CSE 421/521 - Operating Systems
Fall 2019

LECTURE - XVI

# PROJECT-2 DISCUSSION

Tevfik Koşar

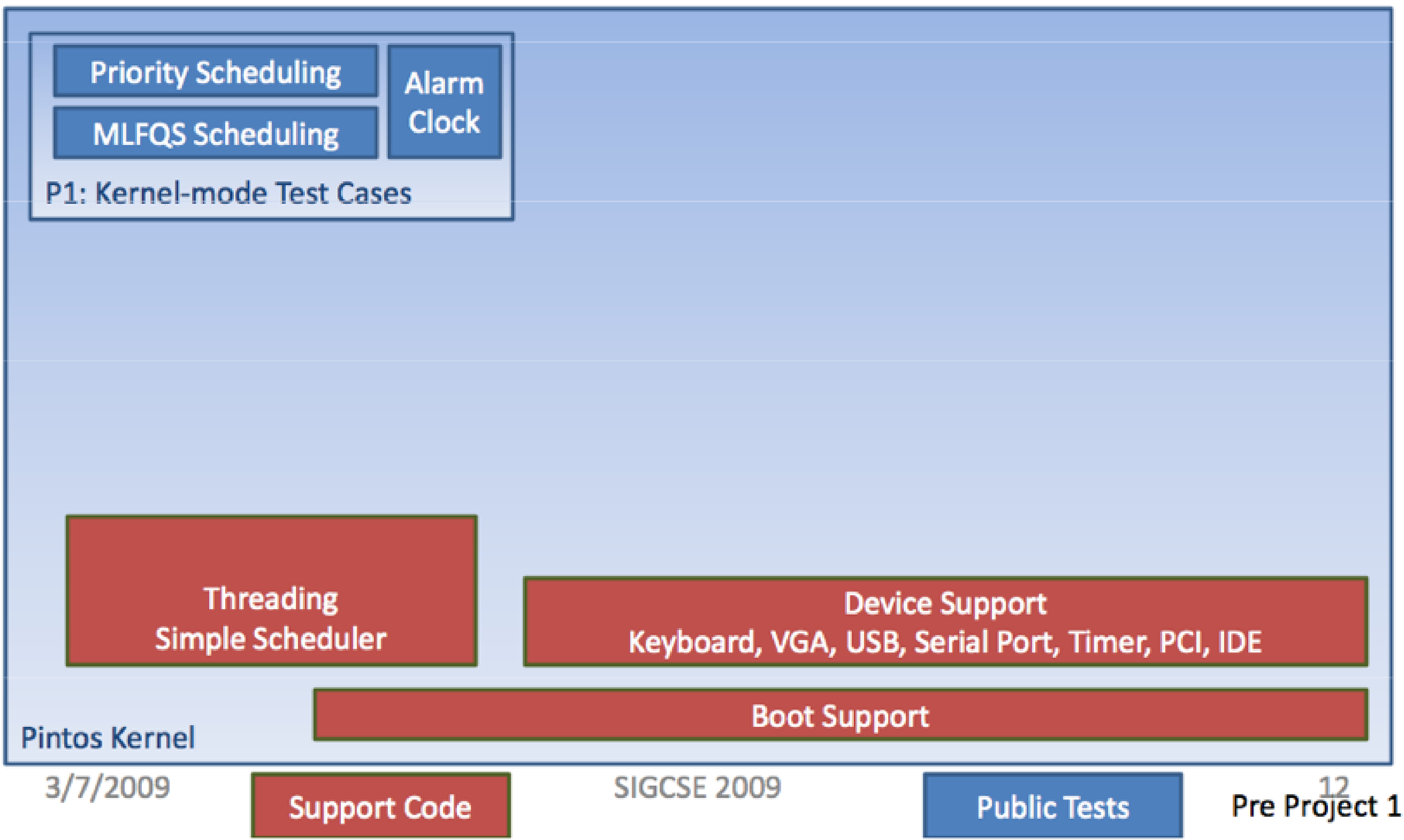
University at Buffalo

October 22nd, 2019

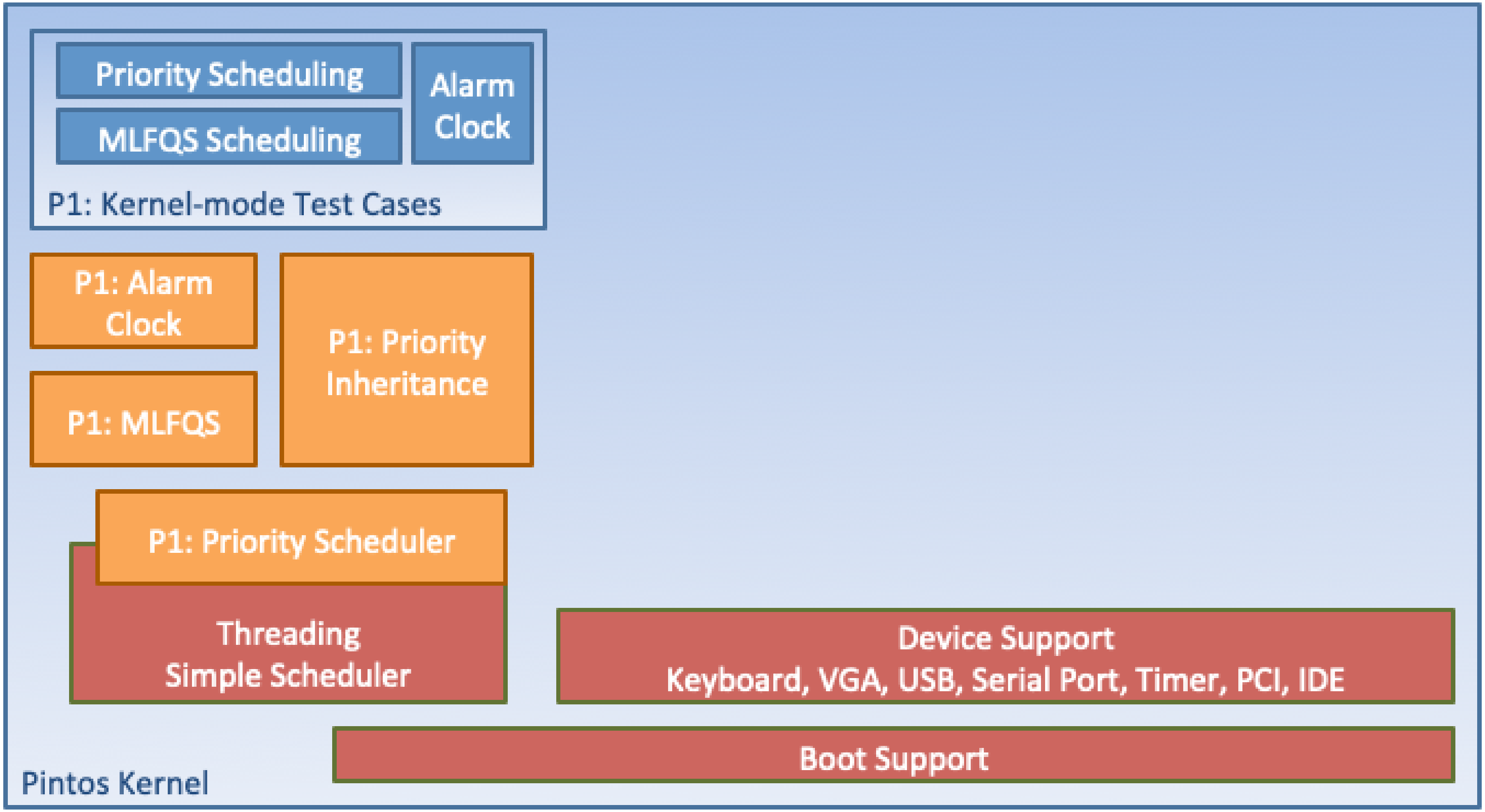
# Pintos Projects

1. Threads
2. User Programs <-- CSE 421/521 Project 2
3. Virtual Memory
4. File Systems

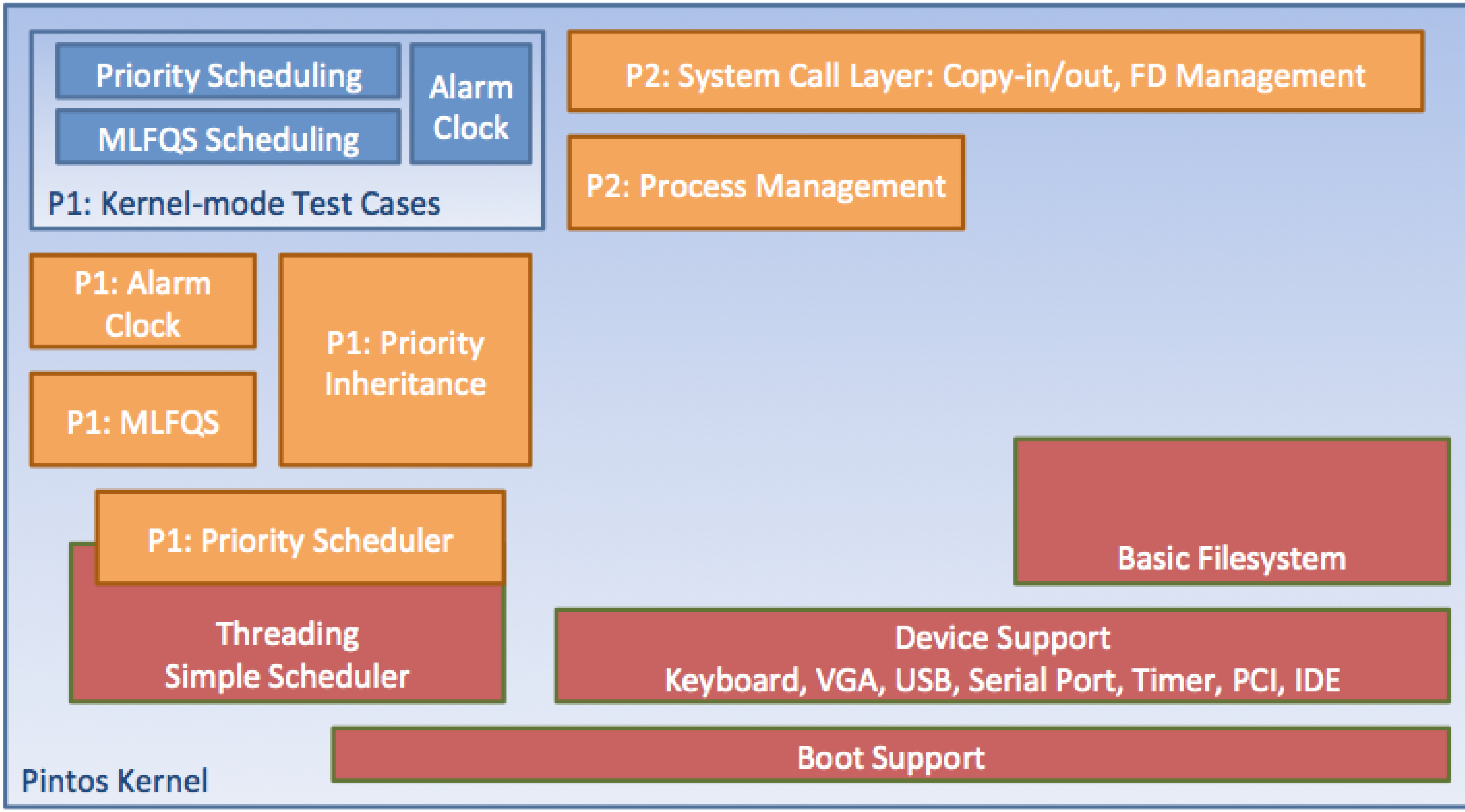
# Yo were provided with this (pre prj-1)



# You implemented this (post prj-1)



# You will now implement this (post prj-2)



# Overview (1)

- The existing Pintos code already supports loading and running user programs, but no argument passing and no interactivity (I/O) is possible.
- In this project, you will enable programs to support arguments and interact with the OS via system calls.

# Overview (2)

- This project requires an understanding of:
  - How user programs run in general
  - Distinctions between user and kernel virtual memory
  - The system call infrastructure / file system interface

# Example (1)

- Sample C program without arguments and systems calls:

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

# Example (2)

- In C, a user program test.c can pass argument
  - int main(int argc, char* argv[])
```
{
    for(int i=0; i<argc; i++)
    {
        char* arg = argv[i];
    }
}
```
  - ./test arg1 arg2 ...

# Example (3)

- test.c can call system libraries
  - ```c
    #include <stdio.h>
    int main()
    {
        FILE* p_file = fopen("myfile.txt","w");
        if (p_file != NULL) fputs("fopen", p_file);
        fclose(p_file);
    }
    ```
  - Get fopen, fputs, fclose by system calls
- Pintos need you to implement
  - Argument passing
  - System calls
    ```

# Example (4)

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

```
myth13:~$ cp -r /usr/bin/temp .
```

1) Shell parses user input
2) Shell calls fork() and execve("cp", argv, env)
3) cp uses file system interface to copy files
4) cp may print messages to stdout
5) cp exits

- These interactions require system calls

# Project Goals

You will need to implement the following:

- Passing command-line arguments to programs
- Safe memory access
- A set of system calls
  - Long list, in section 3.3.4 of the Pintos docs
- Process termination messages
- Denying writes to files in use as executables

# File System Access in Pintos (1)

- A basic file systems is already provided in Pintos
- You will need to interact with this file system
- But, do not modify the file system!
- UNIX like semantics: open, close, read, write…
- Files to take a look at: ‘*filesys.h*’ & ‘*file.h*’

# File System Access in Pintos (2)

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

# File System Access in Pintos (2)

- **Using the Pintos file system**

  `$ pintos-mkdisk fs.dsk 2`

  – Creates a 2MB disk named "`fs.dsk`"

  `$ pintos -f -q`

  – Formats the disk (`-f`) and exits as soon as the format is done (`-q`)

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

  – Put the file "`../../examples/echo`" to the Pintos file system under the name "echo"

  `$ pintos -q run 'echo x'`

  – Run the executable file "echo", passing argument "x"

  `$ pintos -fs-disk=2 -p ../../examples/echo -a echo -- -f -q run 'echo x'`

# Step 1: Preparation

READ:

- Sections 2.3 - 2.4 from Silberschatz
  - Systems Calls

- Chapter 8-9 from Silberschatz
  - Memory & Virtual Memory Management

Lecture slides on Memory & Virtual Memory Management

- Lectures 14 & 15

From Pintos Documentation:

- Section 3: User Programs

# Step 2: Setting Up Pintos (1)

- **Project-2 does not depend on project-1**

**No code from project-1 is required for this assignment**

You can use your existing code base if you wish,

Or, if you think you have totally messed up with project-1, you can grab a fresh copy of the Pintos code base, following the instructions on the project description.

# Step 2: Setting Up Pintos (2)

- Use the Pintos **VM** we have prepared for you:

  http://ftp.cse.buffalo.edu/CSE421/UB-pintos.ova

- It requires **Virtualbox** software

  ==> will work on most Linux, Windows, Mac systems

  https://www.virtualbox.org/wiki/Downloads

- Detailed setup instructions are available on Piazza.

# Step 3: Implementation

1. Argument Passing
2. User Memory Access
3. System Call Infrastructure
4. Implement 13 system calls (Start with Exit and Write)
5. Denying Writes to Executables

# Task 0: Get Familiar with the Code

- The first task is to read and understand the code for the initial userprog system (under the "pintos/src/userprog/" directory).

- For a brief overview of the files in the "userprog/" directory, please see "Section 3.1.1. Source Files" in the Pintos Reference Guide

- You will be mostly working on:
  - /userprog/syscall.c
  - /userprog/process.c

# Task 1: Argument Passing

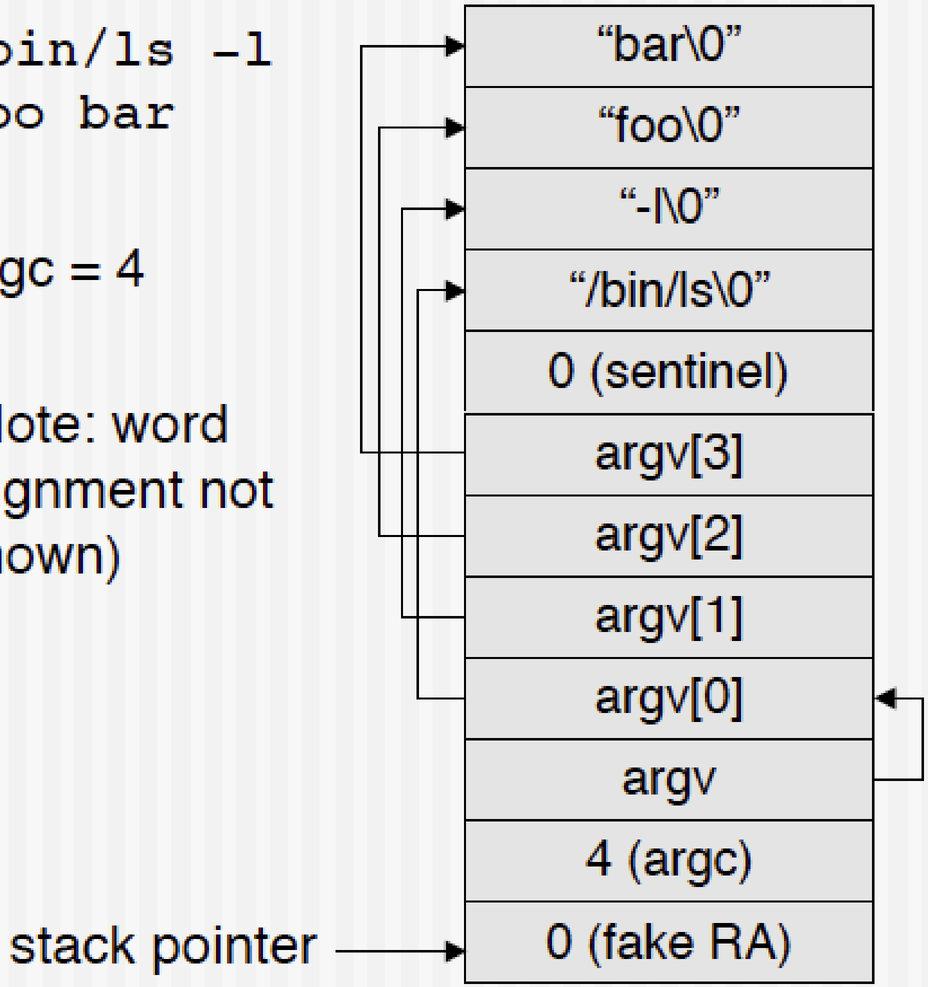
- Before a user program starts executing, the kernel must push the function's arguments onto the stack.
- This involves breaking the command-line input into individual words.
- Consider `"/bin/ls -l foo bar" =>`
  `"/bin/ls", "-l", "foo", "bar"`
- Implement the string parsing however you like in process_execute()
  - one solution is to use strtok_r() in lib/string.c

# Argument Passing (cont.)

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

argc = 4

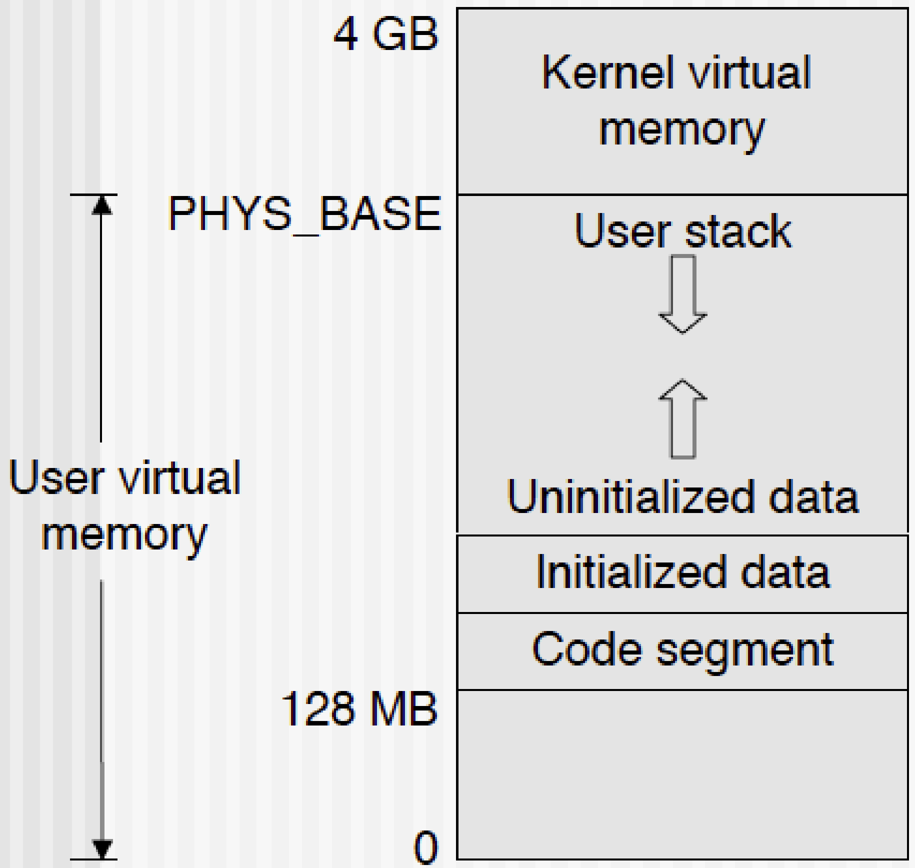
(Note: word alignment not shown)




- 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

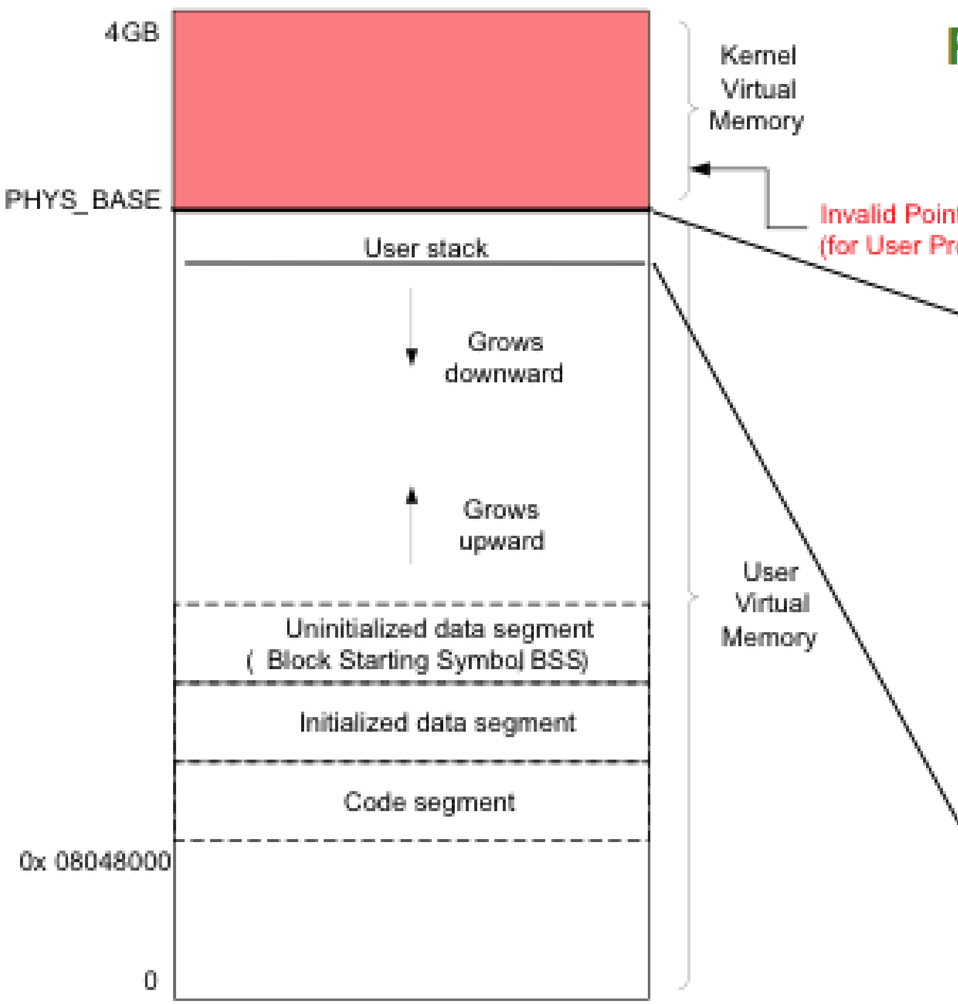
- hex_dump() function in <stdio.h> may be useful for seeing the layout of your stack

# Virtual 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

# Memory Layout




**PHYS_BASE = 3GB**

| 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] | 0xbffffffc | char * |
| 0xbfffffe0 | argv[2] | 0xbffffff8 | char * |
| 0xbfffffdc | argv[1] | 0xbffffff5 | char * |
| 0xbfffffd8 | argv[0] | 0xbfffffed | char * |
| 0xbfffffd4 | argv | 0xbfffffd8 | char ** |
| 0xbfffffd0 | argc | 4 | int |
| 0xbfffffcc | return address | 0 | void (*) () |

# Argument Passing (cont.)

command:
/bin/ls –l foo bar

padding

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 | | | | | |

# Hints

- Pintos currently lacks argument passing!

- Change *esp = PHYS_BASE to *esp = PHYS_BASE – 12 in setup_stack() to get started
- Change process_execute() in process.c to process multiple arguments
- Could limit the arguments to fit in a page(4 kb)
- String Parsing: strtok_r() in lib/string.h

```
pgm.c
main(int argc,
          char *argv[]) {
          ...
   }
$ pintos run  'pgm alpha beta'
argc = 3
argv[0] = "pgm"
argv[1] = "alpha"
argv[2] = "beta"
```

# User Programs in Pintos

`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…) => load(filename…)
- load sets up the stack, data, and code, as well as the start address

# Task 2: 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)
- Be aware of potential problems with buffers, strings, and pointers

# Hints

Two approaches to solving this problem:

- Verify every user pointer before dereference (simpler)
  - Is it in the user's address space, ie below PHYS_BASE? (look at is_user_vaddr in threads/vaddr.h)
  - Is it mapped or unmapped? (look at pagedir_get_page() in userprog/pagedir.c)
  - These checks apply to buffers as well
- Modify fault handler in userprog/exception.c
  - Only check that a user pointer is below PHYS_BASE
  - Invalid pointers will trigger a page fault
  - See 3.1.5 [Accessing User Memory] for more details

# 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
  - Return the value to the user in f->eax

# Hints

- Syscall numbers are defined in lib/syscall-nr.h
  - You will not be implementing all the calls -- see 3.3.4 [System Calls] for a list of required calls for Project 2
- Many of the syscalls involve file system functionality
  - A simple filesys implementation is provided (you do NOT need to modify this, but familiarize yourself with the interfaces in filesys.h and file.h)
  - The file system is not thread-safe
    - 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

# System Calls and Filesys (cont.)

- Reading from the 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:Implement System Calls (1)

- **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()
- A process can perform wait() only for its children
- Wait() can be called twice for the same process
  - The second wait() should fail
- Nested waits are possible: A → B, B → C
- Pintos should not be terminate until the initial process exits

**==> Start with exit()**

# Hints

- int **wait** (pid_t pid)
  - caller blocks until the 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(), and then wait() on top of process_wait()
  - involves the most work of all the syscalls

- 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 **wait**

# Hints

- 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 (use synchronization to ensure this)

# Task 4:Implement System Calls (2)

- **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 of them work on file descriptors

# Hints

- **Implementing 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`" structure
  - Deny writes to a running process's executable file
  - Ensure only one process at a time is executing the file system code

# Task 5: 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 Initial Strategy

- Make a disk and add a few programs (like args-*)
- Temporarily set up stack to avoid immediate page faults
  - In setup_stack() of process.c, change *esp = PHYS_BASE to *esp = PHYS_BASE - 12
  - This will allow execution of programs that take no args
- Implement safe user memory access
- Set up basic syscall infrastructure
  - Read the syscall number, dispatch to function
- Implement **exit** syscall
- Implement **write** syscall to STDOUT_FILENO
  - No tests will pass until you can functionally write to the console
- Change process_wait() to an infinite loop
  - The stub implementation exits immediately, so Pintos will power off before any processes can run

# Creating Disk & Execute Programs

- In order to run programs, you will need to create simulated disks containing user code
- How? From userprog/build folder:
  - pintos-mkdisk filesys.dsk --filesys-size=2 /* creates a 2 MB disk */
  - pintos -f -q /* formats the disk */
  - pintos -p ../examples/echo -a echo -- -q
    /* copies ../examples/echo to disk and names it "echo" */
  - pintos -q run 'echo x' /* runs the program with arg x */
- Example programs are in src/examples, test programs are in src/userprog/build/tests
  - Run "make" in these directories to build the programs
- You may want to make backup disks, in case the disk gets trashed (there is no filesys recovery tool)

# Workspace

```
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(-)
```

# Step 4: Testing

Pintos provides a very systematic testing suite for your project:

1. Run all tests:

```
$ make check
```

2. Run individual tests:

```
$ make tests/userprog/args-none.result
```

3. Run the grading script:

```
$ make grade
```

# Sample Functionality Test

```
/* This program echoes its command-line arguments */
int
main (int argc, char *argv[])
{
 int i;
 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
```

# Sample Robustness Test

```
/* 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"

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

*Expected output:*
bad-read: exit(-1)

# Using GDB

- You can use GDB to debug user code
- Start GDB as usual, then do

```
(gdb) loadusersymbols <userprog name>
```

- You can set breakpoints and inspect data as usual
- Note: a name that appears in both the kernel and the user program will refer to the kernel
  - Fix this by typing `pintos-gdb <userprog name>`
  - Then, `loadusersymbols kernel.o`

# Grading

10% of your project grade will be from the design document, and 90% from the successful implementation of the following components:

- Functionality of system calls, passes all twenty-eight (28) tests.
- Robustness of system calls, passes all thirty-four (34) tests.
- Functionality of features that VM might break, passes one (1) test.
- Functionality of base file system, passes all thirteen (13) tests.

You can use "make grade" to test your implementation and to see what grade you would get from the implementation component.

# Step 5: Design Document

Use the template in Section 3.3.1 of the Pintos documentation.

```
| 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.

                          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?
```

# Submission

1. Design document
   - due by November 17th @11:59PM
   - submit using your submit_cse421 or submit_cse521 script

2. All source code (the full source tree)
   - due by December 8th @11:59PM
   - submit via AutoLab auto-grading system

# Late Policy

Up to 24 hour late submission is accepted with 20% penalty:

- before 11:59pm on December 8th -> no penalty

between 11:59pm on December 8th and 11:59pm on December 9th --> 20% penalty

after 11:59pm on December 9th --> submission not accepted!

# Acknowledgements

- Pintos Manual

- Pintos Notes and Slides by A. He (Stanford), A. Romano (Stanford), J. Sundararaman & X. Liu(Virginia Tech), J. Kim (SKKU)