# CS140 Project 2: User Programs

# Project Requirements

- Argument passing
- Safe memory access
- System calls
- Process exit message
- File system interface
- Denying writes to executables

# Project Overview

Allow user programs to run on top of OS.

Can have multiple processes running simultaneously.

*Nothing that a user program can do should ever cause the OS to crash, panic, fail an assertion, or otherwise malfunction!*

Limitations:

- One thread per process
- Limited set of syscalls (e.g. no malloc)
- Restricted filesystem (project 4….)

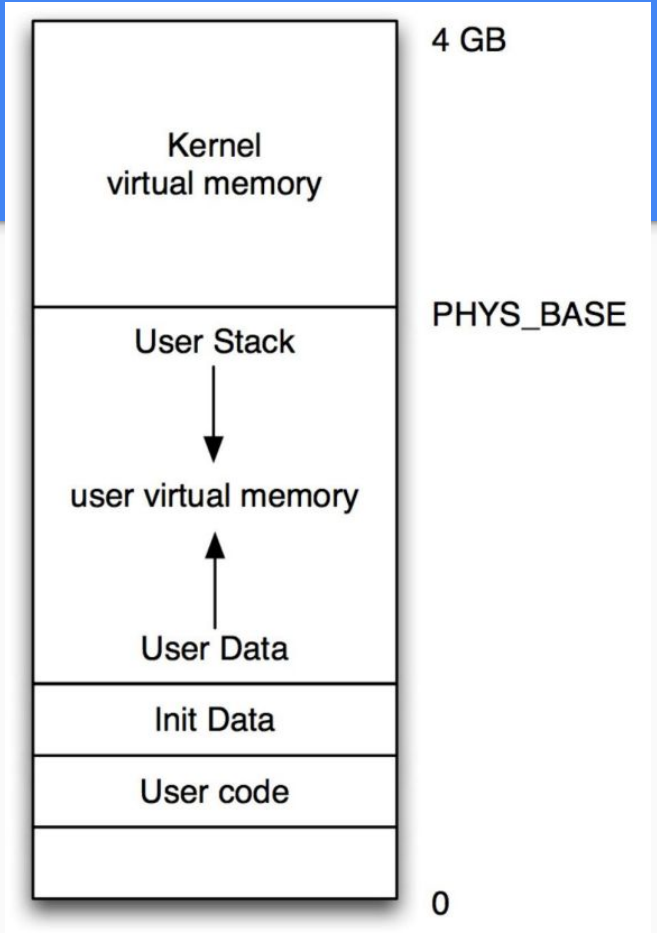
# User program entry point

`userprog/process.c:` **`process_execute()`**

- Creates thread running `start_process()`
- Thread loads executable from disk
- Set up virtual memory (stack, data, code)
- Start executing code from start address

`process_wait()` waits for executable to finish
`process_exit()` frees resources of program

# Memory layout (3.1.4)



# Argument passing (3.3.3)

- `grep foo bar` should run `grep` passing two arguments `foo` and `bar`
- `process_execute()` should allow multiple arguments to be passed in.
- Use `strtok_r()` in `lib/string.c` to break command line into arguments.
- Push the arguments onto the user stack following calling convention.

# Calling Convention (3.5.1)

```
/bin/ls -l foo bar

PHYS_BASE = 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] | 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 (*) () |

# Safe User Memory Access (3.1.5)

- The kernel may access memory through user-provided pointers: buffers, strings, pointers
- Dangers:
  - Null pointers
  - Pointers to unmapped virtual addresses or kernel addresses

1. Validate address.
2. If invalid, kill the process and free its resources.

# Safe User Memory Access (3.1.5)

**Approach #1 (simplest)**: verify every user pointer before dereferencing.

- Check in user address space (`< PHYS_BASE`)
- Check mapped (`userprog/pagedir.c:pagedir_get_page()`)

**Approach #2:** Modify page fault handler in `userprog/exception.c`

- Check in user address space (`< PHYS_BASE`)
- Dereference. Invalid pointers will trigger page faults.
- See code snippet in 3.1.5.

# System calls (3.3.4)

- Allows user programs to invoke kernel functions
- Has syscall number and possible argument(s)
- To user programs, like normal function calls (arguments on stack)
- Execute internal interrupt (int 0x30)
  - `userprog/syscall.c:syscall_handler(struct intr_frame *f)`
- Stack pointer (`f->esp`) at syscall number
- Return value in `f->eax`.

# System calls (3.3.4)

`userprog/syscall.c:syscall_handler()`

- Read syscall number at stack pointer (`f->esp`)
- Dispatch a particular function to handle syscall
- Read (validate!) arguments (above the stack pointer)
  - Validate pointers, buffers, and strings!
  - Syscall numbers defined in `lib/syscall-nr.h`

# Syscalls to implement (3.3.4)

- halt
- exec
- exit
- wait
- create
- remove
- open
- filesize
- read
- write
- seek
- tell
- close

# System calls (3.3.4): `exec`

```
pid_t exec(const char *cmd_line)
```

- Similar to UNIX fork() + execve()
- Creates a child process
- Must not return until new process has been created (or creation failed)
- Creation is successful if child has successfully loaded its executable and there is a thread ready to run.

# System calls (3.3.4): `exit`

```
void exit (int status)
```

- Exit with status and free sources
- Print process termination message
- Parent must be able to retrieve status via `wait`

# Pintos File System (3.1.2)

- Simple file system implementation provided: `filesys.h, file.h`
- Don’t need to modify! Wait for project 4….
- Not thread-safe! **Use a coarse lock to protect it.**
- Syscalls take file descriptors as args
  - Pintos represents files with `struct file *`
  - You must design the mapping
- Special cases: STDIN and STDOUT
  - `write(STDOUT_FILENO, …)` → `putbuf, putchar`
  - `read(STDIN_FILENO, …)` → `input_getc`

# Denying writes to executables (3.3.5)

Pintos should not allow code that is currently running to be modified.

- Use `file_deny_write()` to prevent writes to open file
- Closing file re-enables writes
- Keep executable open as long as the process is running.

# Using GDB for user programs (E.5)

Start GDB as usual, then run:

```
(gdb) loadusersymbols <userprog.o>
```

User symbols will not override kernel symbols.

# Getting Started

You may build on top of Project 1 or start with a fresh copy of pintos.

No code from Project 1 will be required (although some of your timer implementation *might* be helpful for Projects 3 and 4, not necessary).

# Getting Started: file system setup (3.1.2)

Create a simulated disk called `filesys.dsk` with a 2MB Pintos file system partition, and then copy programs and run them.

1. Make disk: `pintos-mkdisk filesys.dsk --filesys-size=2`
2. Format disk: `pintos -f -q`
3. Coy program: `pintos -p ../../examples/echo -a echo -- -q`
4. Run program: `pintos -q run 'echo x'`

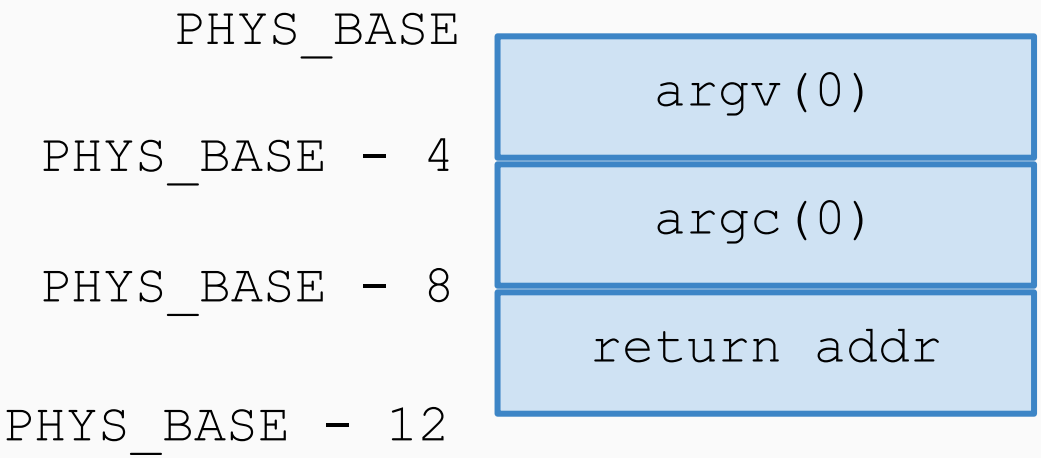
Fast version:

1. `pintos-mkdisk filesys.dsk --filesys-size=2`
2. `pintos -p ../../examples/echo -a echo -- -f -q run 'echo x'`

# Getting Started: Implement this first! (3.2)

- **Argument passing: change `*esp = PHYS_BASE;` to `*esp = PHYS_BASE -12;`**
  - Allows running programs with no arguments
- User memory access
  - All system calls need to read user memory
- System call infrastructure
  - Read the system call number from the user stack and dispatch to a handler
- Exit system call
- Write system call for STDOUT
- **Temporarily change `process_wait` to an infinite loop so pintos doesn't immediately power off**

# Why *esp = PHYS_BASE - 12?



# Getting Started: Tips

- Read the spec *carefully*. There are lots of pieces to this assignment!
- Read the design doc before you get started.
- If you're confused about why a test is failing, read the source code in `tests/userprog`.
- Read the system call APIs carefully, and make sure you validate all user memory addresses.
- **Follow the suggested order of implementation!**
- *Warning:* project 3 is built on project 2, and project 4 is built on project 2 or 3