CSE 421/521
Introduction to Operating Systems

Farshad Ghanei

Project-3 Discussion

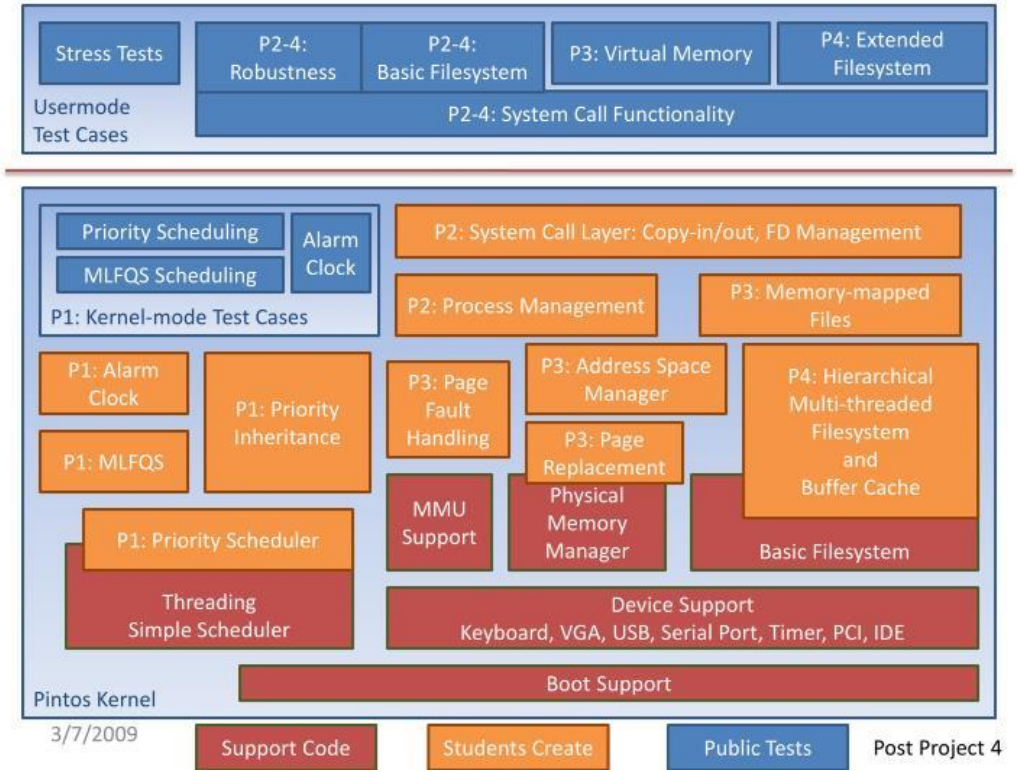
* Slides adopted from Prof Kosar and Dantu at UB, “Operating System Concepts” book and supplementary material by A. Silberschatz, P.B. Galvin, and G. Gagne. Wiley Publishers

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

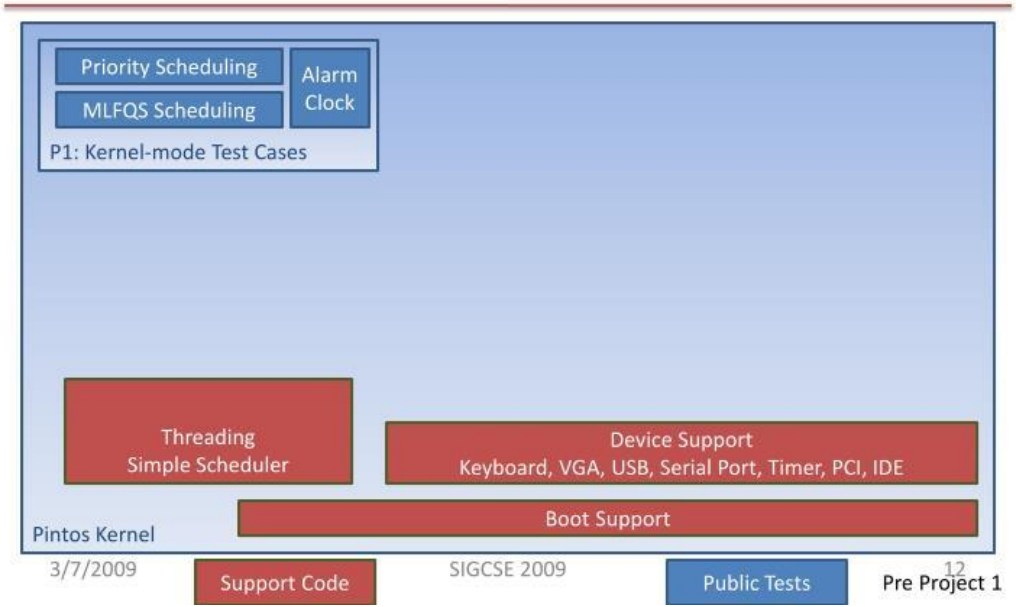
Today

- Pintos projects:
  - Project-1: Threads
  - Project-2: User Programs
  - Project-3: Virtual Memory
    - Step 1: Preparation
    - Step 2: Understanding Pintos
    - Step 3: Design Document
    - Step 4: Implementation
    - Step 5: Testing
  - Project-4: File Systems (Not part of our course)

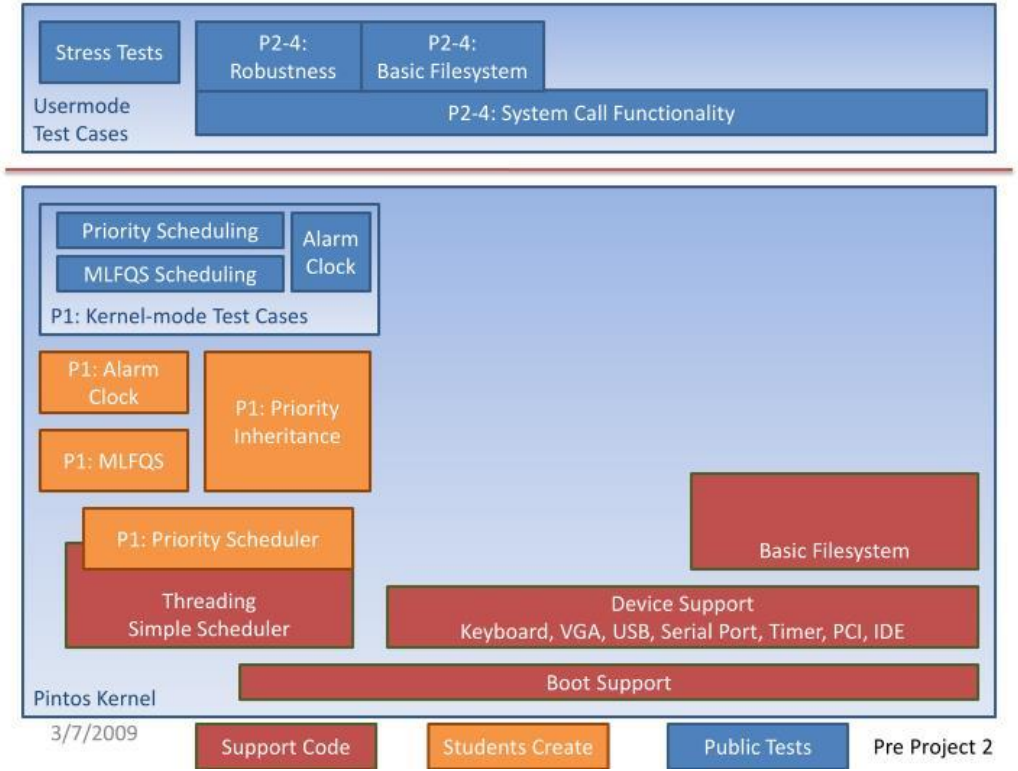
Pintos - After full implementation (Post Project 4)



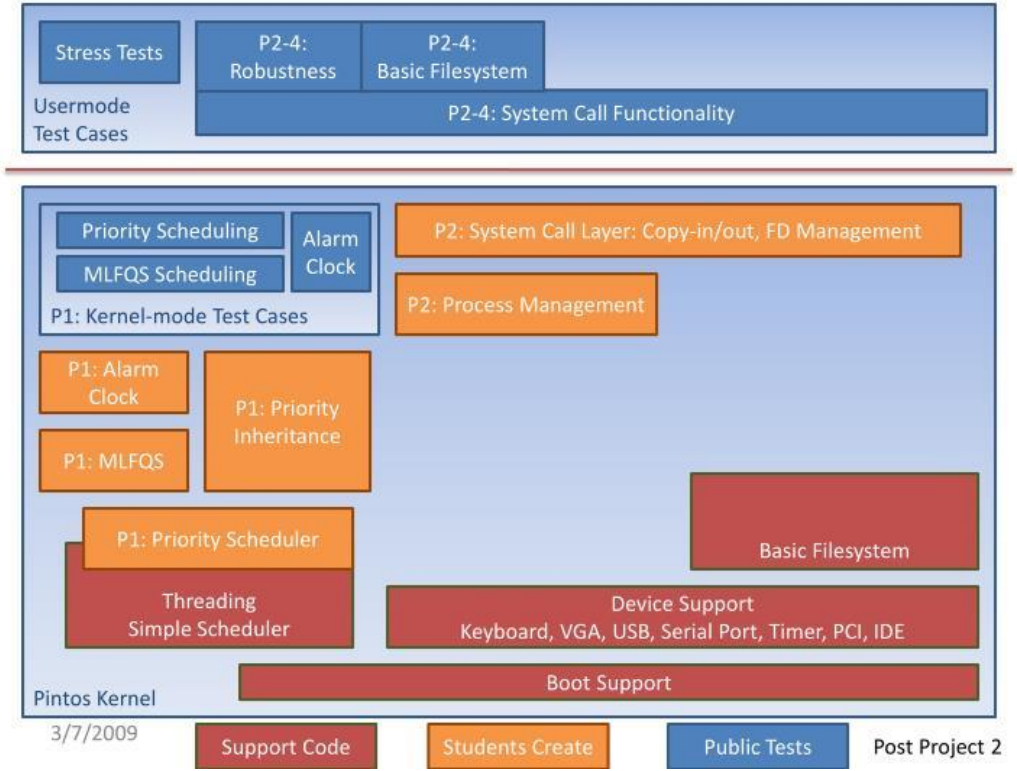
Pintos - You Started from Here (Pre Project 1)



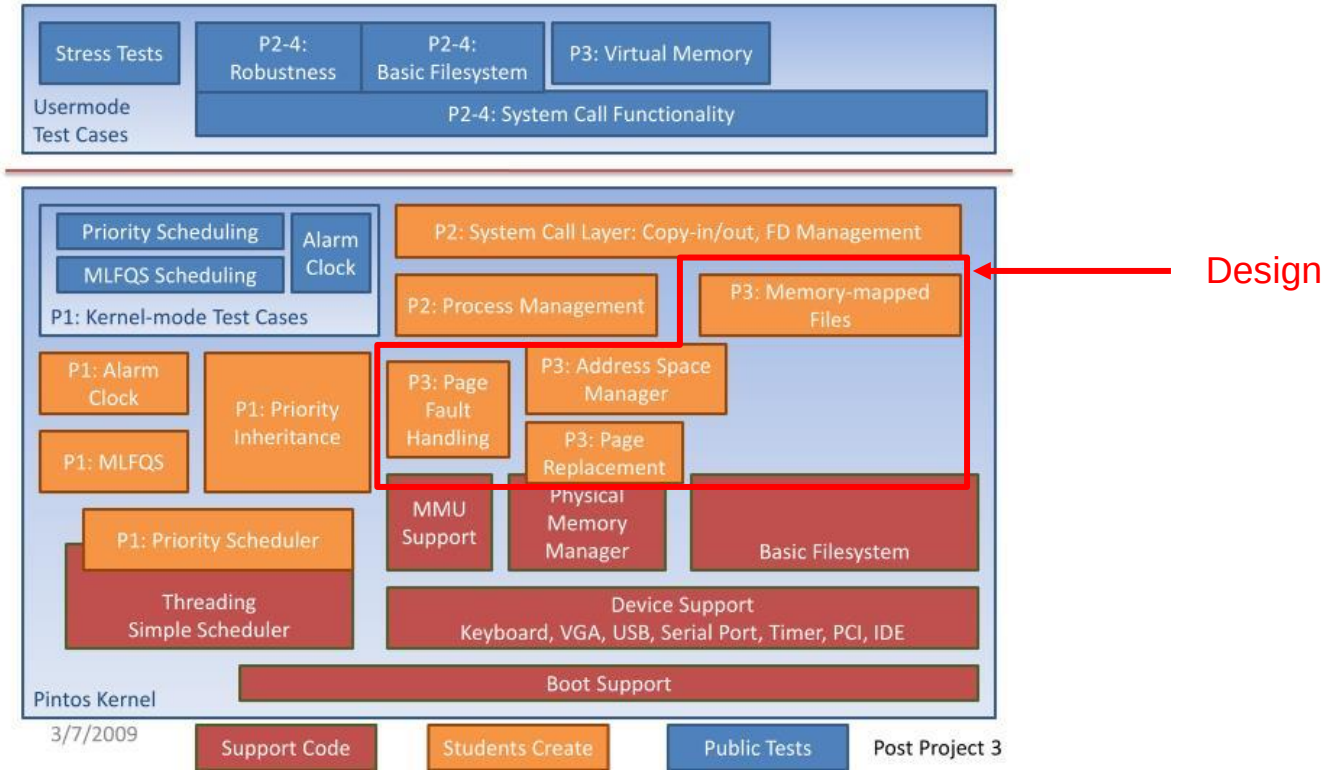
Pintos - You First Reached Here (Pre Project 2)



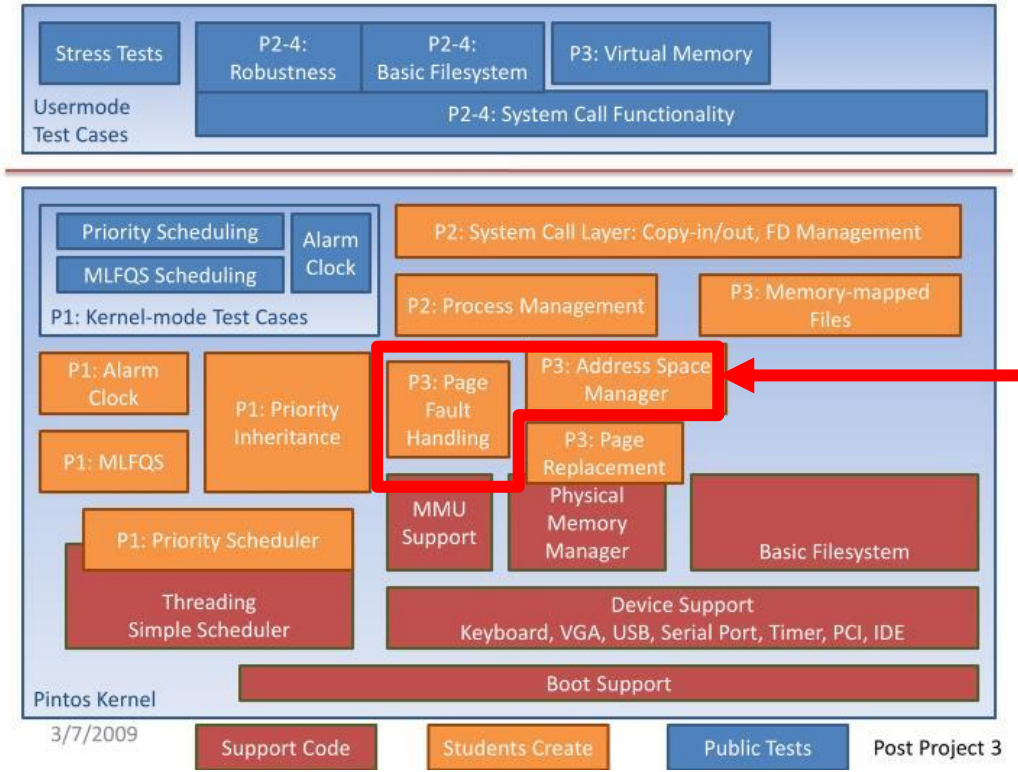
Pintos - You Then Reached Here (Pre Project 3)



Pintos - You Will Implement This (Post Project 3)



Pintos - You Will Implement This (Post Project 3)



Project-3: Virtual Memory

- Step 1: Preparation
- Step 2: Understanding Pintos
- Step 3: Design Document
- Step 4: Implementation
- Step 5: Testing

Step 1: Preparation

Readings from zyBooks:

- Chapters 6, 7

Readings from Pintos documentation:

- Chapter 1 - Introduction
- Chapter 4 - Project 3: Virtual Memory
- Appendix A - Reference Guide [A.1, A.2, A.3, A.4, **A.5, A.6, A.7**]
- Appendix C - Coding Standards
- Appendix D - Project Documentation
- Appendix E - Debugging Tools

Step 1: Preparation

- Project-3 DEPENDS on correct implementation of Project-2
- We will not continue with your Project-2 code
- We will start a new repo specifically for Project-3
- The new repo has other modifications, and currently does not work!
- You'll need to go through the github classroom process again
  - Please pay more attention
  - It is your responsibility to follow instructions and resolve issues.
  - Group name, deadlines, etc.

Step 2: Understanding Pintos

- Terminologies
- How virtual memory works
- Distinctions between user and kernel virtual memory

Step 2: Understanding Pintos

- **Virtual Address:** The only “Address” in the eyes of a process.
- **Page Fault:** Violation while accessing memory: not present, kernel mode, read-only

- **Page (Virtual Page):** A section of virtual memory (4 KB) which is page-aligned.
- **Page Table:** A table used directly by CPU to translate virtual addresses to physical addresses for memory access. (Page number to frame number).

- **Frame (Physical frame):** A section of physical memory (4 KB) which is page-aligned.
- **Frame Table:** Stores information about the frames you have allocated from the physical memory. Used in eviction.

Step 2: Understanding Pintos

- **Swap:** A partition on disk for writing / reading between the memory and the disk.
- **Swap Slot:** A page-size region on swap partition, that is page-aligned
- **Swap Table:** Keeps track of which pages of the swap area are occupied/free

- **Supplemental Page Table:** Additional data about each page, that is going to help you handle page faults, for valid pages that are not present.

- **Table of file mappings:** Processes may map files into their virtual memory space. You need a table to track which files are mapped into which pages.

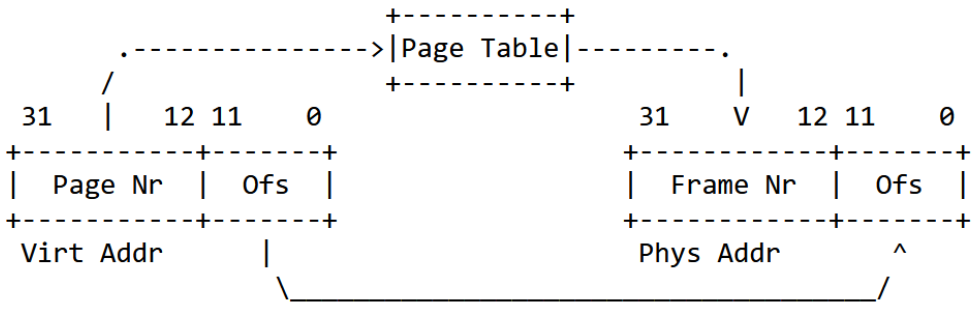
Paging in Pintos

Address consists of page number | offset

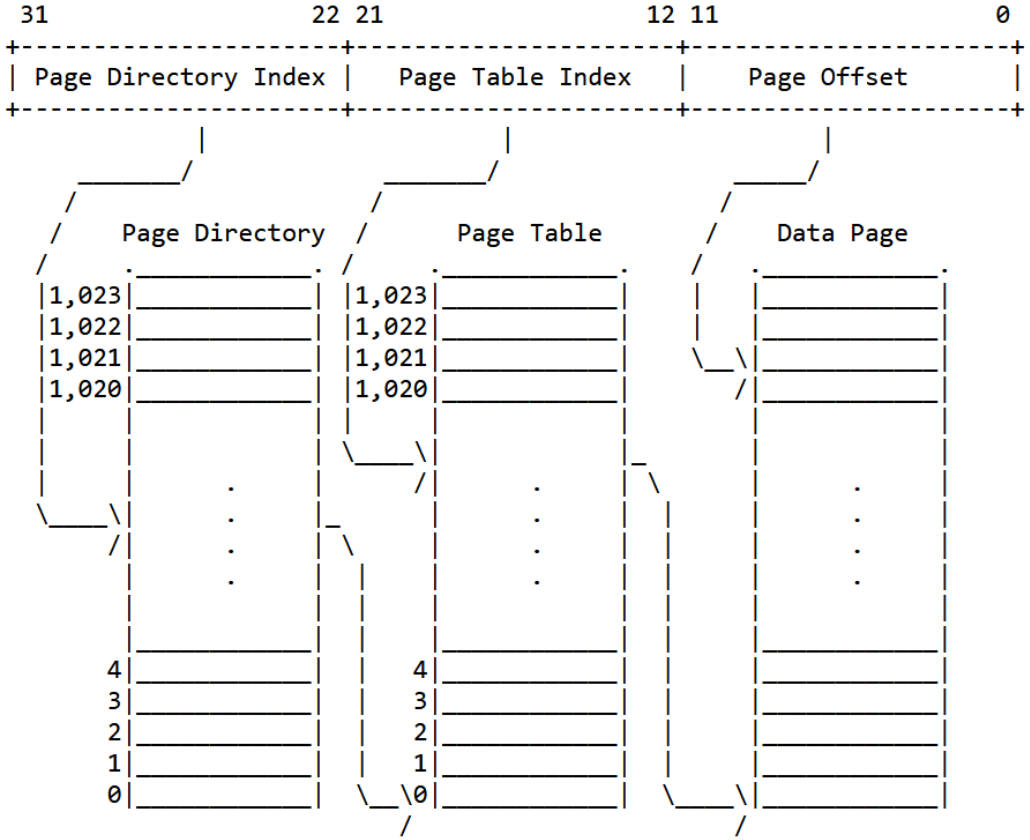
Page number translates to frame number

offset is not modified.

pagedir.c

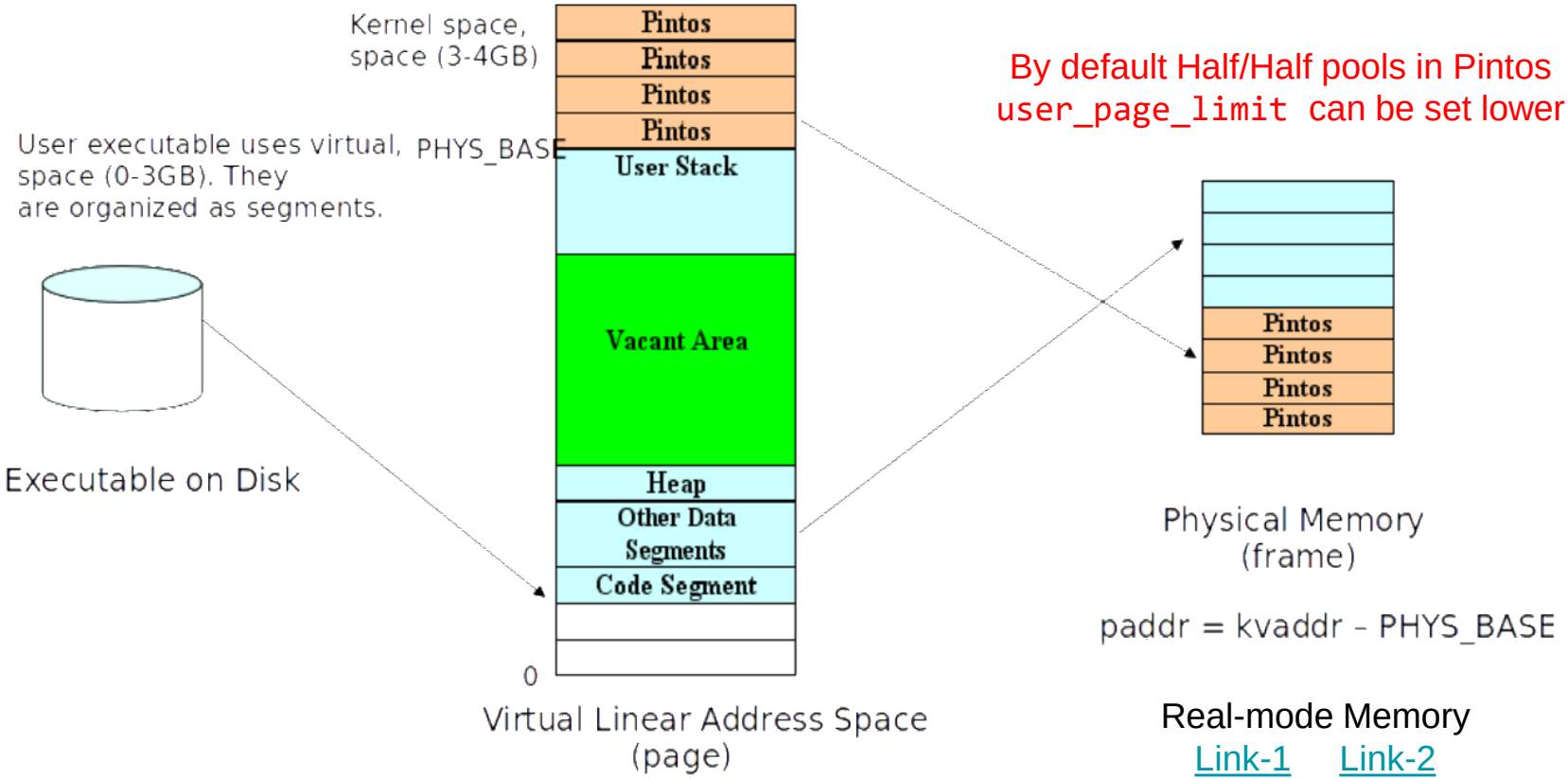


Paging in Pintos



pagedir is a member of struct thread
Kernel entries are shared among all processes. Initially copied directly from init_page_dir.

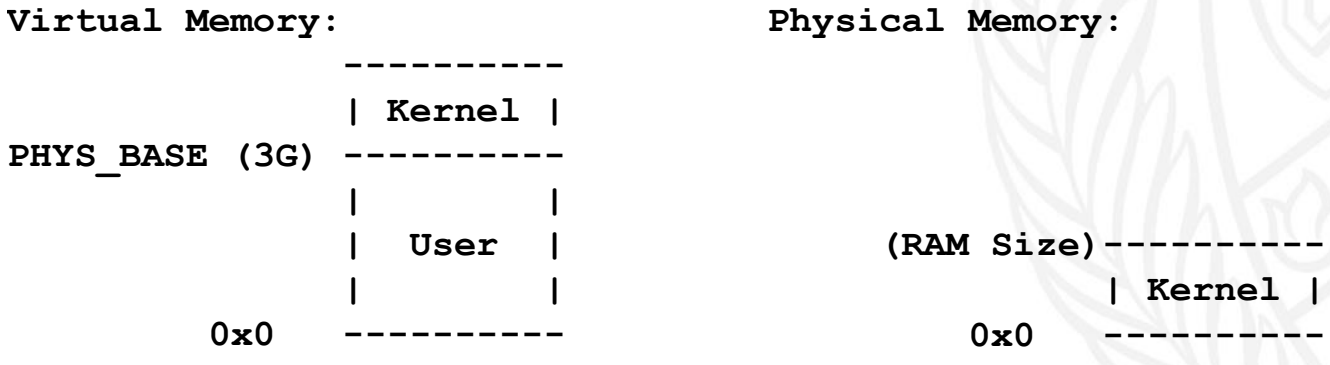
Virtual Memory in Pintos



Virtual Memory in Pintos

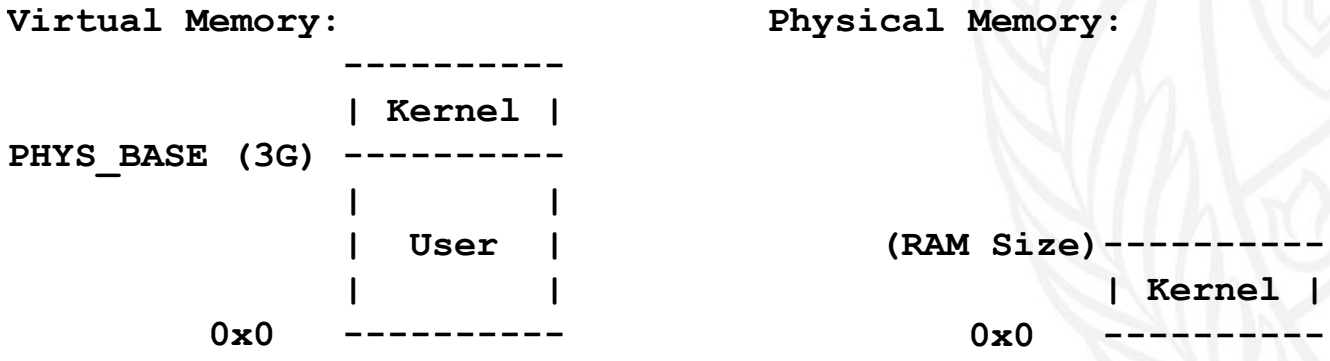
- In Pintos, this mapping is always 1-to-1 for kernel memory, that is, each virtual kernel address maps exactly to a physical address. (unlike user virtual memory!)
- Frames can be accessed through kernel virtual memory!
- The physical address is always

  the kernel virtual address - PHYS_BASE



Virtual Memory in Pintos

- Pintos, provides functions to translate between physical addresses and kernel virtual addresses. Useful later for PTE
- In Pintos, virtual memory is separated into two pools: user and kernel memory. The user memory pool extends from 0x0 to PHYS_BASE, and the kernel memory pool extends from PHYS_BASE to the end of memory.



Pintos File System

- A basic file system is already provided in Pintos
- You will need to interact with this file system
- Do not modify the file system!
- You can use semantics similar to `open`, `close`, `read`, `write`, … from the kernel
- Files to take a look at: `file.h` and `file.c`

Pintos File System

- Pintos file system limitations:
  - No internal synchronization
  - File size is fixed at creation time
  - File data is allocated as a single extent
    - i.e., in a contiguous range of sectors on “disk”
  - No subdirectories
  - File names are limited to 14 characters
  - A system crash may corrupt the disk
    - Happens a lot!
    - Create a safe copy and replace it every time using a script

Pintos File System

```
$ pintos-mkdisk filesys.dsk --filesys-size=2
```

Creates a 2 MB disk named “filesys.dsk”

DO NOT copy from PDF

```
$ pintos -f -q
```

Pintos formats the disk (-f) and exits as soon as the format is done (-q)

```
$ pintos -p ../../examples/echo -a echo -- -q
```

Puts the file “../../examples/echo” to the Pintos file system under the name “echo”

```
$ pintos -q run 'echo x'
```

Run the executable file “echo”, passing the argument “x” (from src/userprog/build)

```
$ pintos --filesys-size=2 -p ../../examples/echo -- -f -q run 'echo x'
```

- This assumes that echo is built! Initially the binary is not there and you’ll get error!

  For that you have to run `make` from /src/examples first!

Important Directories

- `src/vm/`

  Source code for the virtual memory part, which you will modify in project-3.

  We provide some stuff here!

- `src/userprog/`

  Important files such as pagedir.c, process.c and exception.c

- `src/threads/`

  Important files such as palloc.c, pte.h, vaddr.h

- `src/tests/`

  Tests for each project. You can read and modify this code to better understand your implementation. However, we will replace them with originals before we run tests.

Files of Interest

`pte.h and vaddr.h`	Getting to know paging and virtual memory
`pagedir.c and pagedir.h`	You may call some functions from here
`palloc.c and palloc.h`	Getting frames
`block.c and block.h`	You will **ignore** swapping!
`process.c and process.h`	Loads ELF binaries and starts processes
`syscall.c and syscall.h`	You will **ignore** memory mapped files!
`exception.c and exception.h`	We may need to modify how page_fault() behaves

Step 3: Design Document

Use the template in `doc/` directory:

- `threads.tmpl`
- `userprog.tmpl`
- `vm.tmpl`
- `filesys.tmpl`

- These include questions that help you with your design.

  Copy the `vm.tmpl` file for your design doc submission.

  Source: `~/pintos/doc/vm.tmpl`

  Copy to: `~/pintos/src/PA3.txt`

Step 3: Design Document

```
+---------------------------+
|           CS 140          |
| PROJECT 3: VIRTUAL MEMORY |
|        DESIGN DOCUMENT    |
+---------------------------+
```

```
---- GROUP ----

>> Fill in the names and email addresses of your group members.

FirstName LastName <email@domain.example>
FirstName LastName <email@domain.example>
FirstName LastName <email@domain.example>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.
```

Step 3: Design Document

```
                     PAGE TABLE MANAGEMENT
                     =====================

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration.  Identify the purpose of each in 25 words or less.

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for accessing the data
>> stored in the SPT about a given page.

>> A3: How does your code coordinate accessed and dirty bits between
>> kernel and user virtual addresses that alias a single frame, or
>> alternatively how do you avoid the issue?

---- SYNCHRONIZATION ----

>> A4: When two user processes both need a new frame at the same time,
>> how are races avoided?

---- RATIONALE ----

>> A5: Why did you choose the data structure(s) that you did for
>> representing virtual-to-physical mappings?
```

Step 4: Implementation

1) Managing the Supplemental Page Table
2) Managing the Frame Table
3) Handling page-fault
4) Implement Lazy Loading
5) Managing Stack growth
6) Managing Eviction **(Skipping)**
7) Managing the Swap Table **(Skipping)**
8) Managing Memory Mapped Files **(Skipping)**

Task 1: Managing the Supplemental Page Table

- The supplemental page table supplements the page table with additional data about each page. It is needed because of the limitations imposed by the page table's format. Such a data structure is often called a "page table" also; we add the word "supplemental" to reduce confusion.
- On a page fault, the kernel looks up the virtual page that faulted in the supplemental page table to find out what data should be there.
- In project 2, a page fault always indicated a bug in the kernel or a user program. In project 3, this is no longer true. Now, a page fault might only indicate that the page must be brought in from a file or swap.

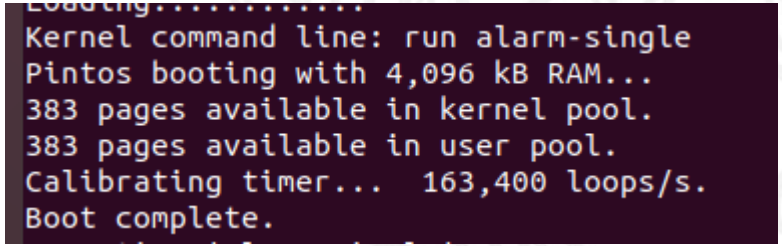
Task 1: Managing the Supplemental Page Table

- page_fault() in userprog/exception.c, needs to do roughly the following:
  1. Locate the page that faulted, in the supplemental page table.
     - If the memory reference is valid, load the page
     - Not valid? Terminate!
  2. Obtain a frame to store the page.
  3. Fetch the data into the frame, by reading it from the file system (lazy loading) or swap, or zeroing it.
  4. Point the page table entry for the faulting virtual address to the physical page. You can use the functions in "userprog/pagedir.c".

Task 2: Managing the Frame Table

- The frame table contains one entry for each frame that contains a user page.
- Each entry in the frame table contains a pointer to the page, if any, that currently occupies it, and other data of your choice.
- The frame table allows Pintos to efficiently implement an eviction policy, by choosing a page to evict when no frames are free.
- palloc_get_page(PAL_USER)
- “user pool” vs. “kernel pool”

Where do these come from?



Task 3: Handling page-fault

- A page-fault in kernel space: Still panic!

  (Unless you implemented safe memory access using page fault)

- A page-fault in user space:

  - Invalid access -> terminate the process and free resources

  + Valid but not present ->

    [Evict a page and] obtain a free frame **(You don't need eviction)**

    - Executable code not loaded -> Lazy loading from file
    - Swapped page -> Swap in **(You don't need swap)**
    - Stack growth -> zero the page
    - Memory mapped file -> load from file **(You don't need mmap)**

Task 4: Implement Lazy Loading

- What page? What offset in the file? Size of the segment?
- Your supplemental page table should retain information you need.
- Same goes with your frame table that should provide a frame from user pool.
- “process.c” and your VM files should work together.

Task 5: Managing Stack growth

- When can such a fault happen? Where can the fault be?
  - + Above esp?
  - + Below esp?
- How can you obtain the esp?

Task 6: Managing Eviction

- The process of eviction comprises roughly the following steps:
  1. Choose a frame to evict, using your page replacement algorithm. The "accessed" and "dirty" bits in the page table … will come in handy.
  2. Remove references to the frame from **any** page table that refers to it.
  3. If necessary, write the page to the file system or to swap.

- CPU sets **accessed_bit** bit and **dirty_bit** in the page table entry (PTE)
- Implementing the clock algorithm

➢ You don't need to implement swapping!

Task 7: Managing the Swap Table

- The swap table tracks in-use and free swap slots.
- It should allow picking an unused swap slot for evicting a page from its frame to the swap partition.
- It should allow freeing a swap slot when its page is read back or the process whose page was swapped is terminated.
- Using BLOCK SWAP block device.

➢ You don’t need to implement swapping!

Task 8: Managing Memory Mapped Files

- The file system is most commonly accessed with read and write system calls.
- A secondary interface is to "map" the file into virtual pages, using the mmap system call. The program can then use memory instructions directly on the file data.

```
#include <stdio.h>
#include <syscall.h>
int main (int argc UNUSED, char *argv[]) {
     void *data = (void *) 0x10000000;     /* Address at which to map. */

     int fd = open (argv[1]);              /* Open file. */
     mapid_t map = mmap (fd, data);        /* Map file. */
     write (1, data, filesize (fd));       /* Write file to console. */
     munmap (map);                         /* Unmap file (optional). */
     return 0;
}
```

➢ You don’t need to implement memory mapped files!

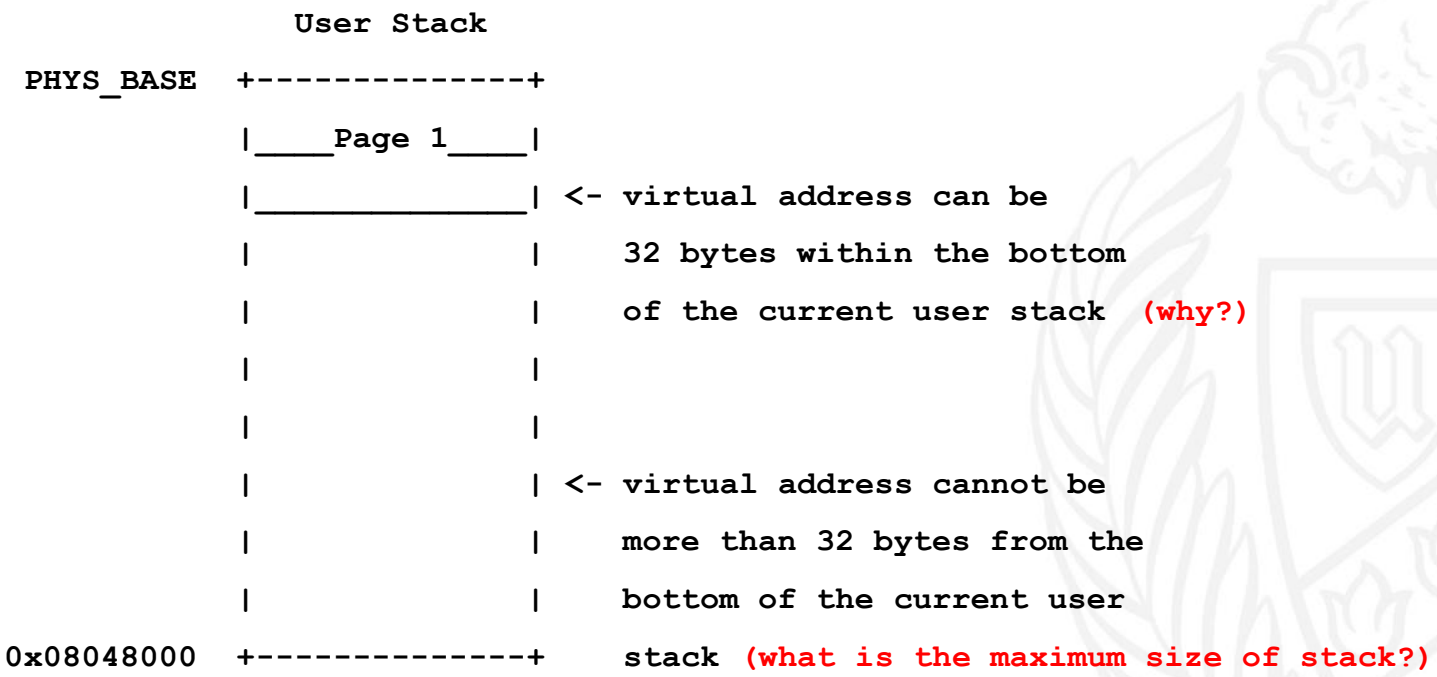
Suggested Order of Implementation

- Read, read read!!!
  - Read the pintos manual
  - Read the pintos base code
  - Read the given codes and modifications
- Brainstorm, and ask Questions
  - Work with your teammates to better understand the concepts and the code.
  - Ask questions anywhere you are not sure.
    - Do above parts first! We'll require you to have gone through them.

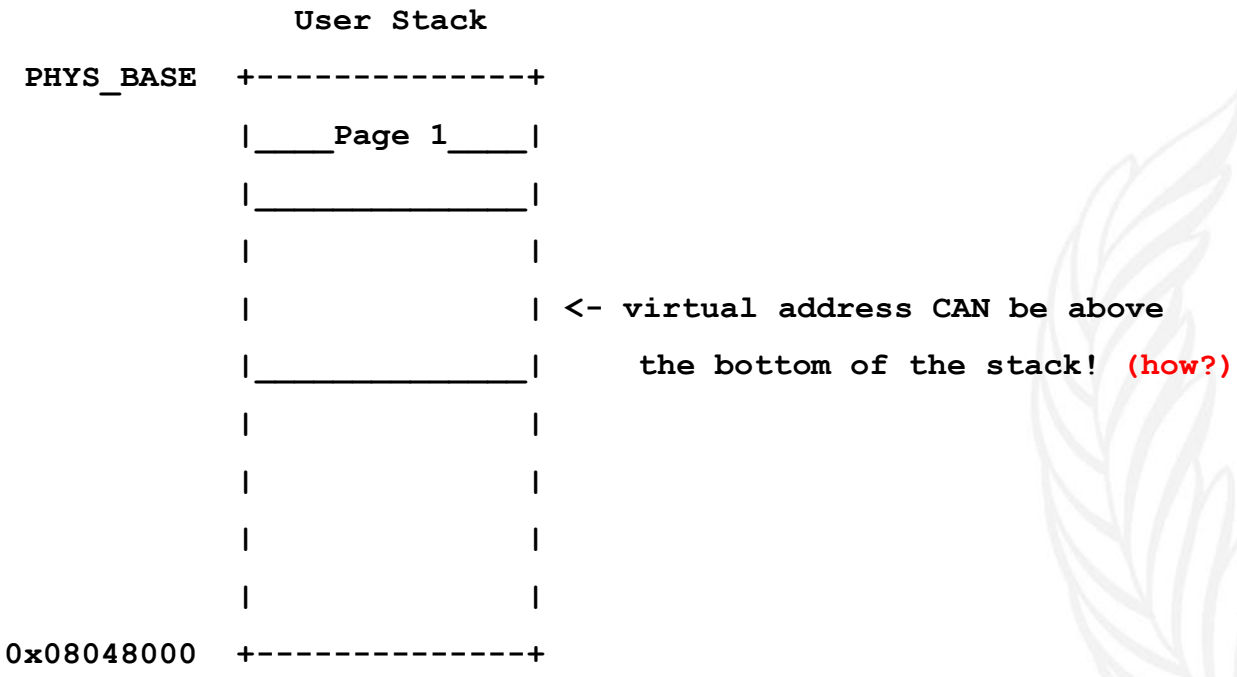
Suggested Order of Implementation

- Understand the given solution and changes in "process.c“, to use with your supplemental page table and frame table allocator
- Design your supplemental page table and frame table
- Iterate over your design, and ask questions
- Implement the missing parts, to allow lazy loading of the executable
    + During start
    + During page-fault
- Your kernel should pass all the project 2 test cases at this point
- Allow stack growth

Stack growth



Stack growth



Stack growth

src/userprog/exception.c

```
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */
  ...
}
```

You can get a new frame with palloc_get_page

You install this frame into a page table with pagedir_set_page

Debugging your Code

- printf, ASSERT, backtraces, gdb
- Running pintos under gdb
  - Invoke pintos with the gdb option (Note the spaces and hyphens).
    - `pintos --gdb -- run testname` Do not copy from PDF
  - On another terminal from build/ directory, invoke gdb
    - `pintos-gdb kernel.o`
  - Issue the command
    - `debugpintos`
  - All the usual gdb commands can be used: step, next, print, continue, break, clear, etc.
  - **Psst**… Use the pintos debugging macros described in manual (e.g. `dumplist`)

Step 5: Testing

- Pintos provides a very systematic testing suite for your project:
  - Compile:
    - `make clean`
    - `make`
  - Run all tests:
    - `make check`
  - Run individual tests
    - `make build/tests/userprog/args-none.result`
  - Run the grading script
    - `make grade`

Step 5: Testing

All tests in “PA2” are included in PA3.
But we may select only a few.

- **`make check`**

```
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
```

```
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
```

Step 5: Testing

All tests in “PA2” are included in PA3.
But we may select only a few.

- **`make check`**

```
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
FAIL tests/vm/pt-grow-stack
FAIL tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
FAIL tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
FAIL tests/vm/pt-write-code2
FAIL tests/vm/pt-grow-stk-sc
FAIL tests/vm/page-linear
```

PA3

Step 5: Testing

All tests in “PA2” are included in PA3.
But we may select only a few

- **`make check`**

```
FAIL tests/vm/page-parallel
FAIL tests/vm/page-merge-seq
FAIL tests/vm/page-merge-par
FAIL tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm
pass tests/vm/page-shuffle
FAIL tests/vm/mmap-read
FAIL tests/vm/mmap-close
pass tests/vm/mmap-unmap
FAIL tests/vm/mmap-overlap
FAIL tests/vm/mmap-twice
FAIL tests/vm/mmap-write
FAIL tests/vm/mmap-exit
FAIL tests/vm/mmap-shuffle
pass tests/vm/mmap-bad-fd
FAIL tests/vm/mmap-clean
FAIL tests/vm/mmap-inherit
FAIL tests/vm/mmap-misalign
FAIL tests/vm/mmap-null
FAIL tests/vm/mmap-over-code
FAIL tests/vm/mmap-over-data
FAIL tests/vm/mmap-over-stk
FAIL tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
26 of 113 tests failed.
```

Grading

- **`make grade`** produces a weighted grade, which is different than our expectations.
- Output is saved in `src/userprog/build/grade`:

```
TOTAL TESTING SCORE: 59.2%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                        Pts Max  % Ttl  % Max
----------------------------------------------- --- --- ------ ------
tests/vm/Rubric.functionality                    19/ 55  17.3%/ 50.0%
tests/vm/Rubric.robustness                       13/ 28   7.0%/ 15.0%
tests/userprog/Rubric.functionality             108/108  10.0%/ 10.0%
tests/userprog/Rubric.robustness                 88/ 88   5.0%/  5.0%
tests/filesys/base/Rubric                        30/ 30  20.0%/ 20.0%
----------------------------------------------- --- --- ------ ------
Total                                                    59.2%/100.0%
```

Grading

- Your grade consists of weighted average of the three phases, and progress/design.

  60% + 15% “design/progress” + 35% PA3-Quiz

  Due Dates: Quiz Apr 30th, PA3_1 May 12th

  Each group **must** meet weekly with a TA for design and progress.

  Test points are weighted. `make grade` will give you a score out of 100% for your code, based on the passed tests and their weight. You can consider that for summary of your tests. Our grading scheme is different, as our requirements are different. **Look only for required tests.**

  Check autograder submission score to be consistent with what you get on your PC.

Submission

- Submission will be via AutoLab autograder.
  - The instructions are on UB Learns. Ensure you follow them completely.
  - You'll have unlimited submissions, submit early and re-submit.
  - Every phase needs registration, group formation, submission etc.

    It is the responsibility of all members to ensure this.

- Due days and times are Fridays 11:59 PM Eastern Time

- Refer to LATE SUBMISSION policy.

Assignments

- Finalize your groups on github classroom and autograder
- Finalize your PA3 TA soon
- You don’t need to wait for anything else, you can get started with existing code
- Start working on your PA-3 design immediately!

Summary

- Pintos projects:
  - Project-1: Threads
  - Project-2: User Programs
  - Project-3: Virtual Memory
    - Step 1: Preparation
    - Step 2: Understanding Pintos
    - Step 3: Design Document
    - Step 4: Implementation
    - Step 5: Testing
  - Project-4: File Systems (Not part of our course)

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from University of Nevada, Reno
- T. Kosar and K. Dantu from University at Buffalo
- Pintos Manual
- Pintos Notes and Slides by A. He (Stanford), A. Romano (Stanford), J. Sundararaman & X. Liu(Virginia Tech), J. Kim (SKKU), S. Chandrashekhara (UB), B. Cheng (USC), S. Tsung-Han Sher (USC)