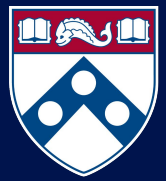
CIS 5480
PennOS Lecture

Tuesday, March 17, 2025



Penn Engineering
University of Pennsylvania

Logistics

- Mid-Semester Survey due EOD 3/24
- Form groups ASAP (random assignments EOD)
- Milestone 0: In the week of 3/24 - 3/28
- Milestone 1: In the week of 4/7 - 4/11
- Final Submission: 4/29
- Demo: Anytime after you have submitted your final submission

Grading Breakdown

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

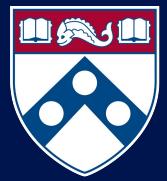
Documentation

- 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

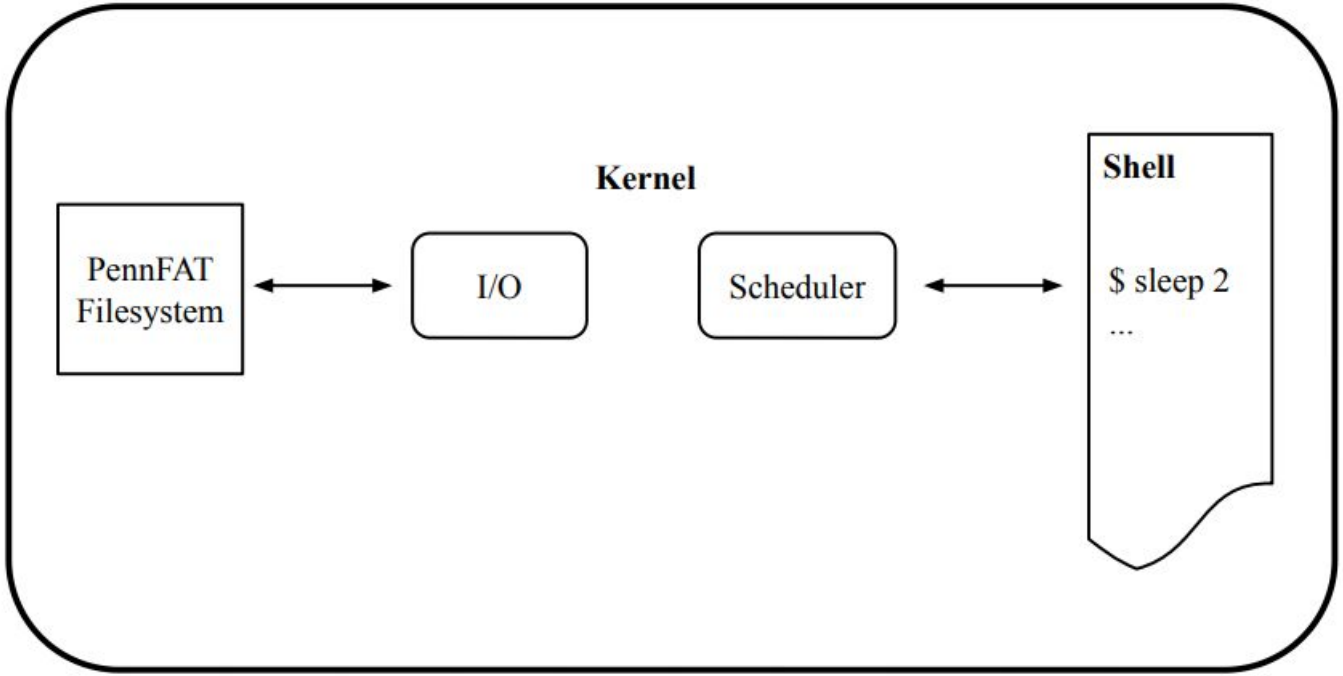


Penn
Engineering
University of Pennsylvania

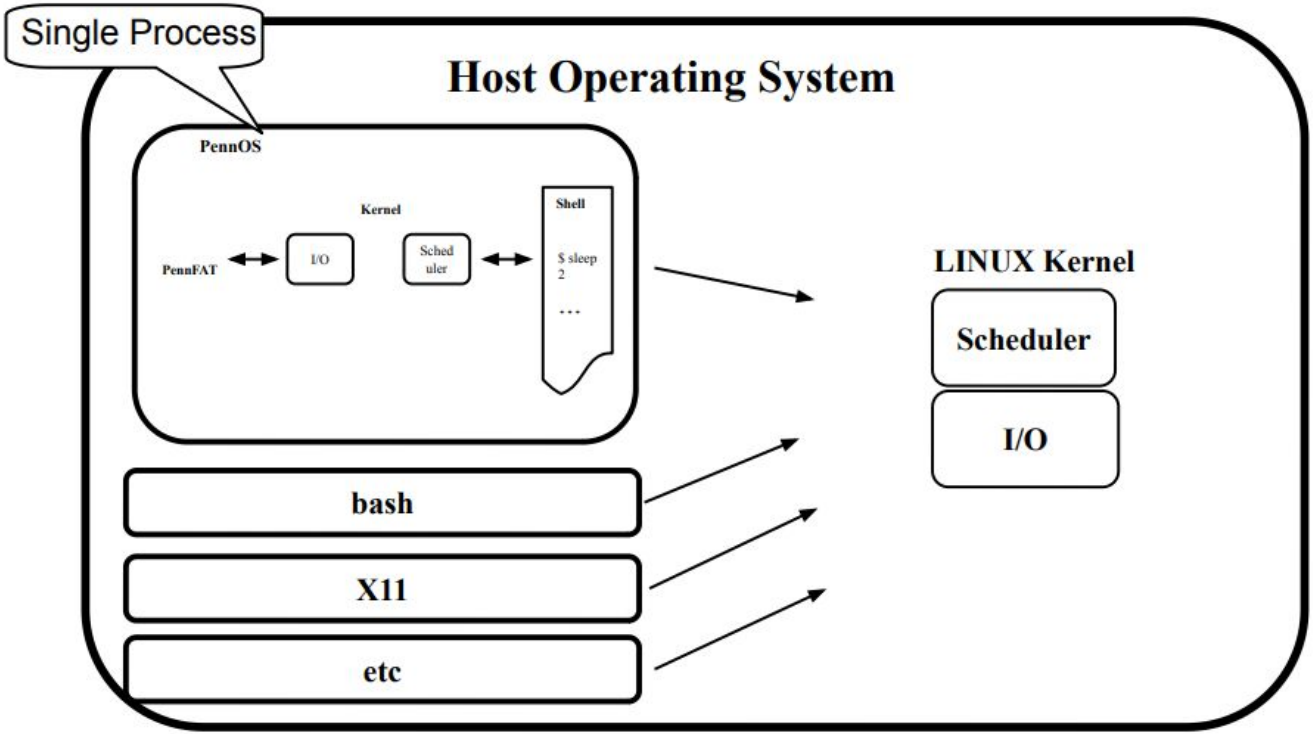
Projects so Far

- 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



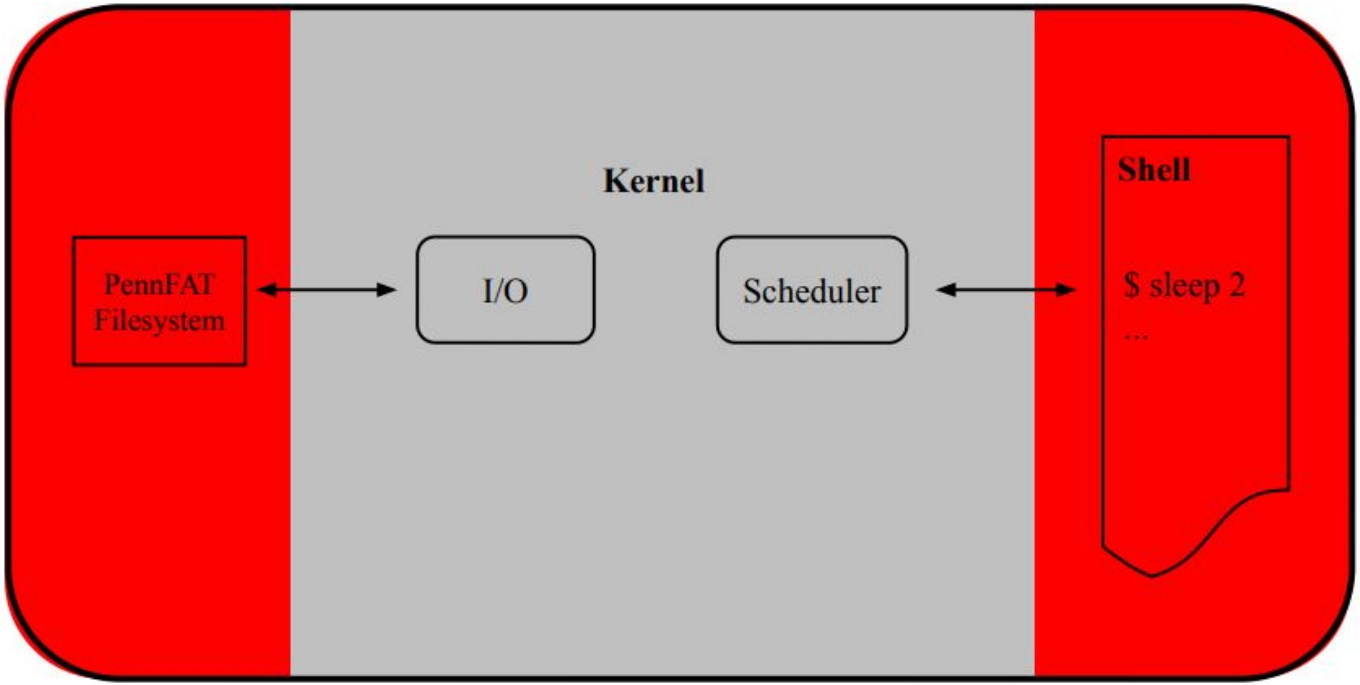
PennOS as a Guest OS



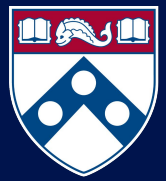
User, System, and Kernel Abstractions

- 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

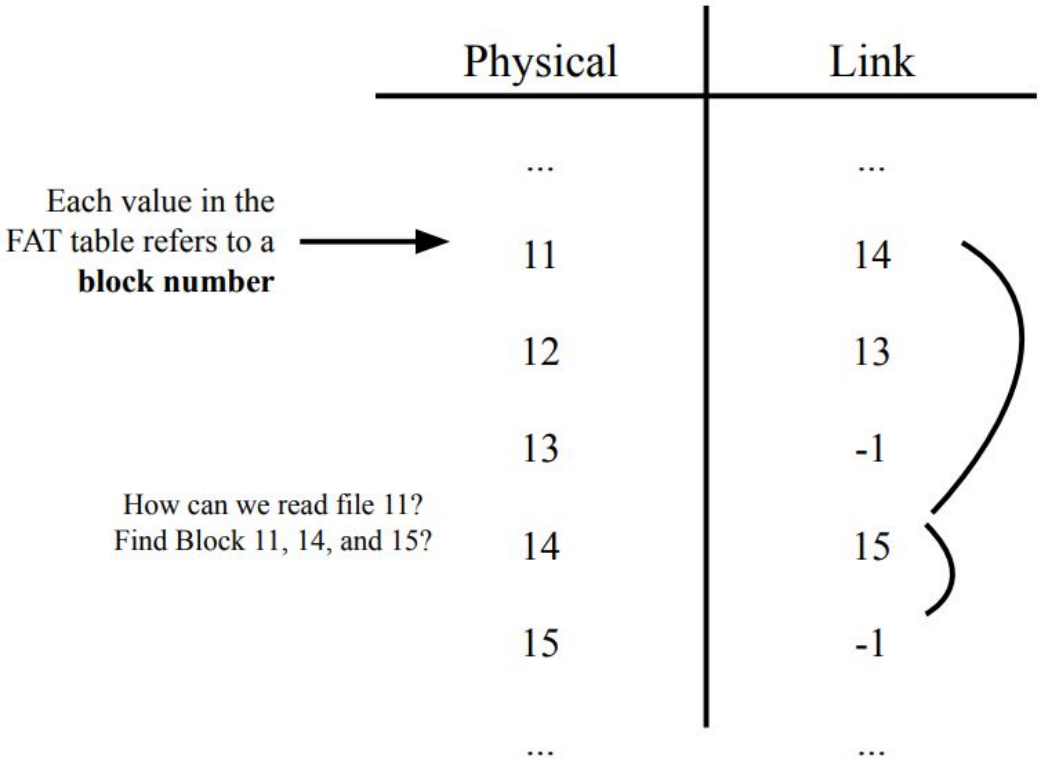
What is a File System

- 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**

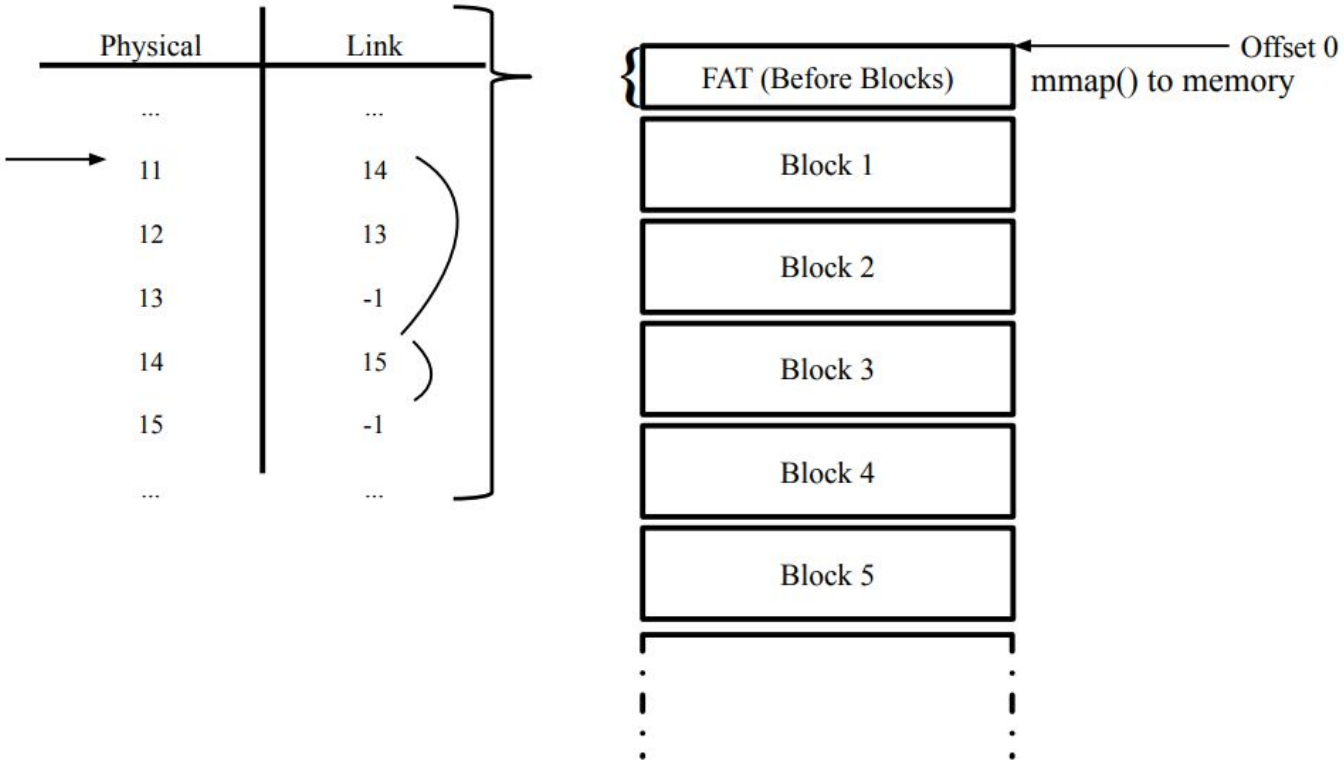
What is FAT?

- 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

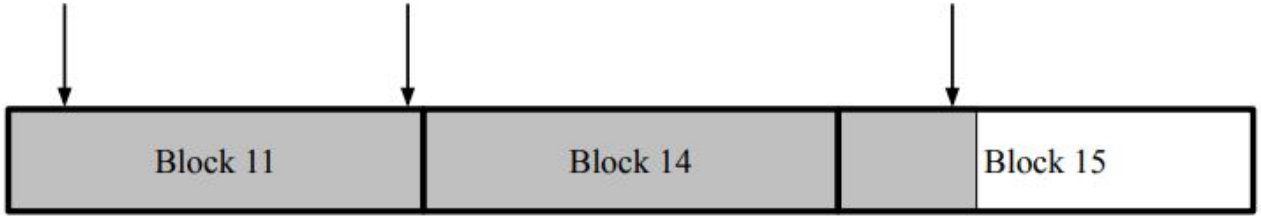

Physical
Link
...
...
Each value in the
FAT table refers to a
block number
11
14
12
13
13
-1
How can we read file 11?
Find Block 11, 14, and 15?
14
15
15
-1
...
...

File System Layout



File Alignment

- Files are distributed along blocks

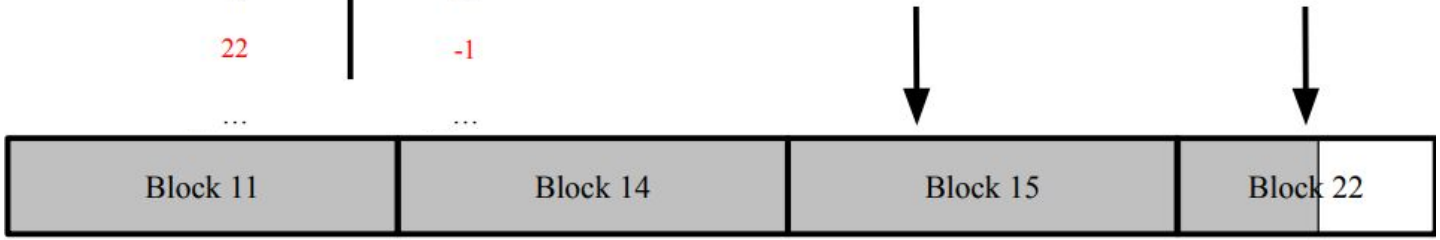



```
lseek(n, F_SEEK_SET, 60)
lseek(n, F_SEEK_SET, block_size - 1)
lseek(n, F_SEEK_SET, block_size * 2 + 100)
```

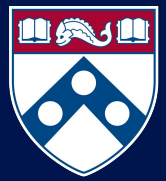
Adjusting File Size

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

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



PennFAT Spec



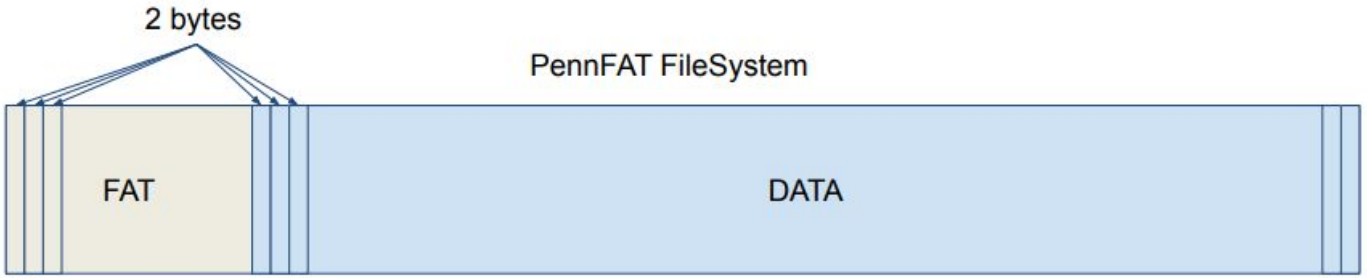
Penn
Engineering
University of Pennsylvania

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 (extra credit)

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
4	symbolic link (extra credit)

Directory entry (cont.)

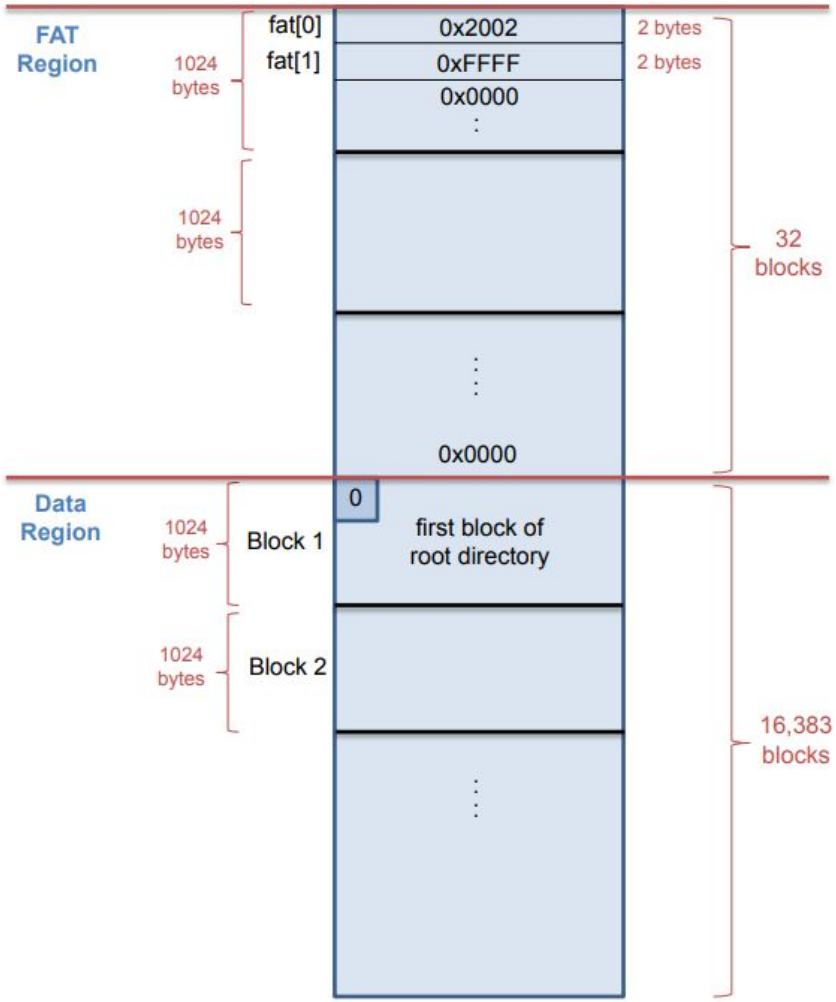
- file permission: 1 byte

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

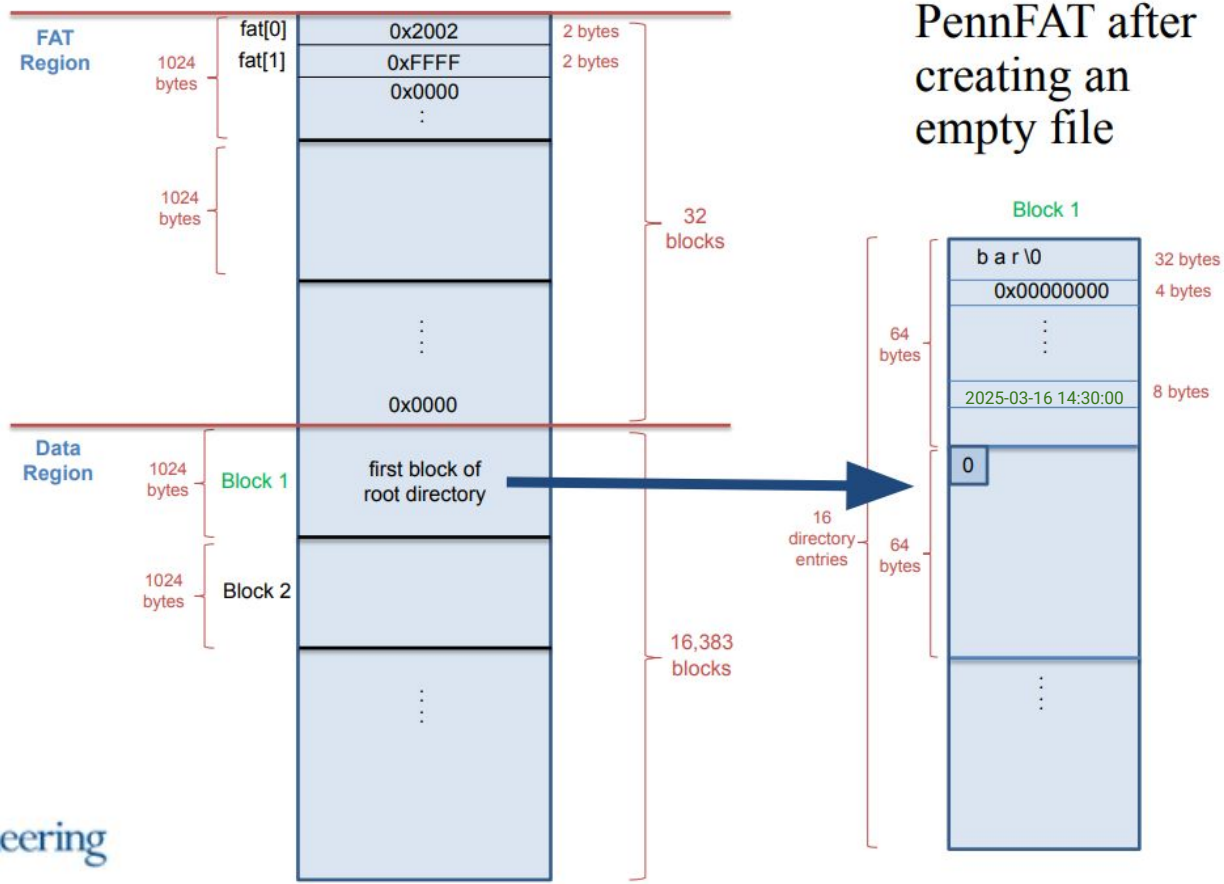
- timestamp: 8 bytes returned by time(2)
- remaining 16 bytes: reserved for E.C

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

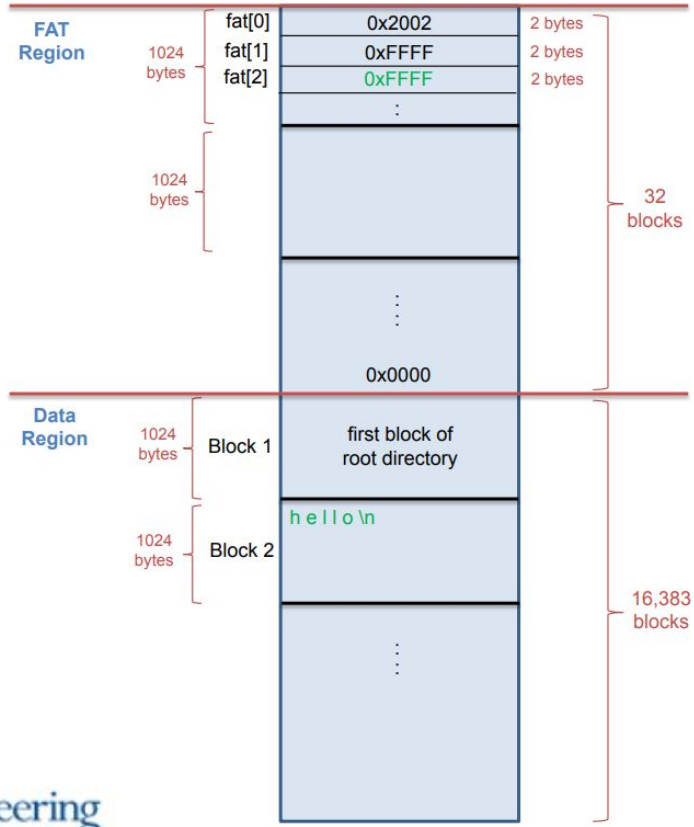


Creating a File

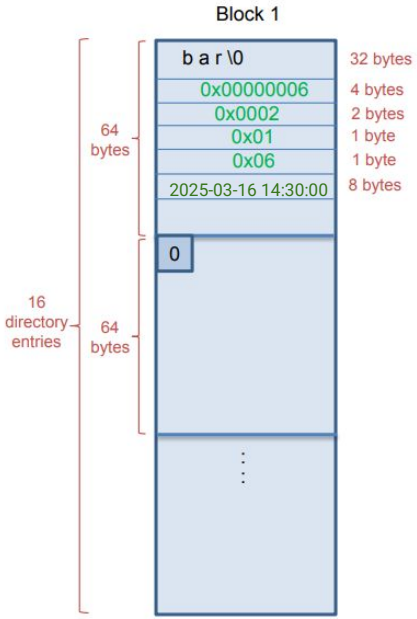


Penn Engineering

Writing to a File

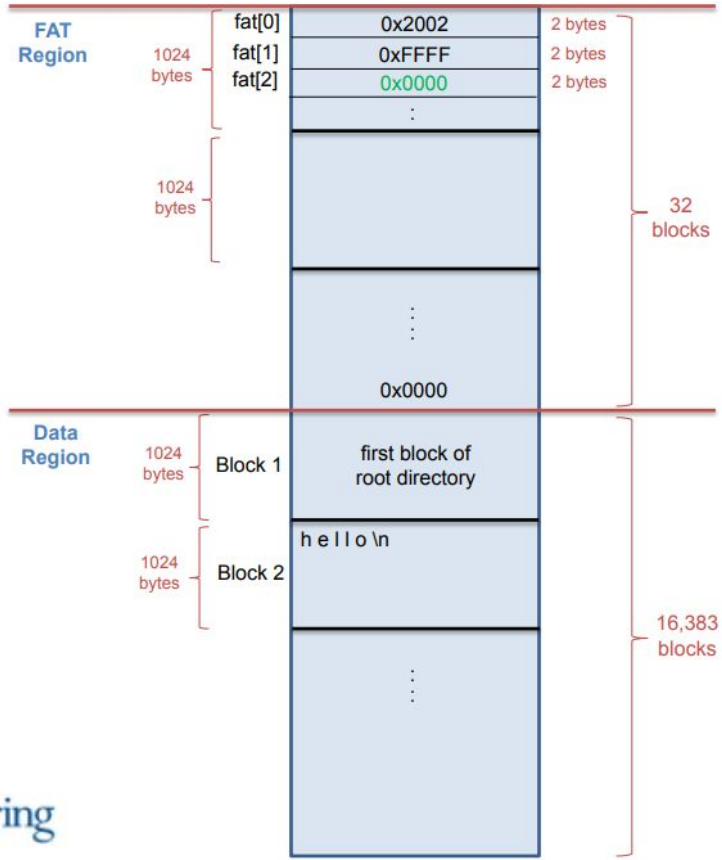


PennFAT after writing to the file

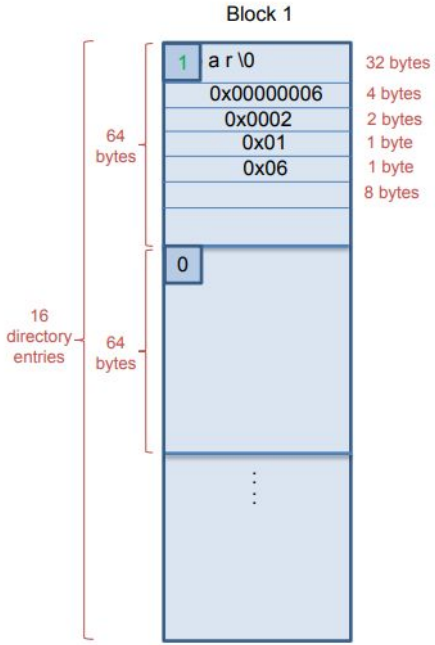


Penn Engineering

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

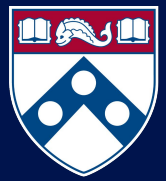
Standalone Routines

- 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 -h flag:
        - read(SOURCE) ← Note this is C sys-call
        - k_write(DEST)
    - else
        - k_read(SOURCE)
        - k_write(DEST)

PennOS Kernel



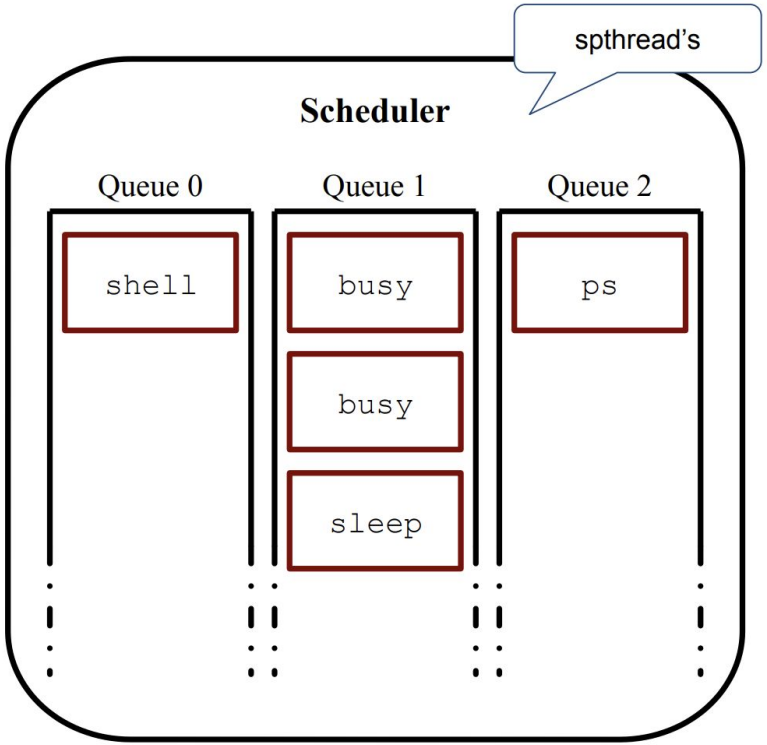
Penn
Engineering
University of Pennsylvania

Scheduling in PennOS

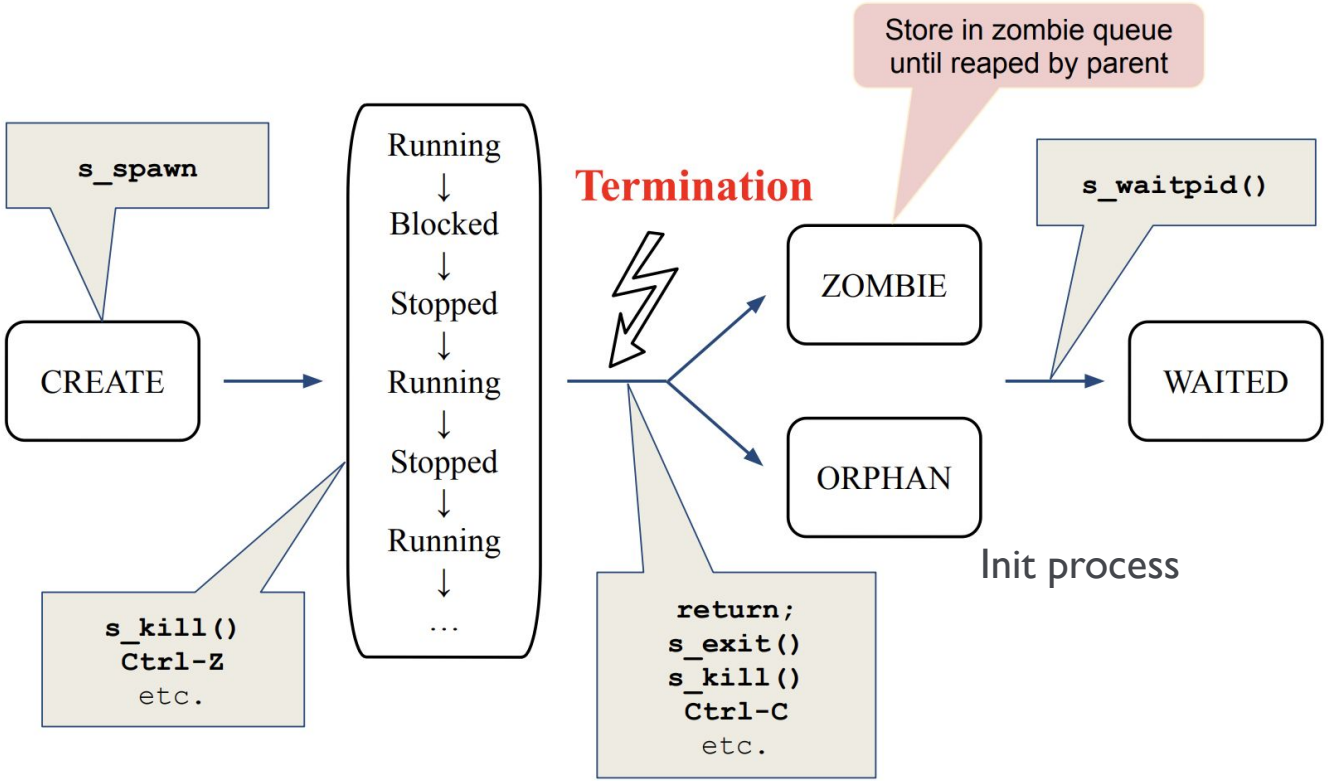
Algorithm: round-robin with queue

Exponential Relationship:

- Queue 0 scheduled 1.5 times more frequently than Queue 1
- Queue 1 scheduled 1.5 times more frequent than Queue 2



Process Life Cycle



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
- etc.

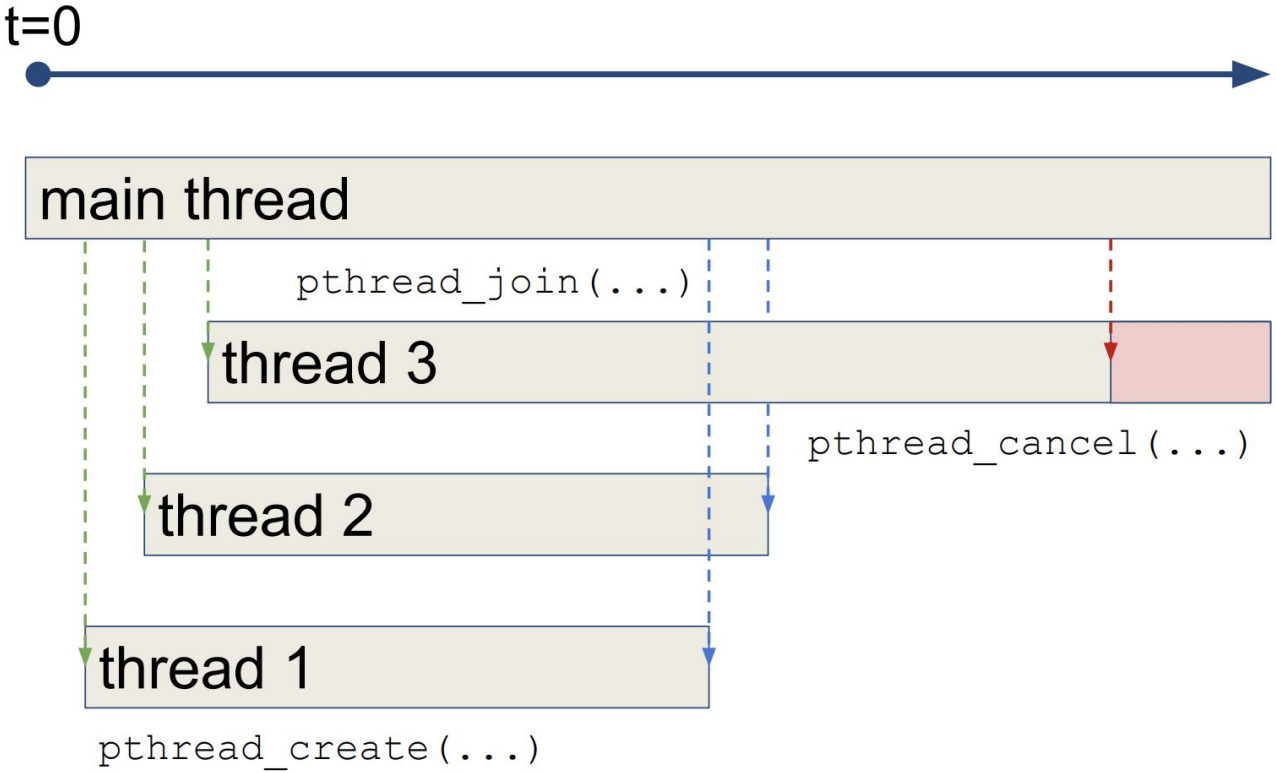
POSIX threads

- 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?

How does pthread work?



Spthread

Wrapper around pthread, provided by us

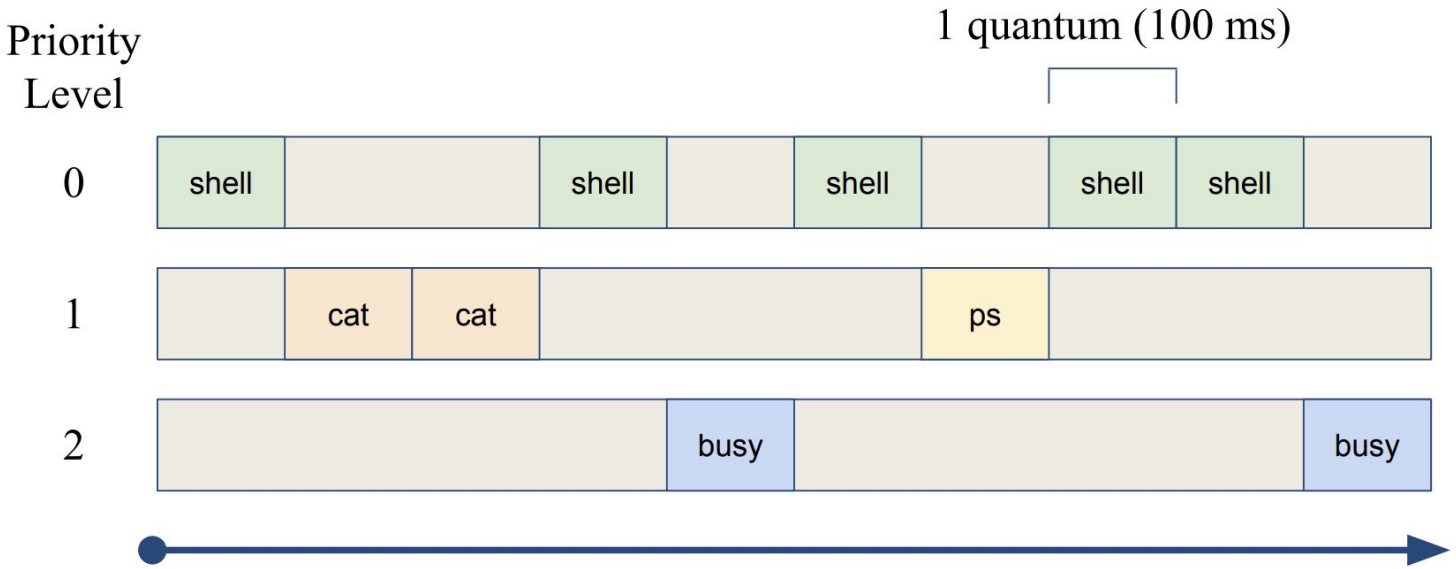
Provides additional tooling to:

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

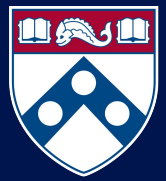
```
spthread_t new_thread;

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

Spthread for Scheduling

Leverage suspend + continue to execute one spthread at a time



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

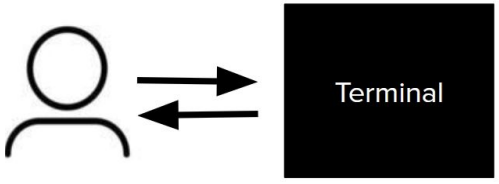
- nice
- nice_pid
- man
- bg
- fg
- jobs
- logout

- 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

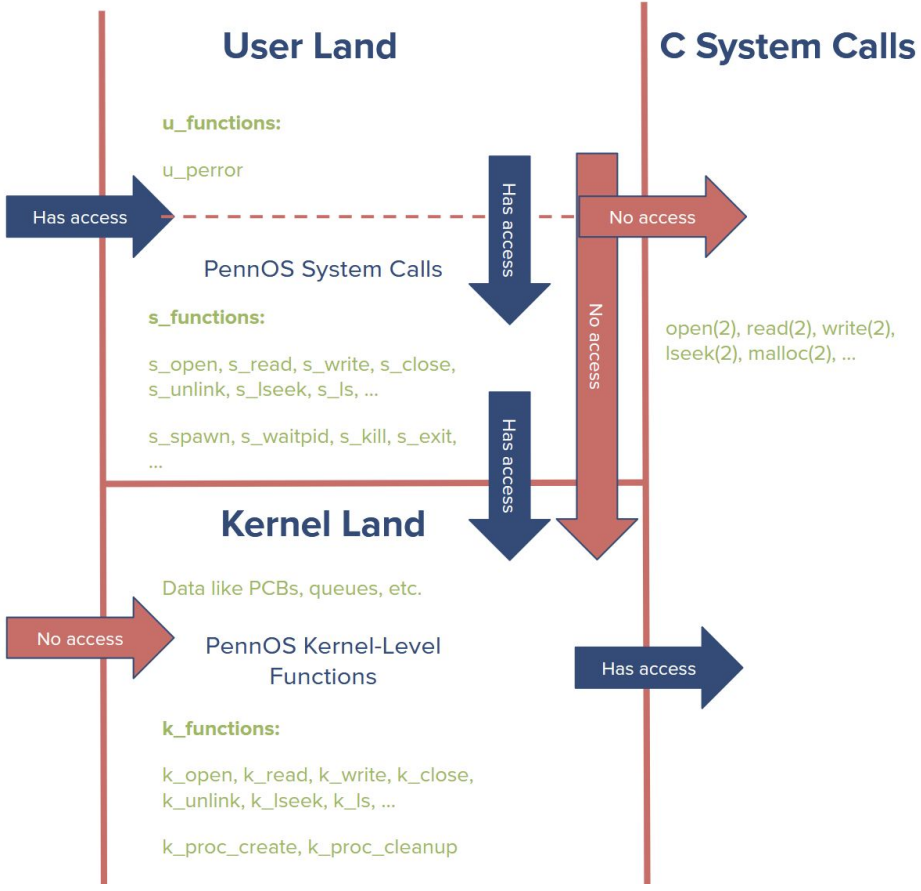
- errno.h
- u_perror
- Have global ERRNO macros
- Call u_perror for PennOS system call errors like s_open, s_spawn
- Call perror(3) for any host OS system call error like malloc(3), open(2)

Maintaining the Abstraction



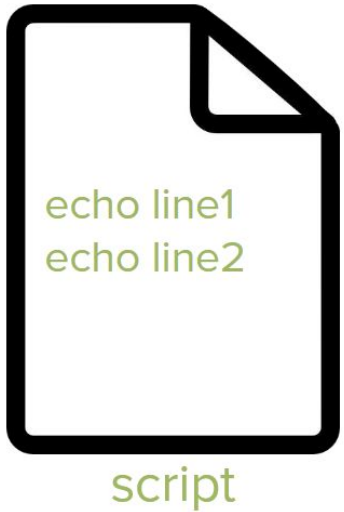
Shell Built-ins:
cat, sleep, busy, echo, ls, touch, mv, cp, rm, chmod, ps, kill, zombify, orphanify

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

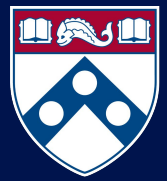


Shell Scripts

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

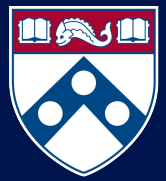


Demo



Penn Engineering
University of Pennsylvania

Questions?



Penn Engineering
University of Pennsylvania