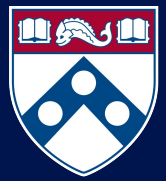
CIS 5480
PennOS Lecture

Tuesday, October 21, 2025



Penn Engineering
University of Pennsylvania

Logistics

- Mid-Semester Survey due EOD 10/31
- Group formation due
- Milestone 0: In the week of 11/03 - 11/07
- Milestone 1: In the week of 11/17 - 11/21
- Last line of code: 12/05 @ 11:59PM.
- Demo: Anytime after you have submitted your final submission

Grading Breakdown

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

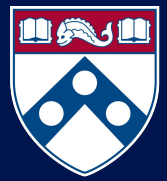
Documentation

- Required to provide a Companion Document
  - Consider this like APUE or K-and-R
  - Describes how OS is built and how to use it
  - Recommended to use Doxygen
- README
  - Describes implementation and design choices

Agenda

- PennOS Overview
- PennFAT file system
- Scheduling & Process Life Cycle
- spthreads
- PennOS Shell
- Demo

PennOS Overview

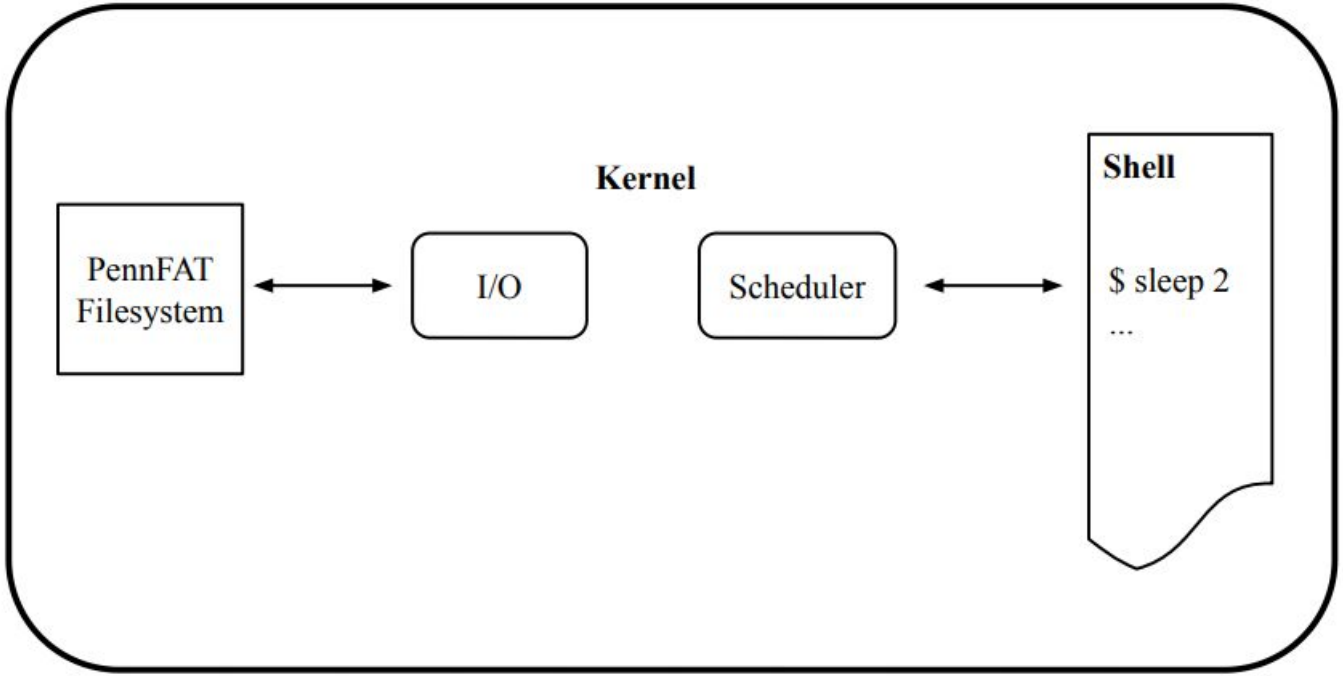


Penn
Engineering
University of Pennsylvania

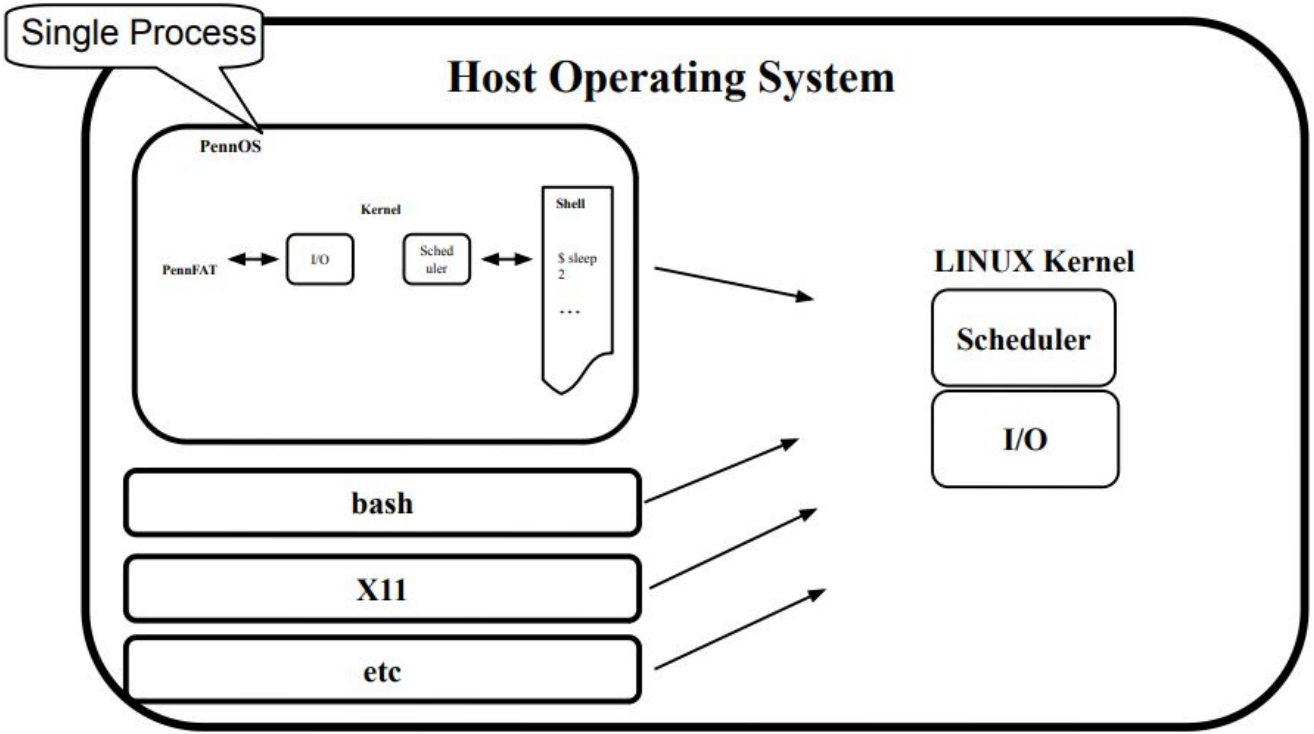
Projects so Far

- Penn Shredder
  - Mini Shell with Signal Handling
- Penn Shell
  - Redirections and Pipelines
  - Process Groups and Terminal Control
  - Job Control
- You will be implementing major user-level calls in PennOS

PennOS Diagram



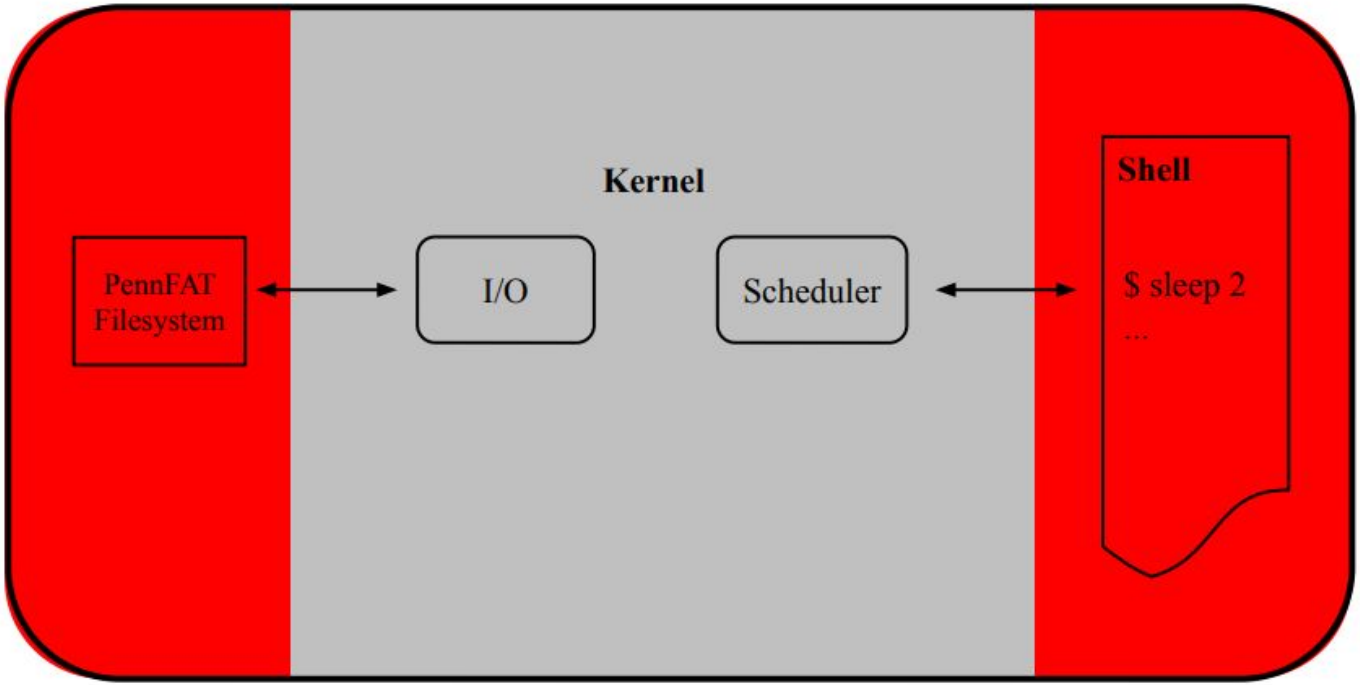
PennOS as a Guest OS



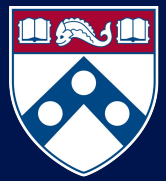
User, System, and Kernel Abstractions

- User Land - What an actual user interacts with
- Kernel Land - What happens 'under the hood'
- System Land - The API calls to connect user land with kernel land

User and Kernel Land



PennFAT File System



Penn Engineering
University of Pennsylvania

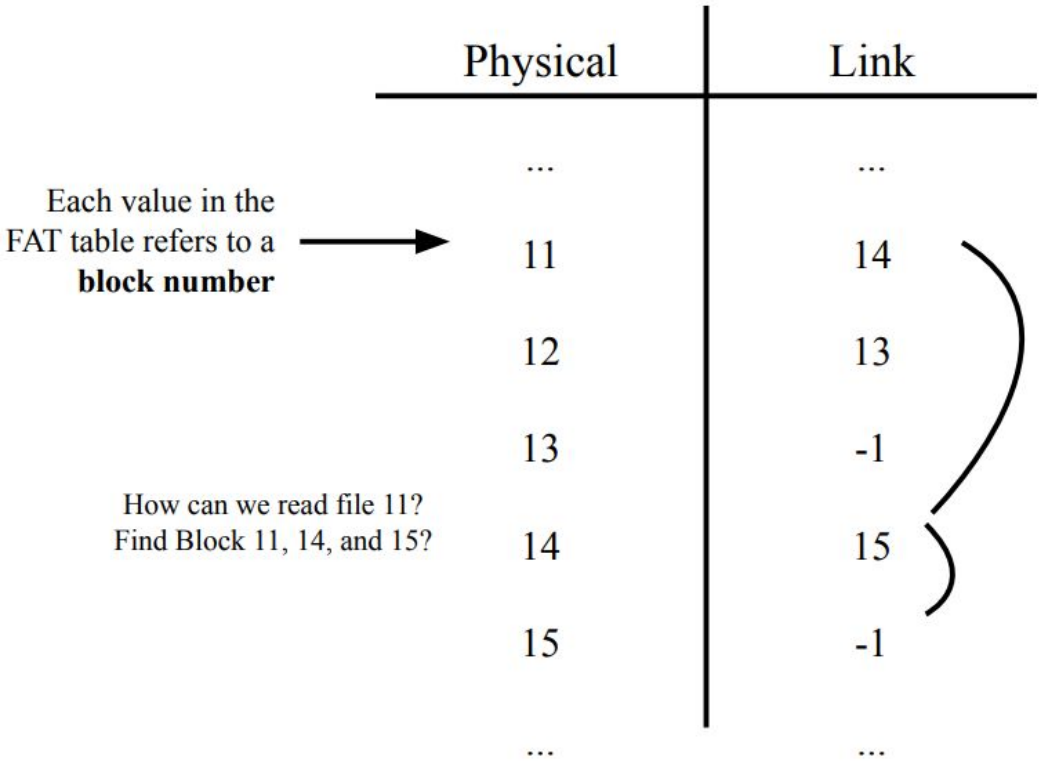
What is a File System

- A File System is a collection of data structures and methods an operating system uses to structure and organize data and allow for consistent **storage** and **retrieval of information**
  - Basic unit: a **file**
- A file (a sequence of data) is stored in a file system as a **sequence of data-containing blocks**

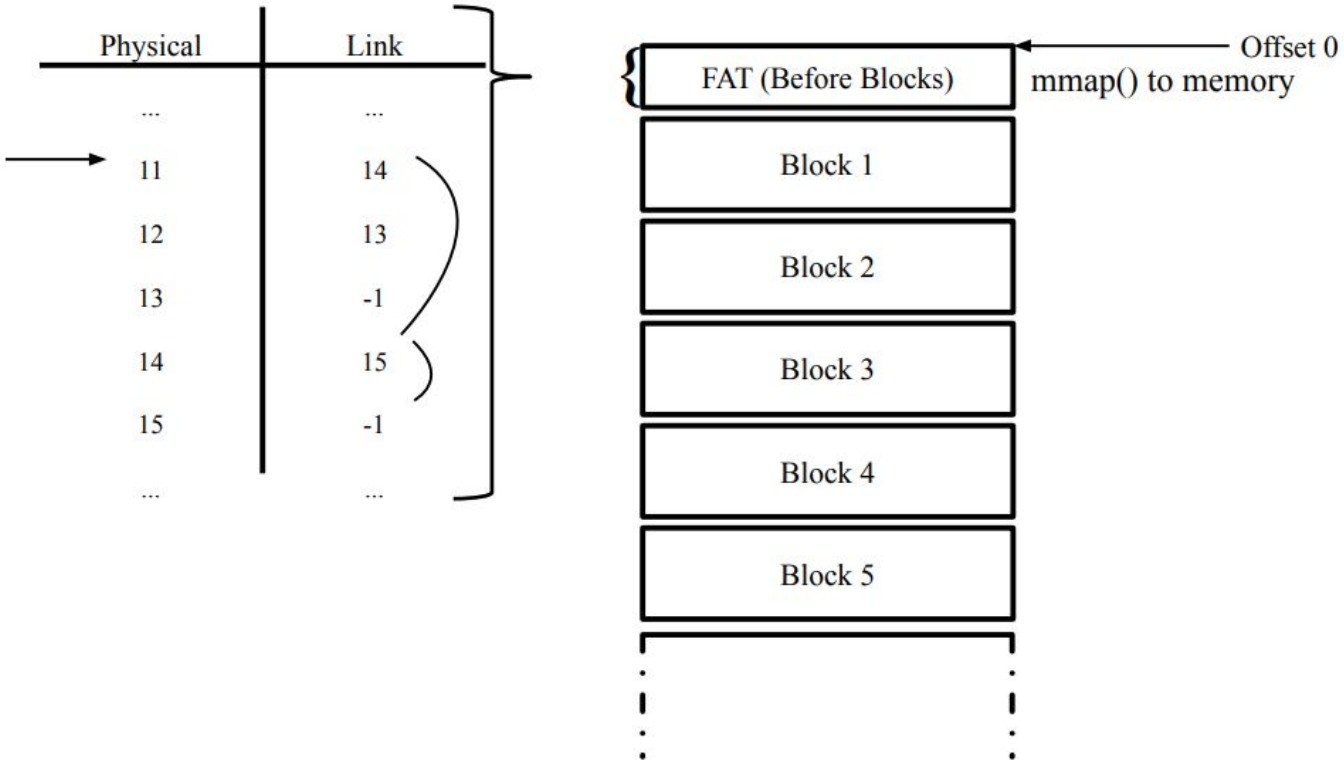
What is FAT?

- FAT stands for file allocation table, which is an architecture for organizing and referring to files and blocks in a file system.
- There exist many methods for organizing file systems, for example:
  - FAT (DOS, Windows)
  - Mac OS X
  - ext{1,2,3,4} (Linux)
  - NTFS (Windows)

FAT Example

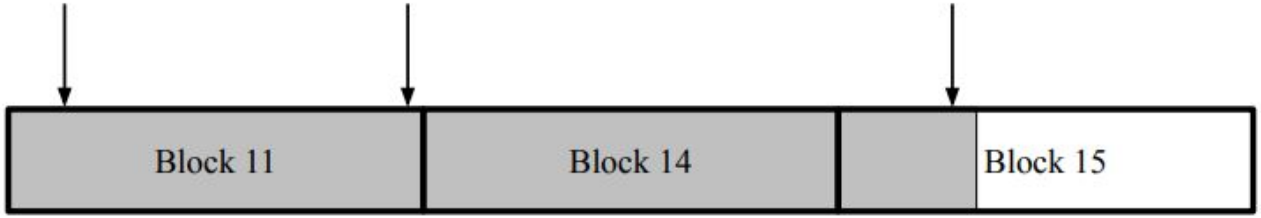

Physical
Link
...
...
Each value in the
FAT table refers to a
block number
11
14
12
13
13
-1
How can we read file 11?
Find Block 11, 14, and 15?
14
15
15
-1
...
...

File System Layout



File Alignment

- Files are distributed along blocks

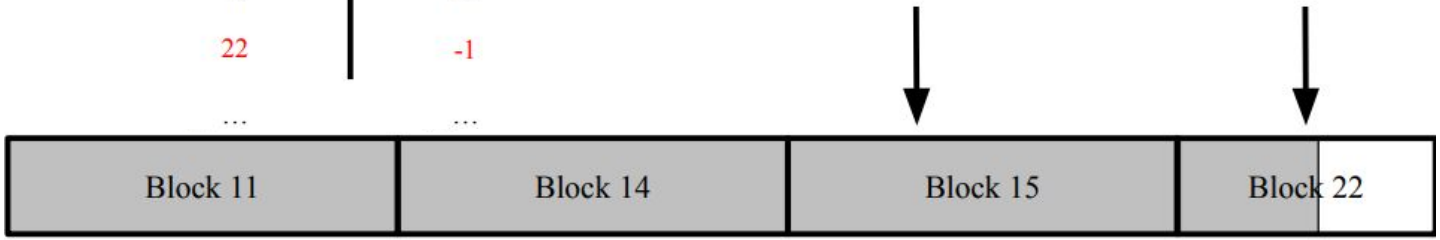



```
lseek(n, F_SEEK_SET, 60)
lseek(n, F_SEEK_SET, block_size - 1)
lseek(n, F_SEEK_SET, block_size * 2 + 100)
```

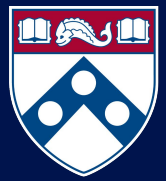
Adjusting File Size

Physical	Link
...	...
11	14
12	13
13	-1
14	15
15	22
...	...
22	-1
...	...

```
write(n, buffer, block_size)
```



PennFAT Spec



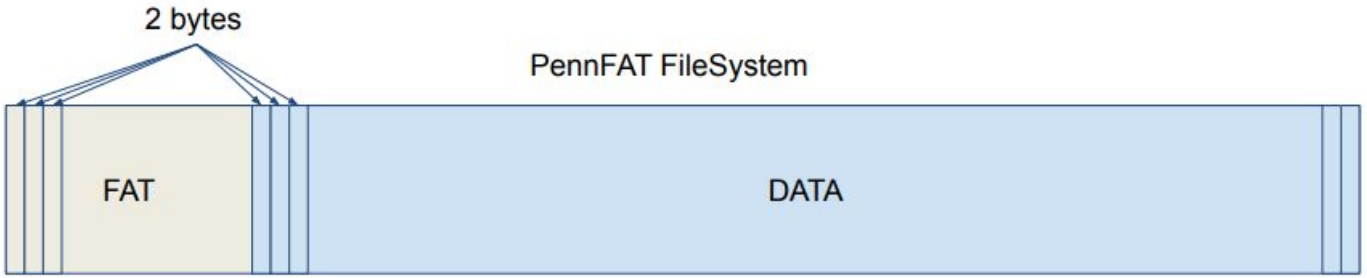
Penn
Engineering
University of Pennsylvania

File System

- Array of unsigned, little endian, 16-bit entries
- mkfs NAME BLOCKS_IN_FAT BLOCK_SIZE
- FAT region and DATA region

Layout

Region	Size	Contents
FAT Region	block size * number of blocks in FAT	File Allocation Table
Data Region	block size * (number of FAT entries – 1)	directories and files



FAT Region

- FAT entry size: 2 bytes
- First entry – special entry for FAT and block sizes
  - LSB: size of each block
  - MSB: number of blocks in FAT

LSB	Block Size
0	256
1	512
2	1,024
3	2,048
4	4,096

FAT first-entry examples

fat[0]	MSB	LSB	Block Size	Blocks in FAT	FAT Size	FAT Entries
0x0100	1	0	256	1	256	128
0x0101	1	1	512	1	512	256
0x1003	16	3	2048	16	32768	16384
0x2004	32	4	4,096	32	131,072	65,536*

* fat[65535] is undefined.

Why?

Other Entries of FAT

fat[i] (i > 0)	Data region block type
0	free block
0xFFFF	last block of file
[2, number of FAT entries)	next block of file

FAT first-entry examples

fat[0]	MSB	LSB	Block Size	Blocks in FAT	FAT Size	FAT Entries
0x0100	1	0	256	1	256	128
0x0101	1	1	512	1	512	256
0x1003	16	3	2048	16	32768	16384
0x2004	32	4	4,096	32	131,072	65,536*

* fat[65535] is undefined.

Why?

- 0xFFFF is reserved for last block of file

Example FAT

Index	Link	Notes
0	0x2004	32 blocks, 4KB block size
1	0xFFFF	Root directory
2	4	File A starts, links to block 4
3	7	File B starts, links to block 7
4	5	File A continues to block 5
5	0xFFFF	Last block of file A
6	18	File C starts, links to block 18
7	17	File B continues to block 17
8	0x0000	Free block

Data Region

- Each FAT entry represents a file block in data region - Data Region size = block size * (# of FAT entries - 1)
  - b/c first FAT entry (fat[0]) is metadata - block numbering begins at 1:
- block numbering begins at 1:
  - block 1 – always the first block of the root directory
  - other blocks – data for files, additional blocks of the root directory, subdirectories (optional extension)

What is a Directory?

- A directory is a file consisting of entries that describe the files in the directory.
- Each entry includes the file name and other information about the file.
- The root directory is the top-level directory.

Directory entry

- Fixed size of 64 bytes each
- file name: 32 bytes (null terminated)
  - legal characters: [A-Za-z0-9._-] (POSIX portable filename character set)
  - first byte special values:

name[0]	Description
0	end of directory
1	deleted entry; the file is also deleted
2	deleted entry; the file is still being used

Directory entry (cont.)

- file size: 4 bytes
- first block number: 2 bytes (unsigned)
- file type: 1 byte

Value	File Type
0	unknown
1	regular file
2	directory

Directory entry (cont.)

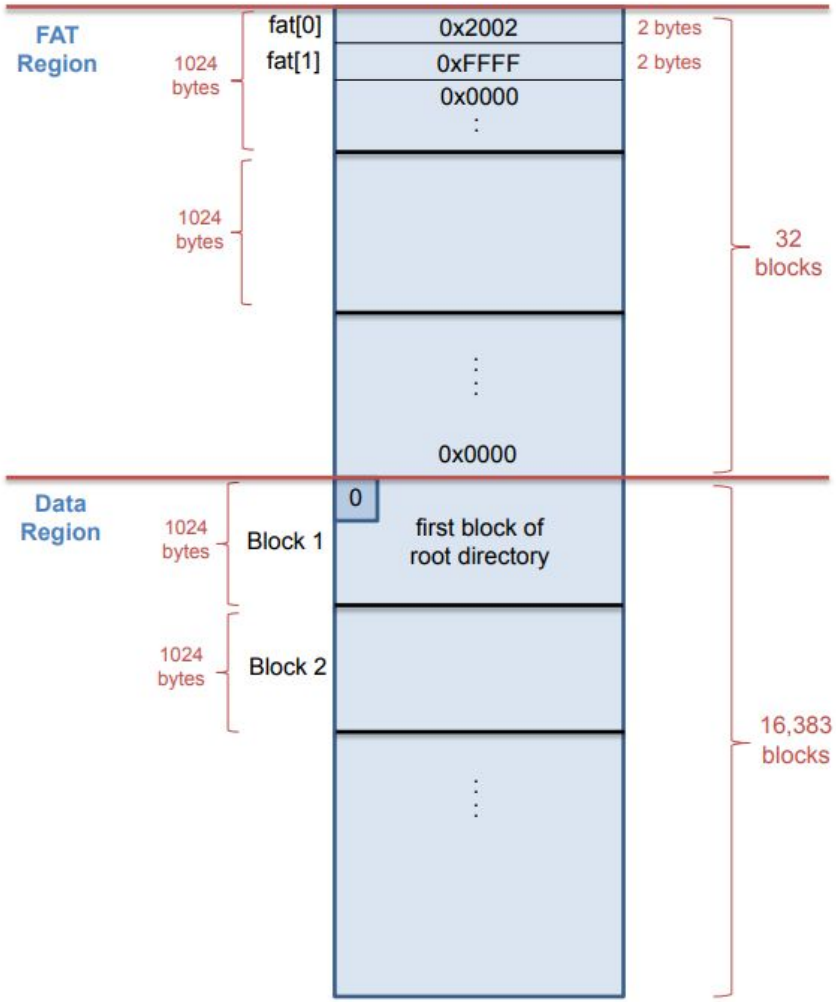
- file permission: 1 byte

Value	Permission
0	none
2	write only
4	read only
5	read and executable
6	read and write
7	read, write, and executable

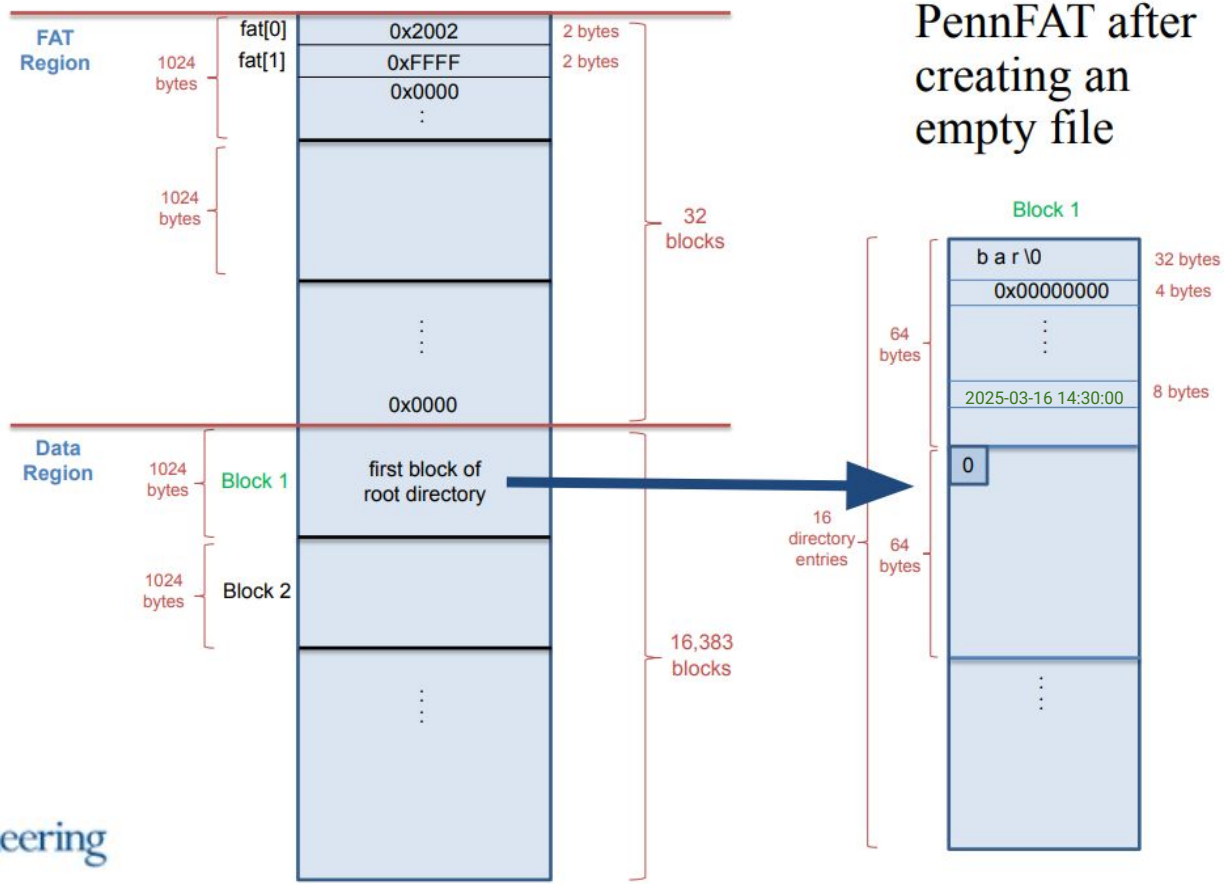
- timestamp: 8 bytes returned by time(2)
- remaining 16 bytes: reserved for extensions

Example

- fat[0] = 0x2002
  - 32 blocks of 1024 bytes in FAT
- First block of Data Region is first block of root directory
- Correspondingly, fat[1] refers to that Block 1, which ends there. So it has value of 0xFFFF

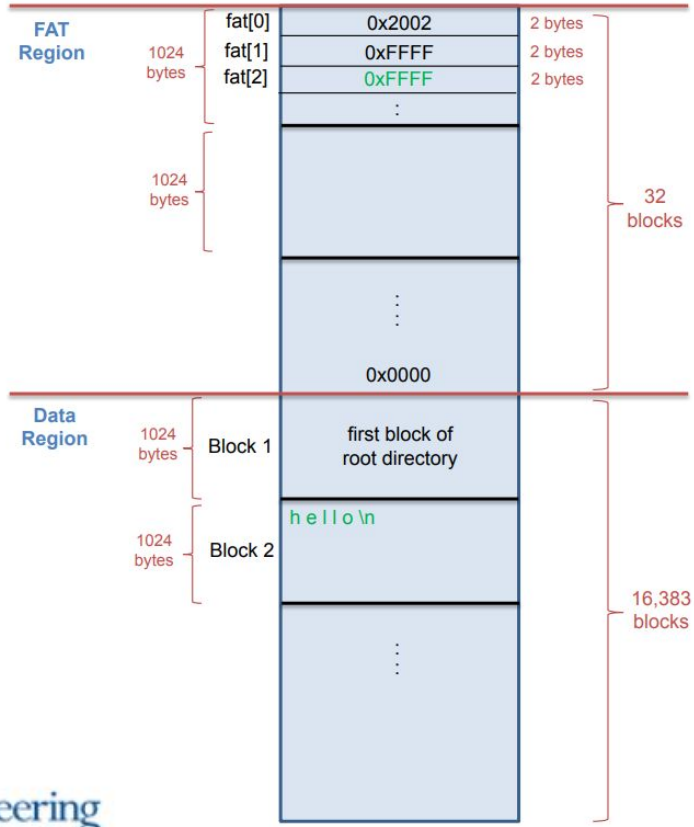


Creating a File

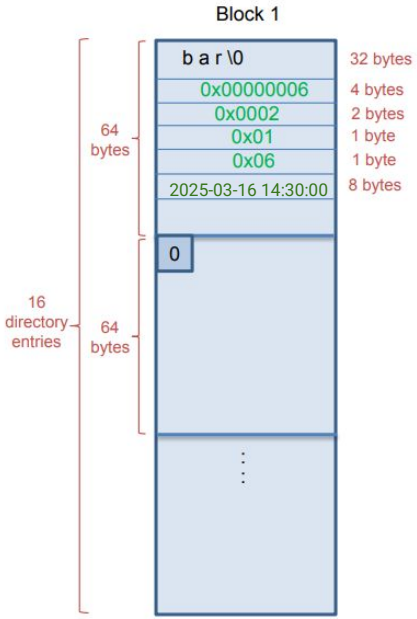


Penn Engineering

Writing to a File

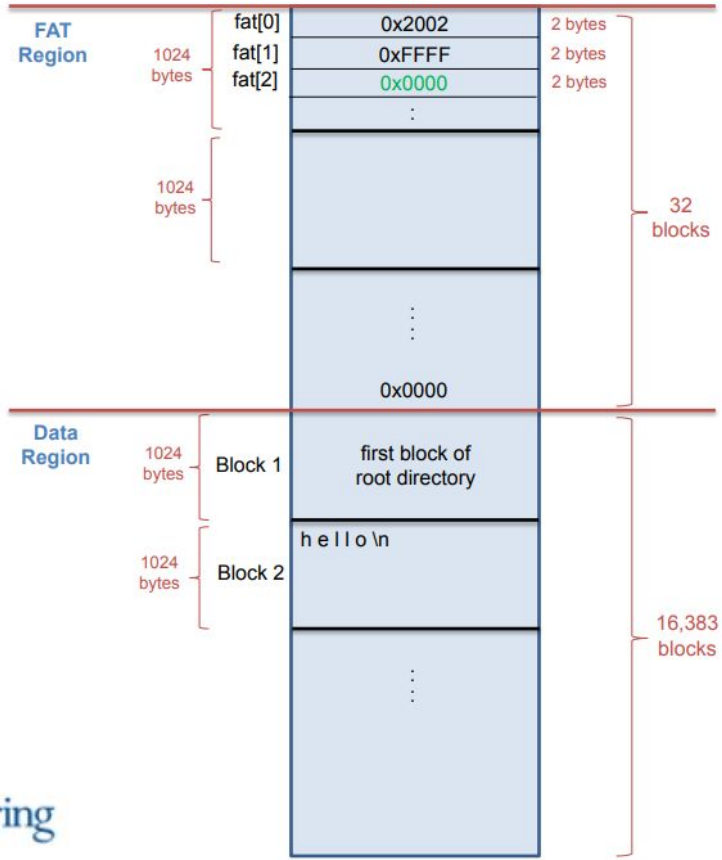


PennFAT after writing to the file

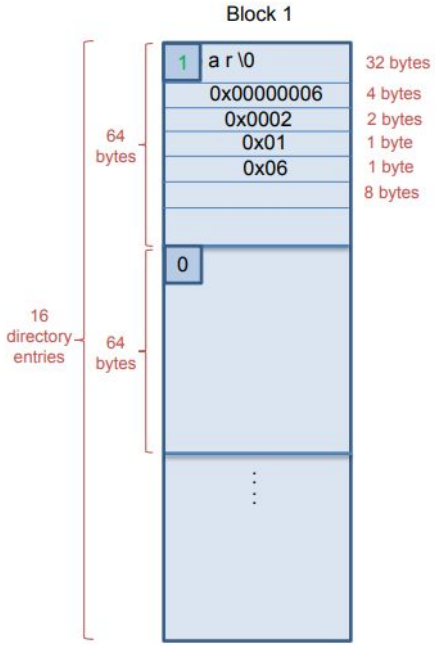


Penn Engineering

Removing the File



PennFAT after removing the file



Penn Engineering

Standalone PennFAT

- Milestone 1
- Implementation of kernel-level functions (k_functions)
- Simple shell for reading, parsing, and executing File system modification routines
- System-wide Global File Descriptor Table

Kernel-Level Functions

- Interacting directly with the filesystem you created
- Also interacts directly with the system-wide Global FD Table
- k_write(int fd, const char* str, int n)
  - Access the file associated with file descriptor fd
    - Access through the FD table
  - Write up to n bytes of str
    - literally modify the binary filesystem you created. This should be loaded in memory, so you can modify the in-memory array

Standalone Routines

- Special Commands
  - mfks, mount, unmount
  - These can be implemented using C System Calls
- Standard Routines
  - touch, mv, rm, cat, cp, chmod, ls
  - These should ONLY use k_functions unless interacting with the HOST filesystem
- Your filesystem: PennFAT binary file you created HOST filesystem: Your docker filesystem

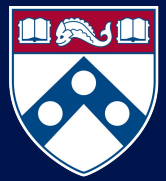
Standalone Routines

- cat FILE … [-w OUTPUT_FILE] - get input from multiple FILE(s), output to stdout or OUTPUT_FILE if specified
  - The following would be logical flow of cat
    - k_open(FILEs)
    - k_read(FILEs)
    - k_write(stdout / OUTPUT_FILE)

Standalone Routines

- cp [-h] SOURCE DEST - copy contents from SOURCE to DEST. If -h flag exists, copy from HOST filesystem
- The following would be logical flow of cp
    - If no -h flag specified:
        - k_read(SOURCE)
        - k_write(DEST)
    - If -h flag specified:
        - read(SOURCE) ← Note this is C sys-call
        - k_write(DEST)

PennOS Kernel



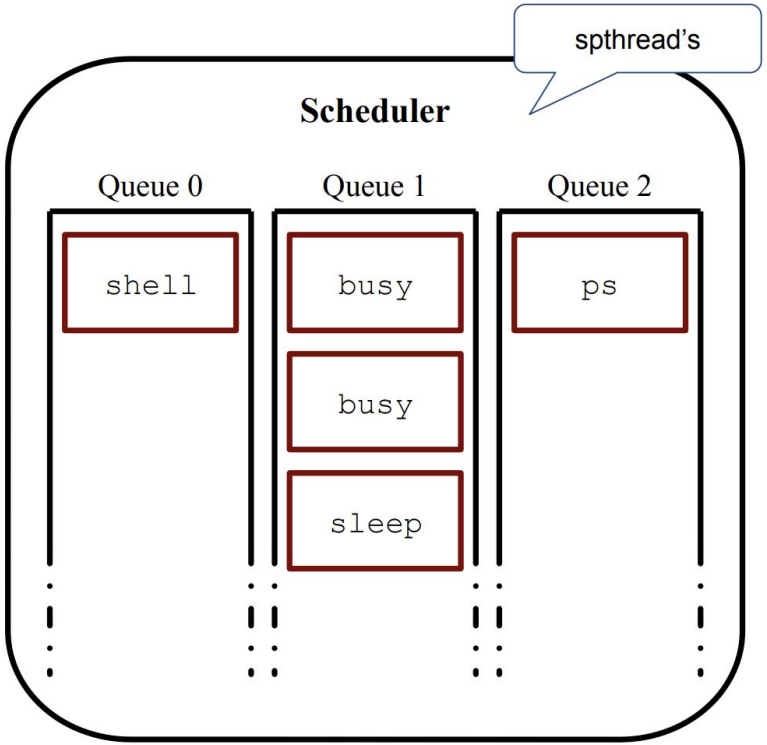
Penn
Engineering
University of Pennsylvania

Scheduling in PennOS

Algorithm: round-robin with 3 different queues

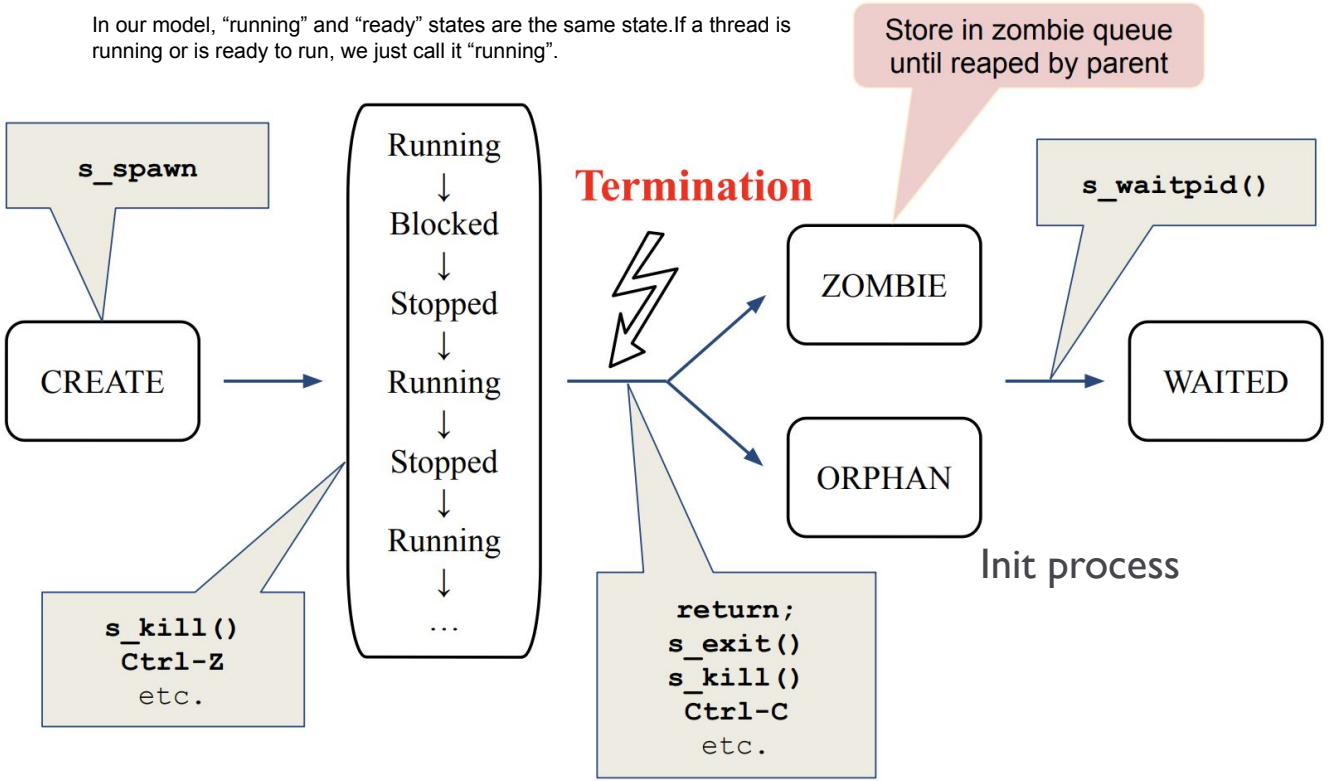
Exponential Relationship:

- Queue 0 scheduled 1.5 times more frequently than Queue 1
- Queue 1 scheduled 1.5 times more frequent than Queue 2
- Deterministic - Not Random



Process Life Cycle

In our model, “running” and “ready” states are the same state.If a thread is running or is ready to run, we just call it “running”.



Process Control Block

```
typedef struct pcb {
    pid_t pid;

    int foo;
    char *bar;

} pcb_t;
```

- handle to the spthread
- PID, parent PID, child PID(s)
- open file descriptors
- priority level
- process state
- Not limited to this!
- Think about sleep
- Think about waitpid and parent - child relationship handling

Scheduler Implementation Tips

- Read, understand sched-demo.c (First TA Catchup!), use this design in scheduler
- Think about where should kernel call the scheduler at?
- What happens when the scheduler is idle?
- DO NOT PUT THE SCHEDULER IN A SIGNAL HANDLER
- Signal handler - as small as possible. The best signal handlers either do nothing or only increment a counter.
- If you want to increment a counter or something, declare the counter of type:
  volatile sig_atomic_t

  This is the only data type that is guaranteed to be signal safe by the standard.

PennOS Signals

- You need to implement your own "signals" for PennOS.
- Use kill, or spthread_kill.
- Use sigaction to register handlers to catch signals (CTRL + C and CTRL + Z) from the terminal but your PennOS should somewhere **manually** handle the "stopping" and "terminating" of the thread.
  - What should happen when a process is killed/terminated?
- You will also likely make use of setitimer and sigsuspend for the scheduler ticks.
  - What should happen at each tick?
  - When should init be scheduled?

More Tips

- With the description of setitimer(), it just says that sigalarm is delivered to the process, not necessarily the calling thread. To make sure sigalarm goes to the scheduler, you may want to make it so that all threads (spthread or otherwise) that aren't the scheduler call something like: **pthread_sigmask(SIG_BLOCK, SIGALARM)**
  - Which will block SIGALARM in that thread.
  - If you want code to always be executed by a thread, a nice way to do it is via a wrapper around the start routine. See spthread.c if you want some inspiration

More Tips

- If you are having issues with the scheduler not running you can try running
  - **strace –e 'trace=!all' ./bin/pennos**
  - You may have to install strace: **sudo apt install strace**
  - This will print out every time a signal is sent to your pennos
  - (Usual fix is the pthread_sigmask thing above)

POSIX threads

- User-level thread management API
- Isolate code execution with distinct threads
- Resource sharing (within same process space)
- Concurrent execution

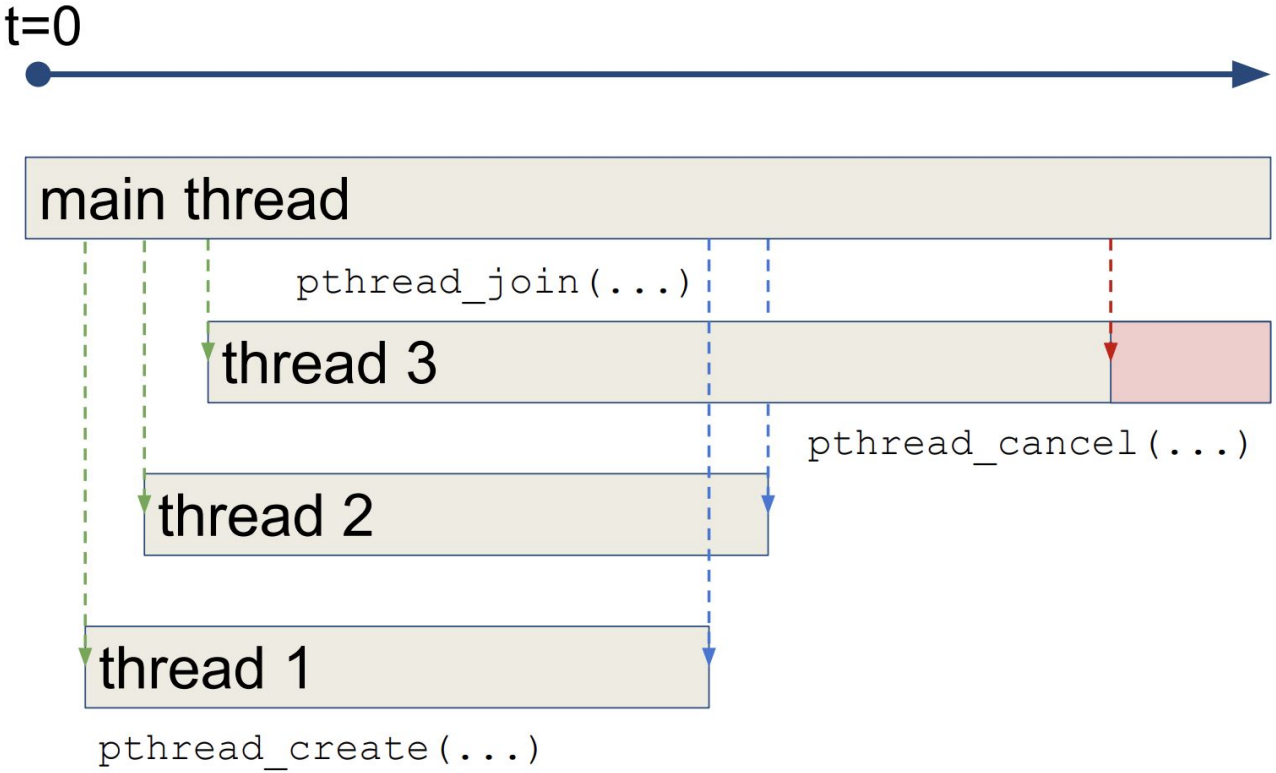
Pros: efficient, lightweight, simple

What are the cons?

pthread_cancel

- Provides us a way to "terminate" a thread.
- Notably, it does not terminate the thread immediately, it sends a "cancellation **REQUEST**". The thread is not cancelled until it hits a cancellation point.
- Read the comments in spthread.h for spthread_cancel to see more.

How does pthread work?



Spthread

Wrapper around pthread, provided by us (READ THE FILE!)

Provides additional tooling to:

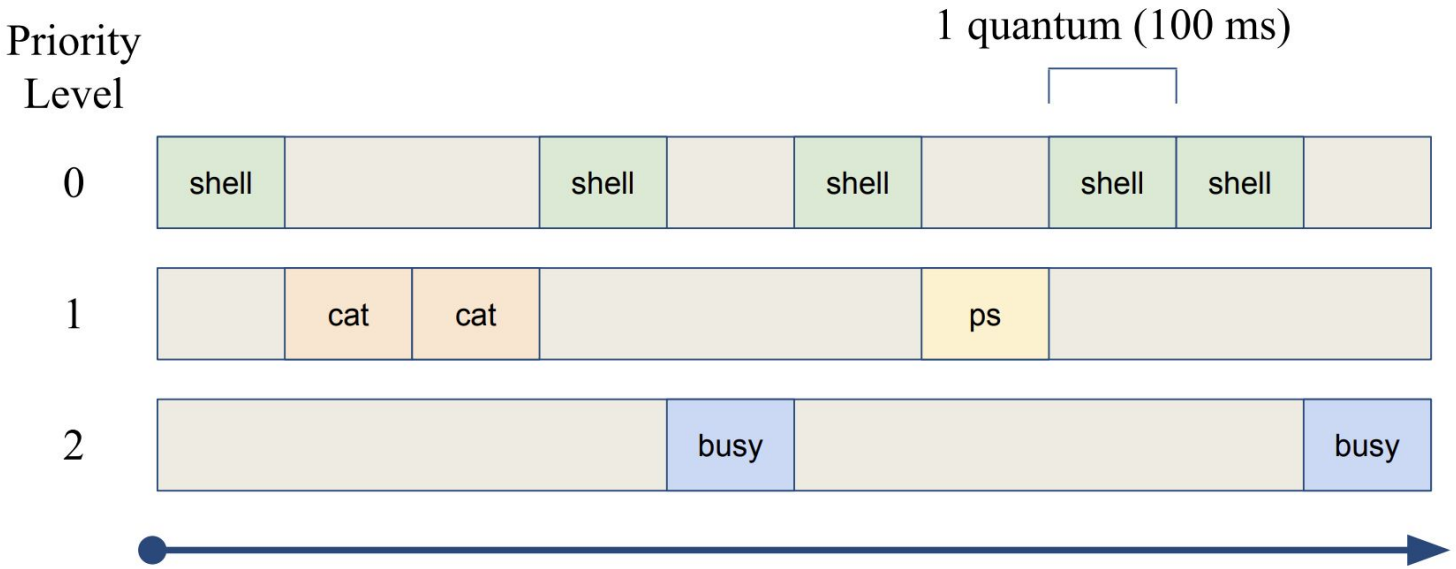
- Create, then immediately suspend the thread
- Suspend a thread
- Continue (unsuspend) a thread

```
spthread_t new_thread;

spthread_create(&new_thread, NULL, routine, argv);
spthread_continue(new_thread);
spthread_suspend(new_thread);
```

Spthread: Unit of Scheduling

Leverage suspend + continue to execute one spthread at a time



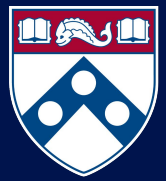
Spthread, Shared Resources

- Unlike normal threads, we aren't going to use a lock to protect resources.
- Instead, we will use a way to "block pre-emption" of a thread to make sure it is the only one running. We will talk about this in the next lecture.

```
spthread_disable_interrupts_self();

spthread_enable_interrupts_self();
```

PennOS Shell



Penn
Engineering
University of Pennsylvania

Shell Requirements

- Synchronous child waiting
- Redirection
- Parsing
- Terminal Signaling
- Terminal Control

Shell Functions

- Basic interaction with PennOS
- Two types:
  - Functions that run as separate processes
  - Functions that run as shell subroutines

Built-ins Running as Processes

- cat
- sleep
- busy
- ls
- touch
- mv
- cp
- rm
- ps

Built-ins Running as Subroutines

- nice
- nice_pid
- man
- bg
- fg
- jobs
- logout

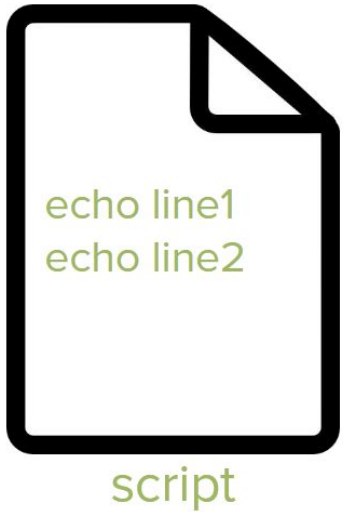
- Quick aside: Why?
  - Think about why it might be problematic/difficult to run these commands from a separate process
- Consider the kernel structure & process lifecycle

Error Handling

- errno.h
- u_perror
- Have global ERRNO macros
- Call u_perror for PennOS system call errors like s_open, s_spawn
- Call perror(3) for any host OS system call error like malloc(3), open(2)

Shell Scripts

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



Should I re-use Penn-Shell?

- Probably not, but you can take inspiration from it and copy *parts* of it.
- Some of it will have to change to support PennOS. Notably the system calls you make are different and behave a little differently.

User, System and Kernel Abstractions

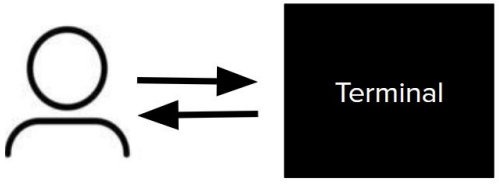
- User Land - What an actual user interacts with
  - Functions that aren't directly interacting with the OS.

    E.g. if you made your own print function or string utility functions
    - String utility doesn't deal with the OS at all
    - Print function uses your system_call "s_write" function to handle the OS.
- Kernel Land - What happens 'under the hood'
  - Deals with the nitty gritty details that the user doesn't need to know about
- System Call Land - The API calls to connect user land with kernel land
  - Similar to the system calls you see available to you in linux and in the past homework assignments.

How to differentiate?

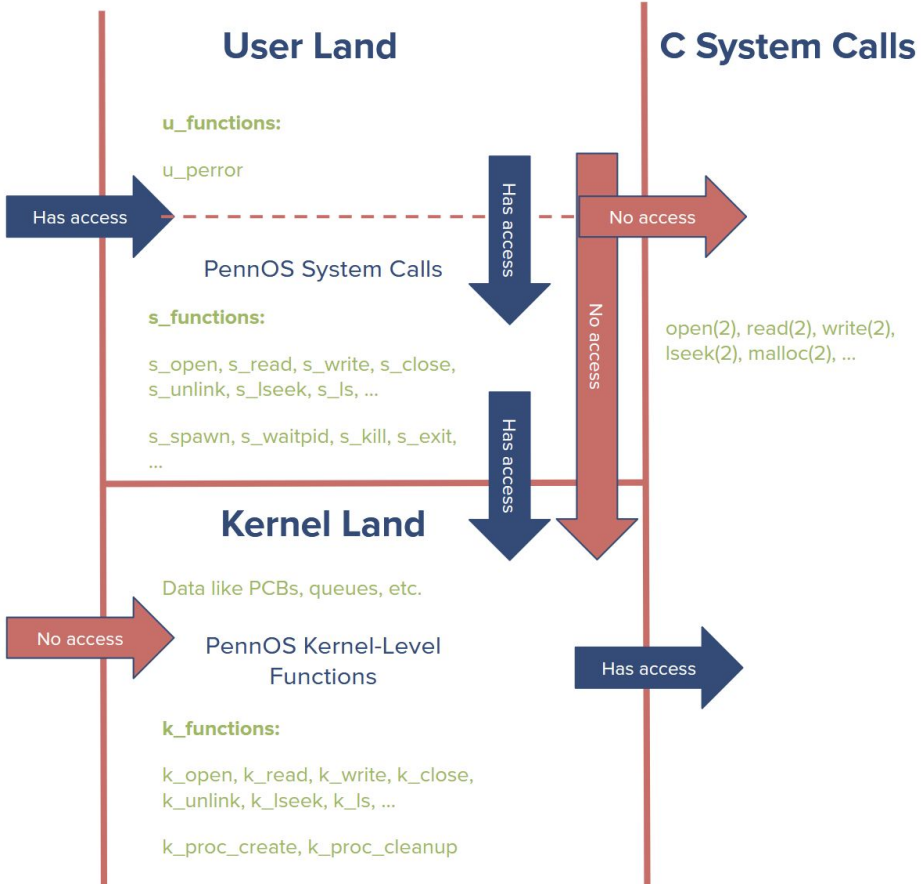
- One way to think about whether something is user /system call / kernel is thinking about who is invoking the function and what info they need to know.
- User level: minimal or no knowledge of the underlying operating system
- System call: some level of the operating system abstraction needs to be understood and deals with the "public" aspects of it
  - Process level file descriptors are "public" parts of the OS interface
- Kernel level functions: deeper knowledge of the OS is needed. Invoker of the function either passes in or gets something "private" to the OS.
  - System wide file descriptors and the PCB are "private"

Maintaining the Abstraction



Shell Built-ins:
cat, sleep, busy, echo, ls, touch, mv, cp, rm, chmod, ps, kill, zombify, orphanify

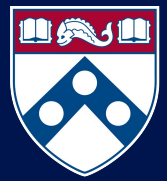
Shell Routines:
nice, nice_pid, man, bg, fg, jobs, logout



Shell Abstraction

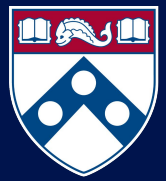
- Your PennOS shell should use the same layer of abstraction as the penn-shell you made.
  - Did your penn-shell access the OS scheduler queues?
  - In penn-shell did you have access to the PCB?

Demo



Penn Engineering
University of Pennsylvania

Questions?



Penn Engineering
University of Pennsylvania