Recitation 5

PennOS!

Table of Contents

Makefile for PennOS

How to structure your files/directories

`src/` .c and .h files

`bin/` executable binary files

`log/` generated log files

`doc/` README, companion doc, etc.

`tests/` .c files for tests (with its own main() function)

```
|-- Makefile
|-- bin
|   |-- pennfat
|   |-- pennos
|   |-- sched-demo
|   `-- test2
|-- src
|   |-- pennfat.c
|   |-- pennos.c
|   |-- spthread.c
|   |-- spthread.h
`-- tests
    |-- sched-demo.c
    `-- test2.c
```

Editing the Makefile for mains

Add **`.c`** files that have a **`int main(...)`**

to these lines

```
TEST_MAINS = $(TESTS_DIR)/cat_test.c $(TESTS_DIR)/list.c

MAIN_FILES = $(SRC_DIR)/pennos.c $(SRC_DIR)/pennfat.c
```

```
|-- Makefile
|-- bin
|   |-- pennfat
|   |-- pennos
|   |-- sched-demo
|   `-- test2
|-- src
|   |-- pennfat.c
|   |-- pennos.c
|   |-- spthread.c
|   |-- spthread.h
`-- tests
    |-- sched-demo.c
    `-- test2.c
```

Using the Makefile

- **`make`** or **`make all`**: create executables of mains in **`src/`**
  - *Be sure to make a **`bin/`** directory before calling **`make`**
- **`make tests`** : create executables of test mains in **`tests/`**
- **`make info`**: list which files are set as main, execs, etc.
- **`make format`**: auto format main, test main, src, and header files
- **`make clean`**: delete **`*.o`** and executable files

demo

C: header guards, extern variables

- Header guards ➔ prevent including code multiple times in same file
- Extern variables ➔ global variables across files

global_state.h

```c
#ifndef GLOBAL_STATE_H
#define GLOBAL_STATE_H

typedef struct GlobalState {
  int id;
} GlobalState;

extern GlobalState gs;

#endif // GLOBAL_STATE_H
```

main.c

```c
#include "global_state.h"
#include "helper.h"

GlobalState gs;

int main() {
  gs.id = 0;
  ...
}
```

helper.c

```c
#include "global_state.h"

void helper_func() {
  gs.id++;
  printf("%d\n", gs.id);
}
```

Tips

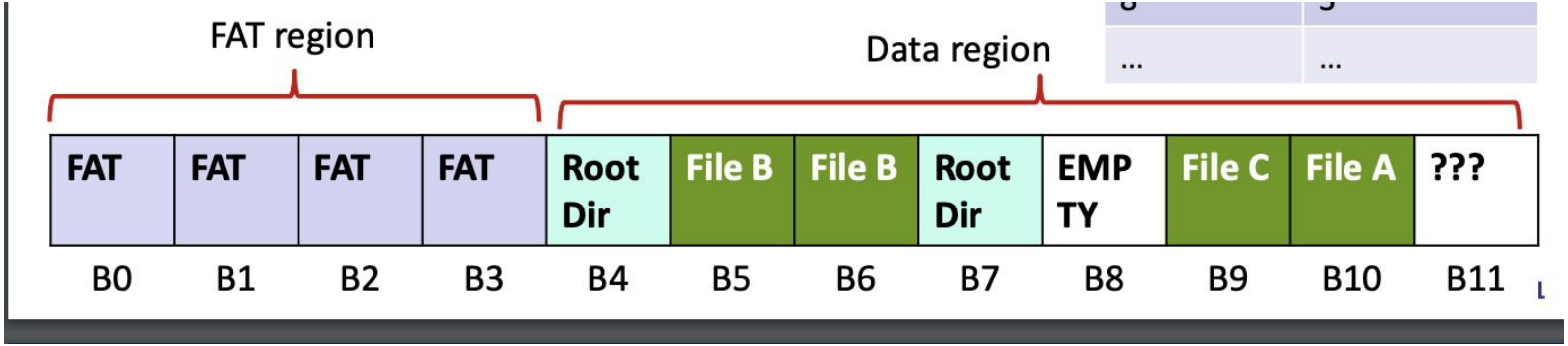
- Functions with varying number of arguments: <stdarg.h>
- Add **`bin/*`**, **`src/*.o`**, and **`.DS_Store`** to your .gitignore
- Check for memory leaks with valgrind (fixing memory leaks ➜ resolve bugs!)
  - Ex: `valgrind bin/pennos`
  - Ex: `valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --verbose bin/pennos`
- Run **`top`** to check CPU usage for kernel
- Using **gdb**:
  - `handle SIGUSR1 nostop`: to not stop whenever a thread is spthread_suspend'd
  - `info threads`: list running pthreads
  - `t N`: switch to thread N

PennFAT

Intro

FAT system splits to two parts:

FAT table and Data blocks



Index	Link
0	0x2004 <— MSB=0x20 (32 blocks in FAT), LSB=0x04 (4K-byte block size)
1	0xFFFF <— Block 1 is the only block in the root directory file
2	5 <— File A starts with Block 2 followed by Block 5
3	4 <— File B starts with Block 3 followed by Block 4
4	0xFFFF <— last block of File B
5	6 <— File A continues to Block 6
6	0xFFFF <— last block of File A
…	…

FAT

Each entry is 2 byte.

First entry give info : # of FAT entries(MSB) and block size(LSB).

Then, all entries are block informations: index is block number, value is next block number.

Second FAT entry must be **ROOT DIRECTORY.**

Which means, FAT[1] is root directory, so first data block must be root directory.

Next entries(FAT[1]......FAT[N])are all file block numbers.

Data block

Root Director and other files.

Root directory stores info of other files.

Metadata(64 bytes)

```
char name[32];
uint32_t size;
uint16_t firstBlock;
uint8_t type;
uint8_t perm;
time_t mtime;
// The remaining 16 bytes are reserved
```

With metadata, we will know first block number of the file, and we can get next block number of the file by indexing FAT table.

FAT[current]=Next.

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

PennFAT thinks itself as a **hard disk,** but actually a **binary file.**

Milestone 1 - Standalone PennFAT

```
# ./pennfat

pennfat> mkfs minfs 1 0
```

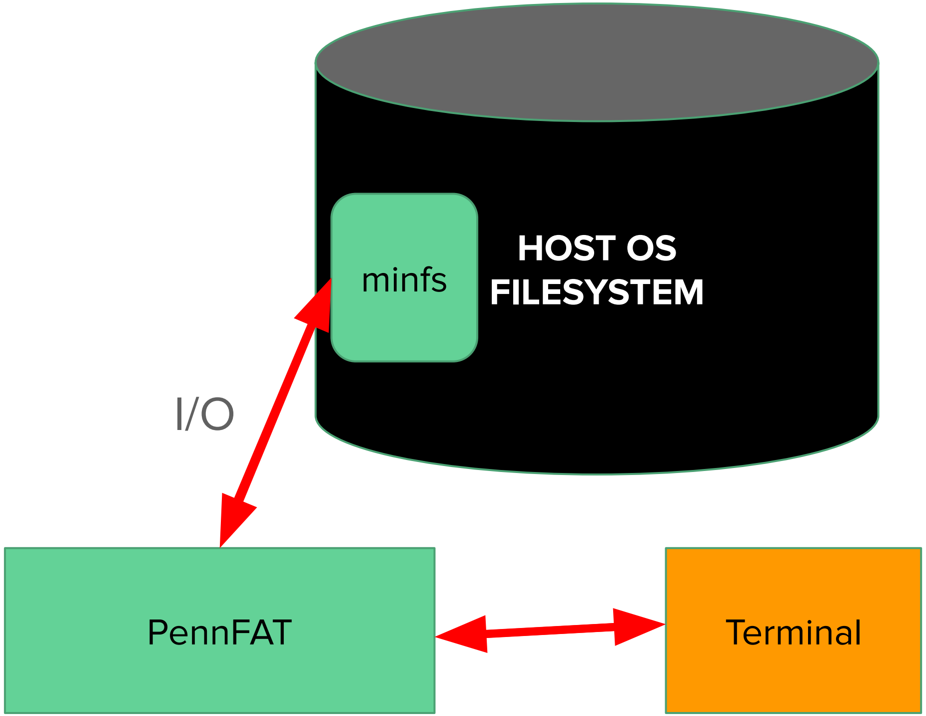
MAKE A FILE SYSTEM!

```
pennfat> mount minfs
```

MOUNT IT!

```
pennfat> touch f1 f2 f3

pennfat> cat -w f1
```



mkfs

- Do not overthink it!

```
TRUNCATE(2)                    Linux Programmer's Manual                    TRUNCATE(2)

NAME
       truncate, ftruncate - truncate a file to a specified length

SYNOPSIS
       #include <unistd.h>
       #include <sys/types.h>

       int truncate(const char *path, off_t length);
       int ftruncate(int fd, off_t length);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       truncate():
           _XOPEN_SOURCE >= 500
               || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
               || /* Glibc versions <= 2.19: */ _BSD_SOURCE

       ftruncate():
           _XOPEN_SOURCE >= 500
               || /* Since glibc 2.3.5: */ _POSIX_C_SOURCE >= 200112L
               || /* Glibc versions <= 2.19: */ _BSD_SOURCE

DESCRIPTION
       The  truncate() and ftruncate() functions cause the regular file named by path or referenced by fd to be trun-
       cated to a size of precisely length bytes.
```

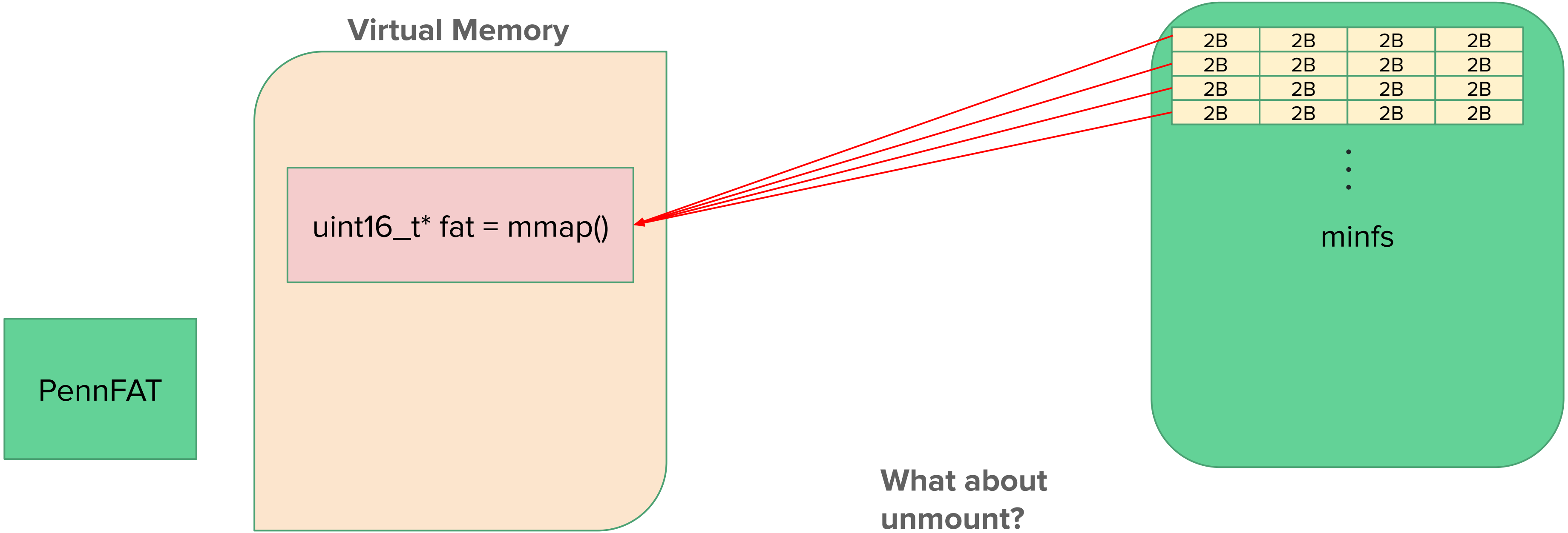
Quick mkfs exercise

```
pennfat> mkfs pikachu 16 2
```

1. Name of Filesystem? pikachu
2. How many blocks in FAT? 16
3. How many entries in FAT? 16*1024/2=8192
4. How many blocks in DATA? 8192-1=8191
5. How big is pikachu in bytes? FAT + DATA = 8192*2 + 8191*1024 = 8403968

mount

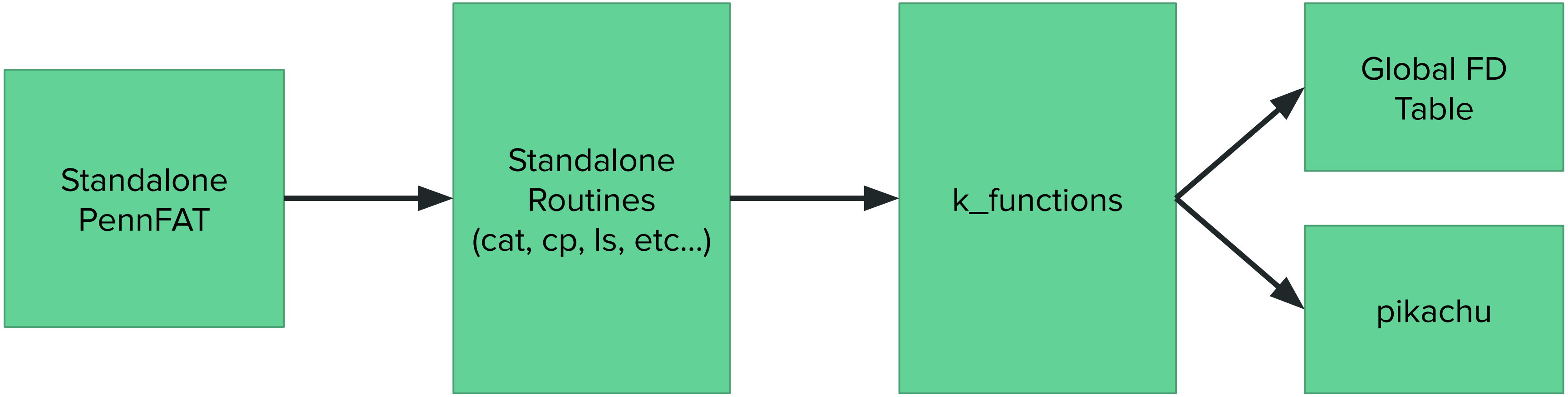
- mmap(2) - creates a new **mapping in the virtual address** space of the calling process.



What about unmount?

k_functions

- Kernel side API specific for PennFAT
- Direct interaction with the PennFAT file system binary
- Direct interaction with the global file descriptor table



Example - k_write(int **fd**, const char *__str__, int **n**)

1. Look for the open file descriptor **fd** in the FD table and retrieve it
2. According to what the *offset* value of the file is, write **n** bytes of **str** from the offset
3. Modify the FAT and Directory entries accordingly

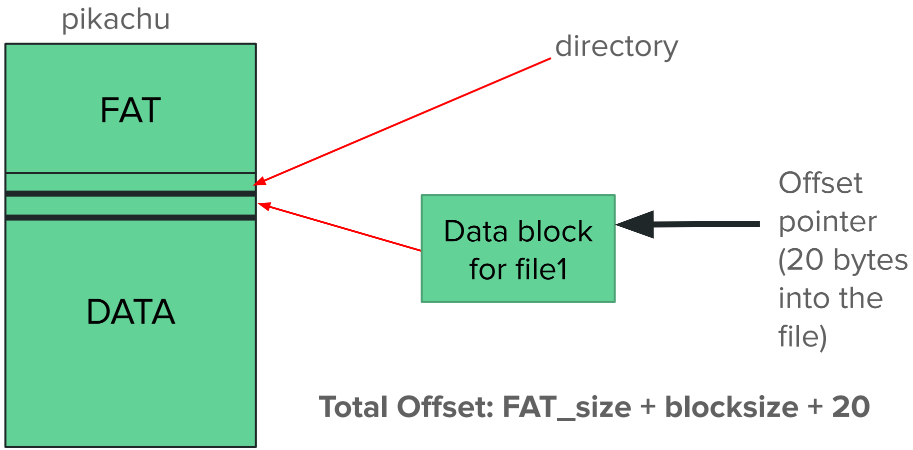
Things to consider when starting

- Think about how you want to structure your file descriptor table. What information do you want to store for each file?
  - Offset, filename, etc...
- What do each k_function want to achieve?
- What happens if you write over a block? What changes in the FAT? The Directory entry?
  - Make sure to update timestamp when you modify a file
- Any error checking?
  - What if there is no more space in the filesystem?
  - What if the file descriptor is open only for reading but you try to write to it?

Comment on Offset

- Each file has their unique *offset*
- Pointer to where in the file a new request to the file will read/write from
- **k_lseek(int fd, int offset, int whence)** can set this *offset* value
- k_read() and k_write() will start reading/writing from this offset pointer
- You can calculate the actual offset of where to write in the filesystem using each file's unique offset value!

File Descriptor Node
Filename: file1
fd: 5
Offset: 20 (bytes)



Standalone Routines

- touch FILE …
  - Creates the file **ONLY**. Does not allocate any memory for it as it has no data written into it.
  - … means multiple files can be created at once
- mv SOURCE DEST
  - Renames SOURCE to DEST **ONLY.**
  - Nothing else. Really.
- cat FILE … [-w/a OUTPUT_FILE]
  - Read contents of FILE(s) and overwrite/append to OUTPUT_FILE
  - Should act like UNIX cat. Exit on ^D (read until EOF)
- cp -h
  - Your HOST OS is files in your **docker container**
  - Everything else are files in **your file system** (pikachu)
- chmod
  - Is included too!

Quick example: cat file1 file2 file3 -w file4

1. **fd1** = k_open(file1), **fd2** = k_open(file2), **fd3** = k_open(file3)
2. k_read(**fd1**), k_read(**fd2**), k_read(**fd3**)
3. **fd4** = k_open(file4)
4. k_write(**fd4**)
5. k_close(**fd1**), k_close(**fd2**), k_close(**fd3**), k_close(**fd4**)

- Note fds and filenames are different
- You may want to have an intermediate buffer to store contents of f1, f2, f3. But you don't need one
- Max number of entries at any time in the FD table during this routine?
  - 7 (stdin, stdout, stderr, f1, f2, f3, f4)
  - min: 4 (stdin, stdout, stderr, and any one file currently being used)

Things to consider

- You are NOT creating a child process to execute something, but rather literally implementing a function that has the functionality of each routines
- These should be implemented using k_functions
  - Only when interacting with host OS, you should be using C system calls
  - Some may not need k_functions
- Function syntax for each routines should be relatively simple!!!
- Check out the examples on the PennOS lecture slides
- You may implement your own k_functions as you need

Some More Clarifications...

- name[0]
  - This is the INTEGER 0 (0x00) not ASCII 0 (0x30)
  - What is 1, what is 2?
- file type
  - What is 0: Unknown, 4: Symbolic Link?
- default permissions
  - Follow UNIX! Read&Write is appropriate here
- Do we mmap FAT only or the entire Filesystem?
  - Up to you. Both ways are valid
- How to handle file deletions?
  - Do we want to zero-out the entire file?
  - Or what is the minimal viable change to indicate a deleted file?
- What if ...?
  - Up to you!

TL;DR

1. Specifications should be followed. (Read the write-up carefully!)
2. When in doubt, follow UNIX behaviors
3. Implementation details are **100% up to you!**
    a. If you think it is appropriate, go ahead!

THIS IS YOUR MILESTONE!

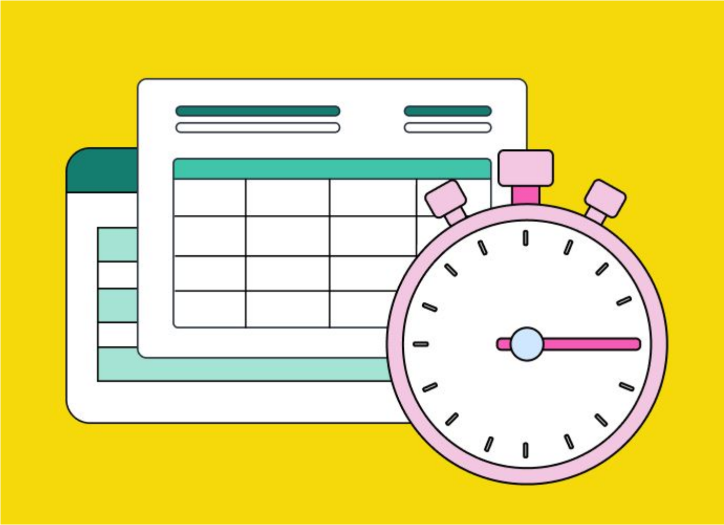
What's After?

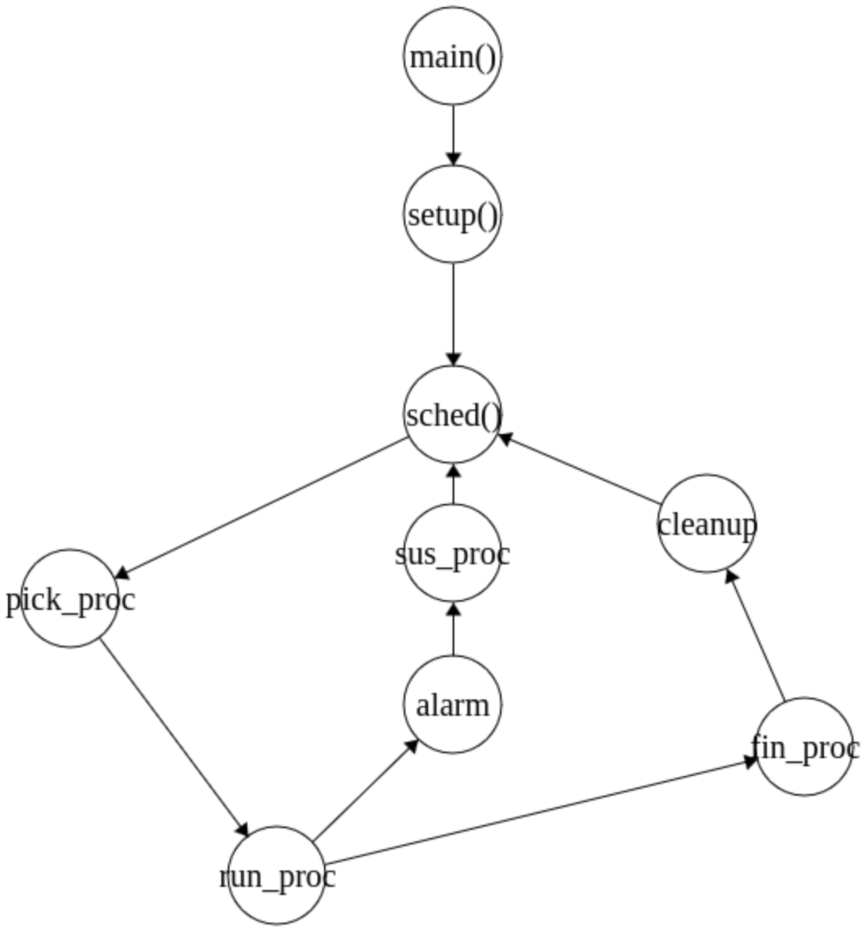
- PennOS and PennFAT Interaction
- u_functions
  - These are your own **system calls!**
  - These provide the connection between PennOS Shell and your File System
- You may use functionalities you implemented in standalone PennFAT to implement u_functions
- You MUST use u_functions to run ANY user-level functions like cat, echo, touch redirections, etc.

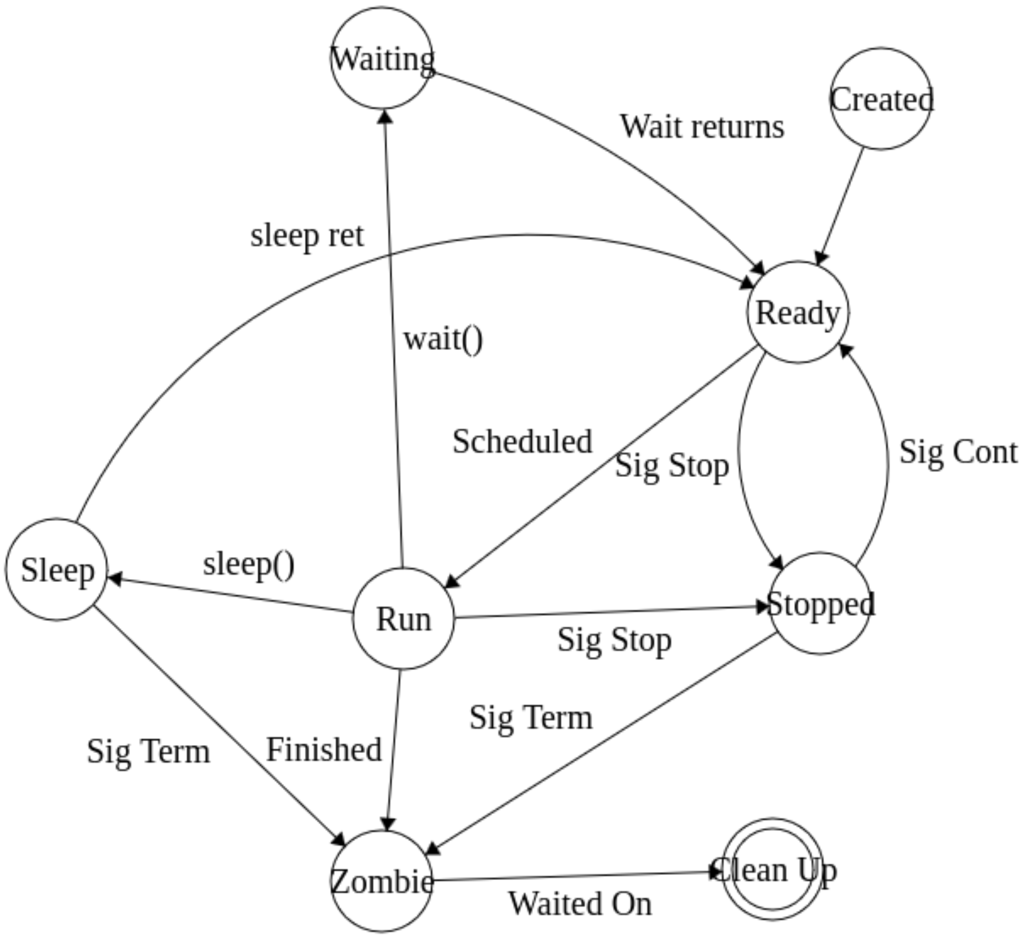
Scheduler

Penn-OS Scheduler Structure

- Runs every "clock" cycle (recurring alarm signal)
- Picks a "process" to run (or the idle process)
- Maintains 3 priority queues, 0, 1, 2
- Lower queue are higher priority
- Maintained ratios of running program




main()
setup()
sched()
sus_proc
cleanup
pick_proc
alarm
fin_proc
run_proc


Waiting
Created
Wait returns
sleep ret
Ready
wait()
Scheduled
Sig Stop
Sig Cont
Sleep
sleep()
Run
Stopped
Sig Stop
Sig Term
Sig Term
Finished
Zombie
Clean Up
Waited On

Functions To Implement the Scheduler

Kernel Level:

- k_proc_create
- k_proc_cleanup

User Level:

- s_waitpid
- s_spawn
- s_kill
- s_exit

k_proc_create

```
/**
 * @brief Create a new child process, inheriting applicable proper
ties from the parent.
 *
 * @return Reference to the child PCB.
 */
pcb_t* k_proc_create(pcb_t *parent);
```

k_proc_cleanup

```c
/**
 * @brief Clean up a terminated/finished thread's resources.
 * This may include freeing the PCB, handling children, etc.
 */
void k_proc_cleanup(pcb_t *proc);
```

s_spawn

```
/**
 * @brief Create a child process that executes the function `func`.
 * The child will retain some attributes of the parent.
 *
 * @param func Function to be executed by the child process.
 * @param argv Null-terminated array of args, including the command name as argv[0].
 * @param fd0 Input file descriptor.
 * @param fd1 Output file descriptor.
 * @return pid_t The process ID of the created child process.
 */
pid_t s_spawn(void* (*func)(void*), char *argv[], int fd0, int fd1);
```

s_waitpid

```c
/**
 * @brief Wait on a child of the calling process, until it changes state.
 * If `nohang` is true, this will not block the calling process and return immediately.
 *
 * @param pid Process ID of the child to wait for.
 * @param wstatus Pointer to an integer variable where the status will be stored.
 * @param nohang If true, return immediately if no child has exited.
 * @return pid_t The process ID of the child which has changed state on success, -1 on error.
 */
pid_t s_waitpid(pid_t pid, int* wstatus, bool nohang);
```

s_kill

```c
/**
 * @brief Send a signal to a particular process.
 *
 * @param pid Process ID of the target proces.
 * @param signal Signal number to be sent.
 * @return 0 on success, -1 on error.
 */
int s_kill(pid_t pid, int signal);
```

s_exit

```
/**
 * @brief Unconditionally exit the calling process.
 */
void s_exit(void);
```

s_sleep

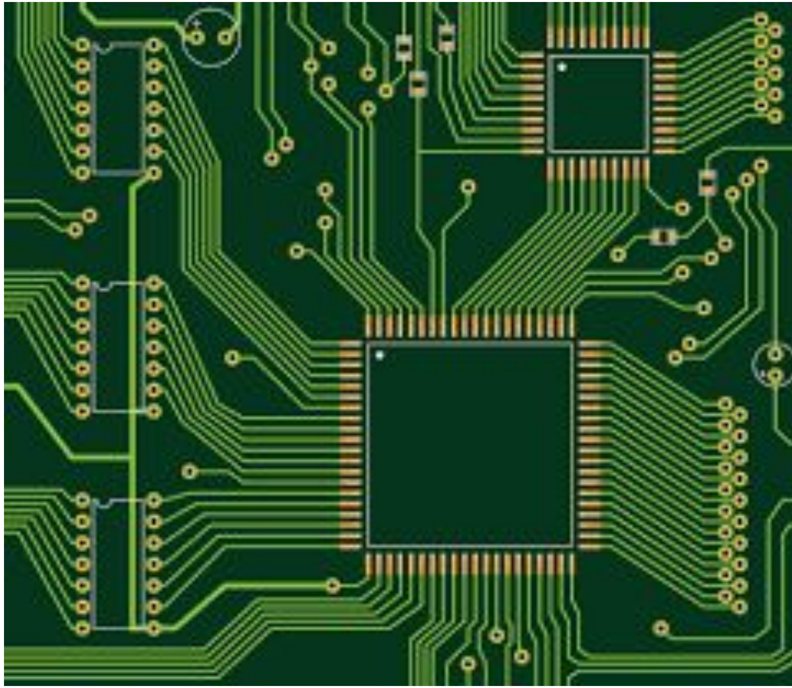
```c
/**
 * @brief Suspends execution of the calling proces for a specified number of clock ticks.
 *
 * This function is analogous to `sleep(3)` in Linux, with the behavior that the system
 * clock continues to tick even if the call is interrupted.
 * The sleep can be interrupted by a P_SIGTERM signal, after which the function will
 * return prematurely.
 *
 * @param ticks Duration of the sleep in system clock ticks. Must be greater than 0.
 */
void s_sleep(unsigned int ticks);
```

s_nice

```c
/**
 * @brief Set the priority of the specified thread.
 *
 * @param pid Process ID of the target thread.
 * @param priority The new priorty value of the thread (0, 1, or 2)
 * @return 0 on success, -1 on failure.
 */
int s_nice(pid_t pid, int priority);
```

PCB Struct

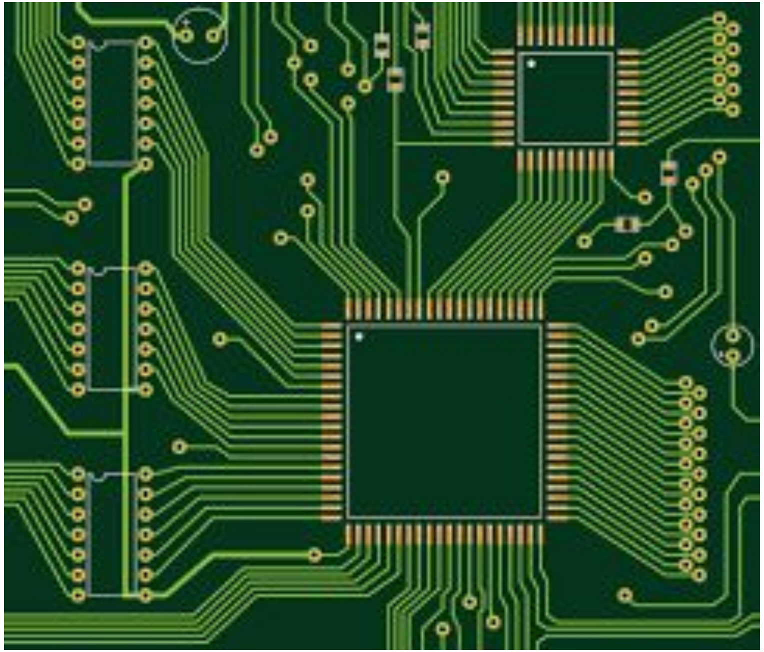
What might we want to have?



PCB Struct

What might we want to have?

- Spthread pointer/struct
- Status of process
- File descriptors
- Parent process identification
- Children process identification
- File descriptors



Ways To Get Started

- Try starting from the ground up. Implement function headers, structs, and constants. Think PCB, signal numbers and function outlines
- Look at sched-demo.c and understand it. Try and implement your own shell which can take an input and based on the input schedule different threads
- Create the outline of the queues and think about how the correct queue will be chosen (and ensured it has a process on it)

Any Questions?