CIS 3800
Penn-OS Lecture

Spring 2024

Milestone and Demo

Milestone 0: **Due by Mar. 25th (TA Meeting by Mar. 29th)**

Meeting with group and TA

General discussion regarding the design of your project

Pass/Fail grade

Milestone 1: **Apr. 9th (TA Meeting Apr 9-12)**

Meeting with group and TA

Autograded Standalone PennFAT, Scheduler & Logging Demo

Pass/Fail grade

Due: **Submission Apr 22nd / Demos Latest May 8th**

Present your PennOS to TA

Demo plan to be released at a later date

Development Grading Breakdown

5% Documentation

45% Kernel/Scheduler

35% File System

15% Shell

Companion Document/README

Required to provide a **Companion Document**

- Consider this like APUE or K-and-R
- Describes how OS is built and how to use it

`README`

- Describes implementation and design choices

Lecture Outline

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

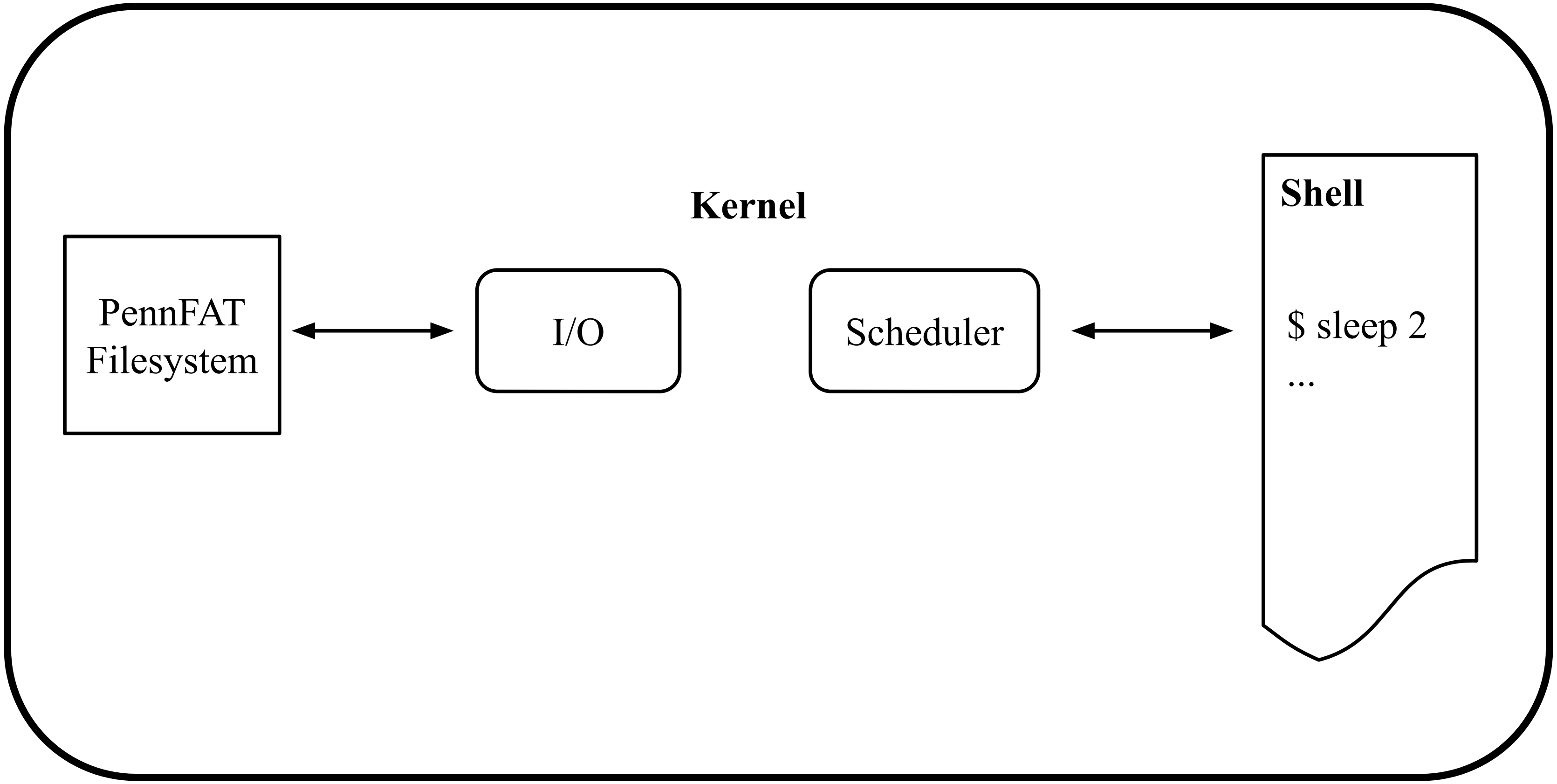
PennOS Overview

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



Kernel

PennFAT
Filesystem

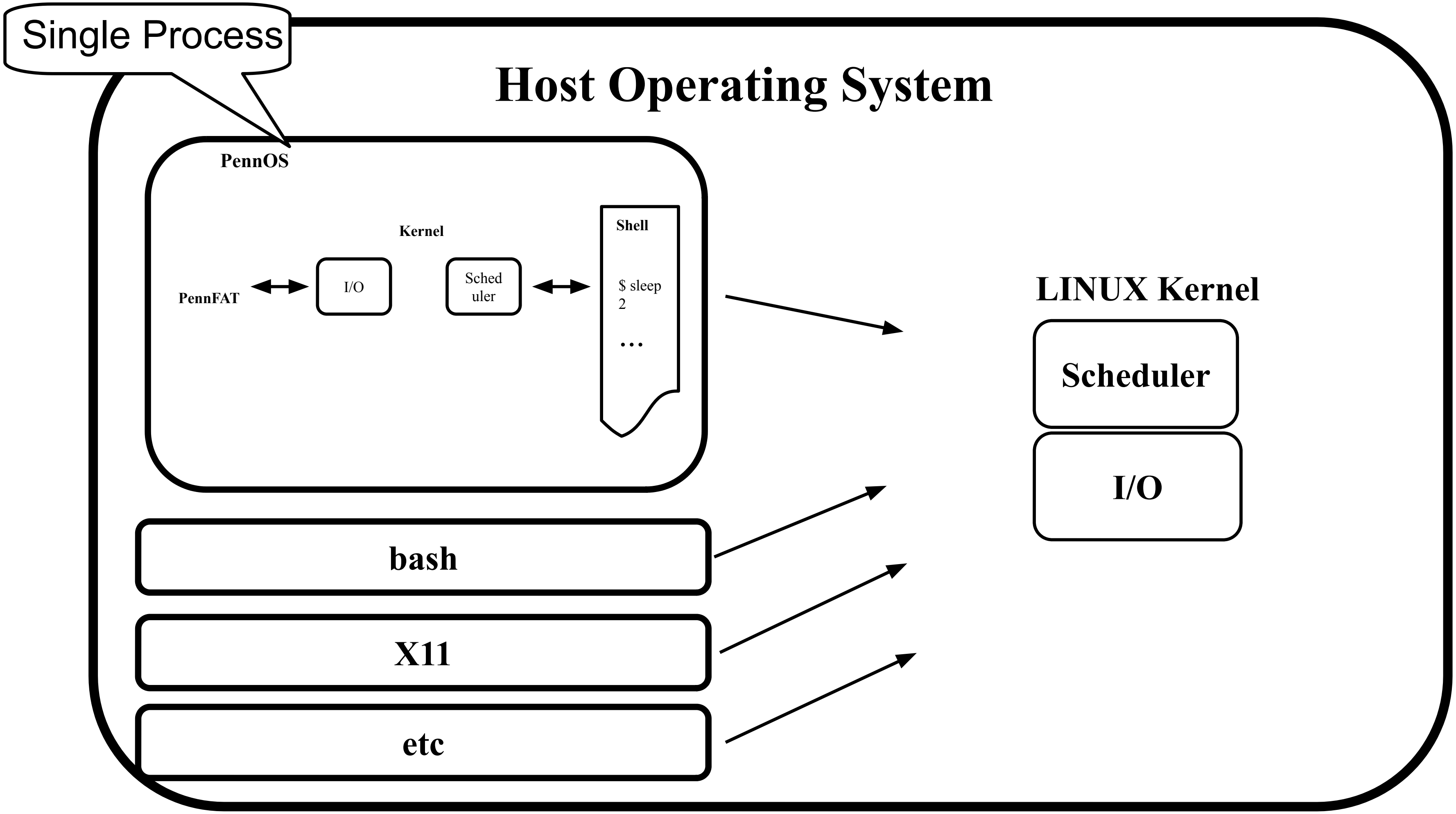
I/O

Scheduler

Shell

$ sleep 2
...

PennOS as a GuestOS



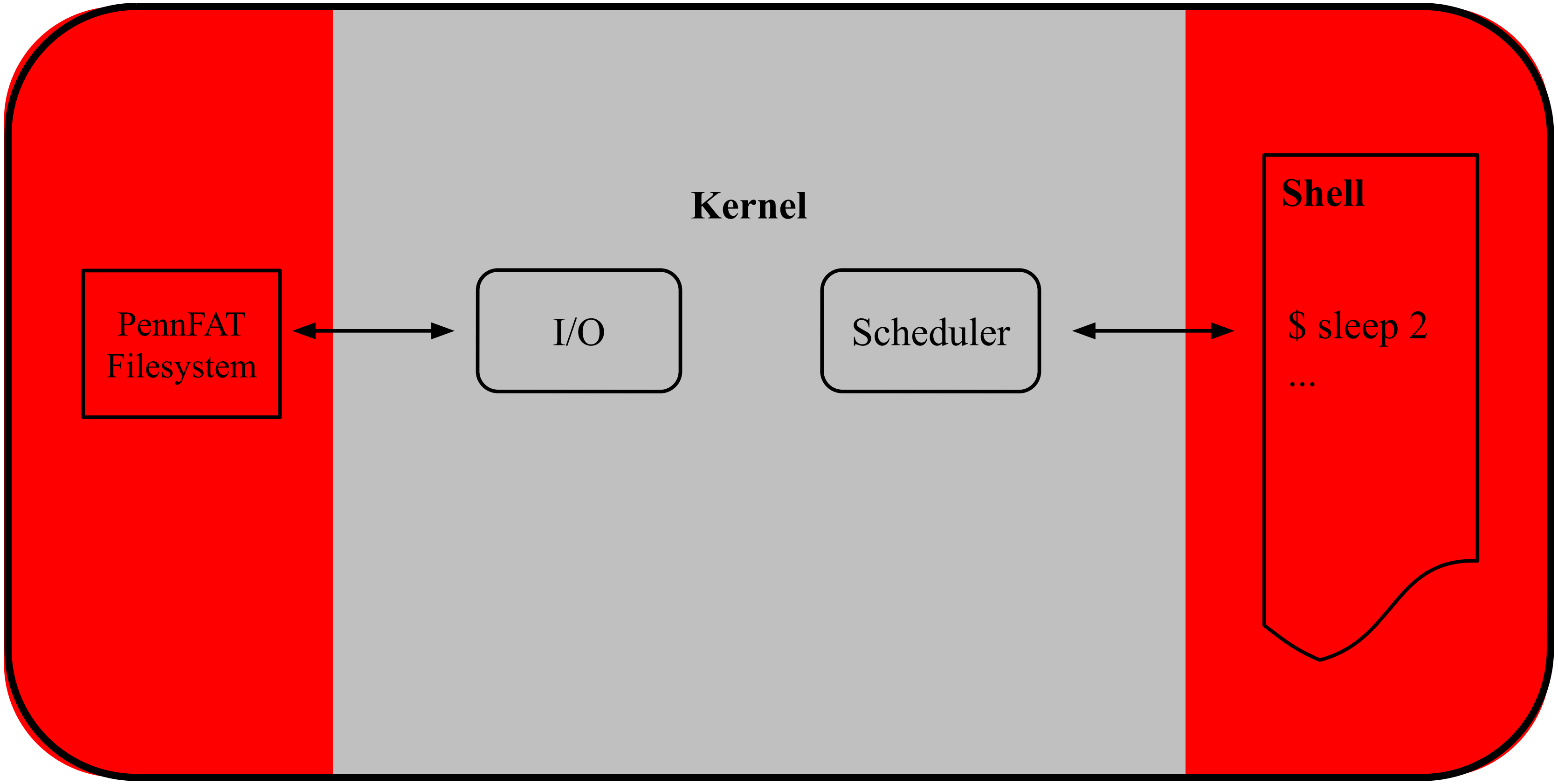
User Land and Kernel Land

User Land - What an actual user interacts with

Kernel Land - What happens ‘under the hood’

System Call - The API calls to connect user land with kernel land

User Land and Kernel Land



Kernel

PennFAT Filesystem

I/O

Scheduler

Shell

$ sleep 2

…

More on this later!!

PennFAT File System

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 a 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; modern operating systems support only their ‘native’ file system, for example:
  - FAT (DOS, Windows)
  - Mac OS X
  - ext{1,2,3,4} (Linux)
  - NTFS (Windows)

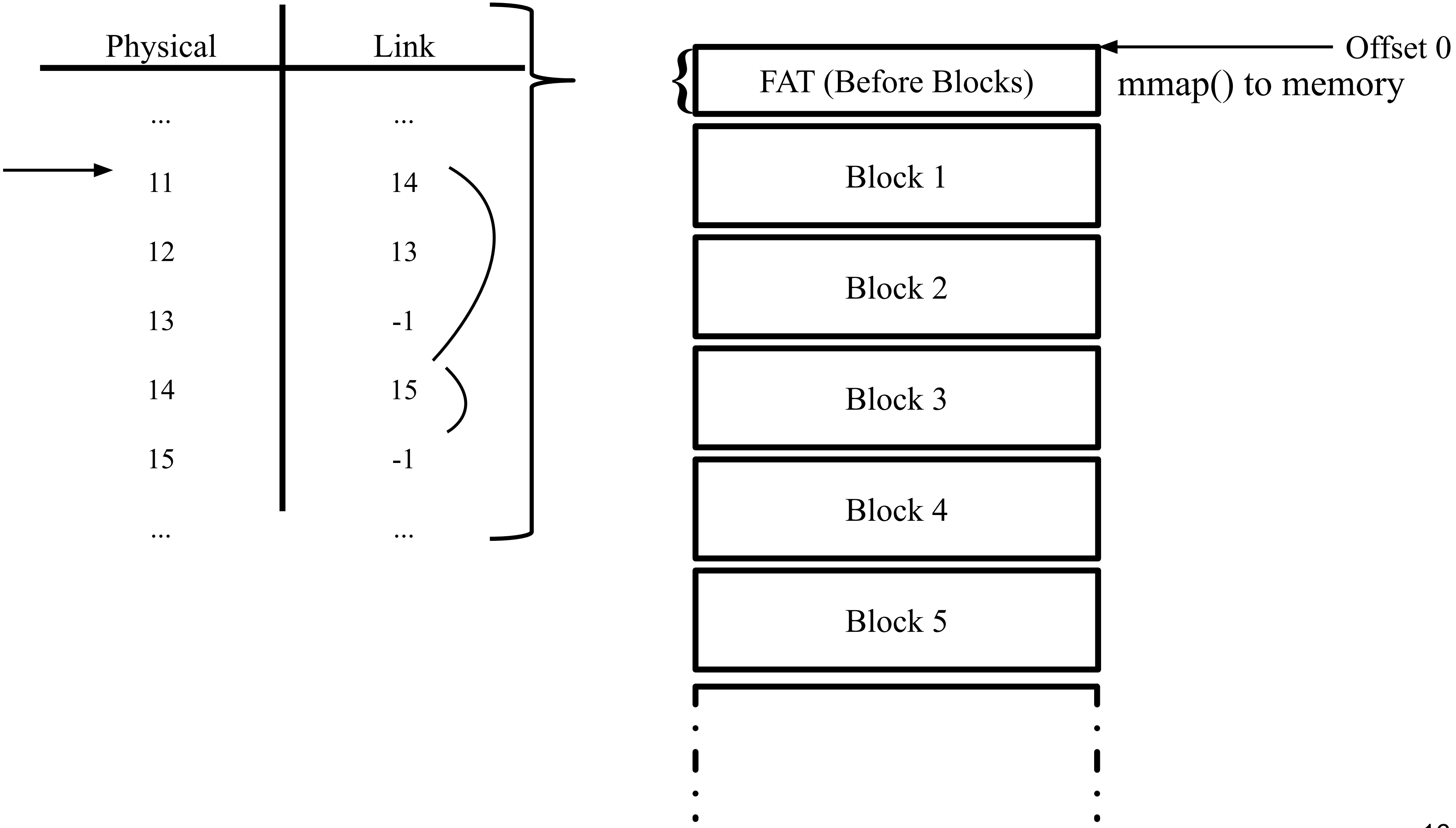
FAT

Each value in the FAT table refers to a **block number**

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

How can we read file 11?
Find Block 11, 14, and 15?

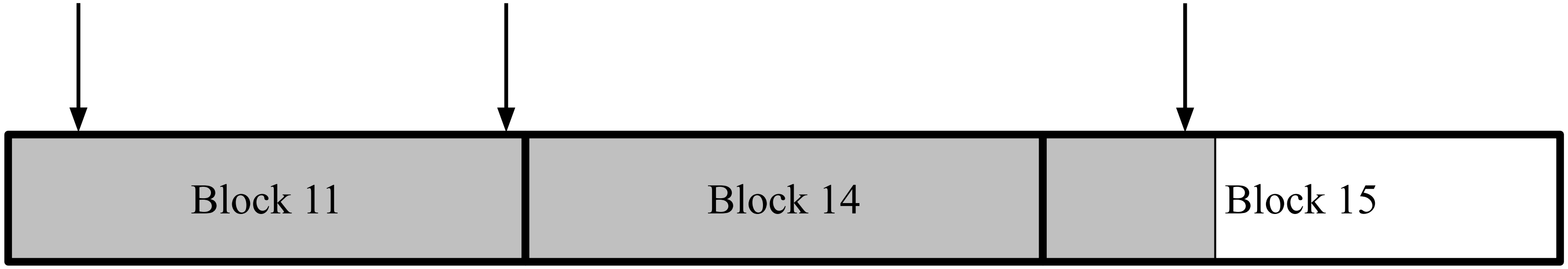
File System Layout



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

File Alignment

Files are distributed across **blocks**

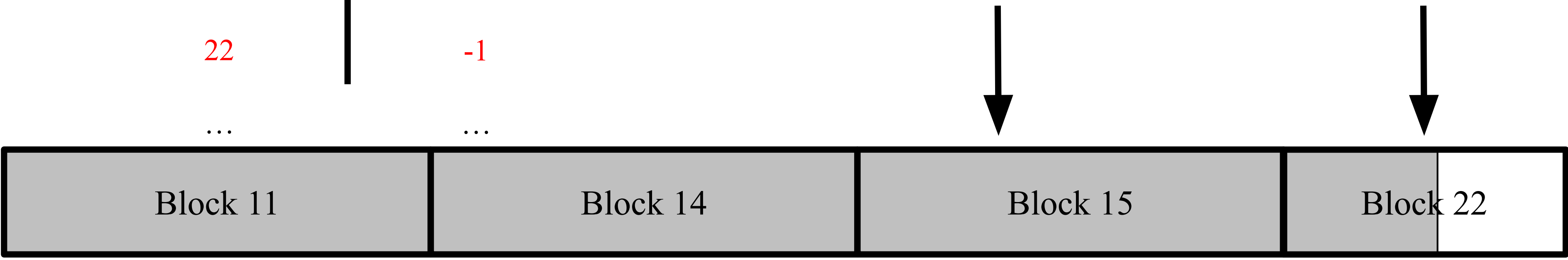


```
lseek(n, F_SEEK_SET, 60)
lseek(n, F_SEEK_SET, block_size - 1)
lseek(n, F_SEEK_SET, block_size * 2 + 100)
```

Adjusting File Size

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

`write(n, buffer, block_size)`



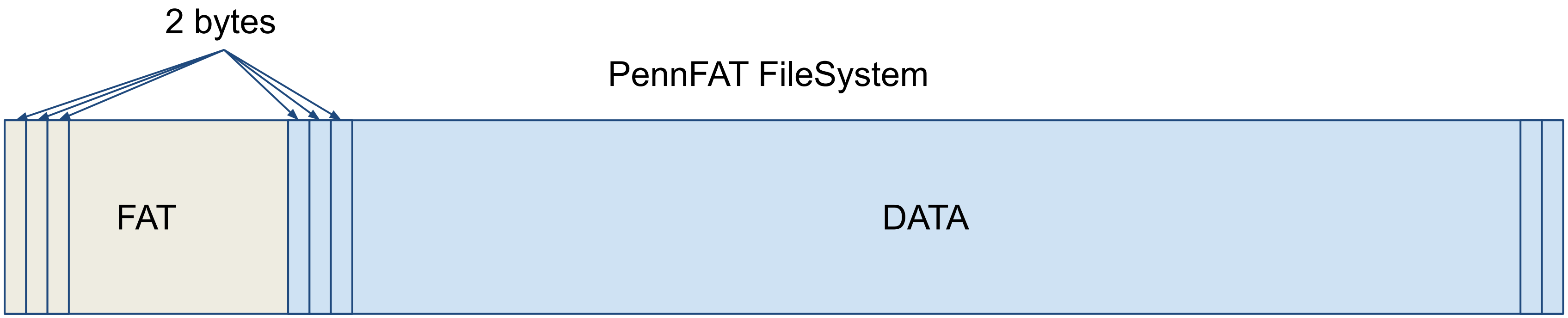
PennFAT Specification

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 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)

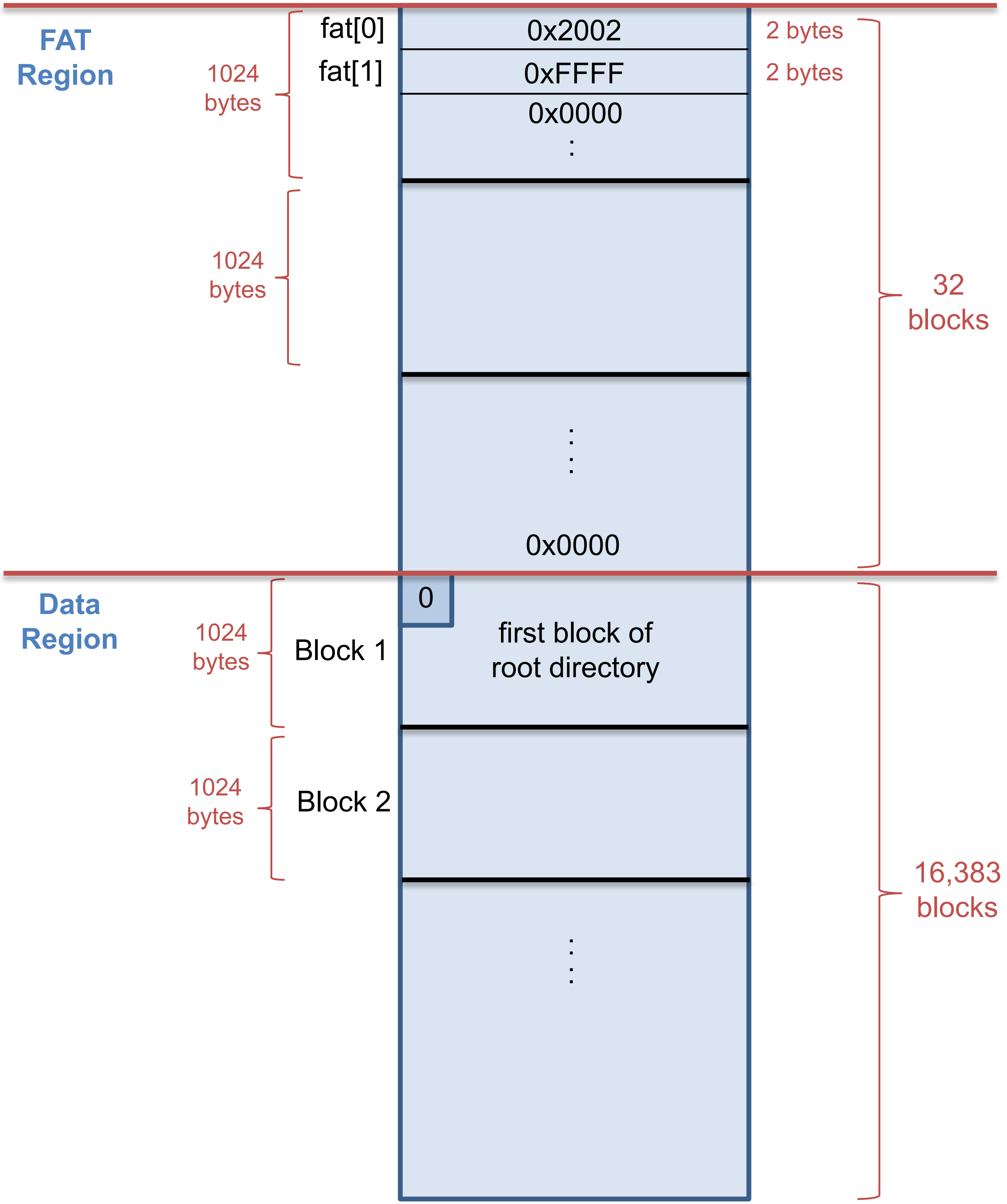
Directory entry (cont.)

- 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

PennFAT after initial formatting



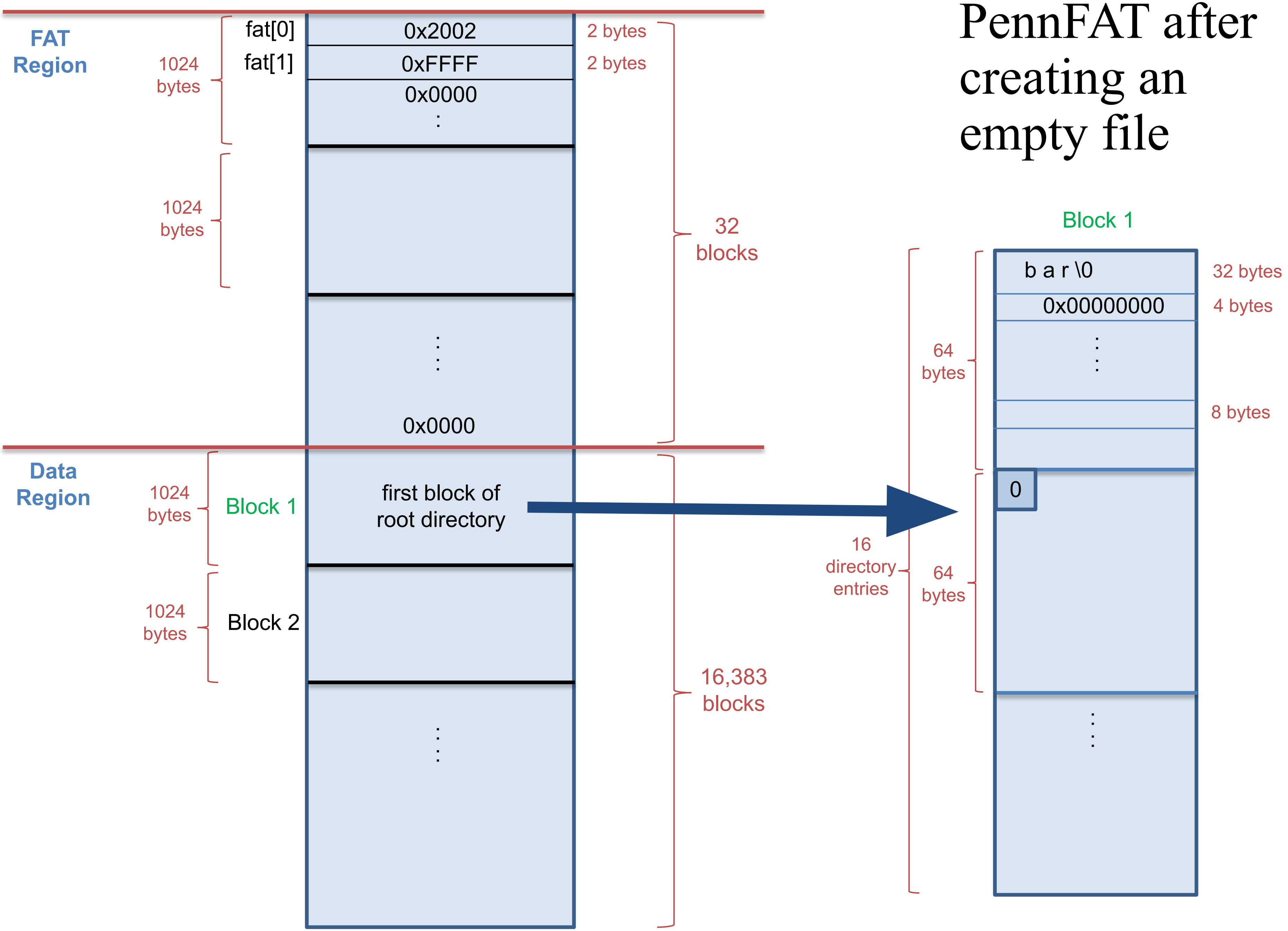
fat[0] = 0x2002

- 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

PennFAT after creating an empty file



FAT Region

1024 bytes

fat[0] 0x2002 2 bytes

fat[1] 0xFFFF 2 bytes

0x0000

1024 bytes

0x0000

32 blocks

Data Region

1024 bytes Block 1

first block of root directory

1024 bytes Block 2

16,383 blocks

Block 1

b a r \0 32 bytes

0x00000000 4 bytes

8 bytes

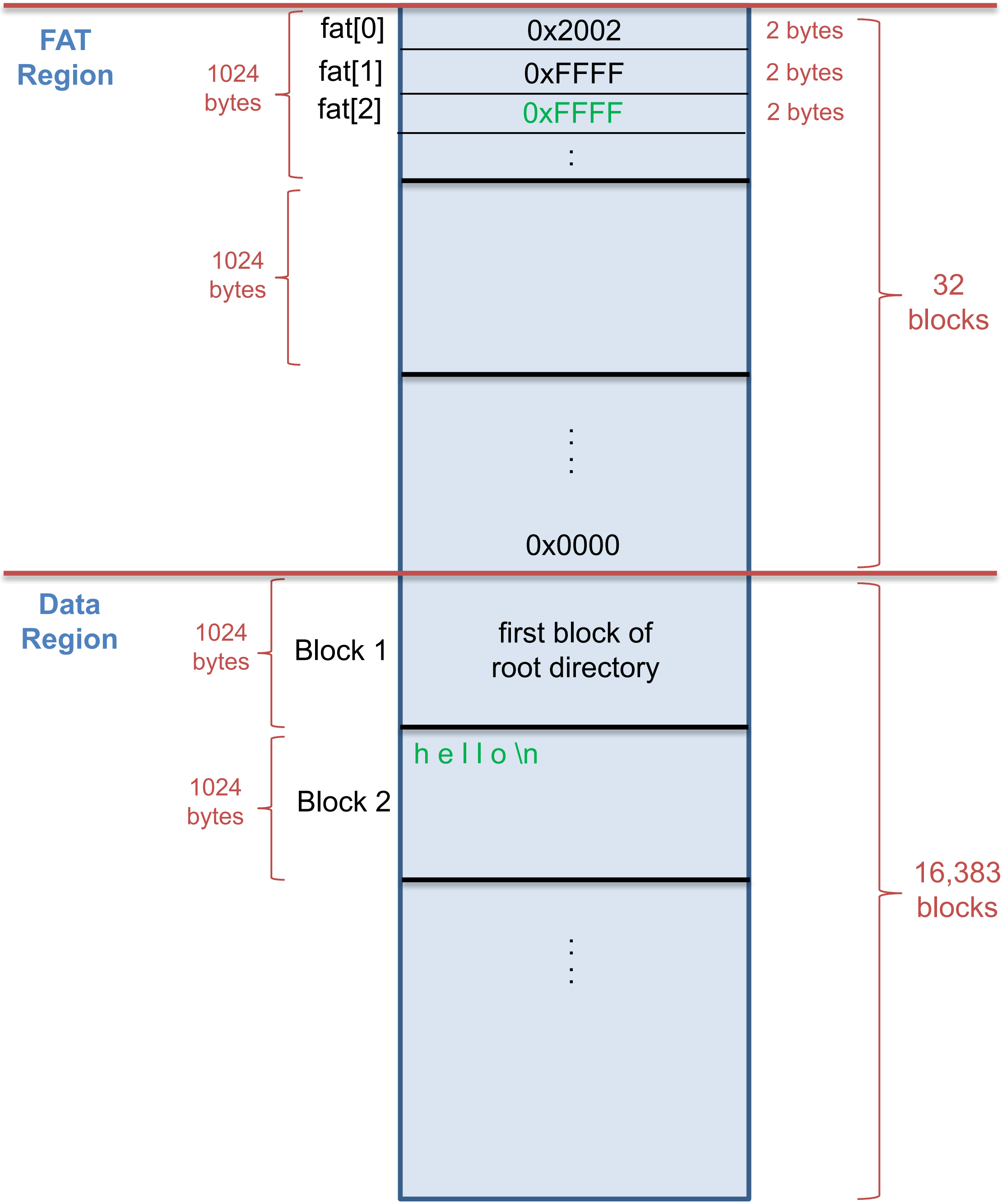
64 bytes

0

64 bytes

16 directory entries

PennFAT after writing to the file



FAT Region

1024 bytes

fat[0]
0x2002
2 bytes

fat[1]
0xFFFF
2 bytes

fat[2]
0xFFFF
2 bytes

:

1024 bytes

⋮

0x0000

32 blocks

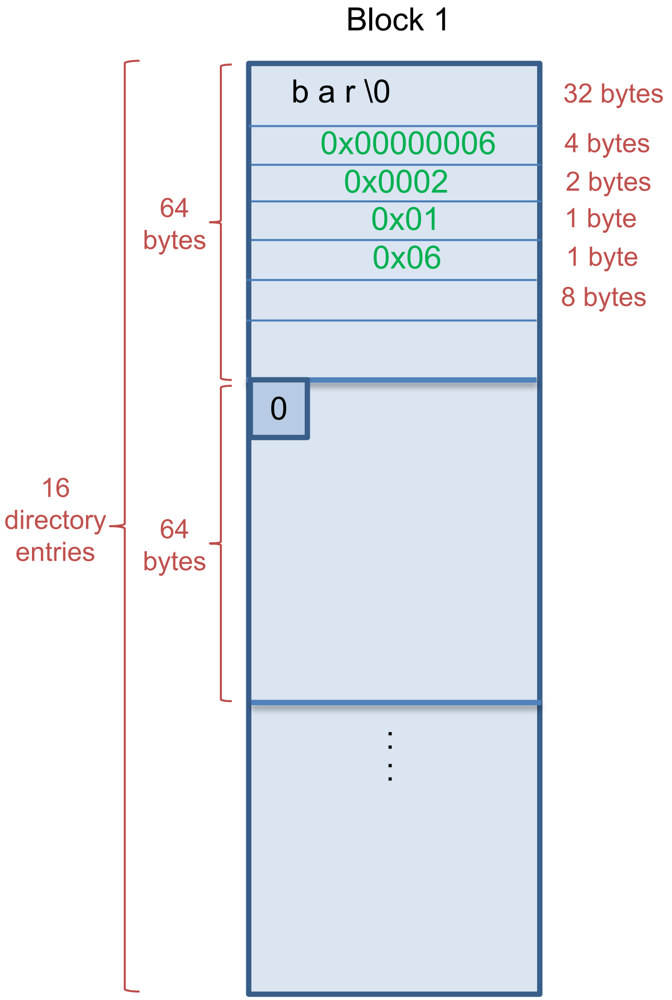
Data Region

1024 bytes
Block 1
first block of root directory

1024 bytes
Block 2
h e l l o \n

⋮

16,383 blocks



Block 1

b a r \0
32 bytes

0x00000006
4 bytes

0x0002
2 bytes

0x01
1 byte

0x06
1 byte

8 bytes

64 bytes

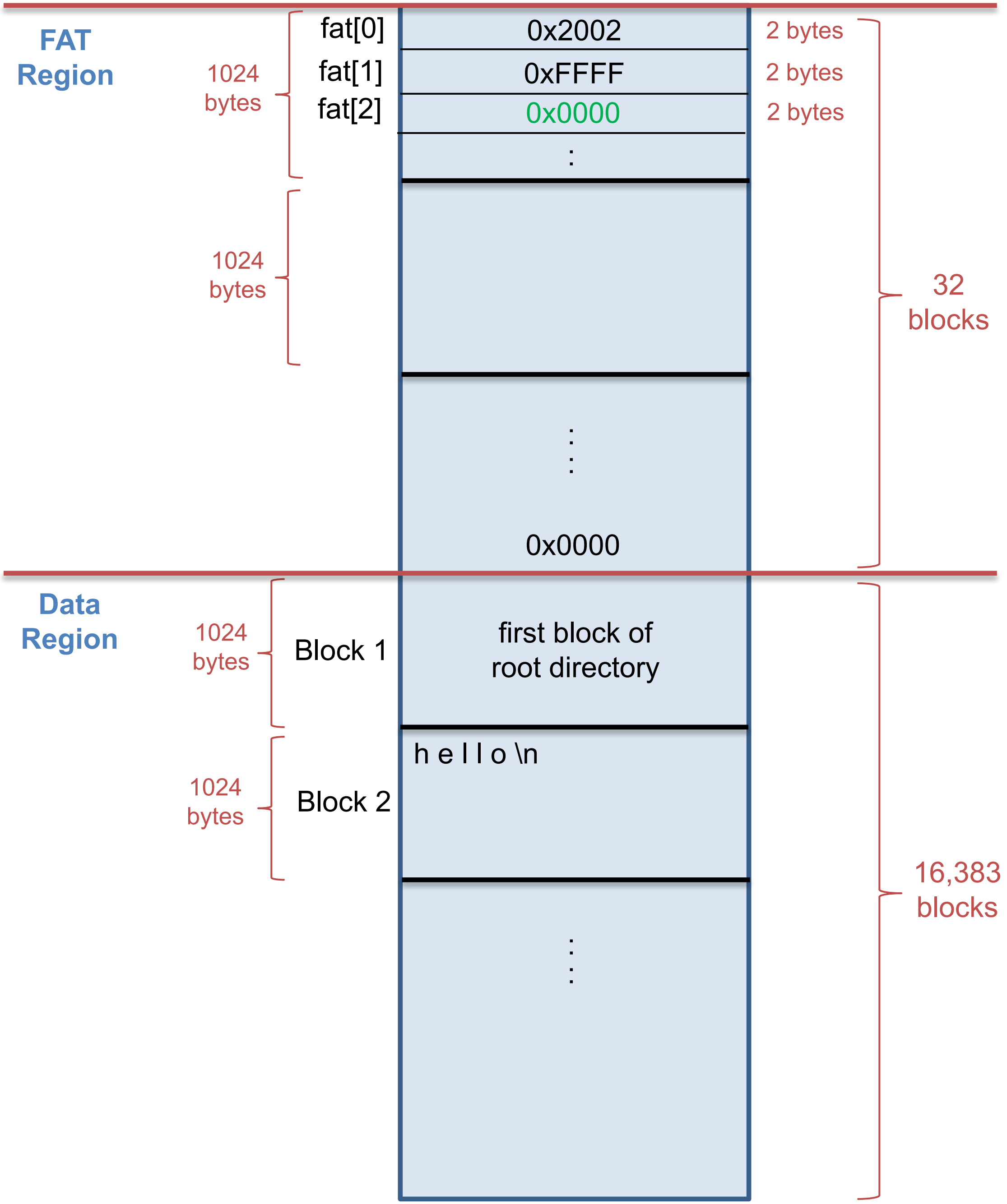
0

64 bytes

⋮

16 directory entries

PennFAT after removing the file



FAT Region

1024 bytes

fat[0] 0x2002 2 bytes

fat[1] 0xFFFF 2 bytes

fat[2] 0x0000 2 bytes

:

1024 bytes

:

0x0000

32 blocks

Data Region

1024 bytes

Block 1

first block of root directory

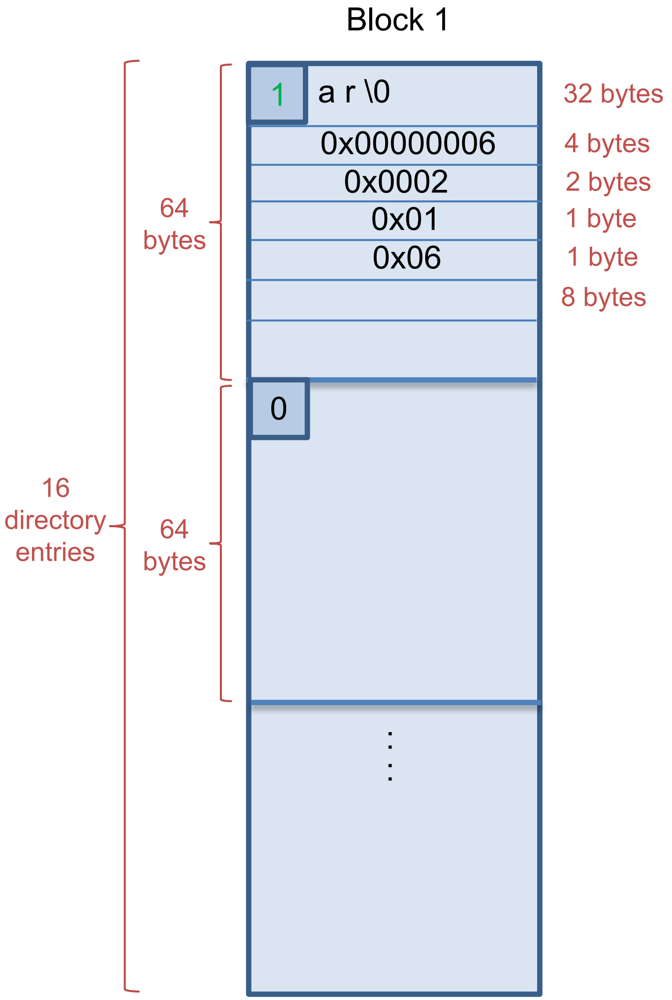
1024 bytes

Block 2

h e l l o \n

:

16,383 blocks



Block 1

1 a r \0 32 bytes

0x00000006 4 bytes

0x0002 2 bytes

0x01 1 byte

0x06 1 byte

8 bytes

64 bytes

0

64 bytes

16 directory entries

:

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
- These API should be **specific** to PennFAT
  - No other filesystem format can use this
- 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 fileystem

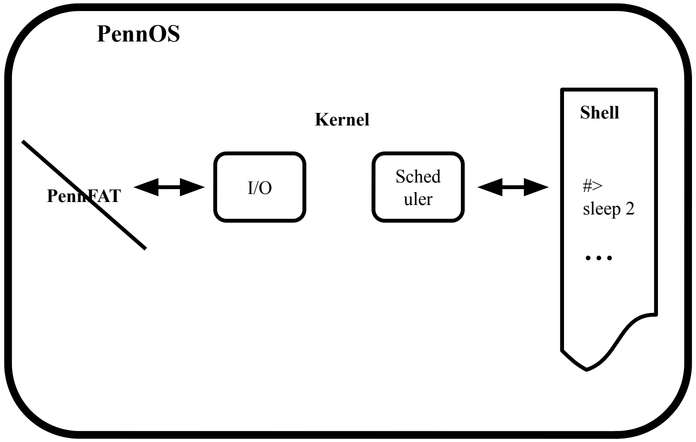
The following would be logical flow of cp

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

Kernel

Scheduling & Process Life Cycle

Scheduling in PennOS



PennOS

Kernel

~~PennFAT~~

I/O

Sched
uler

Shell

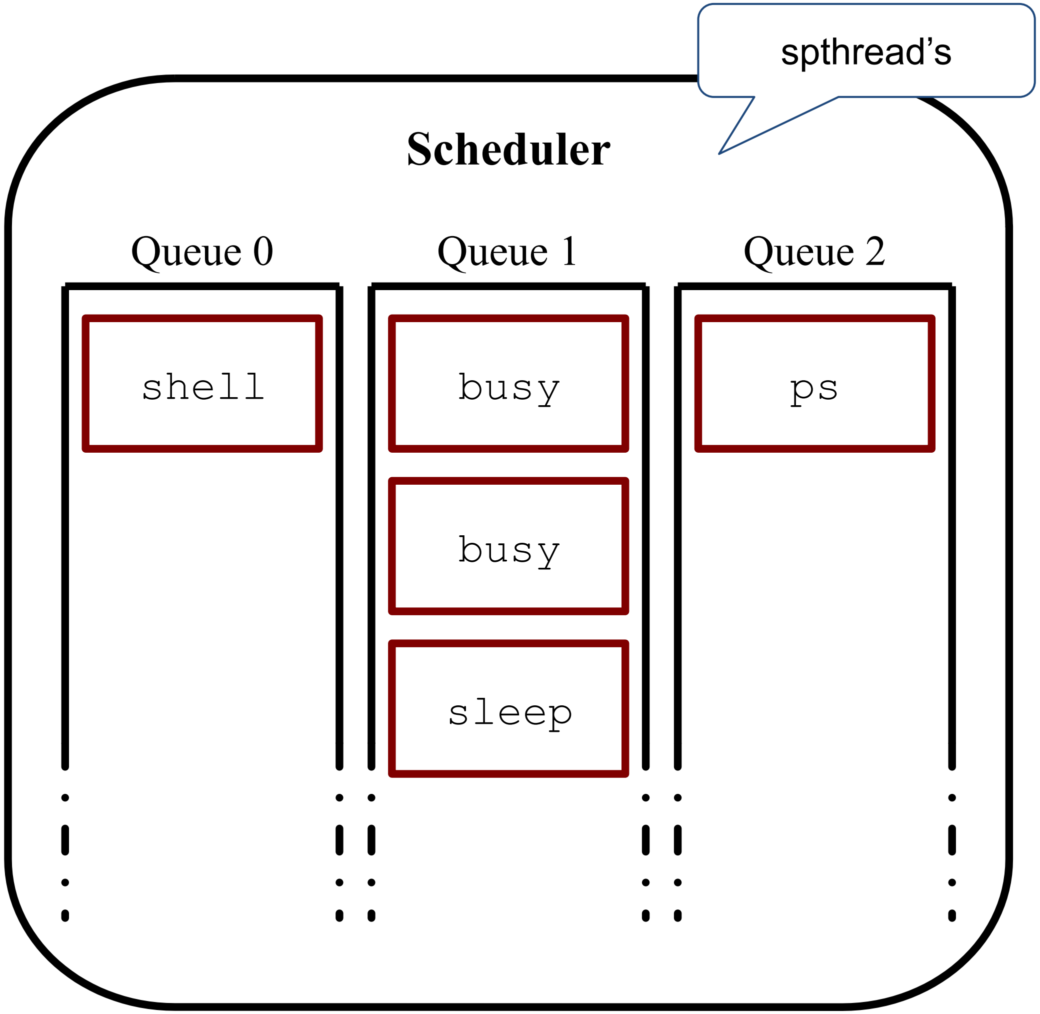
#>
sleep 2
…

Exponential Relationship

Queue **0** scheduled 1.5 times more frequently than Queue **1**

Queue **1** scheduled 1.5 times more frequent than Queue **2**

Round Robin within Queue



spthread's

Scheduler

Queue 0

`shell`

Queue 1

`busy`

`busy`

`sleep`

Queue 2

`ps`

Process Statuses

Running

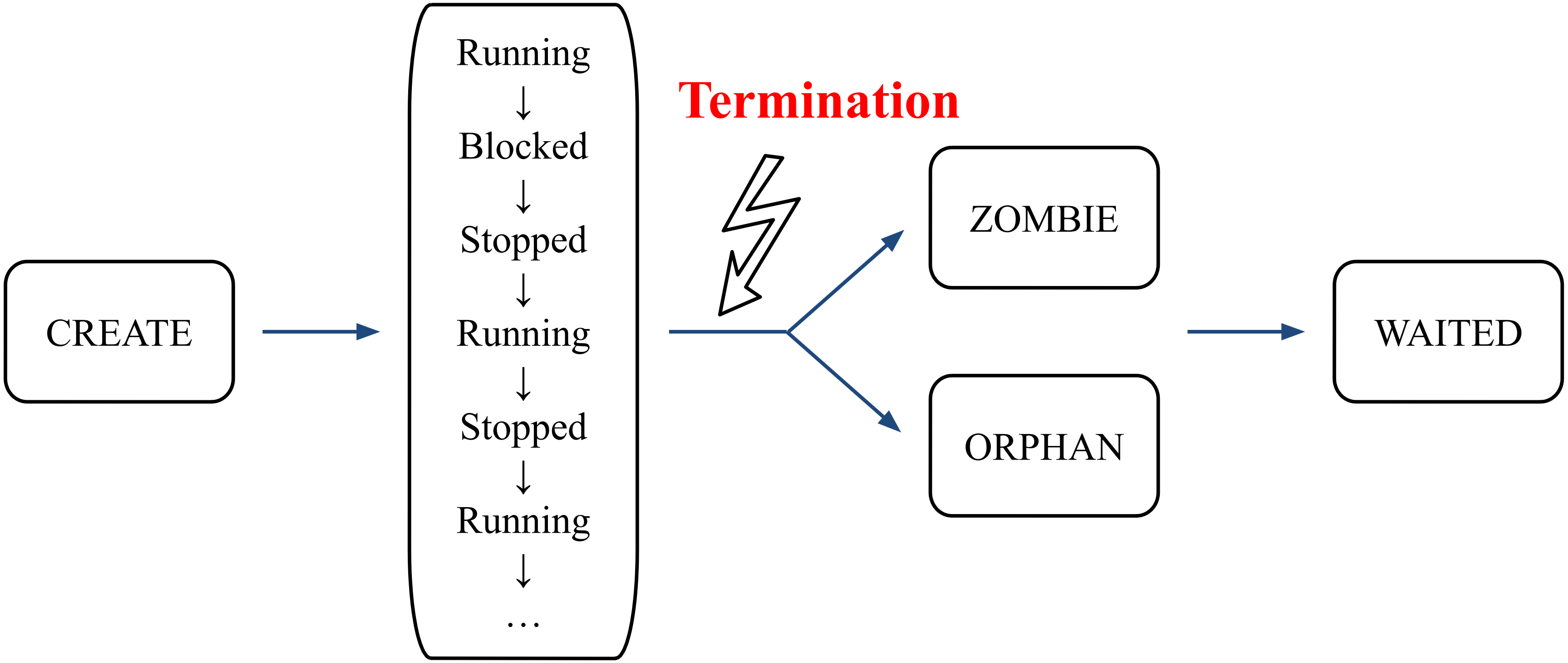
Blocked

Stopped

Zombied

Orphaned

Process Life Cycle



CREATE

Running
↓
Blocked
↓
Stopped
↓
Running
↓
Stopped
↓
Running
↓
…

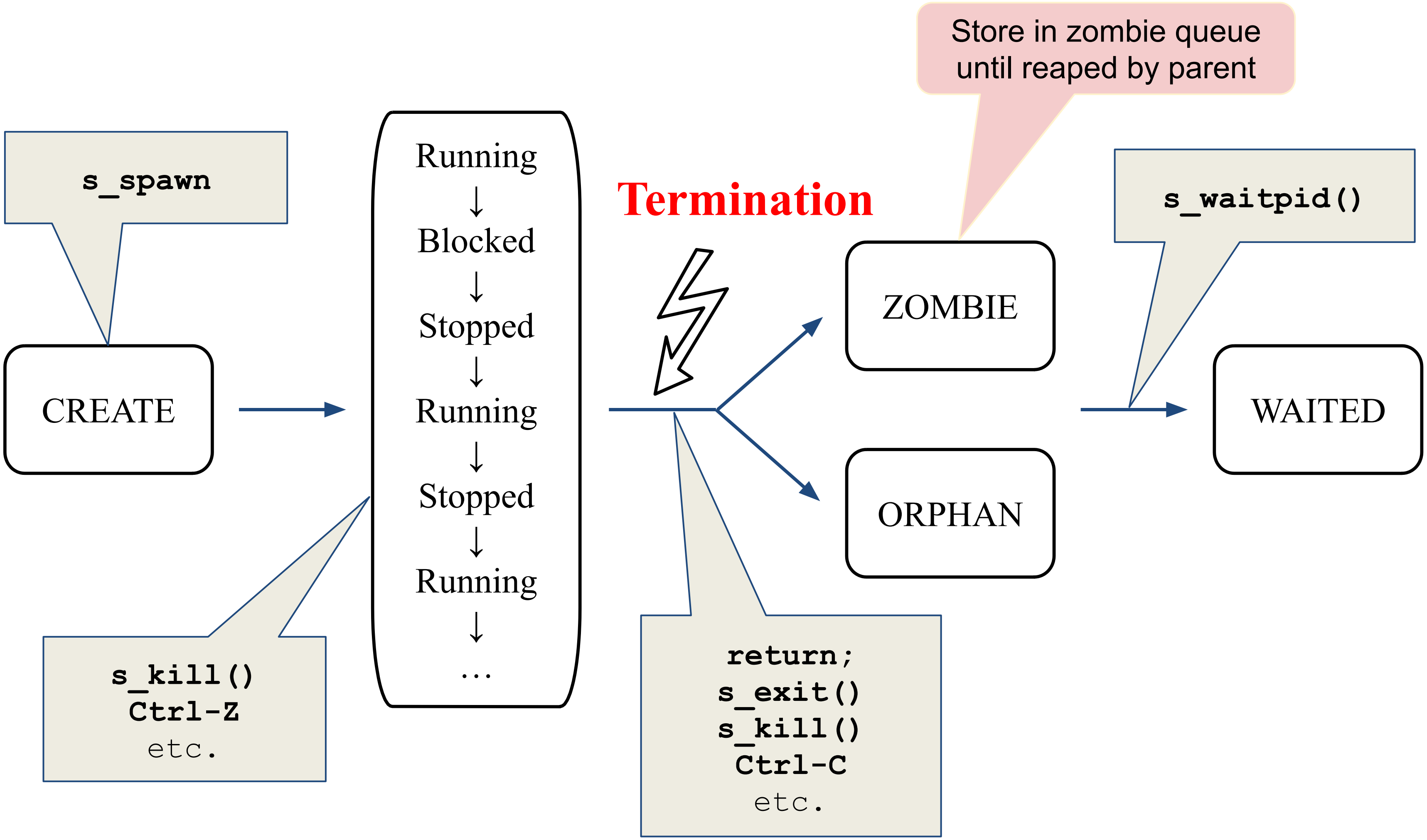
Termination

ZOMBIE

ORPHAN

WAITED

PennOS Process Life Cycle



`s_spawn`

CREATE

Running
↓
Blocked
↓
Stopped
↓
Running
↓
Stopped
↓
Running
↓
…

`s_kill()`
`Ctrl-Z`
`etc.`

Termination

`return;`
`s_exit()`
`s_kill()`
`Ctrl-C`
`etc.`

Store in zombie queue
until reaped by parent

ZOMBIE

ORPHAN

`s_waitpid()`

WAITED

Process Control Block (PCB)

```
typedef struct pcb {
    pid_t pid;

    int foo;
    char *bar;

} pcb_t;
```

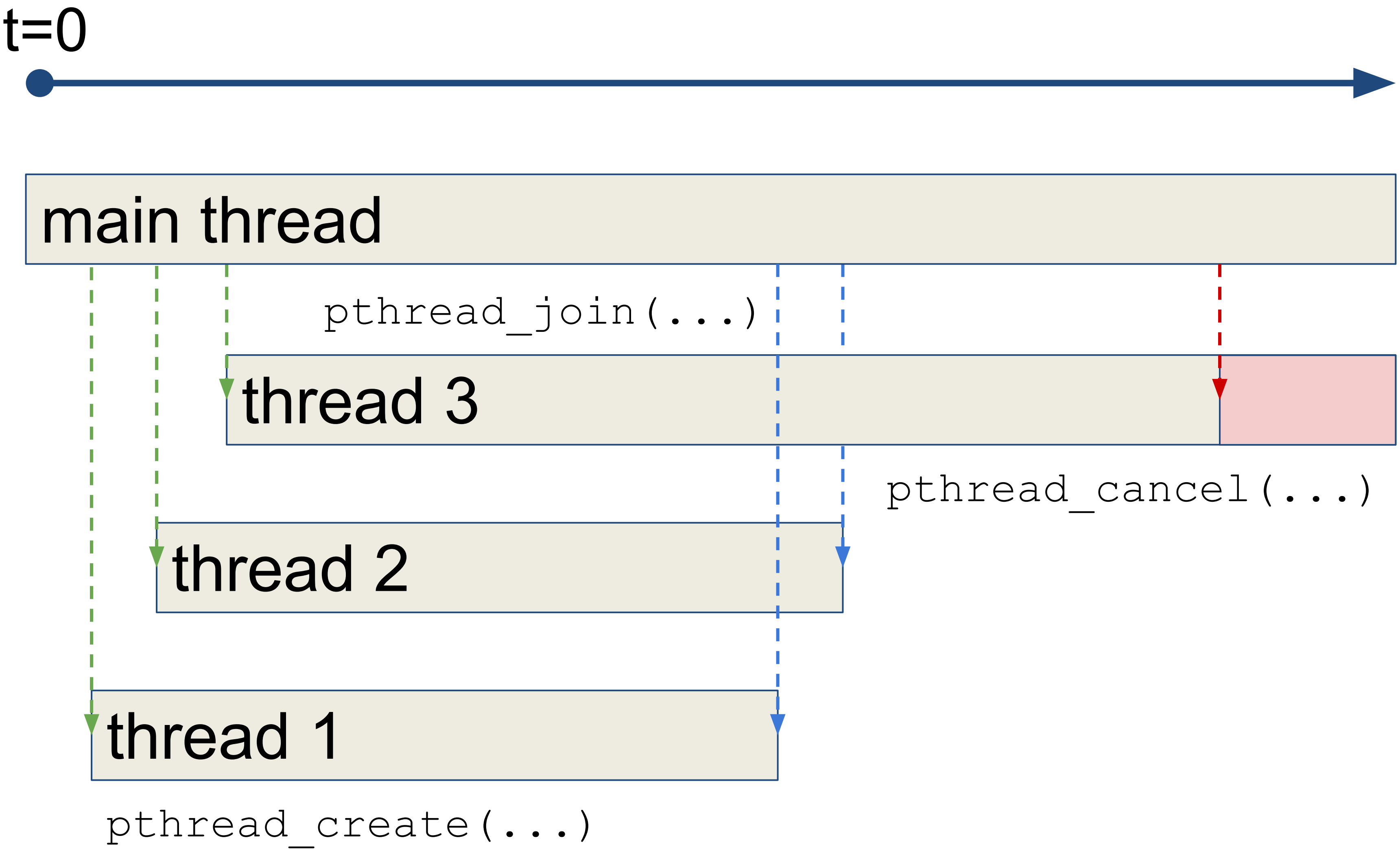
Programming with `spthread`

First, what is `pthread`?

- User-level thread management API
- Isolate code execution within distinct threads
  - Run **`funcA`** in *threadA*, **`funcB`** in *threadB*, etc.
- Resource sharing (within same process space)
- Concurrent execution

Pros: efficient, lightweight, simple

How does `pthread` work?



What is `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);
```

“Hello-World”: a brief tour

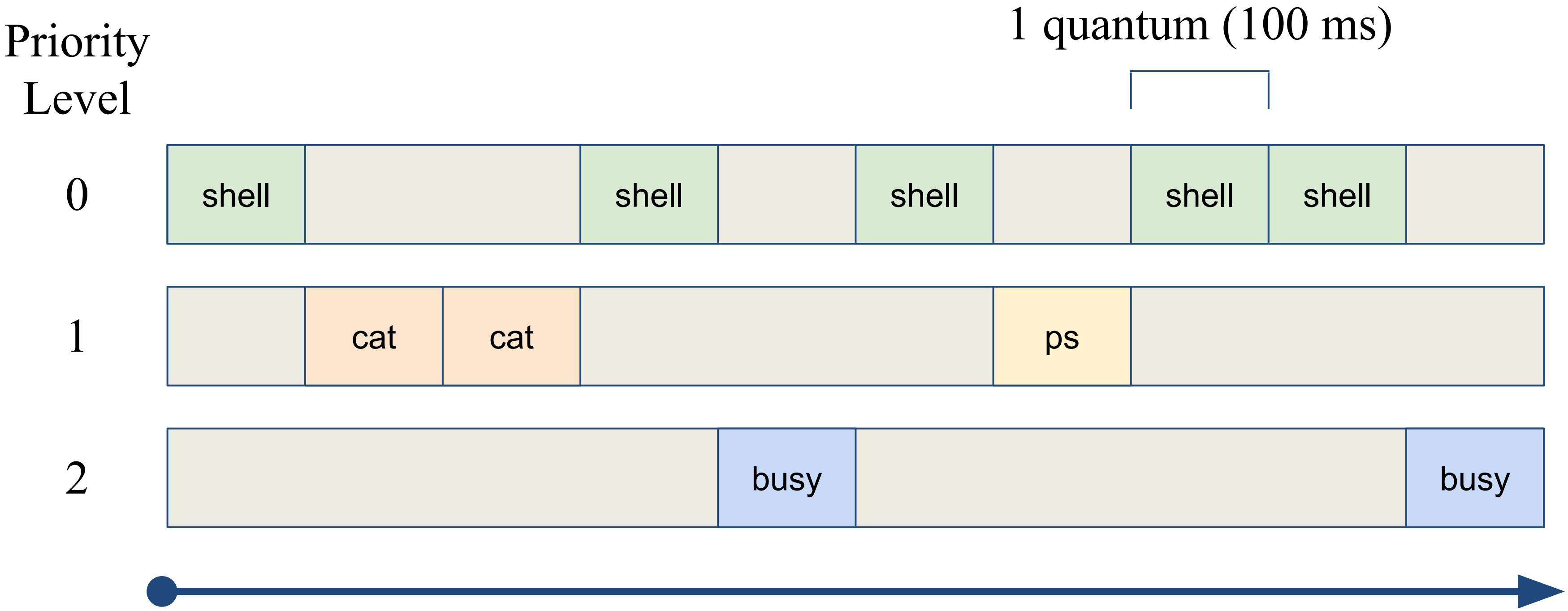
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include "./spthread.h"

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

int main(void) {
   spthread_t hello_thread;

   spthread_create(&hello_thread, NULL, hello_world, NULL);
   spthread_continue(hello_thread);
   spthread_join(hello_thread, NULL);
}
```

“Sleep”: a brief tour

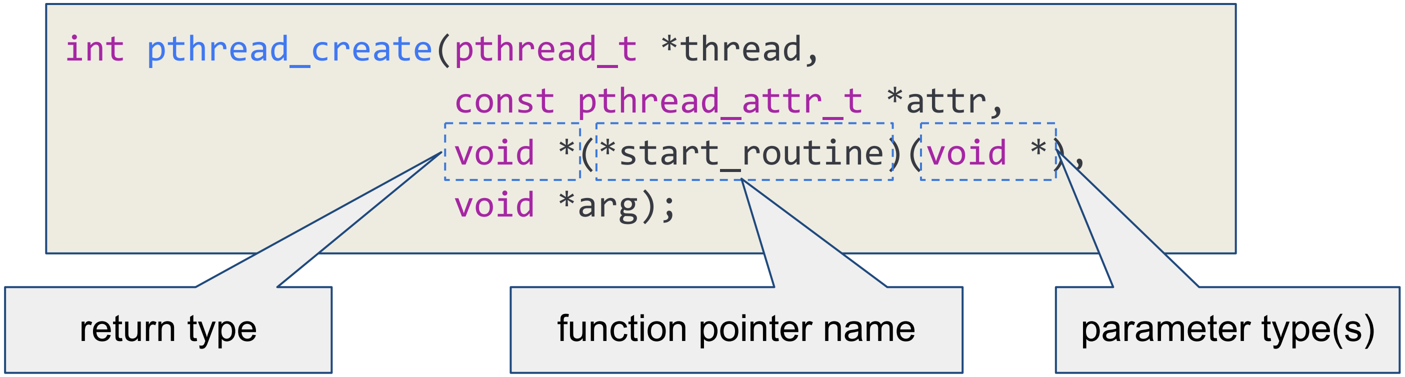
```c
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "./spthread.h"

void sleep_f(int t) {
   while(1) {
     printf("zzzzzzzz\n");
     sleep(t);
   }
}

int main(void) {
  spthread_t sleep_thread;
  int i = 1;

  spthread_create(&sleep_thread, NULL, sleep_f, i);
  spthread_continue(sleep_thread);
  sleep(4);
  spthread_suspend(sleep_thread);
  printf("right here\n");
  sleep(2);
  spthread_continue(sleep_thread);
  spthread_continue(sleep_thread);
  sleep(2);
  spthread_cancel(sleep_thread);
  printf("after sleep\n");
  spthread_join(sleep_thread, NULL);
}
```

Using `spthread` for scheduling

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



Misc: Function Pointers



```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

return type

function pointer name

parameter type(s)

```
void* fun_function(void* args) {
    char** actual_args = (char**)args;
    // ...
    return NULL;
}

void some_helper(void *(*func_ptr)(void*)) {
    char* msgs[] = {"turtle", "turtle", NULL};
    spthread_t thread;
    spthread_create(&thread, NULL, func_ptr, (void*) &msgs);
}

some_helper(fun_function);
```

PennOS Shell

Shell Requirements

Synchronous Child Waiting

Redirection (no pipelines)

Parsing

Terminal Signaling

Terminal Control

Shell Functions

Basic interaction with PennOS

Two types:

- Functions that run as separate process
- Functions that run as shell sub-routines

Examples of Built-ins that Run as a Process

`cat`

`sleep`

`busy`

`ls`

`touch`

`mv`

`cp`

`rm`

`ps`

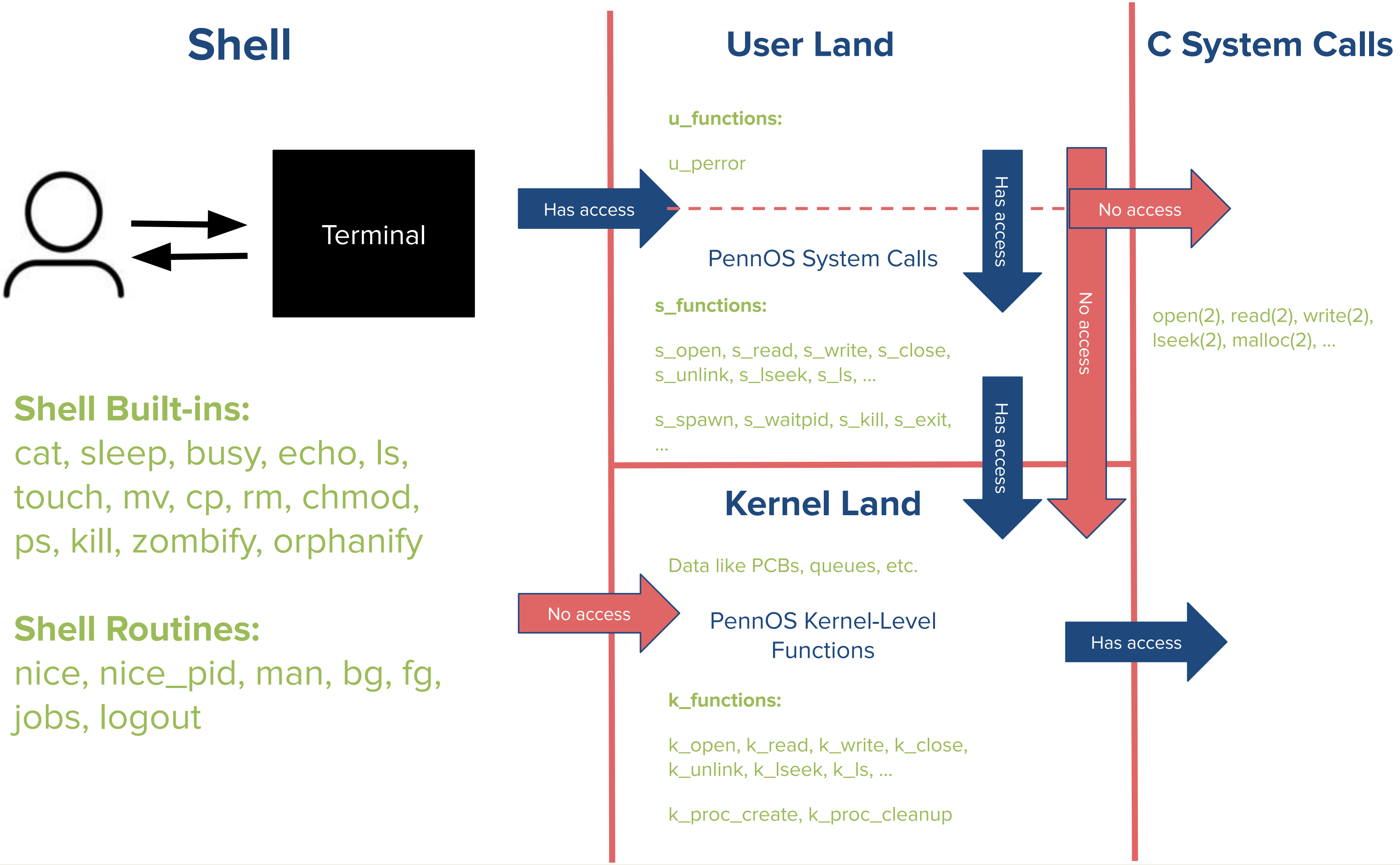
Examples of Built-ins that Run as a Subroutine

```
nice
nice_pid
man
bg
fg
jobs
logout
```

Final Touches: 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)` or `open(2)`

Keeping the Abstraction!



Shell

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

User Land

u_functions:

u_perror

PennOS System Calls

s_functions:

s_open, s_read, s_write, s_close, s_unlink, s_lseek, s_ls, ...

s_spawn, s_waitpid, s_kill, s_exit, ...

Kernel Land

Data like PCBs, queues, etc.

PennOS Kernel-Level Functions

k_functions:

k_open, k_read, k_write, k_close, k_unlink, k_lseek, k_ls, ...

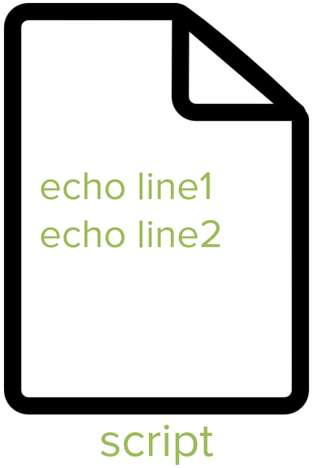
k_proc_create, k_proc_cleanup

C System Calls

open(2), read(2), write(2), lseek(2), malloc(2), ...

Final Touches: 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

Questions?