CIS 3800
Penn-OS Lecture

Fall 2023

Milestone and Demo

Milestone 1: **Due by Nov. 3rd (TA Meeting by 3rd)**

- Meeting with group and TA
- General discussion regarding the design of your project
- Pass/Fail grade

Milestone 2: **Nov. 10th (TA Meeting 10th-14th)**

- Meeting with group and TA
- "Significant Progress" expected (~60% complete)
- Pass/Fail grade

Due: **Submission Nov. 27th / Demos Latest Dec. 6th**

- 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
- ucontext
- PennOS Shell
- Demo

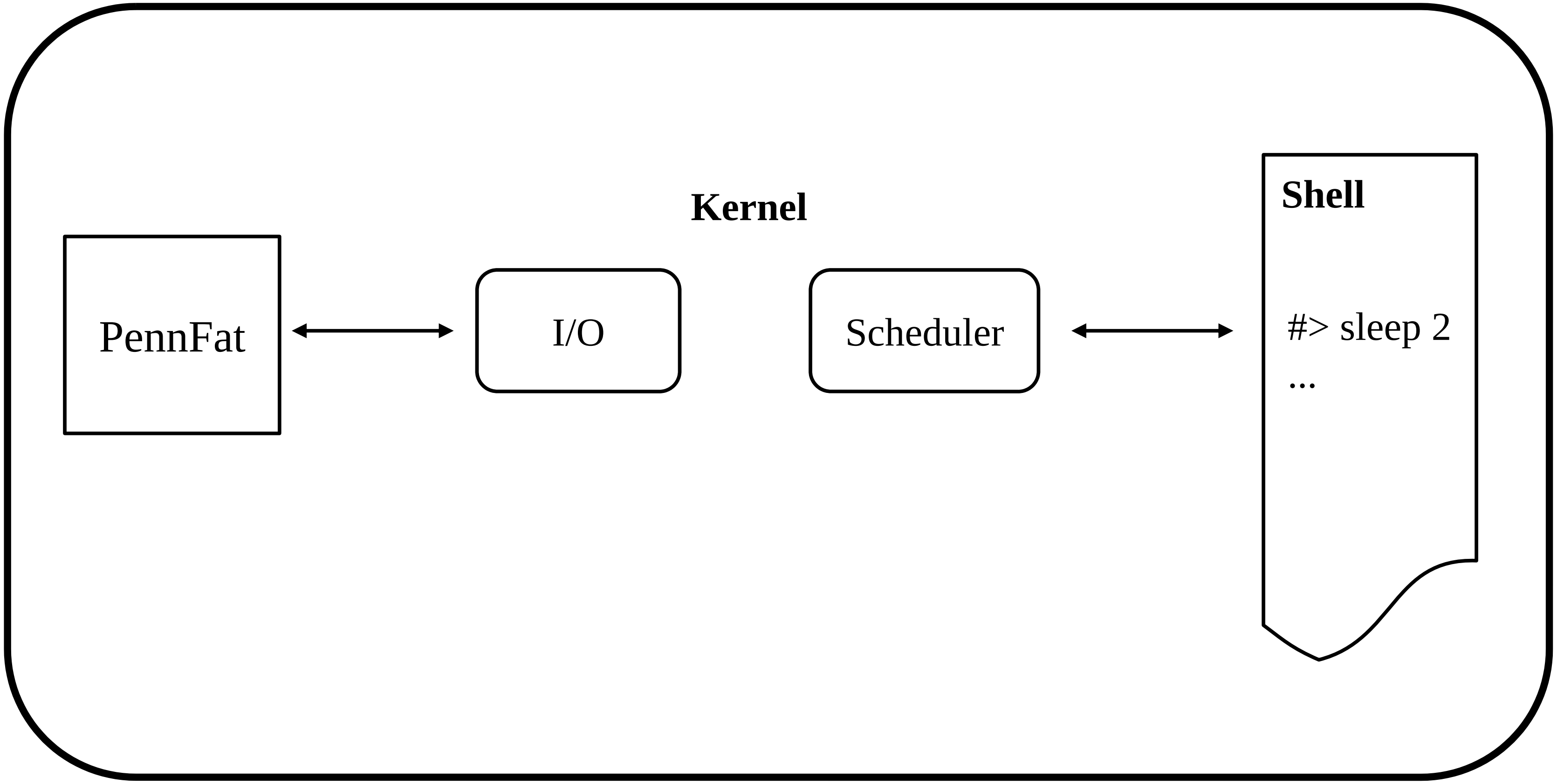
PennOS Overview

Projects So Far

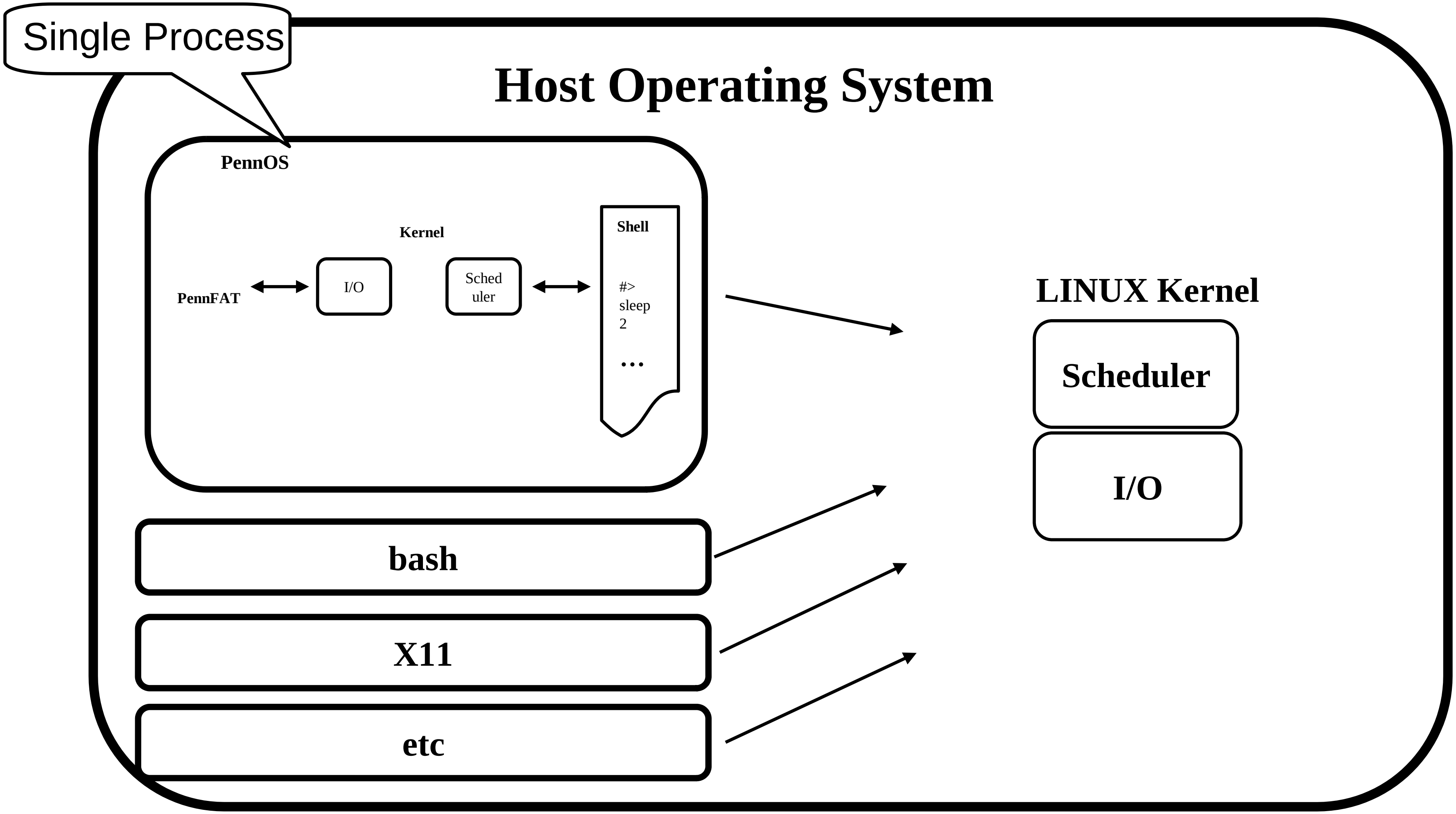
- Penn Shredder
  - Mini Shell with Signal Handling
- Penn Shell
  - Redirections and Pipelines
  - Process Groups and Terminal Control
  - Job Control

You will be implementing major user-level calls in Penn OS

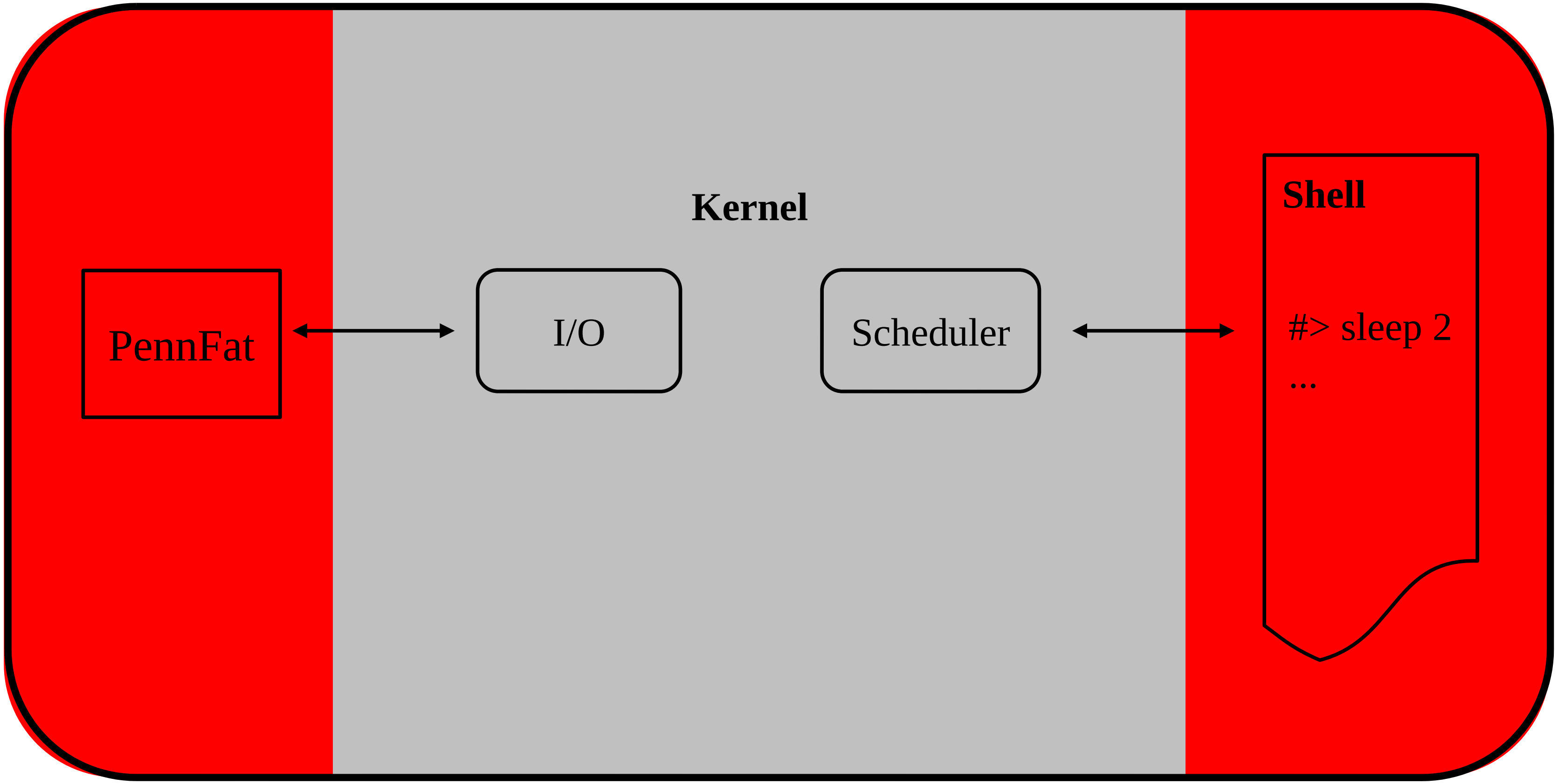
PennOS



PennOS as a GuestOS



User Land Shell Interaction



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)

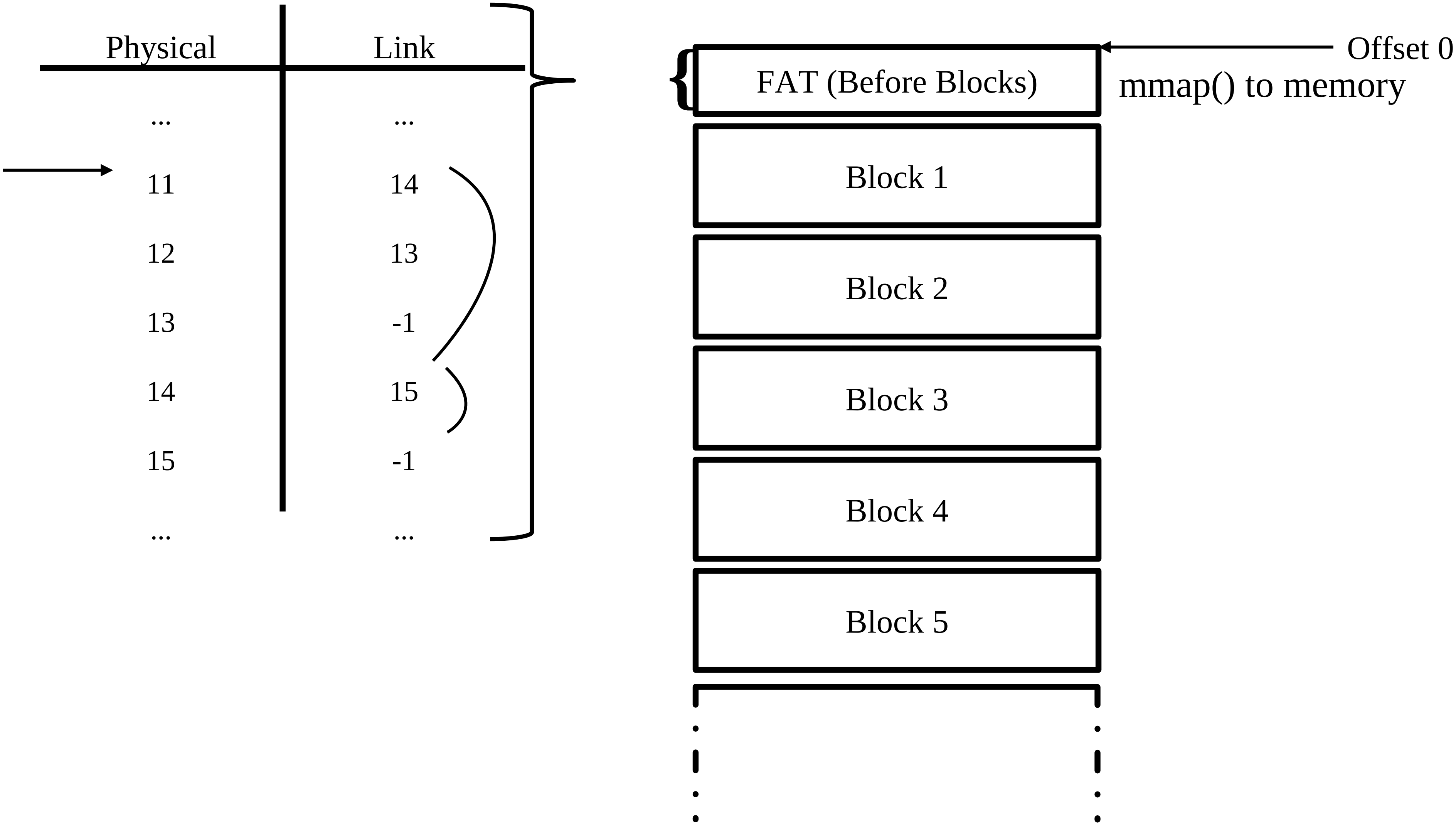
FAT

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

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

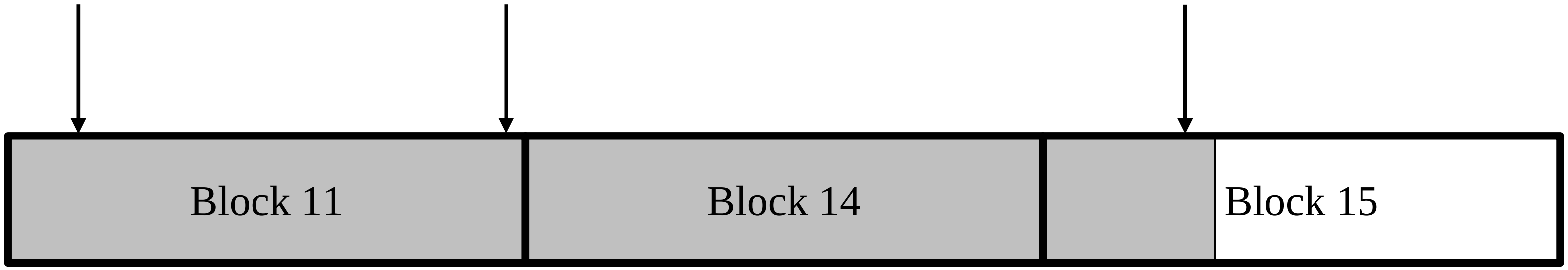
How can we read file 11?
Find Block 11, 14, and 15?

File System Layout



File Alignment

Files are distributed across **blocks**

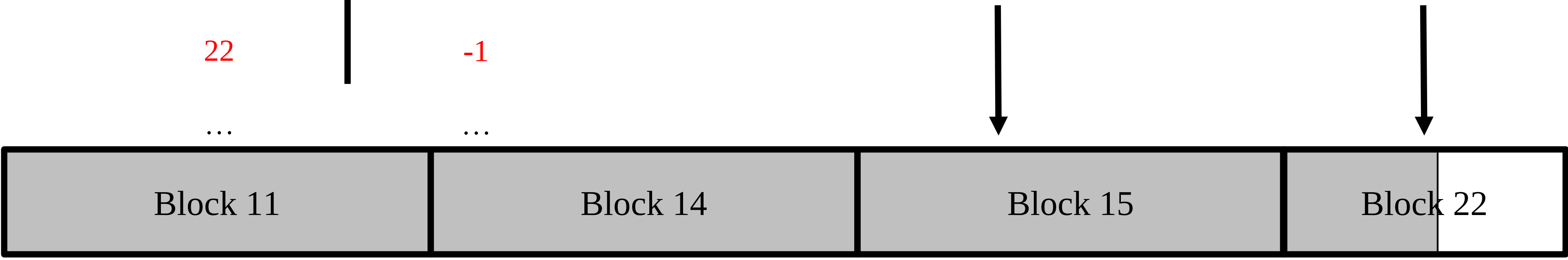



```
f_lseek(n, F_SEEK_SET, 60)
f_lseek(n, F_SEEK_SET, block_size - 1)
f_lseek(n, F_SEEK_SET, block_size * 2 + 100)
```

Adjusting File Size

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

`f_write(n, buffer, block_size)`



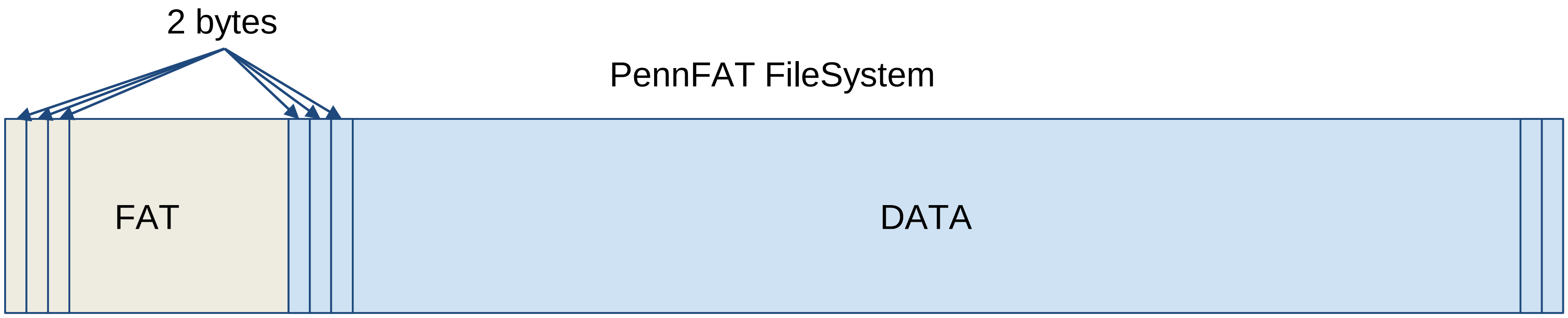
PennFAT Specification

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)

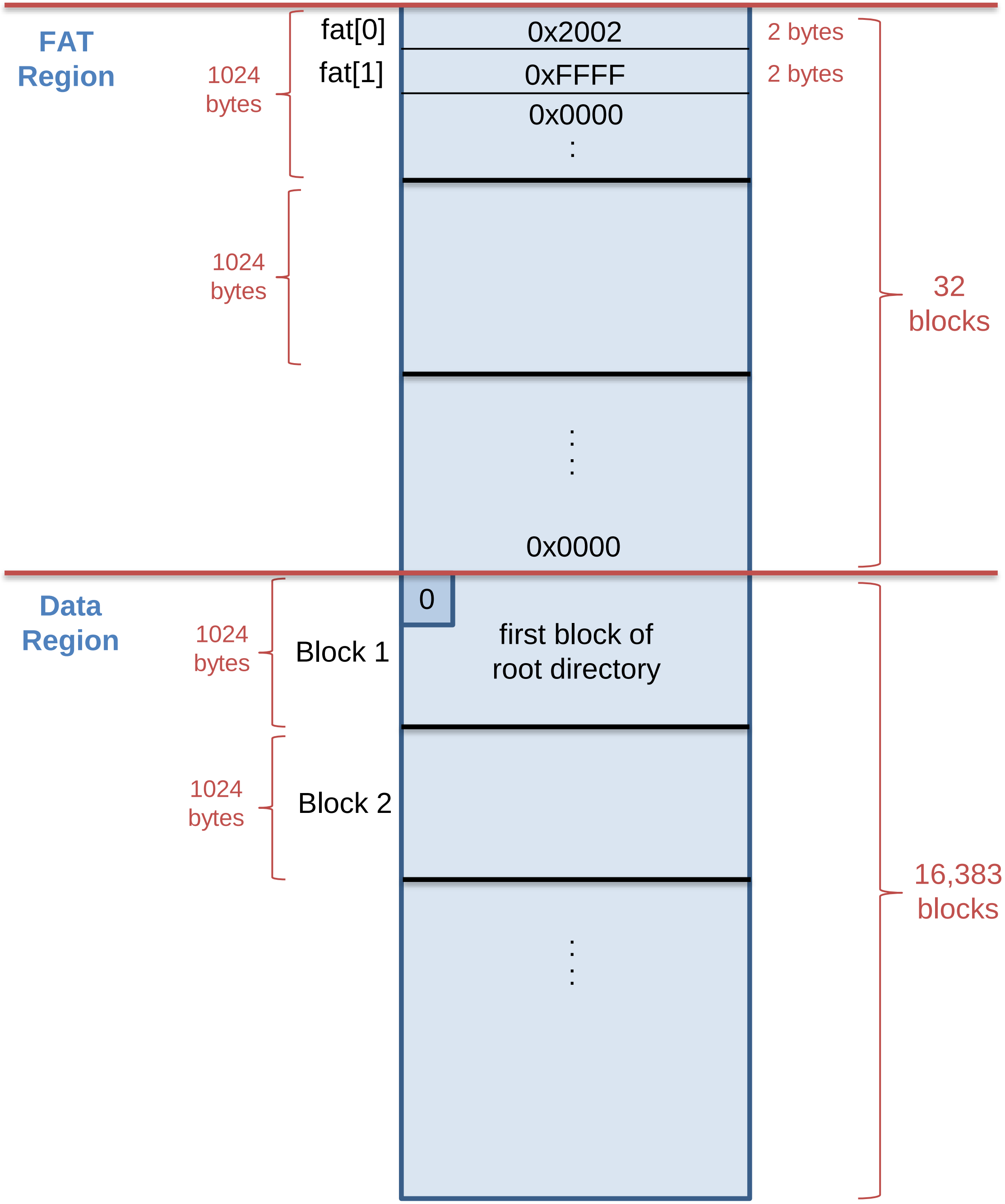
Directory entry (cont.)

- 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



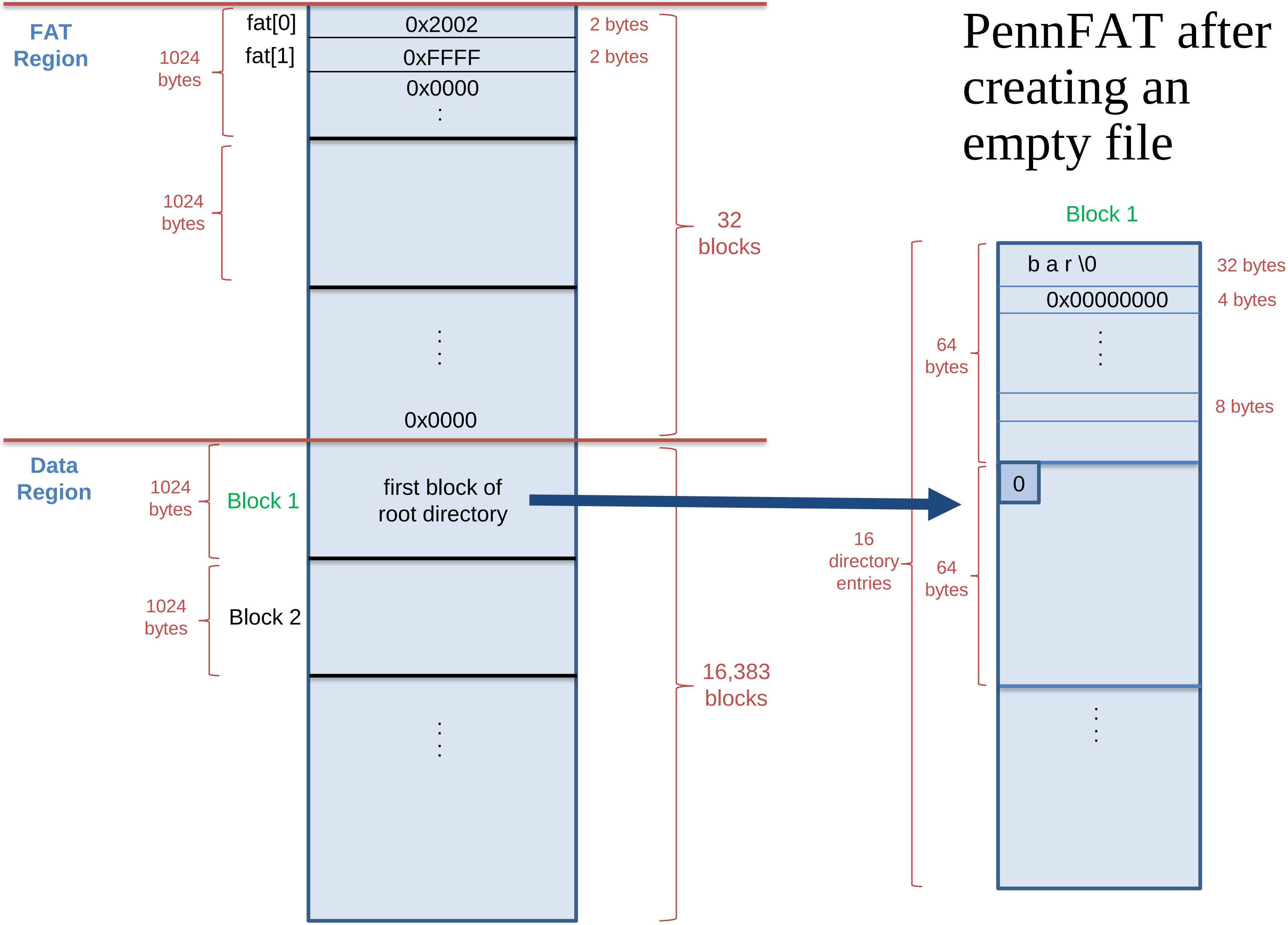
fat[0] = 0x2002

- 32 blocks of 1024 bytes in FAT

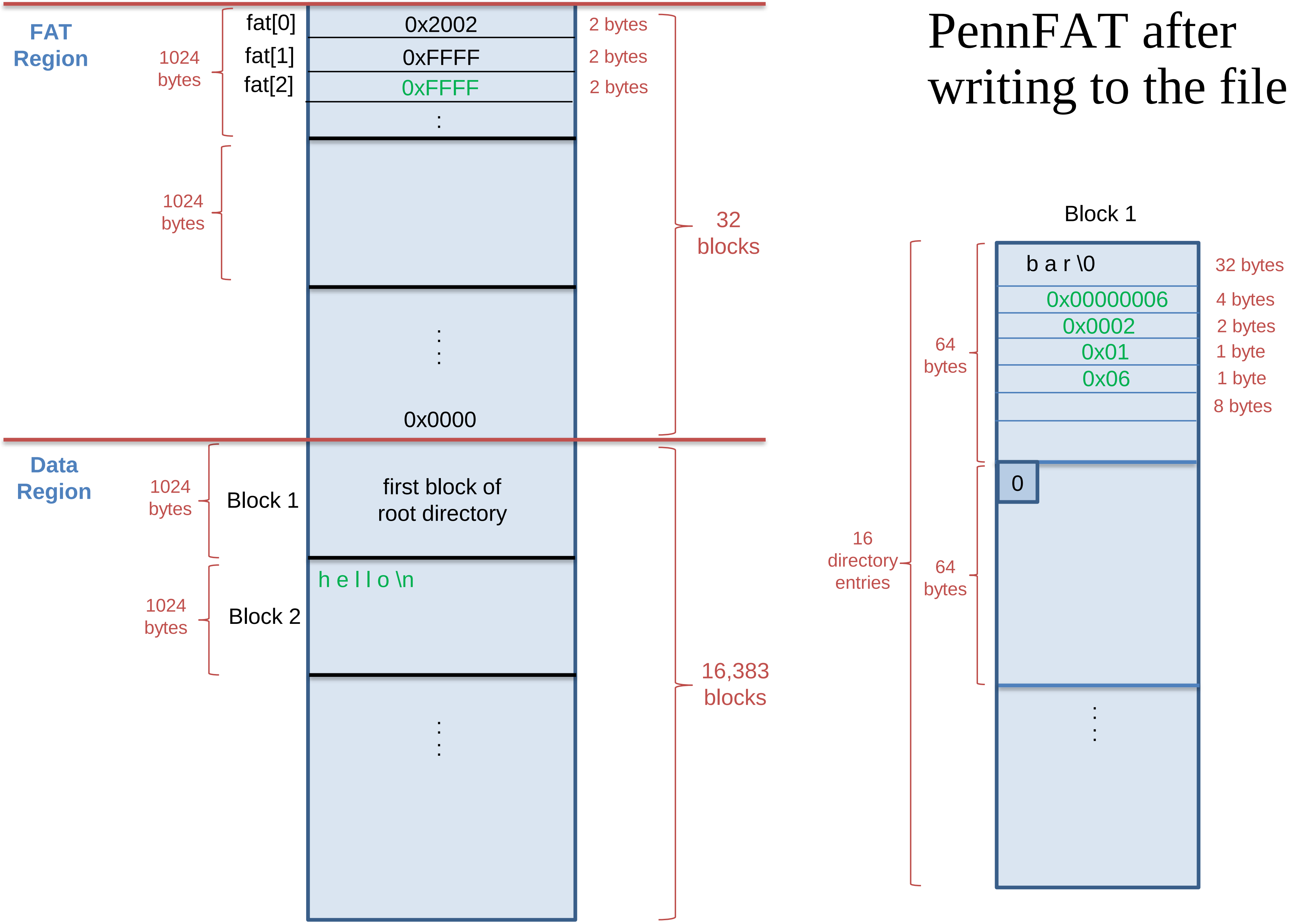
First block of Data Region is first block of root directory

Correspondingly, fat[1] refers to that Block 1, which ends there. So it has value of 0xFFFF

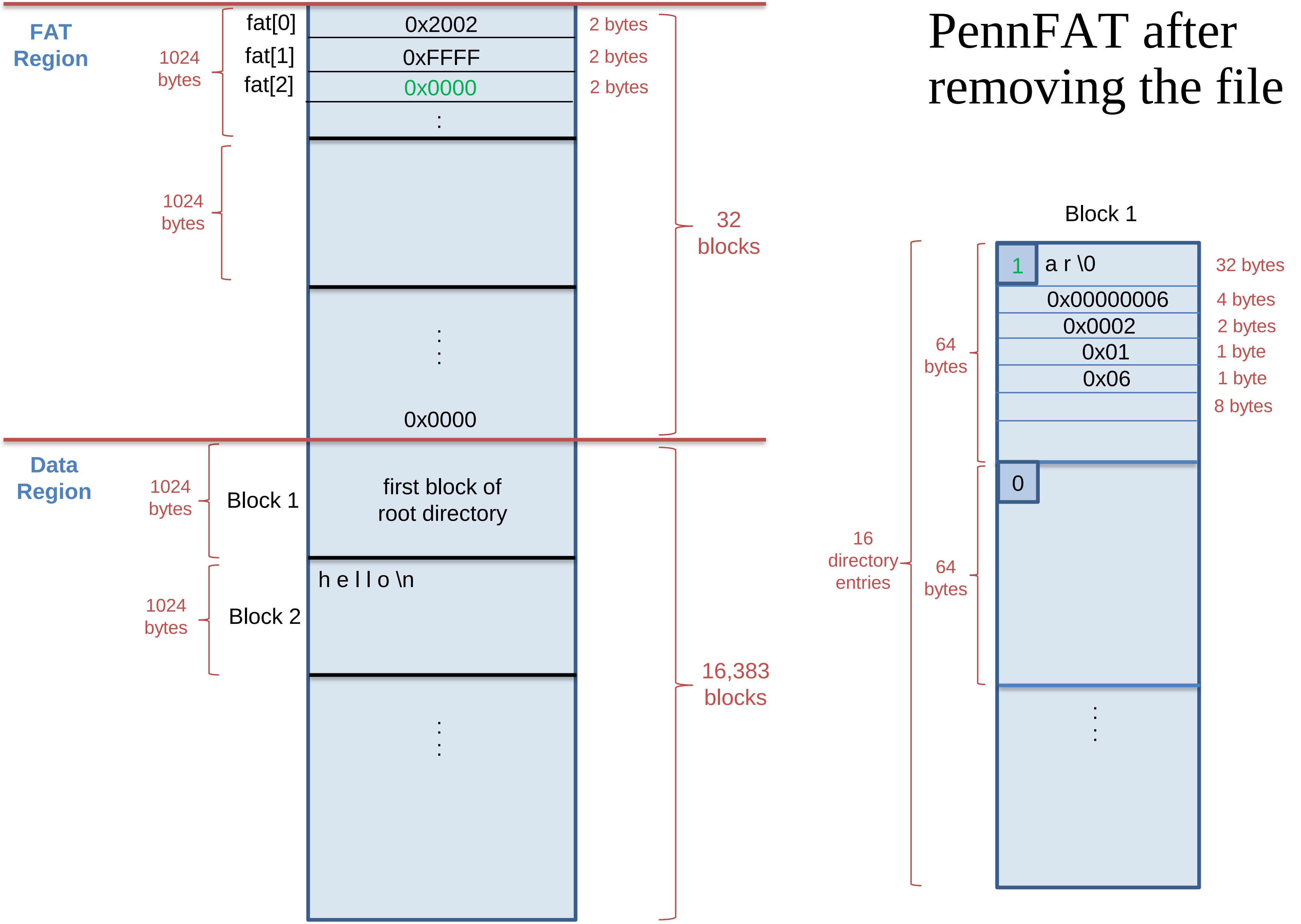
PennFAT after creating an empty file



PennFAT after writing to the file

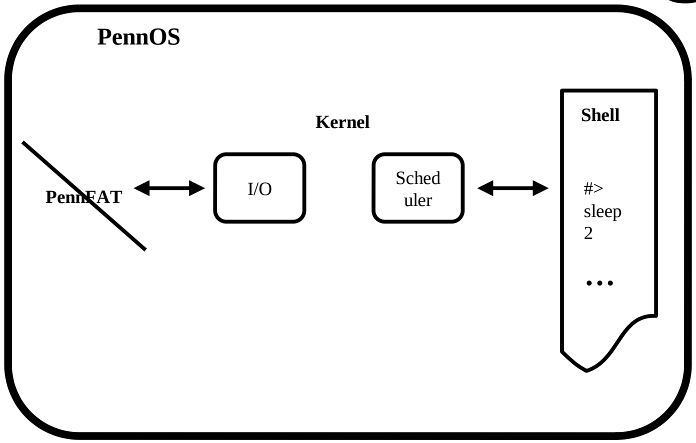


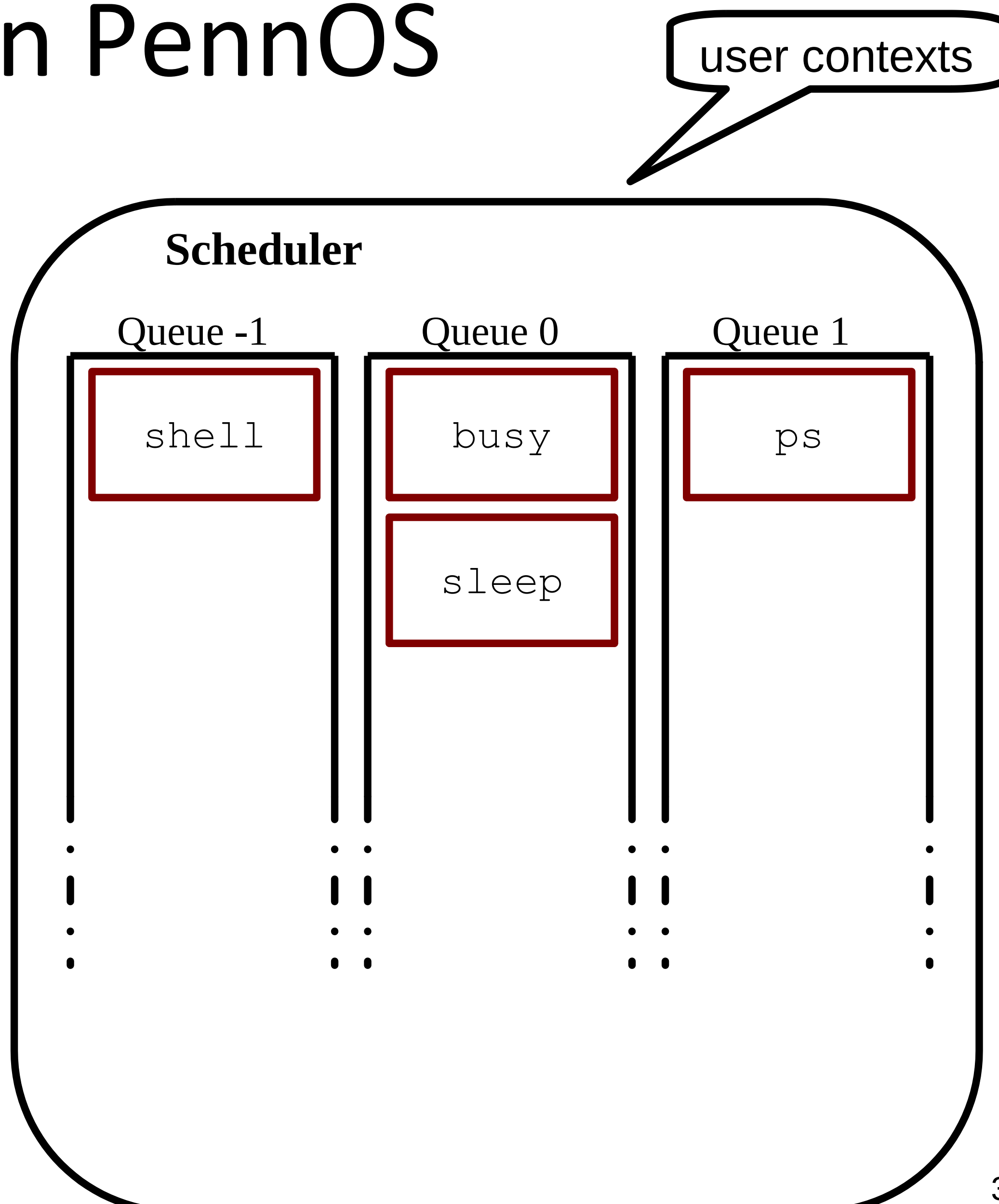
PennFAT after removing the file



Scheduling & Process Life Cycle

Scheduling in PennOS





<u>Exponential Relationship</u>

Queue -1 scheduled 1.5 times more frequently than Queue 0

Queue 0 scheduled 1.5 times more frequent than Queue 1

<u>Round Robin within Queue</u>

Process Statuses

Running

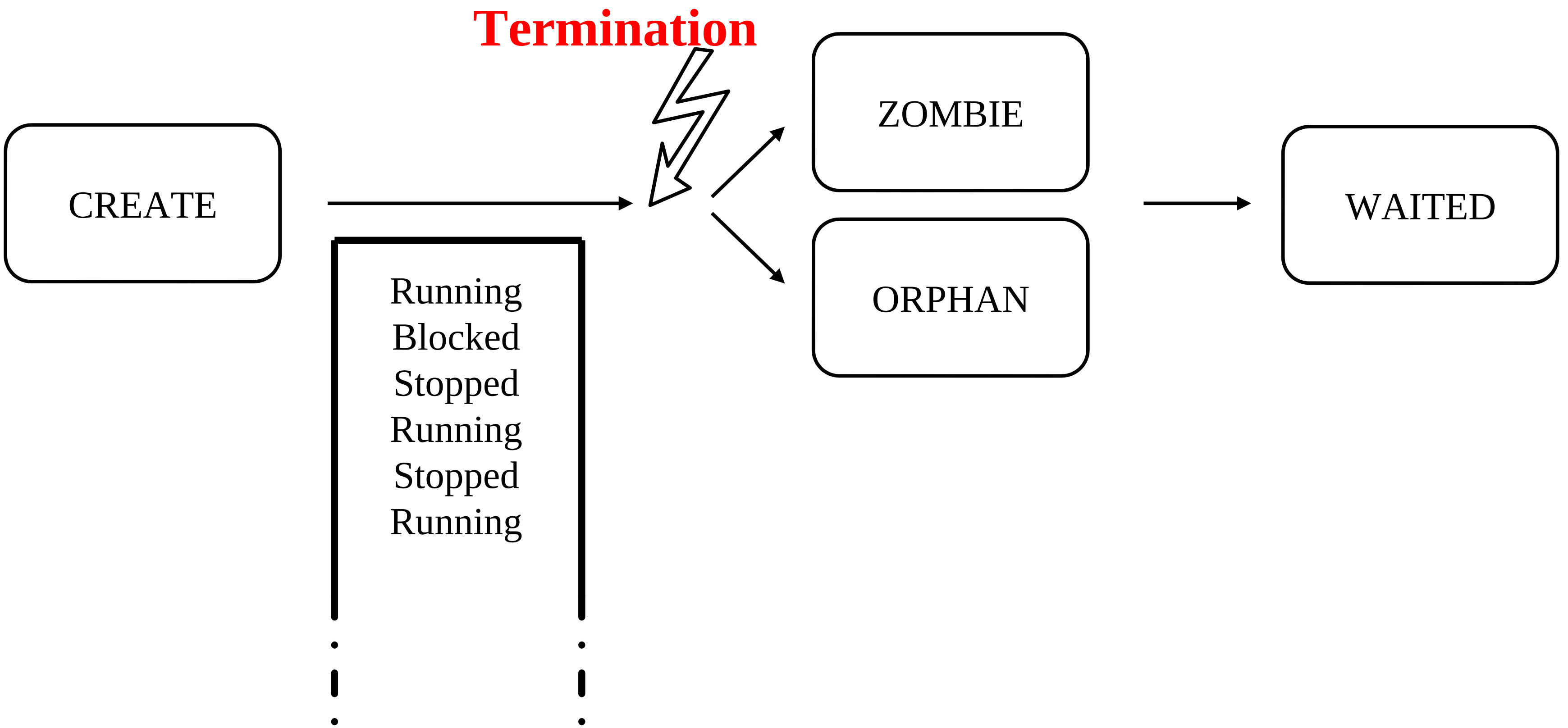
Blocked

Stopped

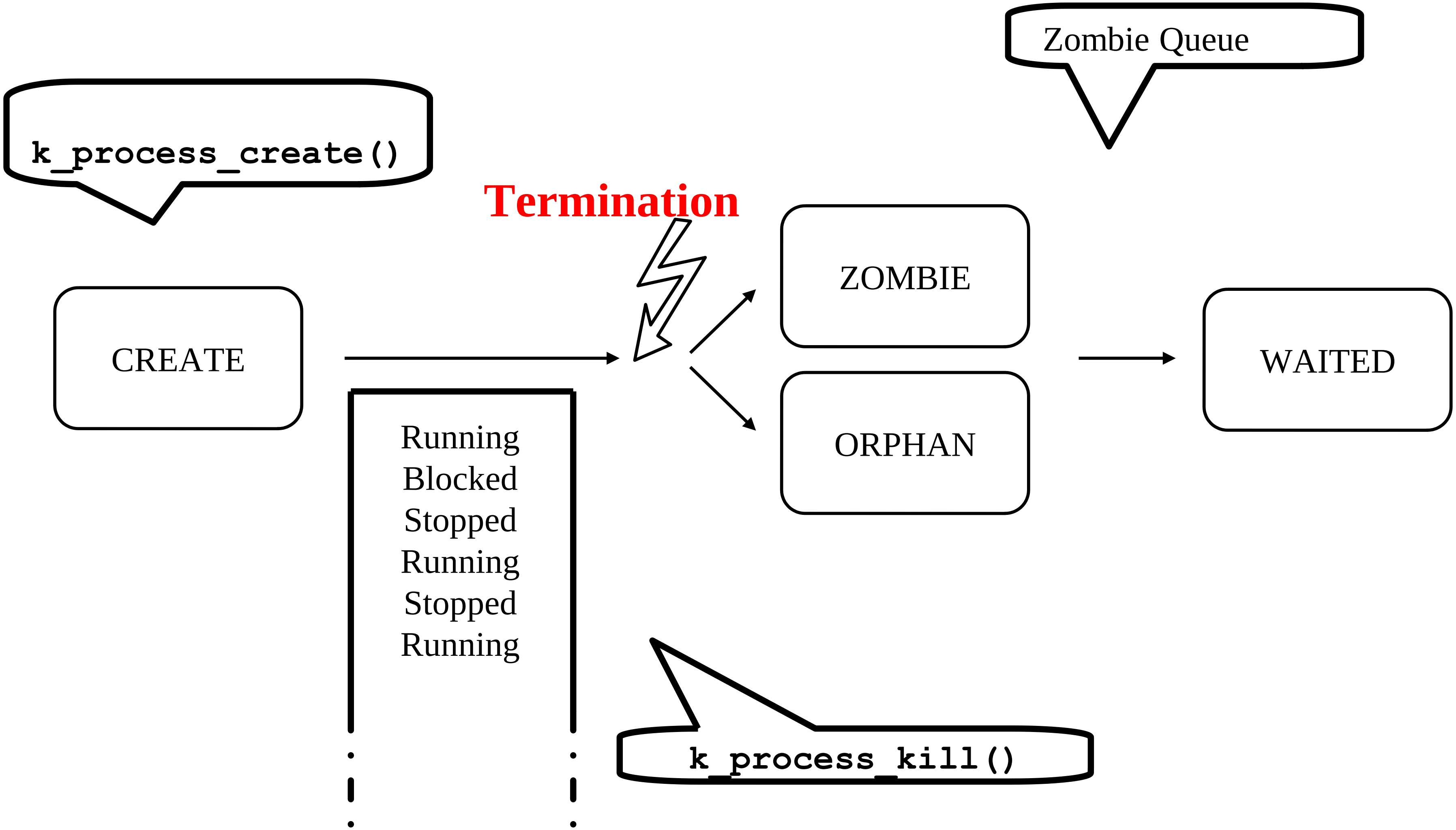
Zombied

Orphaned

Process Life Cycle



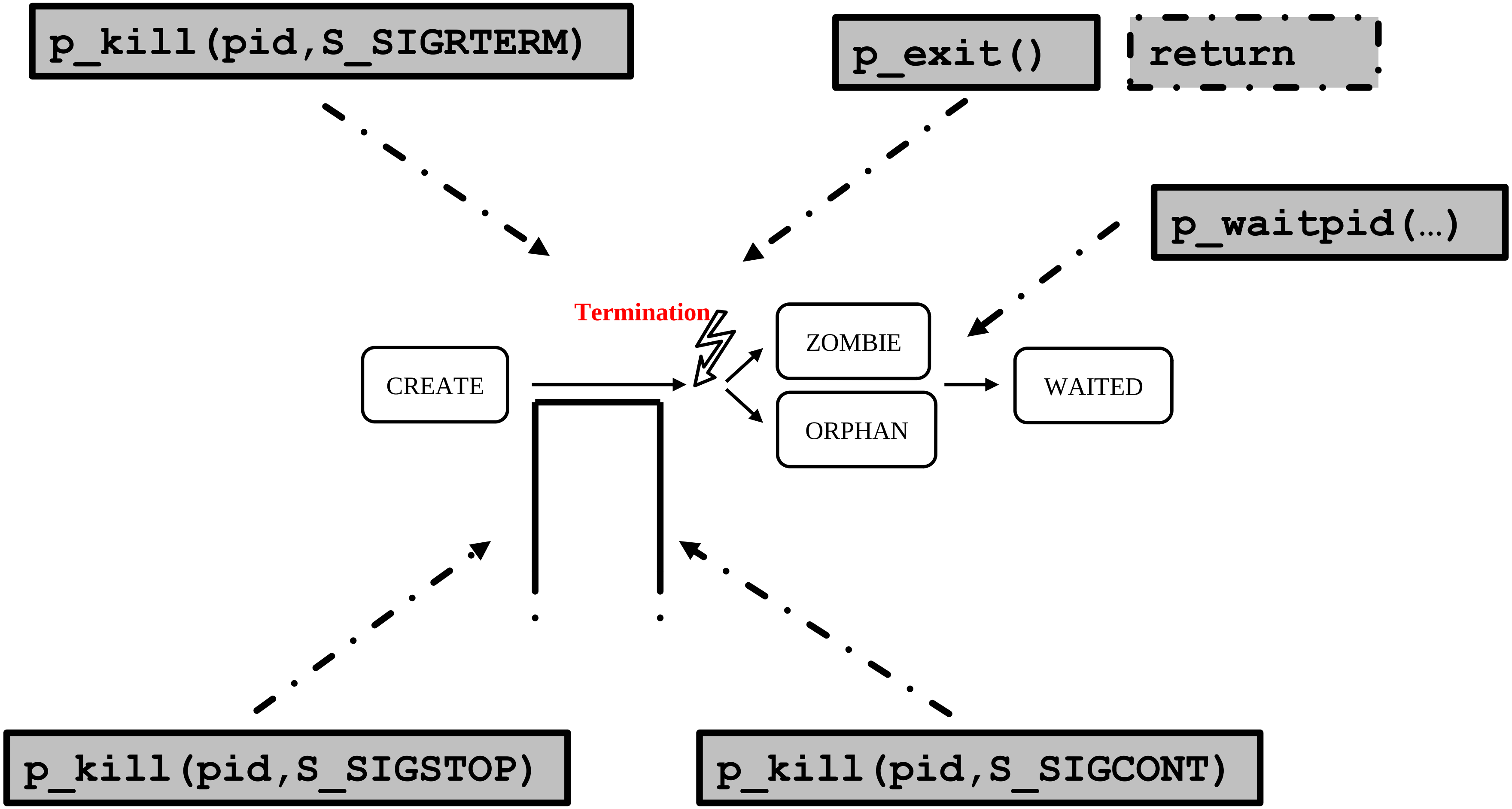
PennOS Kernel Functions



Process Control Block (PCB)

```
typedef struct pcb {









} pcb_t
```

PennOS State Change Functions



Programming with User Contexts

What are User Contexts?

Basic thread-like library

(at the core of `pthread` implementation)

Isolate code execution within a context

Resource sharing

One process can switch between different executions

“Hello Contexts”: a brief tour

```
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");

  return 0;
}
```

ucontext

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

Context run when this one completes

Set of blocked signals for this context

Execution stack for this context

int getcontext(ucontext_t *ucp)

Initializes a **ucontext_t**

- **Does not** initialize **uc_link**, **uc_sigmask**, or **uc_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;
```

```
void makecontext(ucontext_t *ucp,
void (*func)(),
int argc,...)
```

- Specify the function to run when context is activated
  - **`func`** : function to run
  - **`argc`** : number of integer arguments
  - **`...`** : the integer arguments

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

makecontext(&uc, f, 0);
```

```
setcontext(const ucontext_t *ucp)

swapcontext(ucontext_t *oucp,
            const ucontext_t *ucp)
```

Activates a context

- **`setcontext`** : sets the context to **`ucp`**
- **`swapcontext`** : sets context to **`ucp`**, saves current context in **`oucp`**

```
setcontext(&uc);
perror("setcontext");
```

“Hello Contexts”

```
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");

  return 0;
}
```

Ucontext Demo

Many ways to segfault

- Forgetting **`makecontext`**
- Making the stack too small
- Not initializing **`uc_link`**
- Not initializing the context properly with **`getcontext`**
- Re-executing a terminated context

PennOS Shell

Shell Requirements

<u>Synchronous</u> 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`

Demo

Questions?