CSE 421/521
Introduction to Operating Systems

Farshad Ghanei

Lecture - 15
Project-2 Discussion

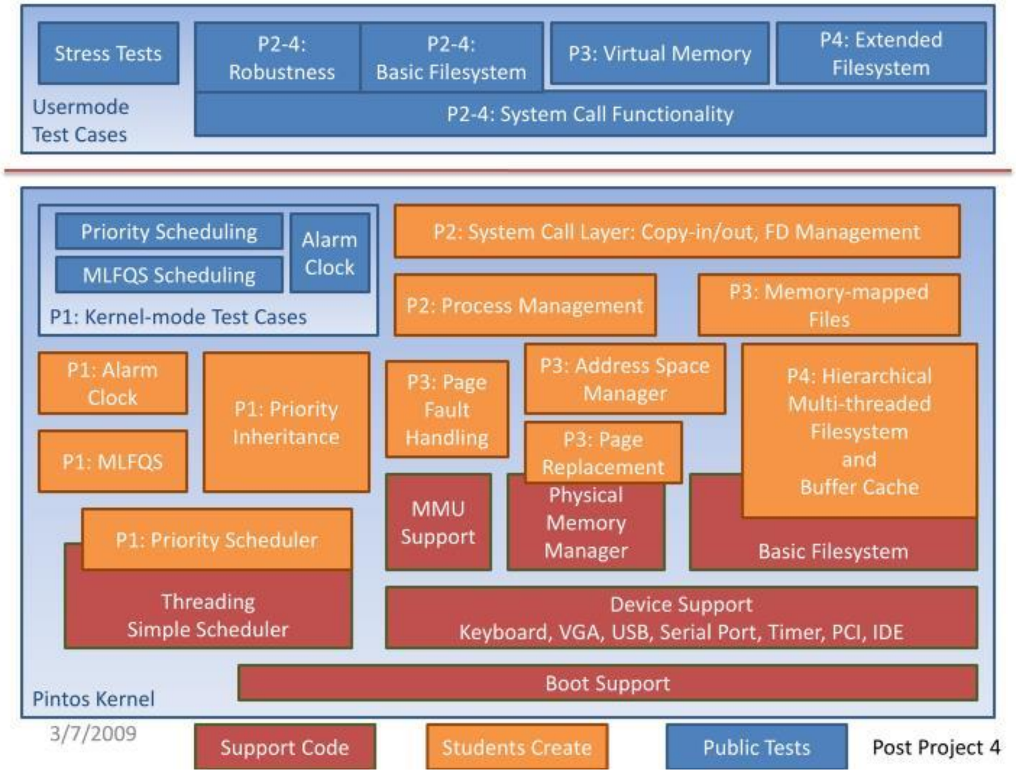
* Slides adopted from Prof Kosar and Dantu at UB, “Operating System Concepts” book and supplementary material by A. Silberschatz, P.B. Galvin, and G. Gagne. Wiley Publishers

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

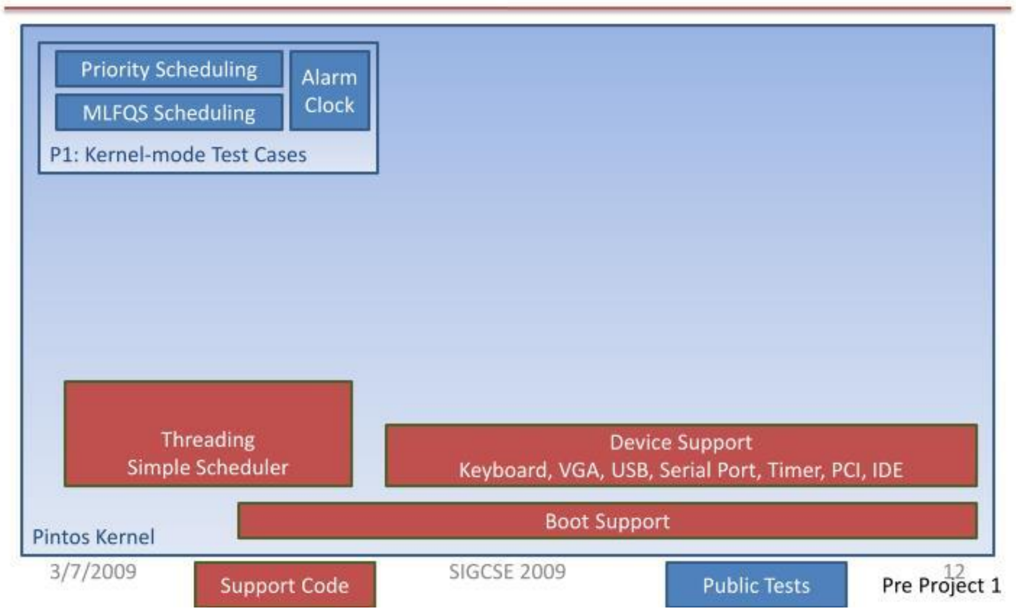
Today

- Pintos projects:
  - Project-1: Threads
  - Project-2: User Programs
    - Step 1: Preparation
    - Step 2: Understanding Pintos
    - Step 3: Design Document
    - Step 4: Implementation
    - Step 5: Testing
  - Project-3: Virtual Memory
  - Project-4: File Systems

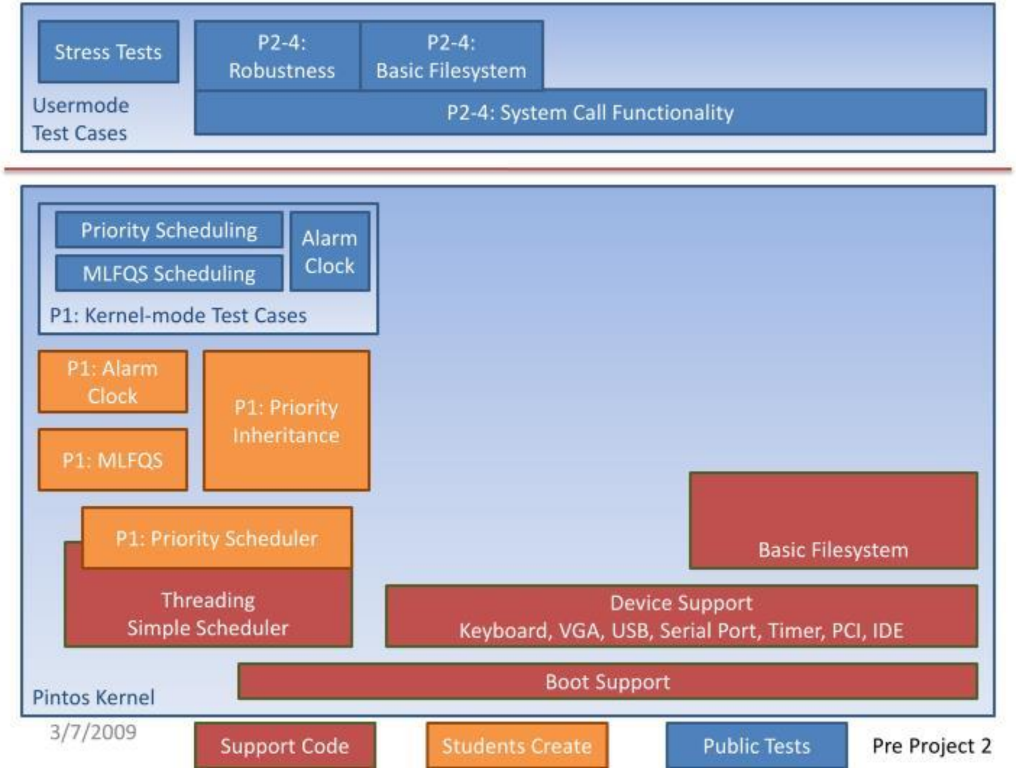
Pintos - After full implementation (Post Project 4)



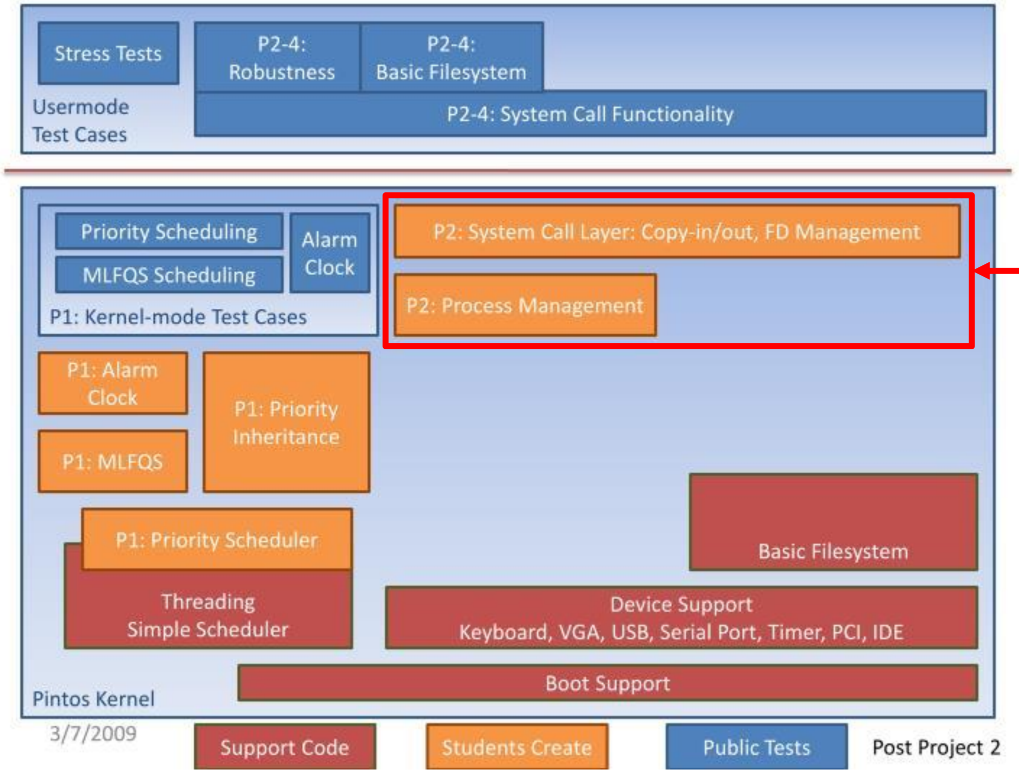
Pintos - You Started from Here (Pre Project 1)



Pintos - You Have Reached Here (Pre Project 2)



Pintos - You Will Implement This (Post Project 2)



Project-2: User Programs

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

Step 1: Preparation

Readings from zyBooks:

- Chapters 2, 6, 7

Readings from Pintos documentation:

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

Step 1: Preparation

- Project-2 does not depend on Project-1
- We will not continue with your Project-1 code
- We will start a new repo for Project-2
- 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

- How user programs run in general
- Distinctions between user and kernel virtual memory
- The system call infrastructure / file system interface

A Simple C Program

```
main() {
        int n = 10;
        int fact = 1;
        for (int i = 1; i <= 10; i++) {
                fact = fact * i;
        }
        printf("%d", fact);
}
```

A Simple C Program with Arguments

```
int main(int argc, char* argv[]) {
        for (int i = 0; i < argc; i++) {
                char* arg = argv[i];
        printf("%s", arg);
        }
}



./test arg1 arg2 ...
```

A Simple C Program with System Calls

```
int main() {
        FILE* p_file = fopen("myfile.txt","w");
        if (p_file != NULL) {
        fputs("Hello", p_file);
        fclose(p_file);
    }
}
```

Example

What happens when a user wants to run the following program in shell?

```
user:~$ cp -r /usr/bin/temp .
```

1) shell parses user input
2) shell calls fork() and execve(“cp”, argv, env)
3) cp parses the arguments
4) cp uses file system interface to copy files
5) cp may print messages to STDOUT
6) cp exits

Pintos User Programs

`threads/init.c`

- main() → run_actions(argv) after booting
- run _actions() → run_task(argv)
  - The task to run is argv[1]
- run_task() → process_wait(process_execute(task))

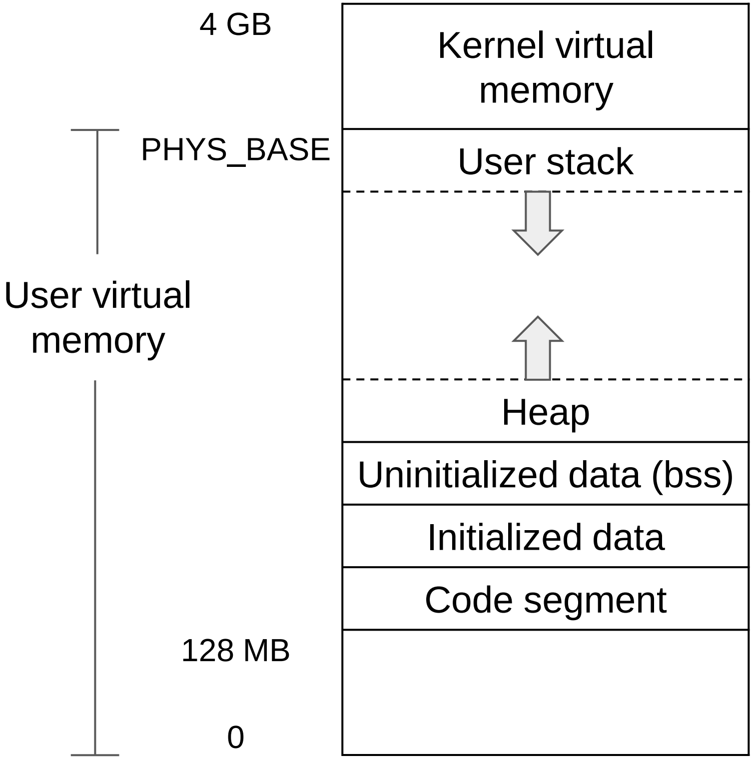
`userprog/process.c`

- process_execute() creates a thread that runs start_process(filename…)
- start_process() → load(filename…)
- load() sets up the stack, data, and code, as well as the start address

➢ You need to modify some of these to set up stack correctly and wait for the process!

Pintos Memory Layout

- Virtual memory is divided between user and kernel virtual memory
- Each user process has its own mapping of user virtual addresses (the page directory)
- User processes are not allowed to access kernel memory or unmapped user virtual addresses
  - This causes **page faults**
- The kernel can also page fault if it accesses an unmapped user virtual address



Get Familiar with the Code

- The first task is to read and understand the code
  - Under the `src/userprog/` directory
- Pintos already implements loading and running user programs, but argument passing and interactivity with OS (I/O) is not implemented.
- For a brief overview of the files in the `src/userprog/` directory, please see **Section 3.1.1 Source Files** in the Pintos documentation.

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

DO NOT copy from PDF

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

Creates a 2 MB disk named “filesys.dsk”

```
$ 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/userprog/`

  Source code for the user program part, which you will modify in project-2.

- `src/filesys/`

  Source code for file system interface.

- `src/examples/`

  User programs that you can use or modify to run on your pintos.

- `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

- `thread.c` and `thread.h`

  Threads need to be modified to allow parent/child access, and file descriptors.

- `process.c` and `process.h`

  Loads ELF binaries and starts processes

- `syscall.c` and `syscall.h`

  Bulk of work in project-2, you have to implement syscall handler skeleton and syscalls

- `exception.c` and `exception.h`

  We may need to modify how page_fault() behaves

- `pagedir.c` and `pagedir.h`

  You may call some functions from the page table

Step 3: Design Document

Use the template in `doc/` directory:

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

- Copy the `userprog.tmpl` file for your design doc submission.

Step 3: Design Document

```
+--------------------------+
|          CS 140          |
| PROJECT 2: USER PROGRAMS |
|     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

```
                        ARGUMENT PASSING
                        ================

---- 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: Briefly describe how you implemented argument parsing.  How do
>> you arrange for the elements of argv[] to be in the right order?
>> How do you avoid overflowing the stack page?

---- RATIONALE ----

>> A3: Why does Pintos implement strtok_r() but not strtok()?

>> A4: In Pintos, the kernel separates commands into a executable name
>> and arguments.  In Unix-like systems, the shell does this
>> separation.  Identify at least two advantages of the Unix approach.
```

Step 4: Implementation

Phase-1

1. Argument passing
   - Passing command-line arguments to programs

Phase-2

1. System call infrastructure
2. “Write” system call to STDOUT and “exit” system call
3. Process termination messages
4. Safe memory access

Phase-3

1. A set of 13 system calls
   - List in **Section 3.3.4 System Calls** in the Pintos documentation.
2. Denying writes to executables

Task 1: Implement Argument Passing

- Before a user program starts executing, the kernel must push the function's arguments onto the stack.
- This involves breaking the command-line input into individual words.
- Consider "`/bin/ls -l foo bar`"

  `-> "bin/ls", "-l", "foo", "bar"`
- Implement the string parsing however you like in process_execute()

  You can use strtok_r() in lib/string.c
- Feel free to define new functions, or modify the existing functions if they need new arguments, etc.

Task 1: Implement Argument Passing

- Pintos currently lacks argument passing, program tries to access its argument in kernel space, and crashes. Change *esp = PHYS_BASE to *esp = PHYS_BASE - 12 in setup_stack to get started (why?) This is temporary (why?)

- Pintos kernel calls `process_wait()` on the user process, currently:

```
int process_wait (tid_t child_tid UNUSED)
{
   return -1;
}
```

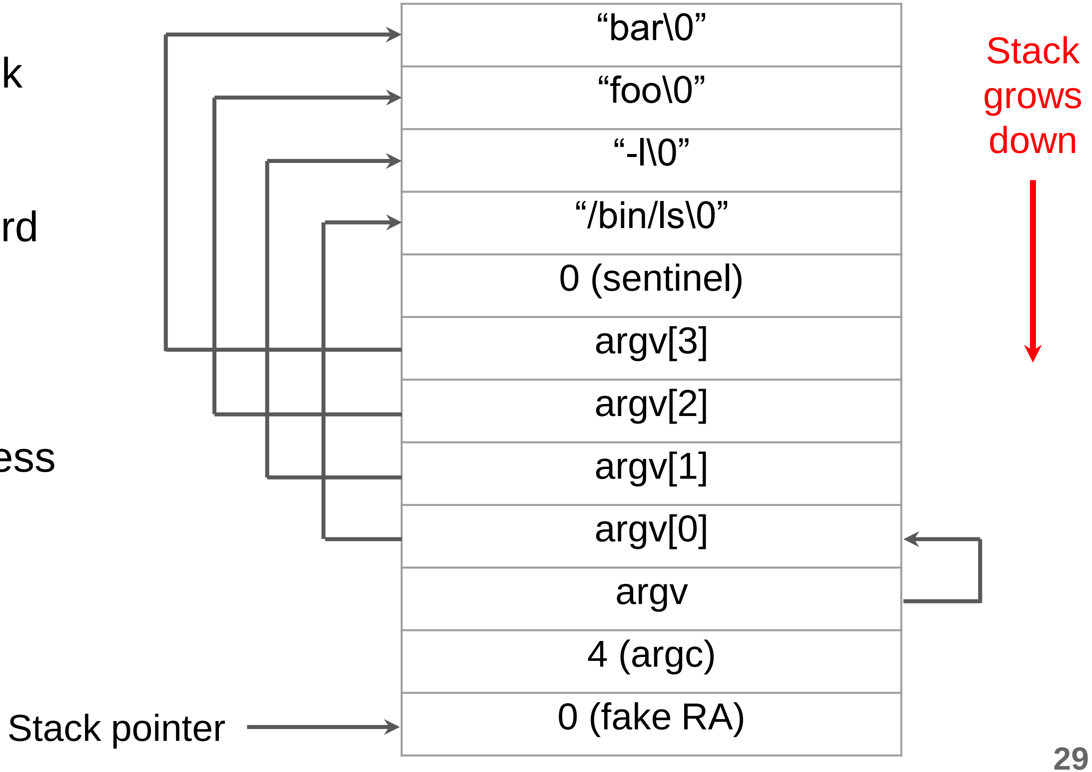
- For now, change it to a infinite/finite loop (why?)

Task 1: Implement Argument Passing

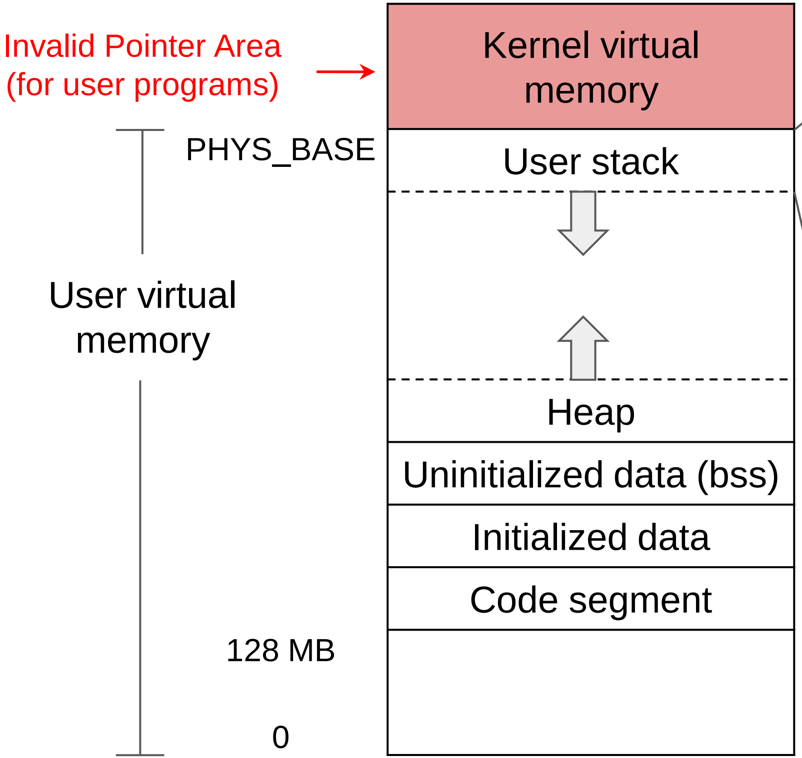
```
/bin/ls -l foo bar
```

- Push the words onto the stack
- Push a null pointer sentinel
- Push the address of each word in right-to-left order
- Push argv and argc
- Push 0 as a fake return address

- hex_dump() in <stdio.h> is your friend!



Task 1: Implement Argument Passing



PHYS_BASE = 3 GB = 0xc0000000

Address	Name	Data	Type
0xbffffffc	argv[3][...]	'bar\0'	char[4]
0xbffffff8	argv[2][...]	'foo\0'	char[4]
0xbffffff5	argv[1][...]	'-l\0'	char[3]
0xbfffffed	argv[0][...]	'/bin/ls\0'	char[8]
0xbfffffec	word_align	0	uint8_t
0xbfffffe8	argv[4]	0	char*
0xbfffffe4	argv[3]		char*
0xbfffffe0	argv[2]		char*
0xbfffffdc	argv[1]		char*
0xbfffffd8	argv[0]		char*
0xbfffffd4	argv		char**
0xbfffffd0	argc	4	int
0xbfffffcc	Return address	0	void (*)()

Task 1: Implement Argument Passing

Command:
/bin/ls -l foo bar

PHYS_BASE=0xc0000000	Kernel space					
0xbffffffc	'b'	'a'	'r'	00		
0xbffffff8	'f'	'o'	'o'	00		
0xbffffff4	00	'-'	'l'	00		
0xbffffff0	'n'	'/'	'l'	's'		
0xbfffffec	00	'/'	'b'	'i'		
0xbfffffe8	00	00	00	00	0	argv[4]
0xbfffffe4	fc	ff	ff	bf	0xbffffffc	argv[3]
0xbfffffe0	f8	ff	ff	bf	0xbffffff8	argv[2]
0xbfffffdc	f5	ff	ff	bf	0xbffffff5	argv[1]
0xbfffffd8	ed	ff	ff	bf	0xbfffffed	argv[0]
0xbfffffd4	d8	ff	ff	bf	0xbfffffd8	argv
0xbfffffd0	04	00	00	00	4	argc
0xbfffffcc	return address					

Padding for Word alignment

Reverse byte order due to Little Endianness

Task 1: Implement Argument Passing

hex_dump() in <stdio.h> is your friend!

Reverse byte order due to Little Endianness

```
bffffc0                                      00 00 00 00 |            ....|
bffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 00 2f 62 69 |............./bi|
bfffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

Task 2: Implement Safe Memory Access

- The kernel will often access memory through user-provided pointers
- These pointers can be problematic:
  - null pointers, pointers to unmapped user virtual memory, or pointers to kernel addresses
  - If a user process passes these to the kernel, you must kill the process and free its resources (locks, allocated memory, etc.)
- Be aware of potential problems with pointers, buffers and strings.

Task 2: Implement Safe Memory Access

Two approaches for solving this problem:

- Verify every user pointer before dereferencing (simpler)
  - Is it in the user's address space, i.e. below PHYS_BASE?
    - is_user_vaddr() in threads/vaddr.h
  - Is it mapped or unmapped?
    - page_dir_get_page() in userprog/pagedir.c
  - These checks apply to buffers as well!
- Modify page fault handler in userprog/exception.c
  - Only check that a user pointer/buffer is below PHYS_BASE
  - Invalid pointers will trigger a page fault, which can be handled correctly
  - **Section 3.1.5 Accessing User Memory**

Task 3: System Call Infrastructure

- Implement syscall_handler() in syscall.c
- What does this involve?
  - Read syscall number at stack pointer
    - SP is accessible as “esp” member of the `struct intr_frame *f` passed to syscall_handler()
  - Read some number of arguments above the stack pointer, depending on the syscall
  - Dispatch to the desired function
    - First implement placeholder skeletons only.
  - Return the value to the user in `f->eax`

Task 4: System Calls

- Syscall numbers are defined in lib/syscall-nr.h
    - You will not implement all the calls. See **Section 3.3.4 System Calls**
- Don’t get confused by the “user” side of system calls, in lib/user/syscall.c
- Many of the syscalls involve file system functionality
    - Use the mentioned pintos file system
    - Use coarse synchronization to ensure that any file system code is a critical section
    - Syscalls take “file descriptors” as arguments, but the Pintos file system uses `struct file *s`
        - You must design a proper mapping scheme

Task 4: System Calls

- Reading from keyboard and writing to the console are special cases with special file descriptors
  - “write” syscall with fd = STDOUT_FILENO
    - Use putbuf(....) or putchar(...) in lib/kernel/console.c
  - “read” syscall with fd = STDIN_FILENO
    - Use input_getc(...) in devices/input.h

Task 4: System Calls - Related to Processes

- System calls related to processes:

```
void    exit(int status);
pid_t   exec(const char *cmd_line);
int     wait(pid_t pid)
```

- All of a process’s resources must be freed on exit()
- The child can exit() before the parent performs wait() (consider other scenarios too!)
- A process can perform wait() only for its children
- wait() can be called twice for the same process
  - The second wait() should fail
- Nested wait() is possible: A → B , B → C (consider other scenarios too!)
- Pintos should not terminate until the initial process exits

Task 4: System Calls - Related to Processes

- `int wait(pid_t pid)`
  - Caller blocks until child process corresponding to pid exits.
    - Use synchronization primitives rather than thread_block()
  - Returns the exit status of the child or -1 in certain cases
    - If wait has already been successfully called on the child
    - If pid does not reference a child of the caller
  - Many cases to think about: multiple waits, nested waits, etc.
  - Suggestion: Implement process_wait(), then wait() on top of process_wait()
  - Involves the most work of all the syscalls

Task 4: System Calls - Related to Processes

- `void exit(int status);`
    - Terminate user program, return status to the kernel.
    - Print a termination message
    - If a child is exiting, communicate exit status back to the parent who called [or may or may not call] wait()

Task 4: System Calls - Related to Processes

- `pid_t exec(const char *cmd_line);`
  - Behaves like unix's fork() + execve(...)
  - Creates a child process running the program in cmd_line
  - Must not return until the new process has successfully loaded or failed
    - Requires synchronization to ensure this.

Task 4: System Calls - Related to Files

- System calls related to files:

```
bool     create(const char *file, unsigned initial_size);
bool     remove(const char *file);
int      open(const char *file);
int      filesize(int fd);
int      read(int fd, void *buffer, unsigned size);
int      write(int fd, void *buffer, unsigned size);
void     seek(int fd, unsigned position);
unsigned    tell(int fd);
void     close(int fd);
```

- create(), remove(), open() work on file names
- The rest work on file descriptors

Task 4: System Calls - Related to Files

- No need to change the code in the filesys directory
- The existing routines in the filesys directory work on the “file” structure

```
struct file *
```

- Maintain a mapping structure from a file descriptor to the corresponding “file” struct
- Different processes must stay independent
- Deny writes to a running process’s executable file
- Ensure only one process at a time is executing the file system code
    - Use coarse synchronization

Task 5: Process Termination Messages

- Whenever a user process terminates, because it called exit or for any other reason, print the process's name and exit code
- Formatted as if printed by printf ("%s: exit(%d)\n", ...);
- The name printed should be the full name passed to process_execute(), omitting command-line arguments. Do not print these messages when a kernel thread that is not a user process terminates, or when the halt system call is invoked. The message is optional when a process fails to load.
- Aside from this, don't print any other messages that Pintos as provided doesn't already print. You may find extra messages useful during debugging, but they will confuse the grading scripts and thus lower your score.

Task 6: Denying Writes to Executables

- Pintos should not allow code that is currently running to be modified
  - Use file_deny_write() to prevent writes to an open file
  - Note: closing a file will re-enable writes, so an executable file must be kept open as long as the process is running

Suggested Order of Implementation

Phase-1

- Argument passing
  - Initially change *esp = PHYS_BASE; in setup_stack() to

    *esp = PHYS_BASE - 12; (why?) This is temporary (why?)
    - This will allow execution of programs with no arguments
  - Initially change process_wait() to an infinite or long enough loop (why?)
    - This will allow the program to run, before the Pintos powers off
  - Implement argument passing so the stack is set up as expected.

Suggested Order of Implementation

Phase-2

- System call infrastructure
  - Read syscall number, dispatch to dummy function
- The exit system call + exit message.
- The write system call to STDOUT_FILENO
  - No Pintos tests will pass until you can functionally write to the console and exit!

Phase-3

- Implement rest of the project (all other system calls and required functionalities)
- First implement functionality, then improve robustness by passing robustness tests.

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.
  - Use the pintos debugging macros described in manual (e.g. `dumplist`)

How Much Code?

```
threads/thread.c        |   13
threads/thread.h        |   26 +
userprog/exception.c    |   8
userprog/process.c      |   247 ++++++++++++++--
userprog/syscall.c      |   468 ++++++++++++++++++++++++++++++-
userprog/syscall.h      |   1
6 files changed, 725 insertions(+), 38 deletions(-)
```

- This reference solution represents just one possible solution.

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`
- We provide testing tools for Phase-1 separately

Step 5: Testing

- **make check**

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

Phase-2

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

Step 5: Testing

- **make check**

```
FAIL tests/userprog/write-bad-ptr
FAIL tests/userprog/write-boundary
FAIL tests/userprog/write-zero
FAIL tests/userprog/write-stdin
FAIL tests/userprog/write-bad-fd
FAIL tests/userprog/exec-once
FAIL tests/userprog/exec-arg
FAIL tests/userprog/exec-bound
FAIL tests/userprog/exec-bound-2
FAIL tests/userprog/exec-bound-3
FAIL tests/userprog/exec-multiple
FAIL tests/userprog/exec-missing
FAIL tests/userprog/exec-bad-ptr
FAIL tests/userprog/wait-simple
FAIL tests/userprog/wait-twice
FAIL tests/userprog/wait-killed
FAIL tests/userprog/wait-bad-pid
FAIL tests/userprog/multi-recurse
FAIL tests/userprog/multi-child-fd
```

```
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
FAIL tests/userprog/wait-simple
FAIL tests/userprog/wait-twice
FAIL tests/userprog/wait-killed
FAIL tests/userprog/wait-bad-pid
FAIL tests/userprog/multi-recurse
FAIL tests/userprog/multi-child-fd
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
FAIL tests/userprog/bad-read
FAIL tests/userprog/bad-write
FAIL tests/userprog/bad-read2
FAIL tests/userprog/bad-write2
FAIL tests/userprog/bad-jump
FAIL tests/userprog/bad-jump2
FAIL tests/userprog/no-vm/multi-oom
```

Phase-2

Step 5: Testing

- **make check**

```
FAIL tests/userprog/no-vm/multi-oom
FAIL tests/filesys/base/lg-create
FAIL tests/filesys/base/lg-full
FAIL tests/filesys/base/lg-random
FAIL tests/filesys/base/lg-seq-block
FAIL tests/filesys/base/lg-seq-random
FAIL tests/filesys/base/sm-create
FAIL tests/filesys/base/sm-full
FAIL tests/filesys/base/sm-random
FAIL tests/filesys/base/sm-seq-block
FAIL tests/filesys/base/sm-seq-random
FAIL tests/filesys/base/syn-read
FAIL tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
80 of 80 tests failed.
```

All tests in “make grade” are included in Phase-3.

warning: test tests/userprog/sc-boundary-3 doesn't count for grading
warning: test tests/userprog/exec-bound-3 doesn't count for grading
warning: test tests/userprog/exec-bound-2 doesn't count for grading
warning: test tests/userprog/exec-bound doesn't count for grading

Step 5: Testing - Functionality

```
/* Prints the command-line arguments. This program is used for all of the args-* tests.
   Grading is done differently for each of the args-* tests based on the output. */

#include "tests/lib.h"

int
main (int argc, char *argv[])
{
  int i;

  test_name = "args";

  msg ("begin");
  msg ("argc = %d", argc);
  for (i = 0; i <= argc; i++)
    if (argv[i] != NULL)
      msg ("argv[%d] = '%s'", i, argv[i]);
    else
      msg ("argv[%d] = null", i);
  msg ("end");

  return 0;
}
```

Expected output for ‘args 1 2’:

```
begin
argc=3
argv[0] = ‘args’
argv[1] = ‘1’
argv[2] = ‘2’
argv[3] = null
end
```

Step 5: Testing - Robustness

```
/* This program attempts to read memory at an address that is not mapped.
   This should terminate the process with a -1 exit code. */


#include "tests/lib.h"
#include "tests/main.h"

int test_main (void)
{
  msg ("Congratulations - you have successfully dereferenced NULL: %d",
        *(volatile int *) NULL);
  fail ("should have exited with -1");
}
```

Expected output:

```
bad-read: exit(-1)
```

Grading

- **make grade**
- src/userprog/build/grade:

```
TOTAL TESTING SCORE: 0.0%
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                      Pts Max  % Ttl  % Max
--------------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality             0/108   0.0%/ 35.0%
tests/userprog/Rubric.robustness                0/ 88   0.0%/ 25.0%
tests/userprog/no-vm/Rubric                     0/  1   0.0%/ 10.0%
tests/filesys/base/Rubric                       0/ 30   0.0%/ 30.0%
--------------------------------------------- --- --- ------ ------
Total                                                   0.0%/100.0%
```

Grading

- Your "source code" consists of weighted average of the three phases.

  (10%, 20%, 70% and Due dates: Oct 29th, Nov 12th, Dec 3rd)
- Your "design document" is 10% of your Project-2 grade, and "source code" is 90%

- The 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 for phase-1, phase-2, phase-3 are different.
- Check autograder submission score to be consistent with what you get on your VM.

Submission

- Submission will be via AutoLab autograder.
  - The instructions will be posted on Piazza.
  - You'll have unlimited submissions, submit early and re-submit.
  - Every phase needs registration, group formation, submission etc. It is the responsibility of both members to ensure this.

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

- Refer to LATE SUBMISSION policy.

Assignments

- Finalize your groups for PA-2 on github classroom and get repos
- Get started as soon as possible towards phase-1
- Read through the Pintos manual Section 3, Appendix A5, A6 and E

- Reading: Chapter 7 (Virtual Memory) from zyBooks

Summary

- Pintos projects:
  - Project-1: Threads
  - Project-2: User Programs
    - Step 1: Preparation
    - Step 2: Understanding Pintos
    - Step 3: Design Document
    - Step 4: Implementation
    - Step 5: Testing
  - Project-3: Virtual Memory
  - Project-4: File Systems

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)