

PennOS Lecture

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Maya Huizar Sid Sannapareddy

Administrivia

❖ PennOS:

- PennOS Due Dates milestones updated
- Groups have been assigned
- TA's have been assigned to groups
- You have the first milestone, which needs to be done before end of day Tuesday the 8th.
- Your group (or at least most of your group) needs to meet with your assigned TA and display the expectations laid out in the PennOS Specification

❖ I will post videos later containing some demos of a functioning PennOS.

Lecture Outline

- ❖ **PennOS Overview**
- ❖ Scheduling & Process Life Cycle
- ❖ Sphreads
- ❖ PennOS Shell
- ❖ PennFAT FS
- ❖ Kernel Functions vs System Calls vs User Level Functions
- ❖ Evaluation Overview

Projects so far

- ❖ Penn Shredder
 - Mini Shell with Signal Handling

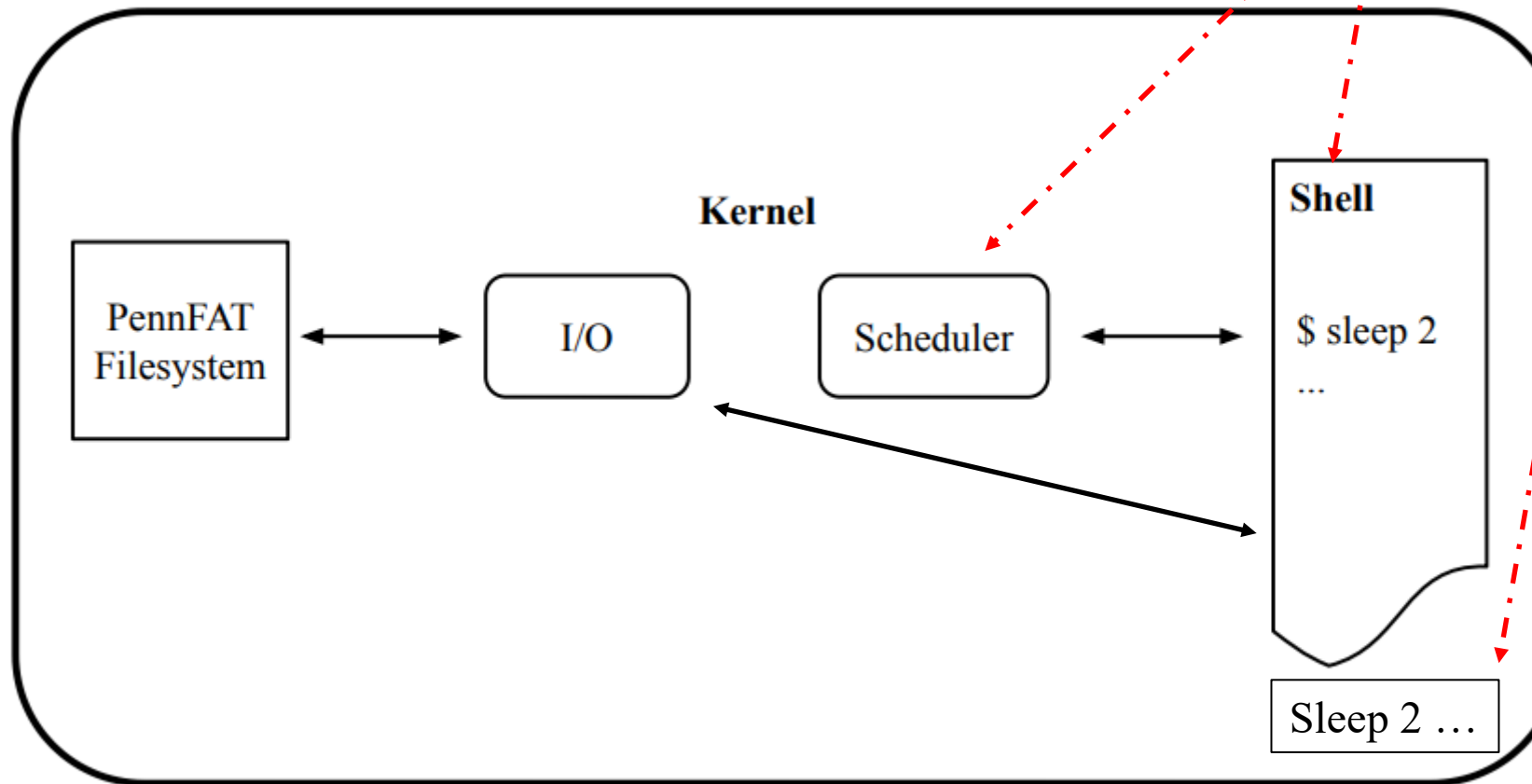
- ❖ Penn Shell
 - Redirections and Pipelines
 - Process Groups and Terminal Control
 - Job Control

- ❖ NEW: PennOS
 - You need to make the system calls that things like Penn-shell use
 - E.g. you will make your own “fork”

PennOS Diagram

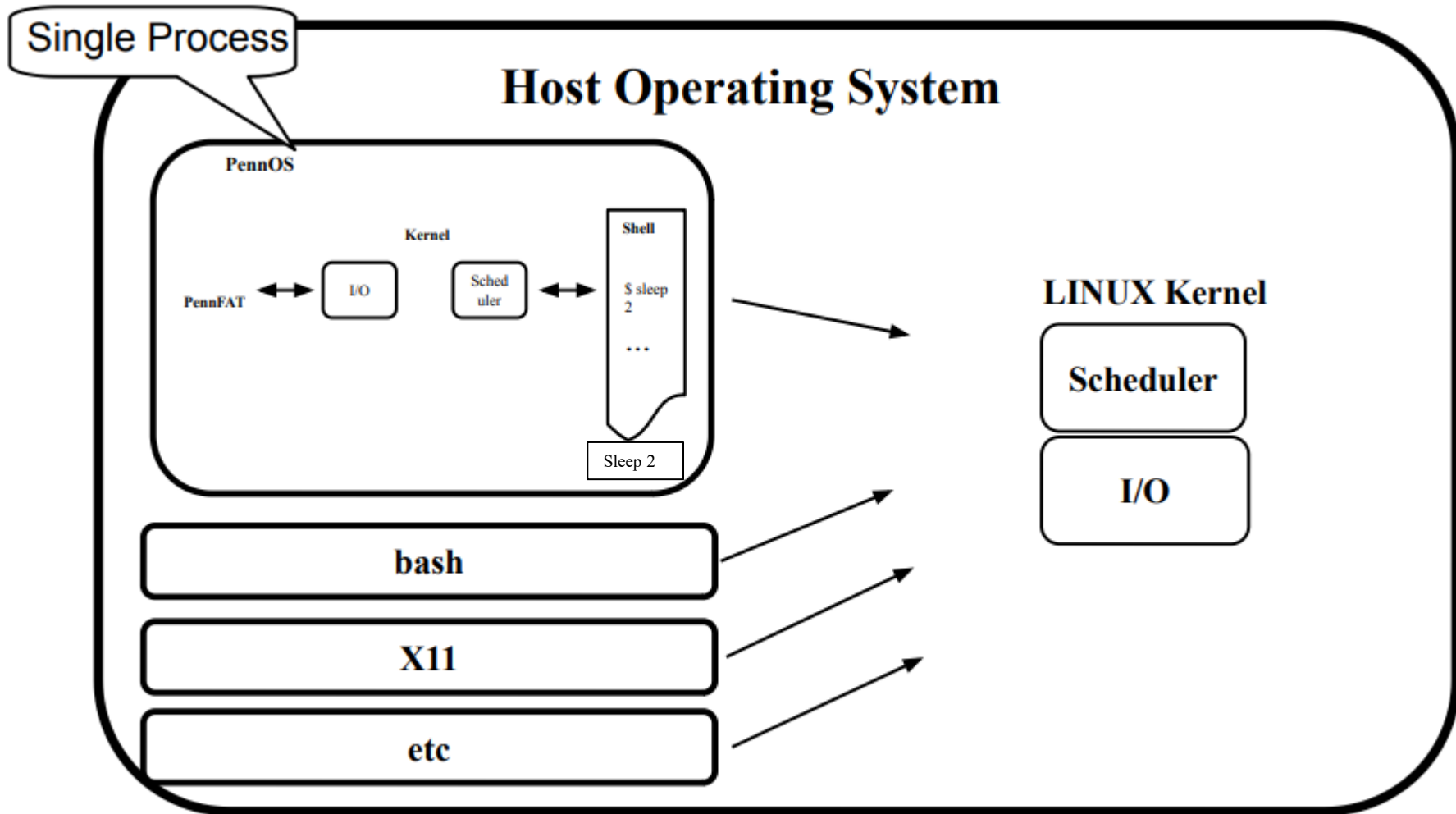
PennOS is made of many components that run as a single process

User programs (and the scheduler) run as separate threads in the process



PennOS as a Guest OS

PennOS runs as a single process on your operating system



PennOS Components

- ❖ Specification has many components, but if you want to split the work, the cleanest divide is between:
 - The file system
 - FAT
 - System Wide file table
 - Process level file descriptors
 - System calls for interacting with a file
 - FAT shell
 - The kernel
 - Scheduler
 - Signals
 - System calls that relate to processes directly (wait, fork, etc.)
 - shell

Brief Demo

- ❖ Brief Demo of how to run PennOS and how it is a single process
- ❖ It runs as a single process, we can see this with `ps`
- ❖ We use `ps -h` to see the individual threads in our PennOS

PennOS Abstraction

- ❖ Since we want you to build the OS yourself, you are mostly forbidden from using anything in the POSIX interface
 - E.g. you should never call “fork” we are making our own version of it
- ❖ There are some instances where it is allowed, but it is only allowed at the lowest level, when there is no alternative:
 - E.g. the read function should only be used at the lowest level of the file system to interact with our “file system” on the host machine. No user or PennOS system call should ever directly call these functions.
 - E.g. the thread functions should only be used at the lowest level of your scheduler/kernel. No user or PennOS system call should ever directly call these functions.

Lecture Outline

- ❖ PennOS Overview
- ❖ **Scheduling & Process Life Cycle**
- ❖ Sthreads
- ❖ PennOS Shell
- ❖ PennFAT FS
- ❖ Kernel Functions vs System Calls vs User Level Functions
- ❖ Evaluation Overview

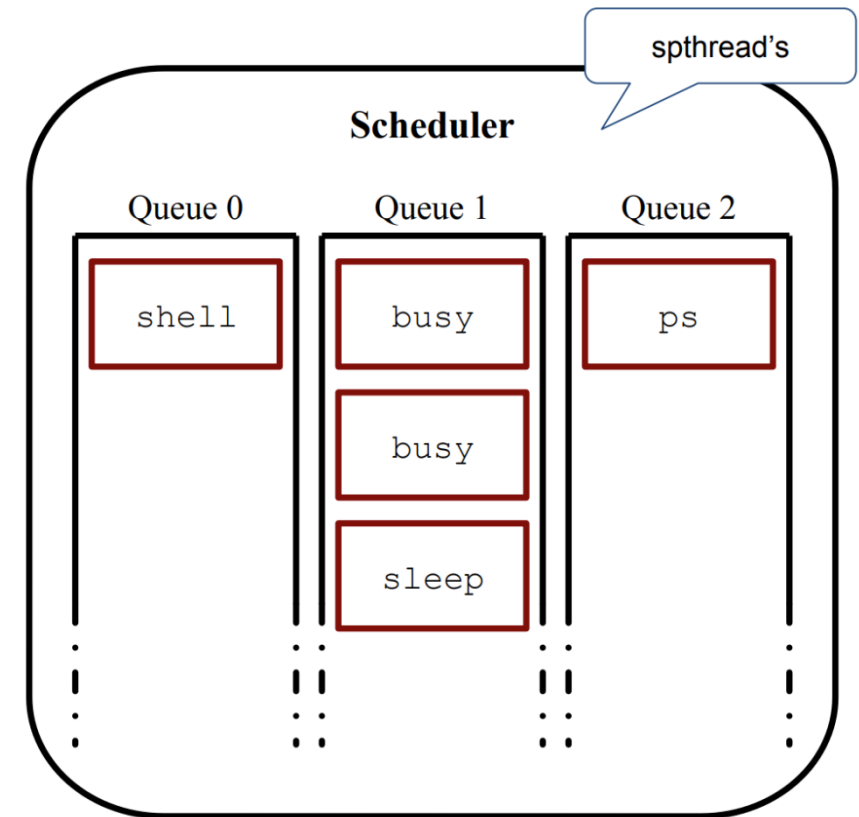
Scheduling in PennOS

❖ Algorithm: round-robin with three different queues

- Each queue acts like normal round robin within the queue
- Unlike some round robin variants, we do not have to empty Queue 0 before we move on to other Queues.

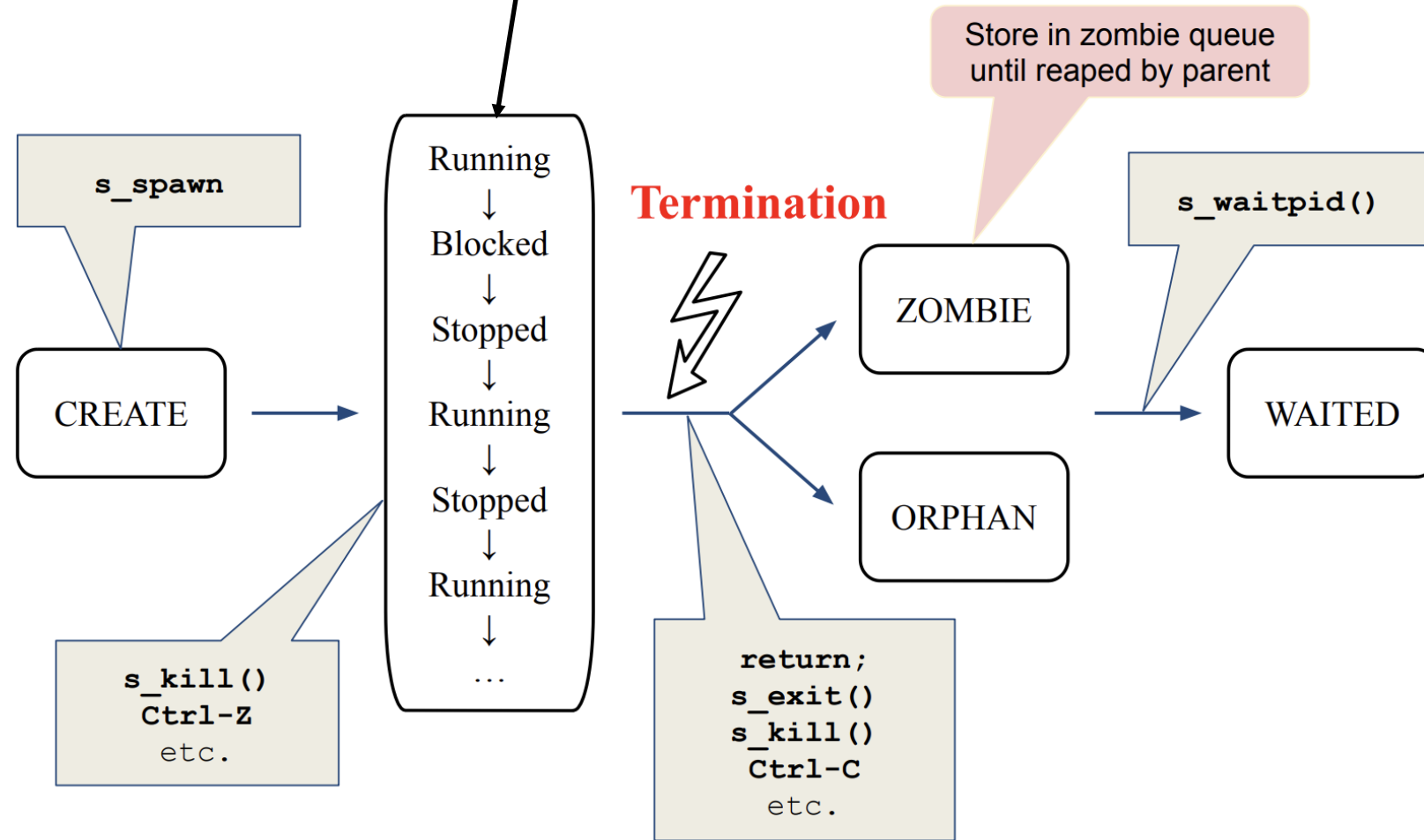
❖ Exponential Relationship:

- Queue 0 scheduled 1.5 times more frequently than Queue 1
- Queue 1 scheduled 1.5 times more frequent than Queue 2
- YOU MUST DO THIS DETERMINISTICLY (no random number generation)



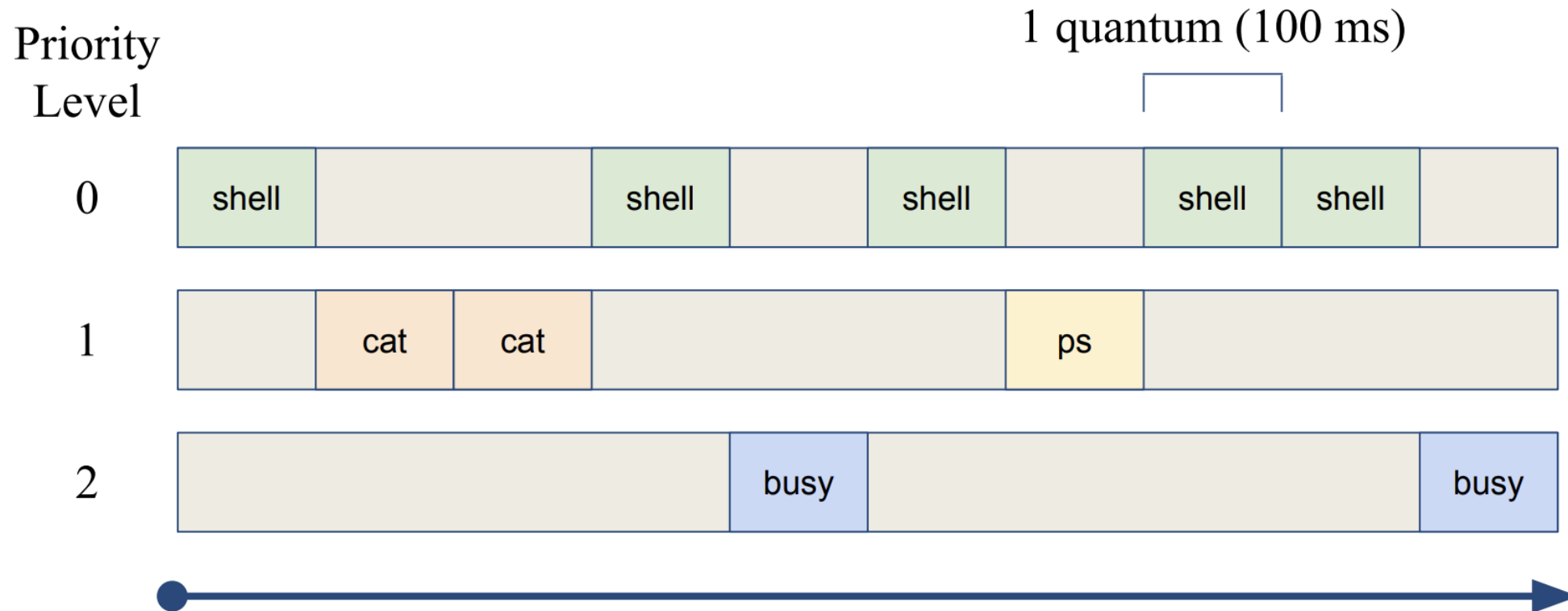
Process Life Cycle

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”.



Spthread is the unit of Scheduling

- ❖ Leverage suspend + continue to execute one spthread at a time



Process Control Block

```
typedef struct pcb {  
    pid_t pid;  
  
    int foo;  
    char *bar;  
  
} pcb_t;
```

- ❖ Should have:
 - handle to the spthread
 - PID, parent PID, child PID(s)
 - open file descriptors
 - priority level
 - process state
 - Probably a bunch more
- ❖ What is needed in a PCB to support waitpid()? What about sleep()?
 - Really think about this for milestone 1!

Implementation Tips:

- ❖ We provide a file called `sched_demo.c`
 - Implements basic scheduling of sptreads.
 - READ IT AND UNDERSTAND IT
 - Probably mimic the design of it for your real scheduler
- ❖ DO NOT PUT THE SCHEDULER IN A SIGNAL HANDLER
- ❖ We recommend putting very little in your signal handler. 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.
- ❖ YOUR SCHEDULER CANNOT USE RANDOM NUMBER GENERATION

PennOS Signals

- ❖ You need to implement our own “signals” for PennOS.
- ❖ Instead of registering handlers, 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.
- ❖ You should use the `kill`, or `pthread_kill`.
- ❖ You will have to 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.
- ❖ You will also likely make use of `setitimer` and `sigsuspend` for the scheduler ticks

Other 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
- ❖ 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)

Lecture Outline

- ❖ PennOS Overview
- ❖ Scheduling & Process Life Cycle
- ❖ **Spthreads**
- ❖ PennOS Shell
- ❖ PennFAT FS
- ❖ Kernel Functions vs System Calls vs User Level Functions
- ❖ Evaluation Overview

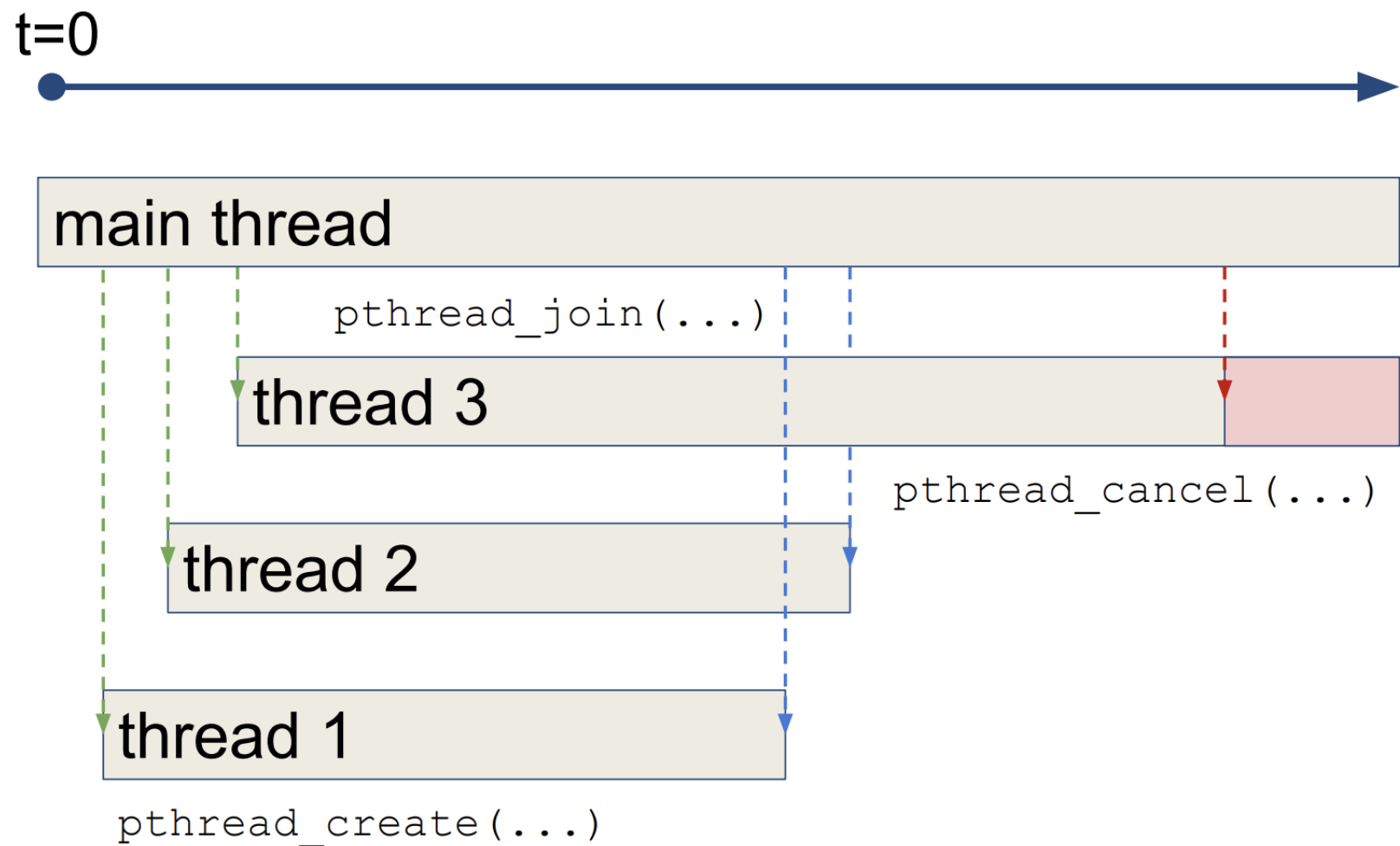
POSIX Threads

- ❖ User-level thread management API
 - ❖ Isolate code execution with distinct threads
 - ❖ Resource sharing (within same address 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 pthreads work



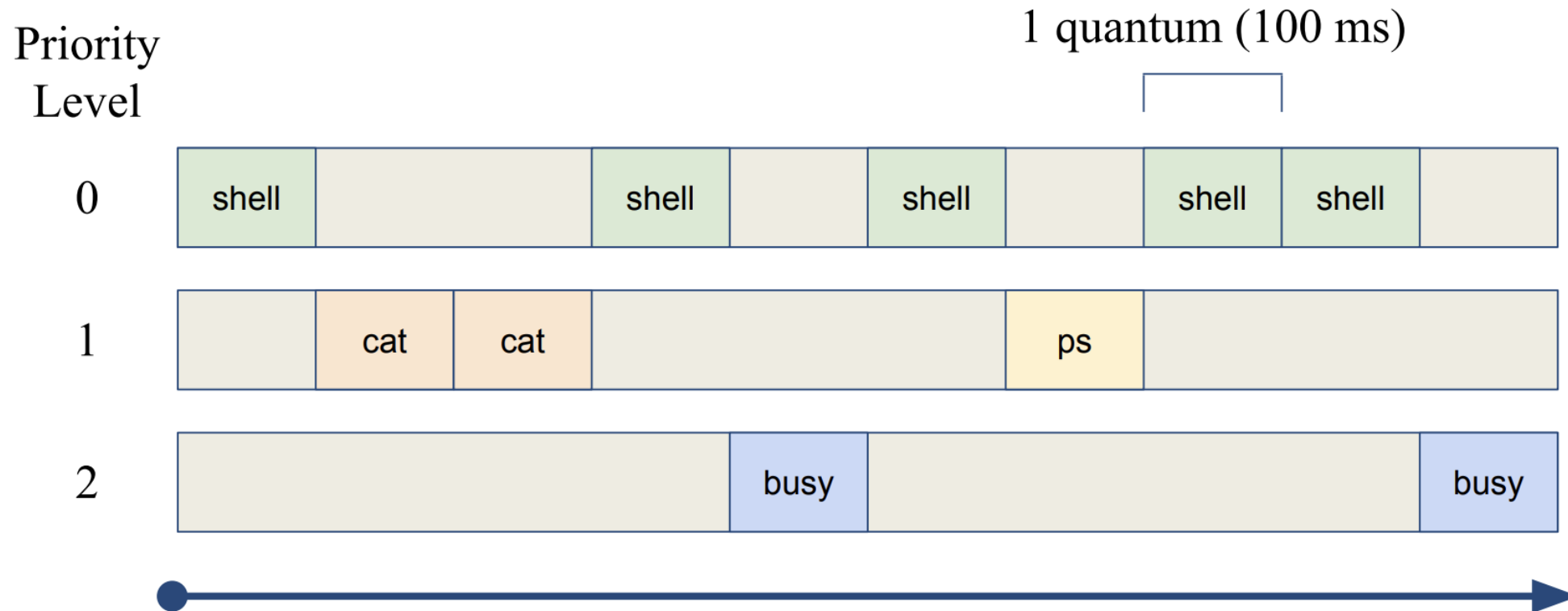
Spthread

- ❖ 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 is the unit of Scheduling

- ❖ Leverage suspend + continue to execute one spthread at a time



Spthreads + 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();`

Lecture Outline

- ❖ PennOS Overview
- ❖ Scheduling & Process Life Cycle
- ❖ Sphreads
- ❖ **PennOS Shell**
- ❖ PennFAT FS
- ❖ Kernel Functions vs System Calls vs User Level Functions
- ❖ Evaluation Overview

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

Shell Scripts

- ❖ Your shell will need to support scripts:

```
$ echo echo line1 > script
```

```
$ echo echo line2 >> script
```

```
$ cat script
```

```
echo line1
```

```
echo line2
```

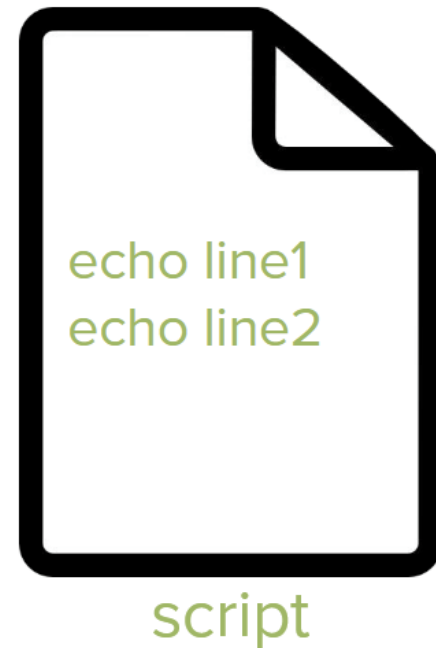
```
$ chmod +x script
```

```
$ 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.
 - Fork is very different than the `s_spawn` function we have you use
 - You don't have to support piping
 - Signals are different entirely

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?

Lecture Outline

- ❖ PennOS Overview
- ❖ Scheduling & Process Life Cycle
- ❖ Sphreads
- ❖ PennOS Shell
- ❖ **PennFAT FS**
- ❖ Kernel Functions vs System Calls vs User Level Functions
- ❖ Evaluation Overview

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 (File Allocation Table)

- ❖ This table is called the **File Allocation Table (FAT)**
- ❖ This table is in memory when it is running
- ❖ Table stored in disk initially, loaded into memory when computer is booted.

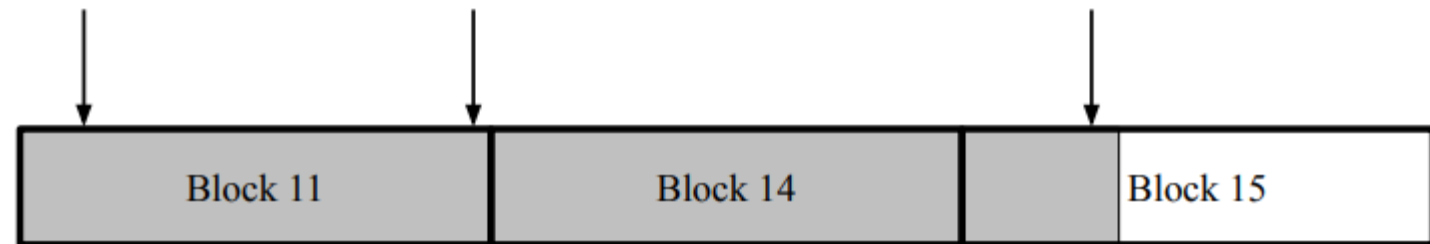
| Block # | Next |
|---------|----------------|
| 0 | BITMAP/SPECIAL |
| 1 | END |
| 2 | 6 |
| 3 | 9 |
| 4 | END |
| 5 | EMPTY / UNUSED |
| 6 | 3 |
| 7 | END |
| 8 | END |
| 9 | END |
| 10 | 8 |
| 11 | END |

Disk:

| | | | | | | | | | | | |
|-----|----------|--------|--------------|--------|-------|--------------|--------|--------------|--------------|--------|--------|
| FAT | Root Dir | File D | File D Blk 3 | File B | Empty | File D Blk 2 | File A | File C Blk 2 | File D Blk 4 | File C | File E |
| B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | B10 | B11 |

File Alignment

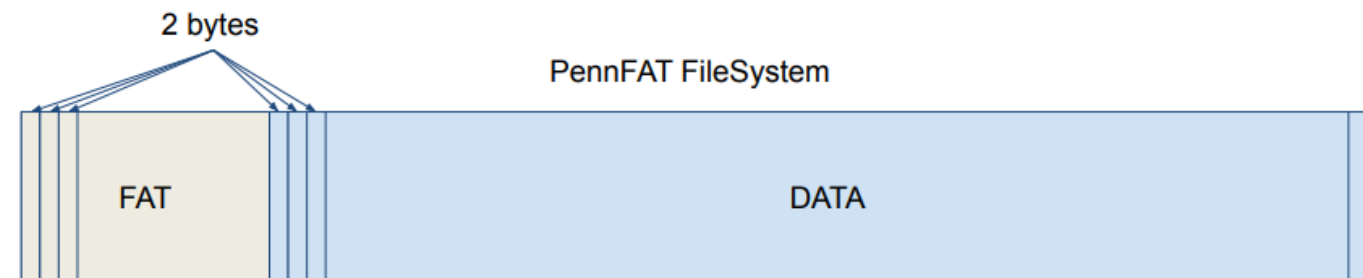
- ❖ Files are distributed along blocks
- ❖ Note: this is the logical view of a file. From the user of the file, all data in the file is contiguous, but it is really split up across three blocks that may not actually be contiguous.



FAT File System Layout

- ❖ FAT region and DATA region
 - The FAT region is an array of unsigned, little endian, 16-bit entries
 - The DATA region contains the actual data blocks of files & directories

| 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 |



PennOS FAT Details

- ❖ If we have N entries in the File Allocation Table, we only have $N - 1$ references to data blocks in the FAT
- ❖ The first File Allocation Table entry **FAT** [0] holds meta data about the FAT, so it doesn't refer/point to a “real” block
- ❖ An entry is 16-bits, which is 2 bytes.
- ❖ Consider the example 2-byte value: 0x2004
 - We can split this into two bytes for **FAT** [0]
 - The MSB (Most Significant Byte) 0x20 -> 32 in decimal
 - The LSB (Least Significant Byte) 0x04 -> 4 in decimal

PennOS FAT[0] MSB






- ❖ The first FAT entry **FAT** [0] holds meta data about the FAT, so it doesn't correspond to a “real” block
- ❖ Consider the example 2-byte value: 0x2004
 - We can split this into two bytes
 - The MSB (Most Significant Byte) 0x20 -> 32 in decimal
 - The LSB (Least Significant Byte) 0x04 -> 4 in decimal
- ❖ The MSB is size of the File Allocation Table in units of blocks
 - in this example, the FAT is 32 blocks

PennOS FAT[0] LSB

- ❖ The first FAT entry **FAT**[0] holds meta data about the FAT, so it doesn't correspond to a "real" block
- ❖ Consider the example 2-byte value: 0x2004
 - We can split this into two bytes
 - The MSB (Most Significant Byte) 0x20 -> 32 in decimal
 - The LSB (Least Significant Byte) 0x04 -> 4 in decimal
- ❖ The LSB is between 0 and 4, and specifies the size of the blocks for the file system

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

PennOS FAT Entry Special Values

- ❖ A PennFAT entry is 16-bits and only contains the block number of the next block in the file.
- ❖ There are two special values a PennFAT entry can hold
 - ❖ 0x0000 (0 in decimal)
 - Indicate the block is free.
 - We start indexing into our blocks in the data region starting with index 1     
 - ❖ 0xFFFF (65535 as unsigned, -1 as signed)
 - Indicates that there is no block after this logically in the file
 - That this is the last block in the 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?

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?

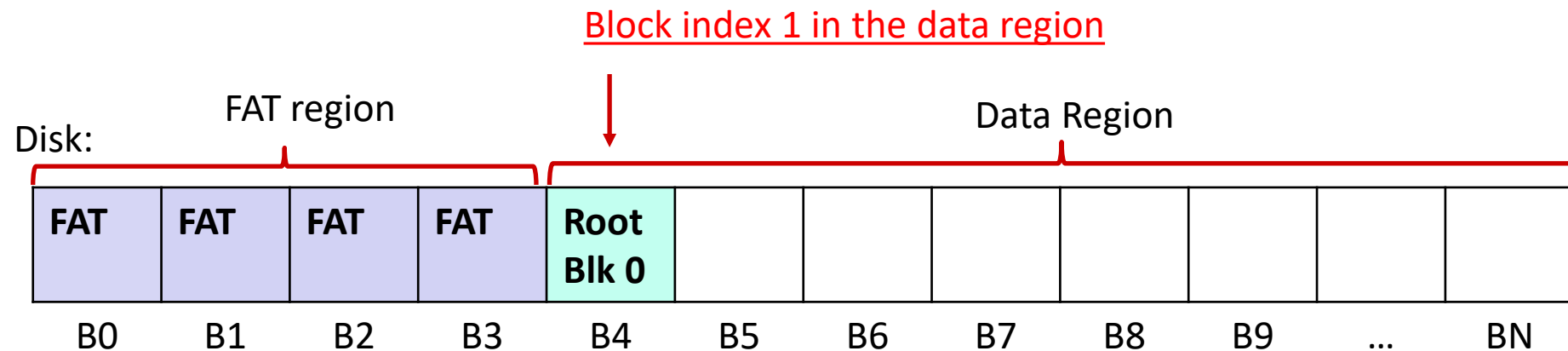
- 0xFFFF is reserved for last block of file

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 |

PennOS root Directory

- ❖ PennFAT has a special value for **FAT** [1] as well.
- ❖ It still corresponds to a data block, but that data block is the first block of the root directory
- ❖ This means we always know where the root directory starts. (at index 1 into the data region), and from there we can find all other files
 - *...pathname resolution soon...*



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. Should contain AT LEAST:

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

- ❖ Notably: the file name is 32 bytes (null terminated) and the first item in the directory entry.
 - legal characters: [A-Za-z0-9._-] (POSIX portable filename character set) letters, digits, . _ -
- ❖ first byte (char) 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

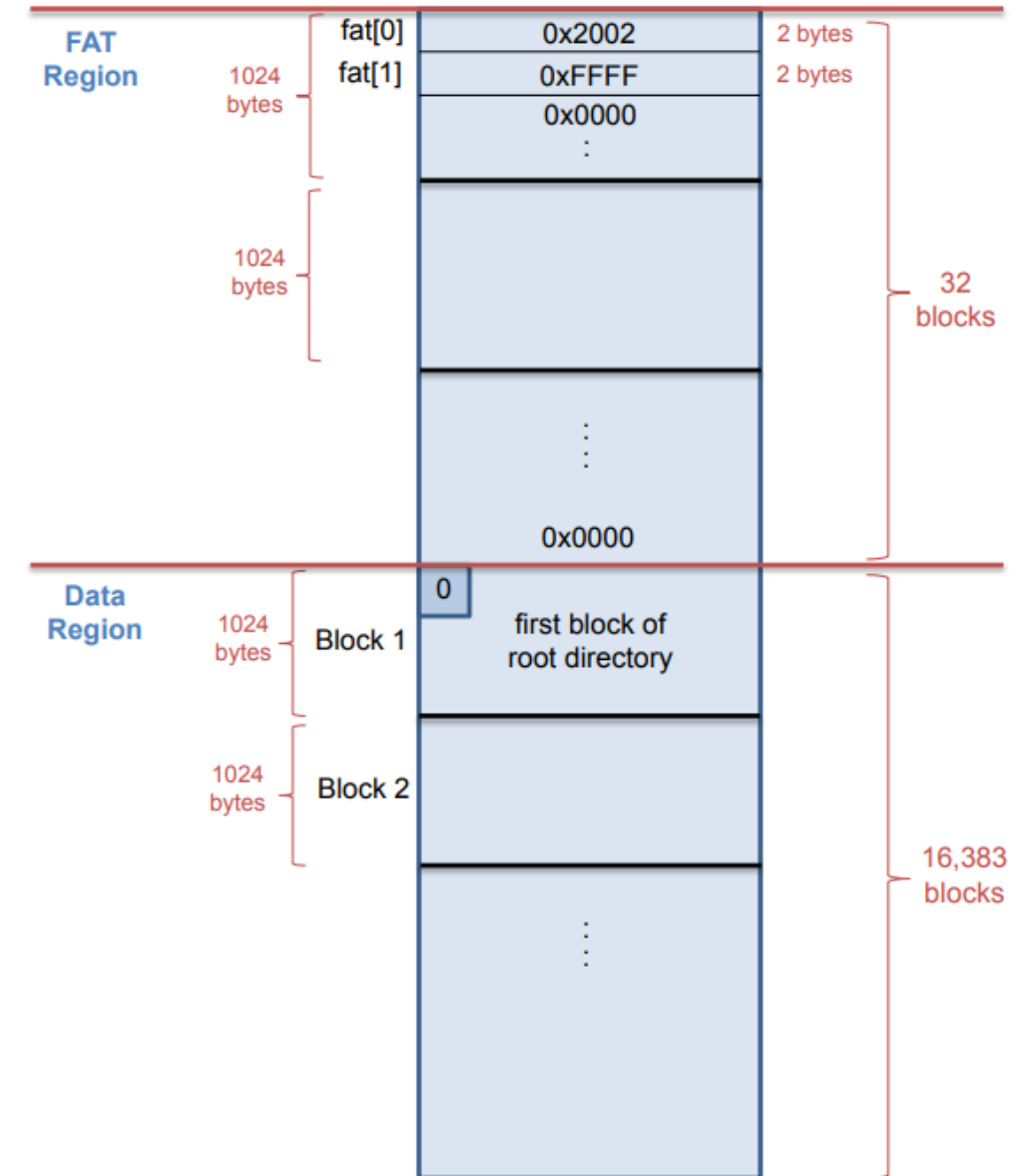
| Value | File Type |
|-------|------------------------------|
| 0 | unknown |
| 1 | regular file |
| 2 | directory |
| 4 | symbolic link (extra credit) |

- ❖ file permission: 1 byte
- ❖ timestamp: 8 bytes returned by time(2)
- ❖ remaining 16 bytes: reserved for E.C
(If you don't have EC you may have to pad the struct out to be 64 bytes)

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

Example

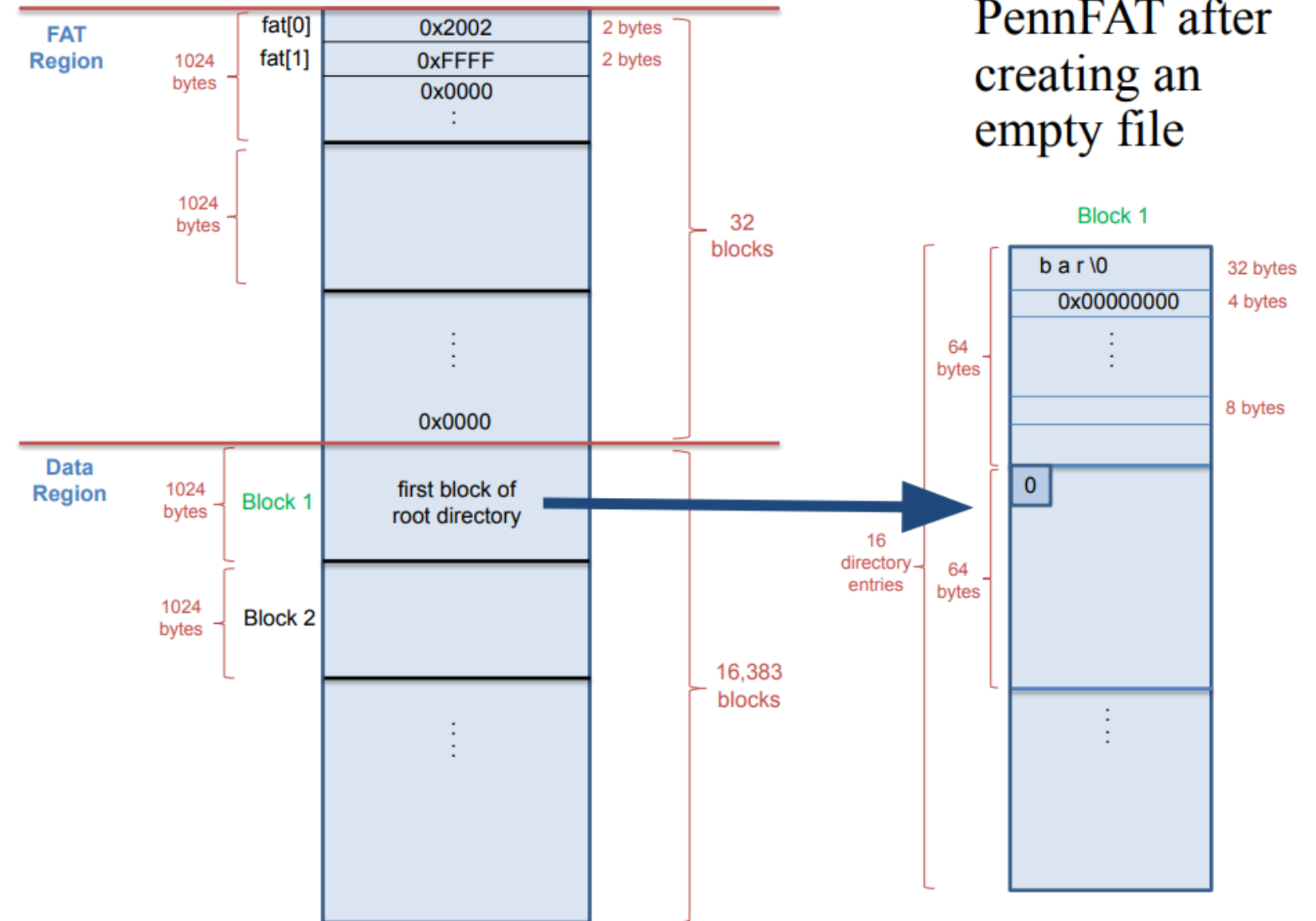
- ❖ $\text{fat}[0] = 0x2002$
 - 32 blocks of 1024 bytes in FAT
- ❖ First block of Data Region is first block of root directory
- ❖ Correspondingly, $\text{fat}[1]$ refers to that Block 1. That file (root directory) end on block 1, so $\text{fat}[1]$ has value of $0xFFFF$



Creating a File

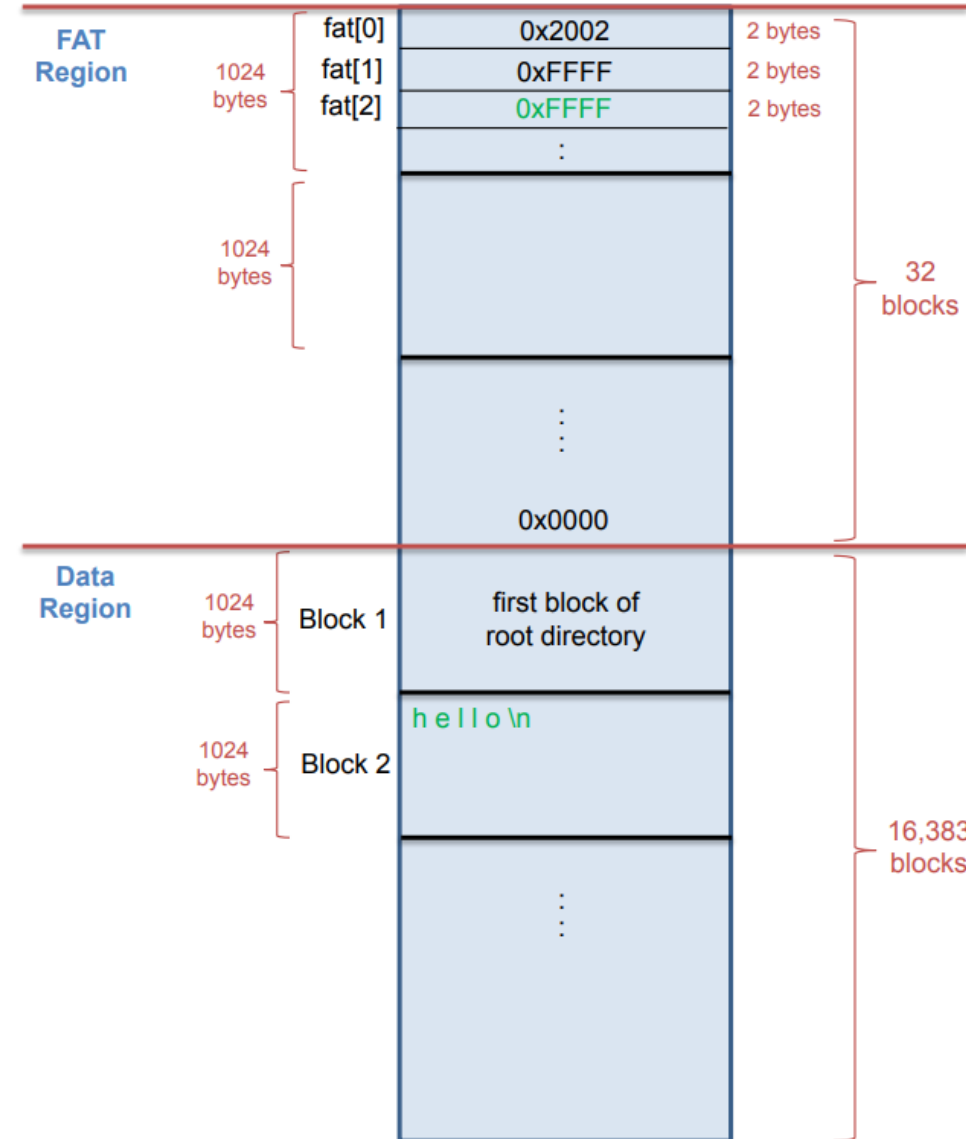
After creating an empty file called “bar”.

Note that the file bar starts at size = 0 and does not have a data block allocated to it until its size grows

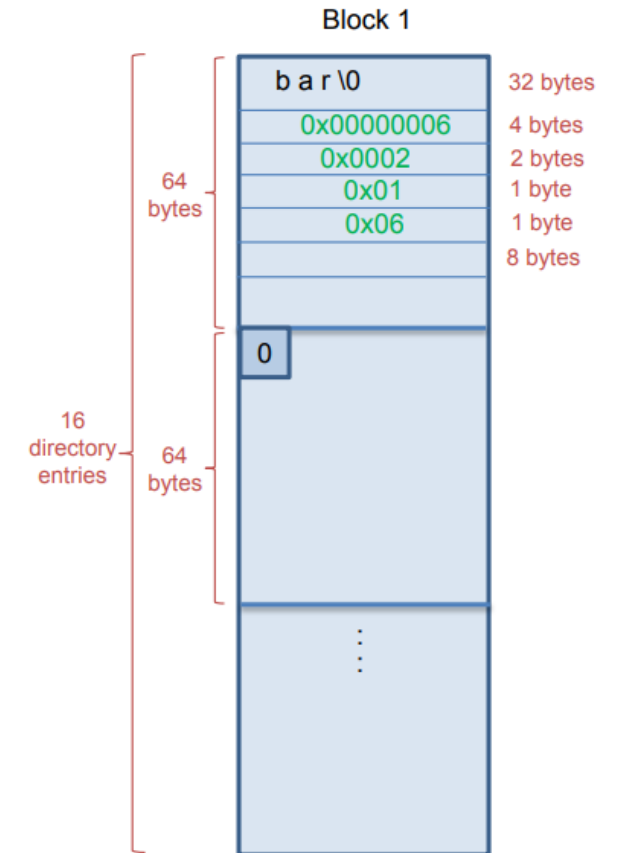


Writing to a File

- ❖ After writing “hello\n” to the file.
- ❖ FAT gets updated to represent the new block allocated to a file
- ❖ Directory entry also gets updated

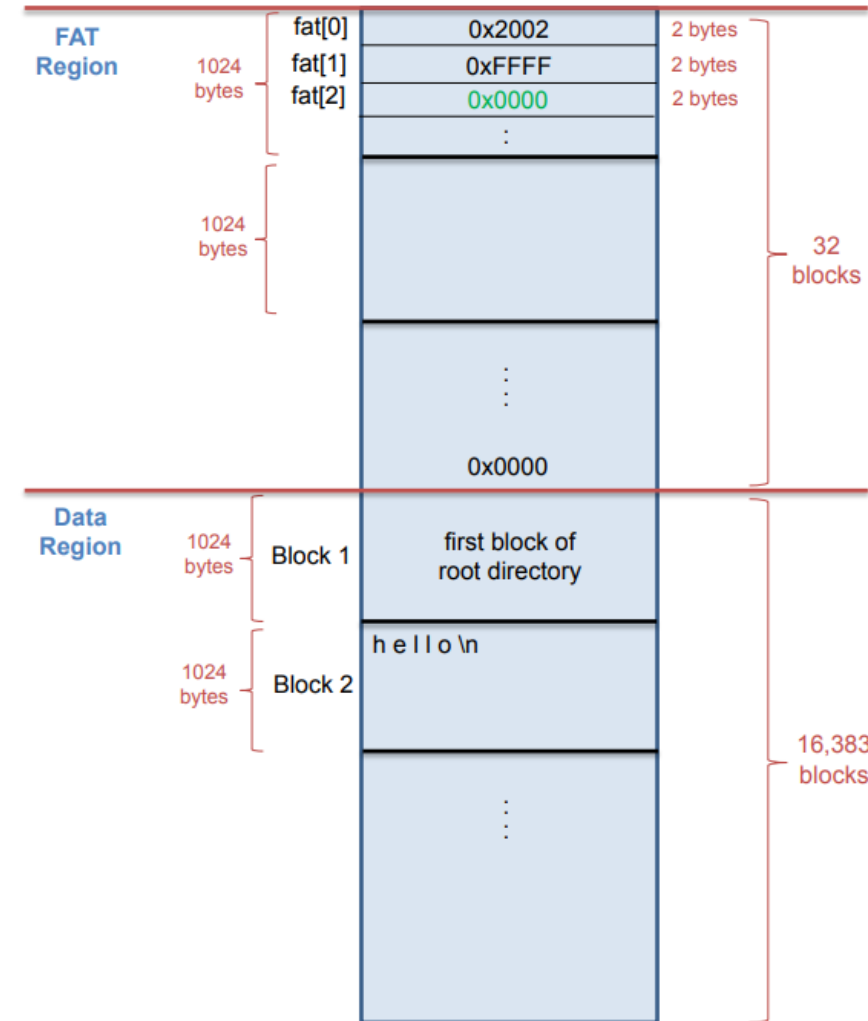


PennFAT after writing to the file

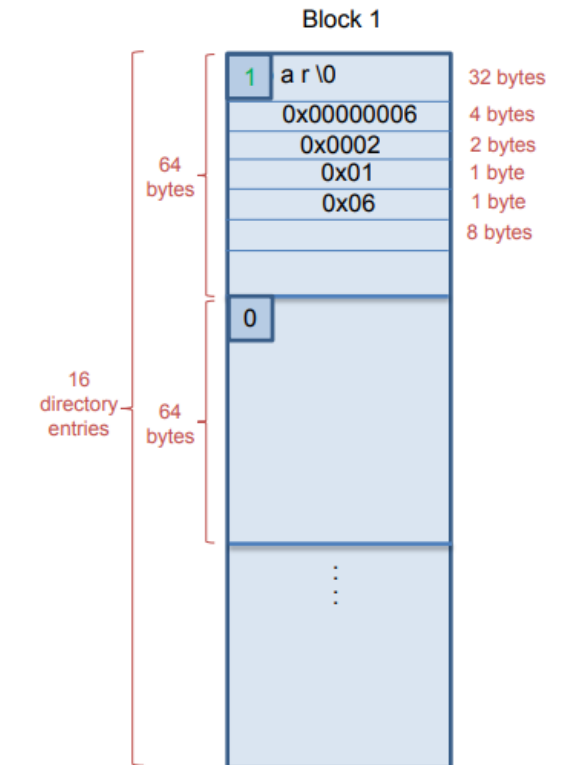


Removing the file

- ❖ After removing the file “Bar”
- ❖ Notably all we did was set the first byte of the directory entry to 1 and file contents are still in a data block.
 - “Lazy deletion”
 - You don’t have to follow this strictly



PennFAT after removing the file



Standalone PennFAT

- ❖ Milestone 1
- ❖ Implementation of kernel-level functions (k_functions)
 - k_functions deal with and manage the global system-wide file table.
- ❖ 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`
 - (which is an index into global system-wide table)
 - Write up to `n` bytes of `str`
 - literally modify the binary filesystem you created.

Tip: defining new types

- ❖ You may find it useful to define new types to make a clearer distinction for whether a variable is a file descriptor for:
 - The system wide file descriptor table
 - The per-process file descriptor table
- ❖ Typedef to the rescue!
 - May be tempted to do this:

```
typedef int sys_fd;  
typedef int proc_fd;
```

but still allows for easy conversion between two types accidentally... ☹

```
sys_fd x = 5;  
proc_fd y = x; // no compiler warnig or error
```
 - If we define one or both as structs, it is a bit clunkier to use but also harder to accidentally convert from one to another.

```
typedef struct {  
    int fd;  
} sys_fd;
```

Standalone Routines

❖ Special Commands

- mfs, 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)`
 - `k_close(FILEs)`

Standalone Routines

- ❖ `cp [-h] SOURCE DEST` - copy contents from SOURCE to DEST in pennfat file system. 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)`

IMPORTANT IMPORTANT IMPORTANT

❖ PennOS Advice:

- In your FAT code you may do something like this:

```
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);
```

- Sometimes though, the write and lseek will return a success, but it won't actually write to your file system
- Most commonly happens with blocks near the end of the FAT (as in blocks not in the allocation table but show up shortly after the end of the allocation table)
- Most likely related to an issue between mmap and write
- Shows up inconsistently!
- What's the fix?
Just do it twice, that usually fixes it.
- Or: don't use mmap

```
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);  
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);
```

IMPORTANT IMPORTANT IMPORTANT

- ❖ If you mmap, you can't mmap the whole fs. You are only allowed to mmap the FAT region, not the data region.

Lecture Outline

- ❖ PennOS Overview
- ❖ Scheduling & Process Life Cycle
- ❖ Sphreads
- ❖ PennOS Shell
- ❖ PennFAT FS
- ❖ **Kernel Functions vs System Calls vs User Level Functions**
- ❖ Evaluation Overview

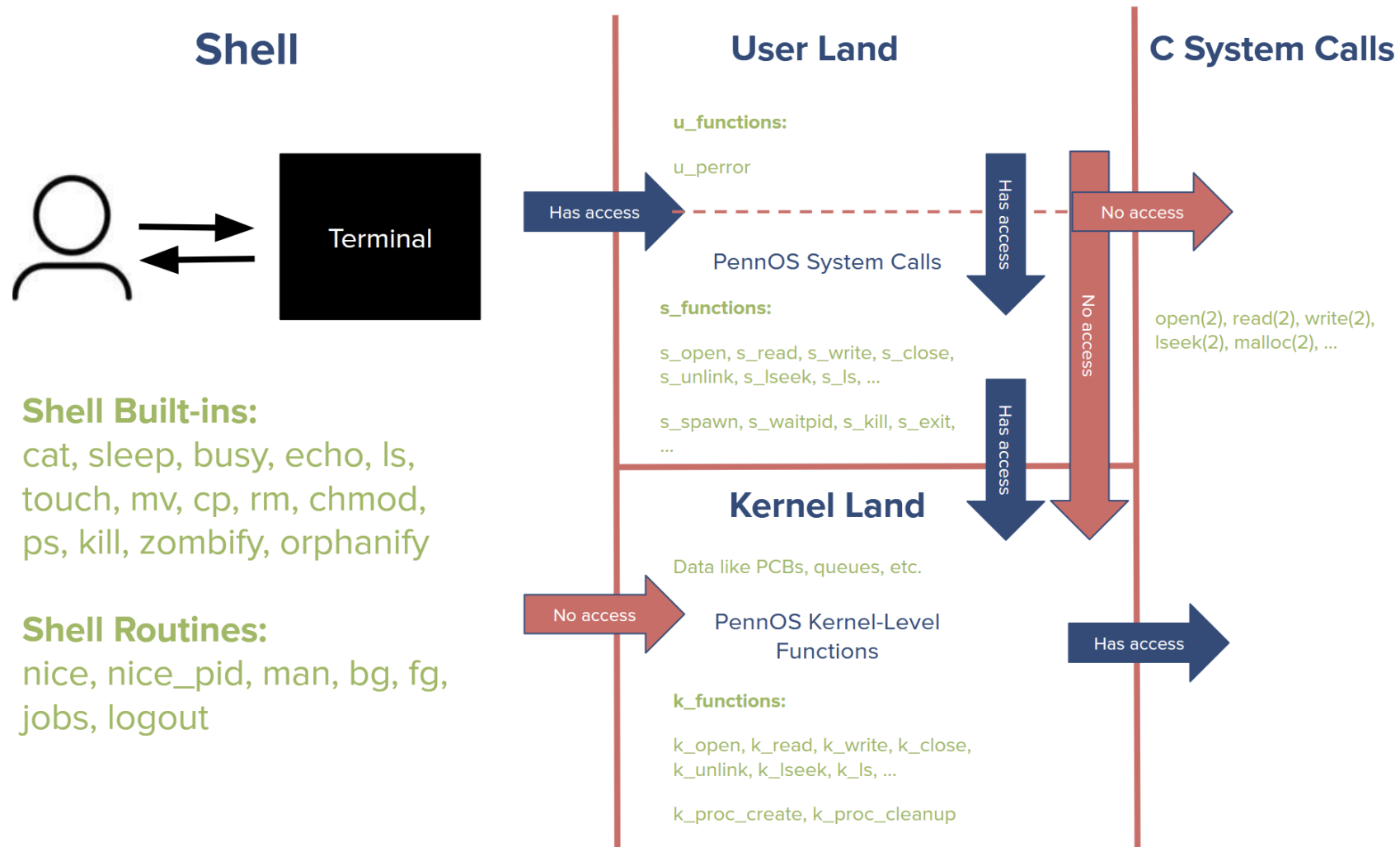
User, System, and Kernel Abstractions

- ❖ 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.

User, System, and Kernel Abstractions (Rationale sort of)

- ❖ 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



Error Handling

- ❖ make your own `errno.h` with your own `P_ERRNO` variable
 - Declare this `P_ERRNO` variable `_Thread_local`
 - Thread local variables are seemingly global, but there is a unique global per thread.
 - This makes sure that `errno` is not accidentally a data-race across threads.
- ❖ Make your own `u_perror` function to report `errno` issues
- ❖ 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)`
(should not happen often)

Lecture Outline

- ❖ PennOS Overview
- ❖ Scheduling & Process Life Cycle
- ❖ Sphreads
- ❖ PennOS Shell
- ❖ PennFAT FS
- ❖ Kernel Functions vs System Calls vs User Level Functions
- ❖ **Evaluation Overview**

Approximate Grading Breakdown

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

Due Dates

- ❖ Milestone 0: by July 8th
- ❖ Milestone 1: July 14th through 18th
- ❖ Final Submission: July 25th
- ❖ Demo: Anytime after you have submitted your final submission
 - We will ask you to pull the latest commit of your git repository that is before the submission deadline.
 - If you need to make a fix, you can “hotfix” it in the demo if needed for a minor deduction
- ❖ PRO TIP: give yourself more time for the integration than you think you need.

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 (Look into this sooner rather than later so you don't have to spend the last minute writing a bunch of comments. Comments will also help you and your teammates reason about and learn code)

- ❖ README
 - Describes implementation and design choices

That's all!

❖ See you next time!