Intro to CS 169 & Processes Help Session

Spring 2019

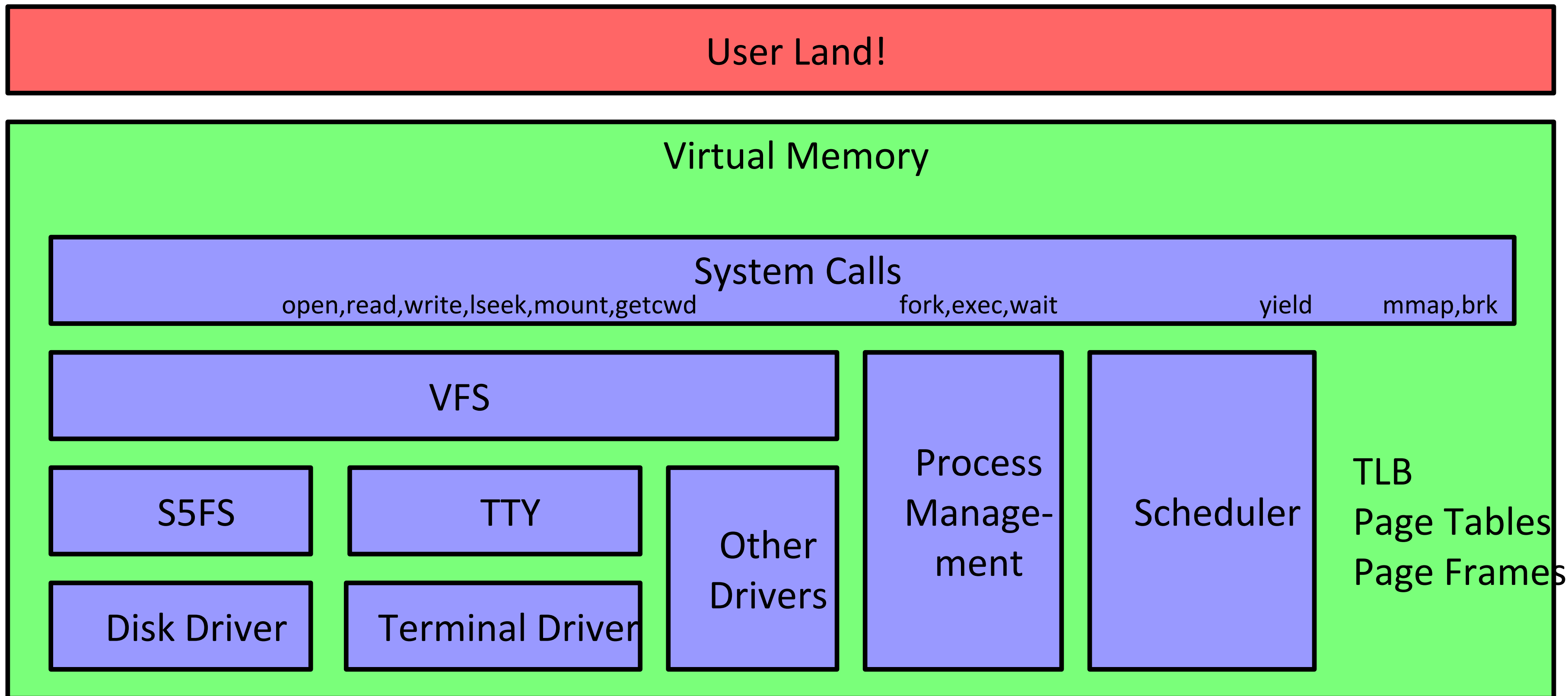
Course Outline

- Write an actual operating system (Weenix)
- Five projects:
  - Processes (Procs) - February 11th
  - Drivers - February 20th
  - Virtual File System (VFS) - March 11th
  - System 5 File System (S5FS) - April 8th
  - Virtual Memory (VM) and final integration - May 6th
- Once you are done, you will be able to run many simple C programs on your own OS!
- You can switch back to CS 167 at any time

Getting and Building Weenix

- `git clone <your repo url>`
- Read the `README`
- `make -j 4`
- `./weenix -n`
- Make sure to change the flags in `Config.mk` when you start a new assignment
    - E.g. if you are on VFS: `DRIVERS=1,VFS=1,S5FS=0,VM=0,DYNAMIC=0`
- If you work remotely over SSH, make sure to forward X11 (`ssh -Y`)
- If you don't know git, the TAs can help you! (stick around after)
- You'll be using a **private** GitHub Classroom repository

Weenix Layout



Weenix - Documentation

- The `README` helps you get started
- All the documentation is in `doc/latex` and on the course website
- Need to run `make` first in this directory to build the PDF
- The documentation is your friend and guide
- There will not be individual assignment handouts posted on the website

Tools - `nyi`

- `make nyi`
- Run this to get a list of functions you still need to implement

Tools - gdb

- `./weenix -d gdb`
- Learn how to use breakpoints and inspect data if you do not already
- You can add GDB commands you want to run every time you boot weenix by adding them to `init.gdb`
- **Please use GDB**

Tools - `cscope`

- `cscope` allows you to search the entire source code
  - Where a function or variable is declared or defined
  - Where something is used or called
  - What functions call a particular function
- Build database with `make cscope` in the `kernel/` directory
- Run `cscope` while in the `kernel/` directory
- `cscope` plugins exist for common text editors

Tools - kshell

- Kernel mode shell provided by the TAs
- Easily extensible (easy to add new commands)
- Add your test cases here
- kshell itself depends on / tests your Drivers code (so you can’t use it in Procs)
- Put your procs test cases directly in `initproc_run` (the first thread that runs in the kernel, see the documentation)

Reference - manpages

- Many (perhaps most) of the functions in weenix emulate actual system calls in unix systems. These are named as `do_<function_name>`. Reference the manpages of these calls for expected behavior, assumptions, and error codes.
- Doing so not only maintains sensible standards, it may illuminate edge cases which you may not have yet considered.

References - OSDev Wiki

- https://wiki.osdev.org/Main_Page
- Wiki about hardware and operating systems development
- Goes into lots of depth - more detail than you need, but a great resource

Debugging - Logs

- Use `dbg()` macro to log information to standard output
- `dbg(DBG_THR, “Debug value is %d\n”, val)`
- `DBG_THR` is a “debug mode”, of which there are many, and which are color-coded in the output
- Enable / disable whether certain debug modes are printed out by setting the DBG variable in `Config.mk`
- E.g. `DBG=-all,thr` will only print `DBG_THR` messages, all other debug statements will be silenced

Debugging - panics

- Panics occur when Weenix enters an illegal state
- Halts the processor (i.e., Weenix crashes)
- `panic(“YOU DID BAD THING NUMBER %d!!!\n”, badthing);`
- Better to crash in a controlled way like this, rather than farther down the line

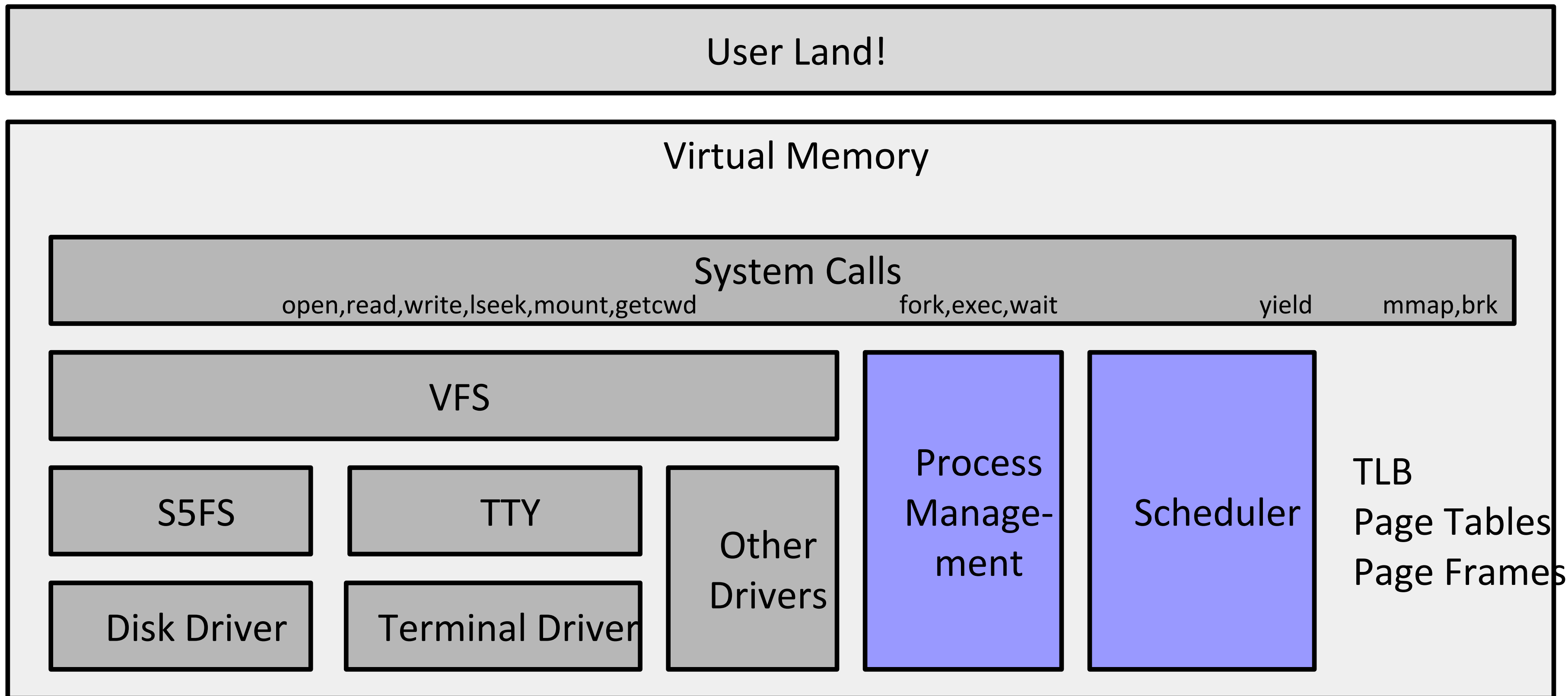
Debugging - kasserts

- Use the KASSERT macro liberally
- Causes a kernel panic when assertion fails
- `KASSERT(NULL != input);`
- `KASSERT(NULL != input && “input to my-func should not be null”);`
- KASSERT whenever something must be true for your kernel to function correctly
- You can also assert two values are equal (casted to `ints`) using KASSERTEQ. This will print out the values if the assert fails. Similar assertions exist for <, <=, >, >=, and !=

Overview - Processes

- Processes
- Threads
- Locks (provided!)
- Scheduling (partially provided!)

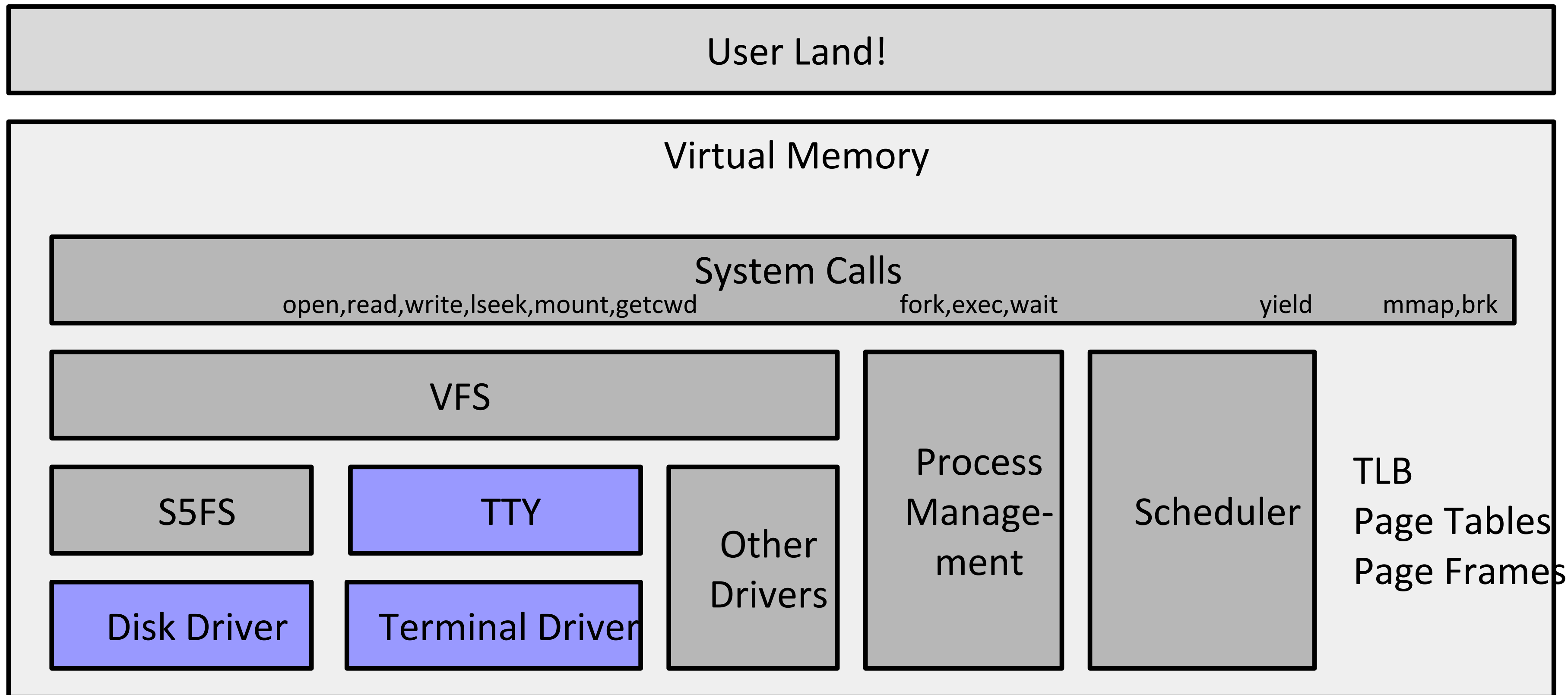
Weenix Layout (Processes)



Overview - Drivers

- How the OS interacts with hardware devices
- TTY driver
  - Line Discipline
  - TTY
  - Basically, how you can type things in to your OS, have backspace work, pass typed in text to applications, etc.
- Disk driver
  - Used later in S5FS and VM

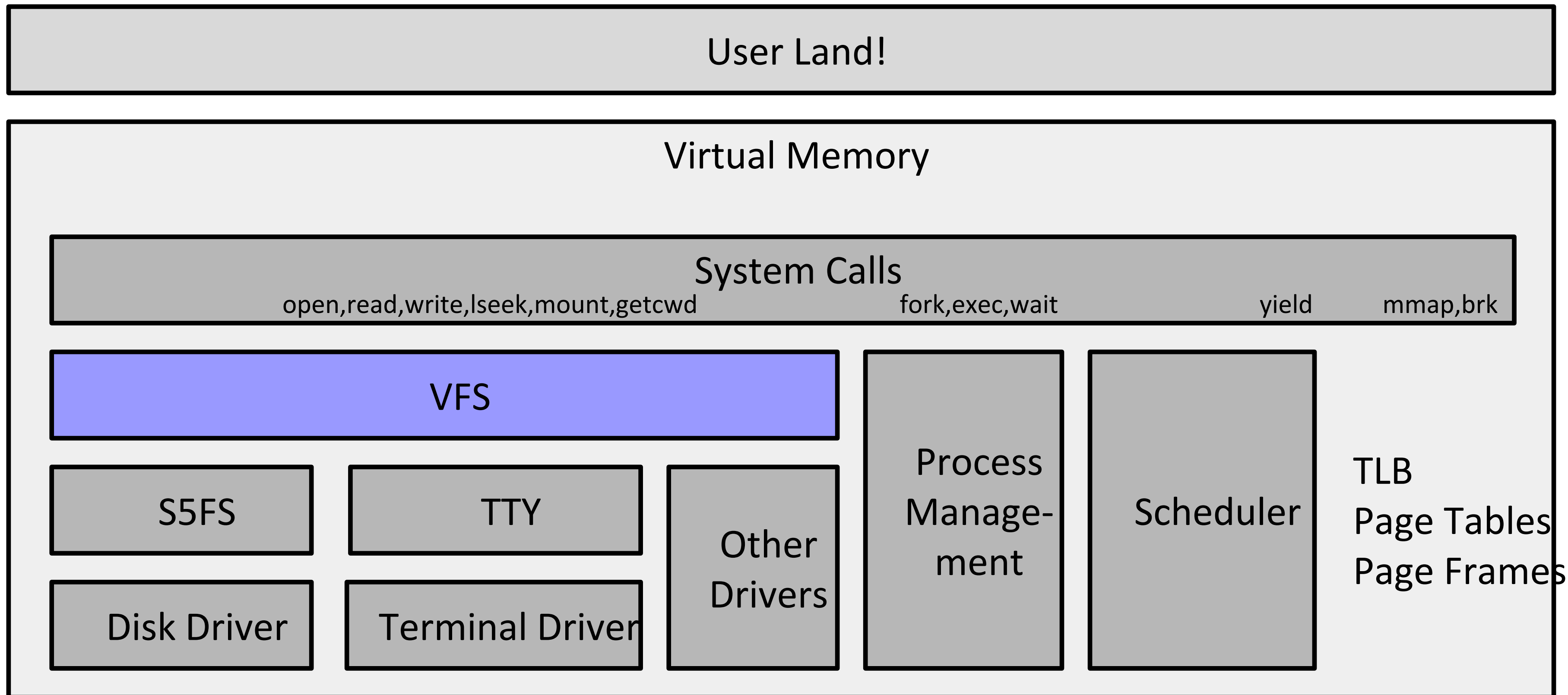
Weenix Layout (Drivers)



Overview - VFS

- Gives “abstract” view of the filesystem, like an API
    - Everything is a file or a directory
- Hides implementation of in-memory or on-disk filesystems behind a consistent API
- Introduction to “reference counting,” a memory management technique

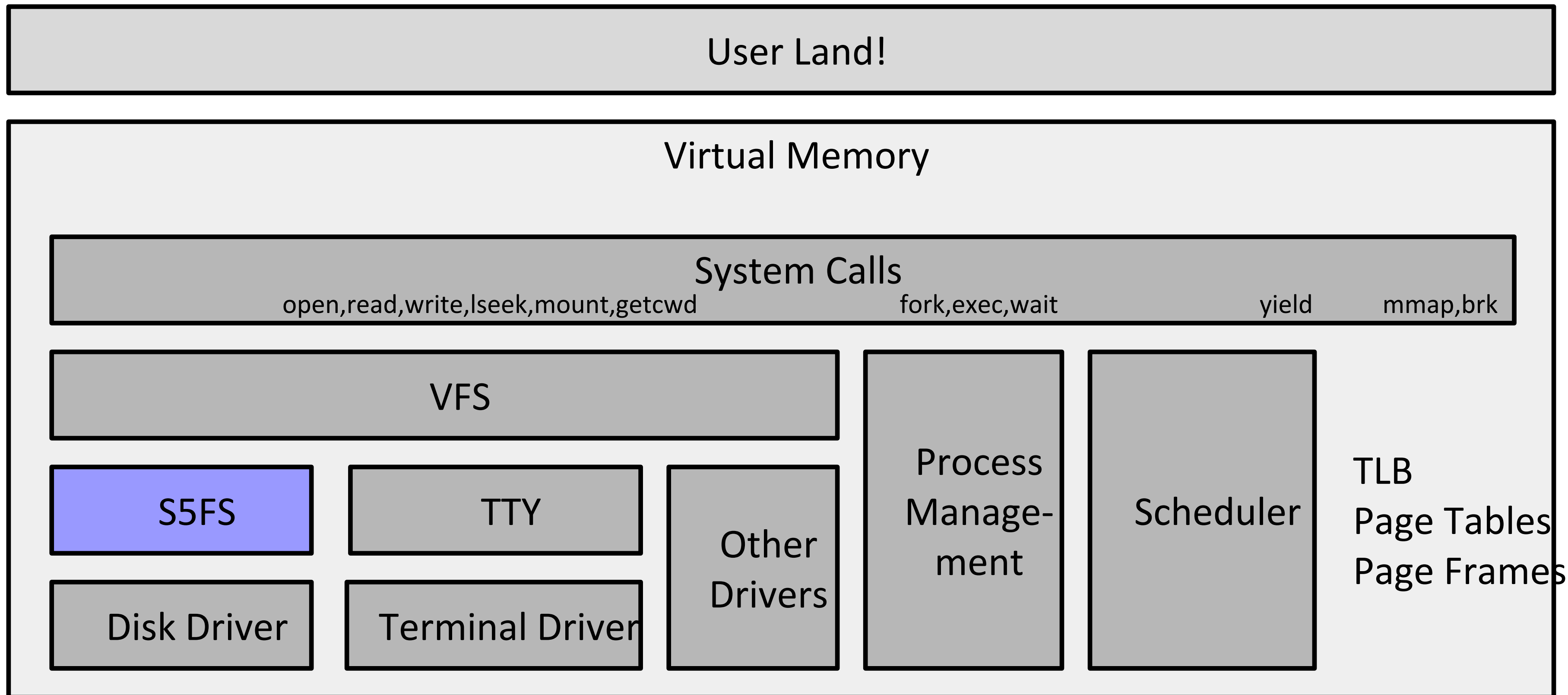
Weenix Layout (VFS)



Overview - S5FS

- On-disk filesystem using your disk driver
- Implements backend for files, directories
- Learn about the page frame system and memory objects before VM
- Until this project, you will not be able to persist any changes to disk
    - Every boot is fresh

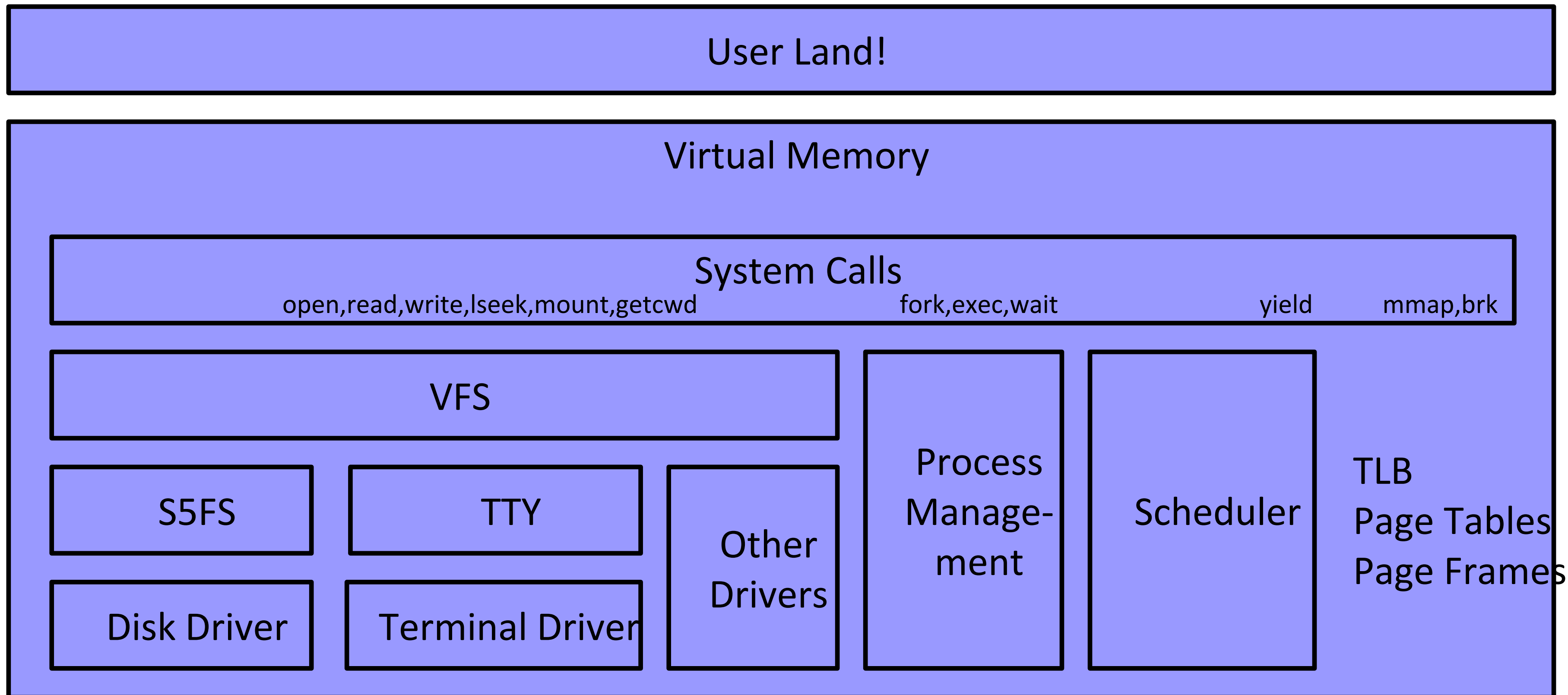
Weenix Layout (S5FS)



Overview - VM

- Uses every subsystem extensively
    - You will find bugs in code you wrote months ago
- Allows processes to share memory and map files into their memory
- **Allows you to run processes in userland!**
    - And then jump up from your seat in the Sunlab and scream, “I DID IT!”

Weenix Layout (VM)



Processes Help Session

Processes - Details

- Process management
- Scheduling
- Synchronization primitives

Processes - “contexts”

- How do we switch between threads?
- We need to save the state of the currently running thread (e.g. `eip`, `esp`, `ebp`, the kernel stack pointer, etc.) somewhere
- We use a `context_t` struct to store this information
- Each thread needs its own context

Processes - `kthread`

- Defines a kernel thread
- Each `kthread_t` contains a context
- Set up this struct in `kthread_create`
- Clean it up in `kthread_exit`
- Think about threads as roughly “units of execution” and processes as “units of resources”

Processes - `proc`

- `proc_t` defines a process
- In the version of Weenix which you will be implementing, **each process only contains one kthread**
- Set this struct up in `proc_create`
- Clean it up in `proc_cleanup`

Processes - sched

- Like uthreads!
- The run queue is used for threads waiting to run (no priorities)
- Alternatively, a thread may be waiting on a sleep queue
  - Waiting for a mutex
  - Waiting for a child process to exit
- **A thread that is not currently running must be on some queue**

Structure of the Files

- `.c` files in `kernel/proc` and `kernel/main`
  - Your code goes here!
  - Functions in these files have super helpful comments describing how to implement them
- `.h` files in `kernel/include/proc`
  - List function declarations and structs
  - Have comments explaining what the functions do and return, and what the struct fields are for

Linked lists in Weenix

- Defined as macros in `kernel/include/util/list.h`
- The comments in the file cover topics such as:
  - Iterating through a linked list
  - Inserting/removing items from the list
  - Accessing items from the list
- Necessary to finish Procs!
  - `proc_t` and `kthread_t` structs both have linked list fields.

Getting Started

```
Not yet implemented: PROCS: bootstrap, file main/kmain.c,
  line 130
panic in main/kmain.c:132 bootstrap(): weenix returned to
  bootstrap()!!! BAD!!!
```

- Start by making this scary error go away!
  - `proc_create()`
  - `kthread_create()`
  - `bootstrap()`
- Then move onto other functions in `proc.c` and `kthread.c`, then `sched.c` and `main.c`
- **Write tests** in `initproc_run()`

Common Mistakes / Recommendations

- Make sure that you read the provided stencil code before you start.
- Ensure proper memory allocation and deallocation in creation and deletion of processes / threads. This includes not only the timing of allocation / deallocation but also where (what function calls are used) the memory is coming from (things which may work now may not work later).
- With the above point, be careful about the memory that a parent process is responsible for cleaning up (rather than the child process).
- Test `do_waitpid()` and `sched_switch()`

Mentor TAs

- Each student is assigned one mentor TA for all of CS 169
- Questions should be directed to the TA's email or Piazza (ask your mentor which they prefer), but your mentor TA will keep an extra close eye on your questions and make sure that they get answered
- After every project you will demo your code for your mentor TA, and have a chance to ask them questions
- Mentor TAs are very nice people who are often willing to answer questions off hours, but remember that you should only go to them once you have tried very hard to solve your problem for yourself

The Scary Slide

- `#define early now`
- Start early!
- Test your code well after each project
- Think before you code - try to understand the code flow before you start coding
- If something is broken, spending a little time just logically reasoning through your code flow will save a lot of time spent in the debugger or the logs
- **But you can do it!**

Introduction to Git

Introduction to Git

- Running the command `git clone <your repo url>` creates a *git repository* for you
- A git repository is a form of version control which allows you to keep track of the history of files, compare changes over time, and go back to old versions
- You should snapshot your changes periodically by “committing” them
- The git documentation is very comprehensive and can be found at http://git-scm.com/doc
- TAs are willing to help, but please Google first and reference the man pages first, as there are answers for nearly all git questions out there

Git Workflow

- Make some changes
- `git status`
  - Tells you what files have been modified since last commit
- `git diff`
  - Shows you what's changed since your last commit or add of the file
- `git add file1 file2 …`
  - File1, file2, … are modified files whose changes you want to record
  - Only add changes that you want to *commit*
- `git commit -m "an informative message"`
  - This stores the snapshot of the current state of the files, so that you can refer back to them

Other Useful Git Commands

- `git log`
  - View previous commits
- `git checkout <commit> <file>`
  - Replace your current version of `<file>` with `<commit>`'s version
- `git stash` and `git stash pop`
  - "Stash" current changes without committing them

Introduction to Testing

Automated Testing Practices

As you begin Weenix, we highly encourage you to write your own unit test suites for each part. These tests will help maintain your sanity as well as make integration smoother after completing VM. On the next slide is a general recommended flow for creating test suites.

Testing Flow

- Write tests as functions in a test suite file, e.g. `proc_tests.c`
- Create a wrapper function to call all testing functions, e.g. `int run_proc_tests()`
- Create header file with appropriate function declarations, e.g. `proc_tests.h`
- Include header file in `kmain.c`, e.g. `#include {path_to_tests}/proc_tests.h`
- Invoke testing function wrapped in preprocessor, (PROCS is always defined so using DRIVERS as example) e.g.

```
#ifdef __DRIVERS__
    run_drivers_tests()
#endif
```