system.
- Demonstration of correctness of implementation (i.e. new files always use unused blocks). Be sure to demonstrate removal of files and the reuse of blocks.
- At the very least, a plan for the unfinished parts of the user-facing file API.

Kernel

- Working scheduler that can schedule processes, taking into account priorities.
- Demonstration (with logs and with top | grep [userid] to check CPU percentage usage per process) that scheduler allocates processes correctly according to priorities.
- A **shell** with I/O loop (including a `read` inside a while loop) that calls `p_spawn`, waits on the child, and logs to the logfile correctly.
- Demonstrate signaling and explain your code for the `k_process_kill` and other helper functions.

Additionally, you should make sure that your companion document has documentation for each of the files and functions you've implemented by this point in the project. You should also have descriptions of the data structures you've implemented thus far for your OS, like the PCB and file system node structs. You should update your document as you make changes to your project heading toward the final submission deadline.

10 Attribution

This is a large and complex project using arcane and compactly documented APIs. We do not expect you to be able to complete it without relying on some outside references. That said, we do expect you to struggle a lot, which is how you will learn about systems programming, by *doing things yourself*.

The primary rule to keep you safe from plagiarism/cheating in this project is to attribute in your documentation any outside sources you use. This includes both resources used to help you understand the concepts, and resources containing sample code you incorporated in your OS. The course text and APUE need only be cited in the later case.

You should also use external code sparingly. Using most of an example from the `getcontext(2)` man page is okay; using the `CreateContextsAndImplementContextSwitching()` function from *Foo's Handy Write Your Own OS Tutorial* is not (both verbatim, and as a closely followed template on structuring your OS).

A User Context: “Hello World”

```c
#include <ucontext.h>
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define STACKSIZE 4096

void f(){
  printf("Hello World\n");
}

int main(int argc, char * argv[]){

  ucontext_t uc;
  void * stack;

  getcontext(&uc);

  stack = malloc(STACKSIZE);

  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = 0;

  sigemptyset(&(uc.uc_sigmask));

  uc.uc_link = NULL;

  makecontext(&uc, f, 0);

  setcontext(&uc);
  perror("setcontext"); //setcontext() doesn't return on success

  return 0;
}
```

B Logger Examples

```
vagrant@cis548Dev:/vagrant/PennOS$ ./a.out
$ sleep 100
^Z
[1]+  Stopped     sleep 100
$ ps
PID PPID PRI STAT CMD
  1    0  -1  B   shell
  2    1   0  S   sleep
  3    1   0  R   ps
$ kill -cont 2
$ ps
PID PPID PRI STAT CMD
  1    0  -1  B   shell
  2    1   0  B   sleep
  4    1   0  R   ps
$ logout
vagrant@cis548Dev:/vagrant/PennOS$ cat log
[   0]  CREATE       1   0  shell
[   0]  NICE         1   0 -1 shell
[   1]  SCHEDULE     1  -1  shell
[  51]  CREATE       2   0  sleep
[  51]  BLOCKED      1  -1  shell
[  51]  SCHEDULE     2   0  sleep
[  51]  BLOCKED      2   0  sleep
[  73]  STOPPED      2   0  sleep
[  73]  UNBLOCKED    1  -1  shell
[  73]  SCHEDULE     1  -1  shell
[  73]  WAITED       2   0  sleep
[ 211]  CREATE       3   0  ps
[ 211]  BLOCKED      1  -1  shell
[ 211]  SCHEDULE     3   0  ps
[ 211]  EXITED       3   0  ps
[ 211]  ZOMBIE       3   0  ps
[ 211]  UNBLOCKED    1  -1  shell
[ 211]  SCHEDULE     1  -1  shell
[ 211]  WAITED       3   0  ps
[ 382]  CONTINUED    2   0  sleep
[ 382]  WAITED       2   0  sleep
[ 402]  CREATE       4   0  ps
[ 402]  BLOCKED      1  -1  shell
[ 402]  SCHEDULE     4   0  ps
[ 402]  EXITED       4   0  ps
[ 402]  ZOMBIE       4   0  ps
[ 402]  UNBLOCKED    1  -1  shell
[ 402]  SCHEDULE     1  -1  shell
[ 402]  WAITED       4   0  ps
[ 470]  EXITED       1  -1  shell
[ 470]  ORPHAN       2   0  sleep
vagrant@cis548Dev:/vagrant/PennOS$
```