Administrivia

- **Project 1: Threads was due at 10am, unless one member of your team is here**
- **Project 2: Userprog is due Friday, Feb. 10**

Overview

- **Project 2: Userprog**
  - build support for running user processes
- **Project requires good understanding of:**
  - steps for running a user program
  - distinctions between user and kernel virtual memory
  - system call inferface and handling
  - kernel file system interface

User Programs

- **What happens when a user runs (in the shell):**

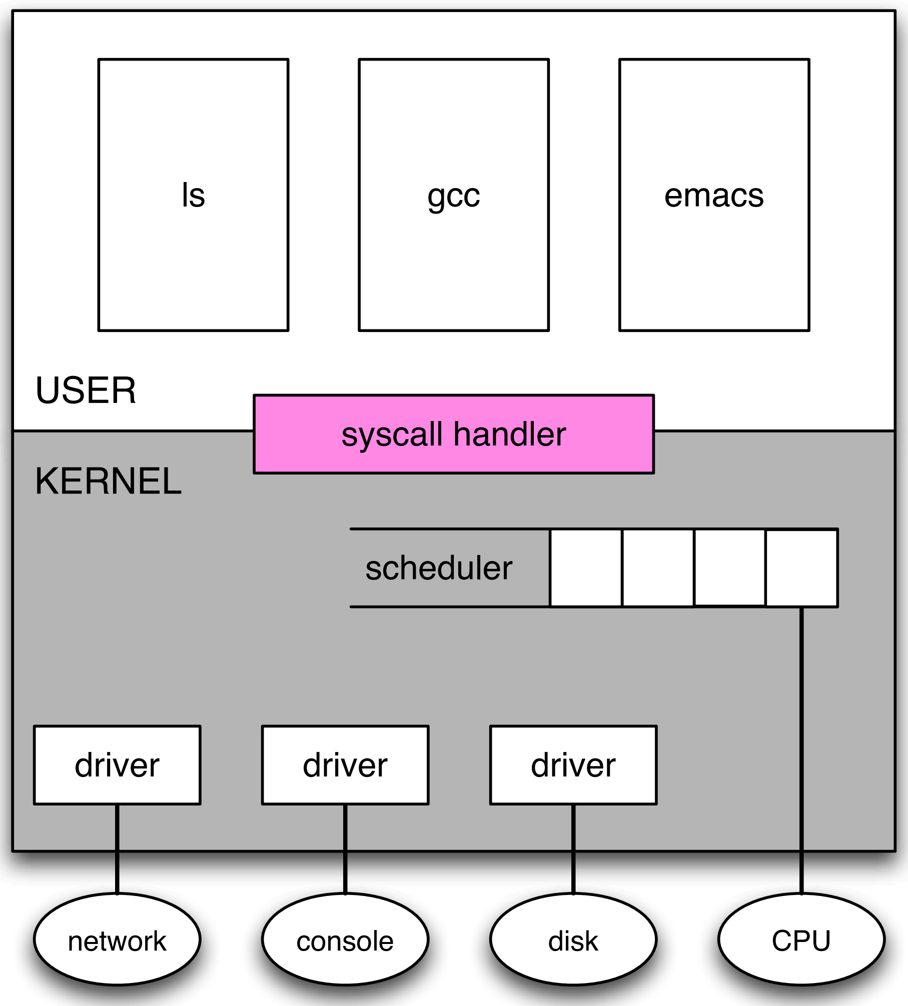
```
% cp -r herp derp
```

- `shell` **parses user input**
  - `argc = 4, argv = {"cp", "-r", "herp", "derp", NULL}`
- `shell` **calls** `fork()` **and** `execve(argv[0], argv, env)`
- `cp` **uses system call interface to read/write files**
- `cp` **may print messages to** `stdout`
- `cp` **exits**

User Programs in Pintos

- `threads/init.c`
  - `run_actions () --> run_task (argv)`
  - `run_task () --> process_wait (process_execute (task))`
- `userprog/process.c:process_execute()`
  - creates thread running `start_process()`
  - thread loads executable file
  - sets up user virtual memory (stack, data, code)
    - ▷ user programs have no heap/malloc
  - starts executing user process (start address)

System Calls

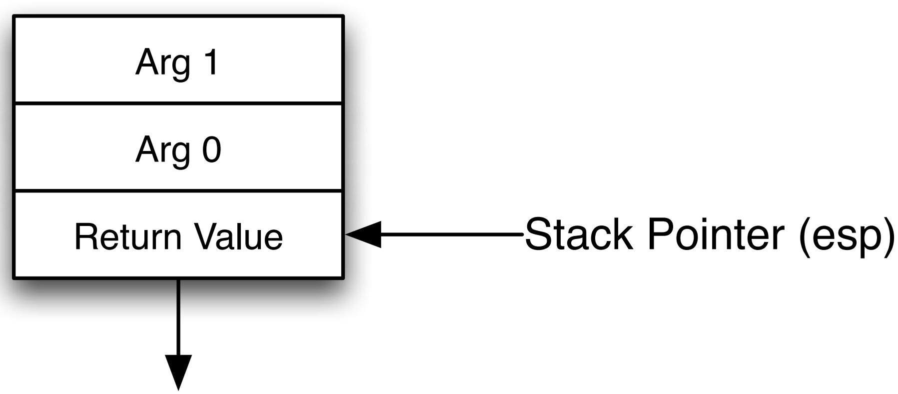


Starting User Process

- `lib/user/entry.c`

```
void _start (int argc, char *argv[]) {
  exit (main (argc, argv));
}
```

- **pass process start arguments on user stack**



Project 2 Requirements

You will need to implement:

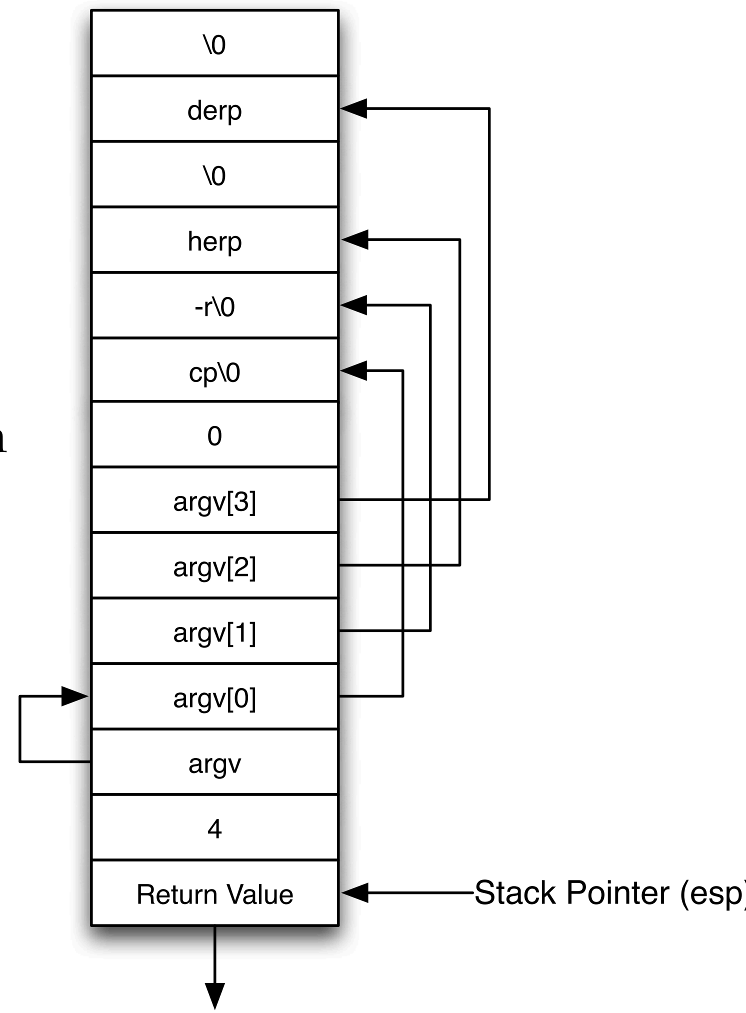
- **Argument Passing**
- **Safe memory access**
- **System calls**
- **Process exit messages**
- **Denying write to in-use executable files**

Argument passing

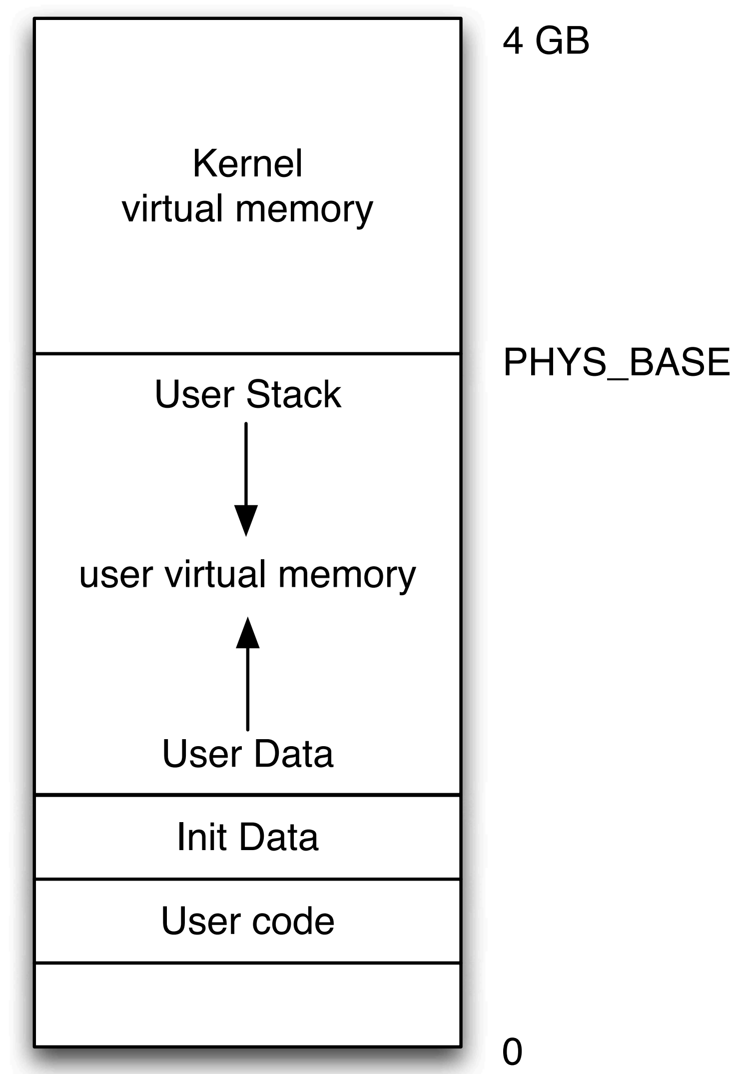
- **In preparation to start a user process, the kernel must push the command line arguments onto the stack**
- **Break command-line input to individual tokens**
  - from: `"cp -r herp derp"`
  - to: {`"cp", "-r", "herp", "derp"`}
- `strtok_r(...)` **in** `lib/string.c` **is helpful.**

Argument passing (stack)

push tokens (c-strings)
push null sentinel
push address of each token
(right to left)
push argv, push argc
push return address (0)



Safe Memory Access



Safe Memory Access

- **The kernel will often access memory through user-provided pointers**
- **This is dangerous!**
  - null pointers
  - pointers to unmapped virtual addresses
  - pointers to kernel addresses
- **kill the process (free its resources, e.g. locks, memory)**
- **be careful with buffers, strings, any other pointers**

Safe Memory Access

Two approaches to solving this problem:

- **Verify every user pointer before dereference (simpler)**
  - ensure it is in user's address space (i.e. below `PHYS_BASE`)
  - ensure it is mapped (look at `userprog/pagedir.c:pagedir_get_page()`)
  - for buffers, ensure for the entire buffer.
- **Modify fault handler in `userprog/exception.c`**
  - ensure pointer (or buffer) is below `PHYS_BASE`
  - invalid pointers will trigger page faults
  - See 3.1.5 - Accessing User Memory for more details

System Calls: how do they work?

- **work like normal function calls (args in stack)**
- **execute internal interrupt (`int` instruction)**
  - `syscall_handler(struct intr_frame *f)`
- **stack pointer (`f->esp`) at syscall number**
- **calling thread data available**
  - to pass args to handler
  - to return value to user process
- **return value just like functions (`f->eax`)**

System Calls: Implementation

- `userprog/syscall.c:syscall_handler()`
- **read syscall number at stack pointer**
- **dispatch a particular function to handle syscall**
- **read (validate!) arguments (above the stack pointer)**
  - above the stack pointer
  - validate pointers and buffers!
- **syscall numbers defined in** `lib/syscall-nr.h`
- **see 3.3.4 - System Calls for Project 2's required calls.**

System Call: File System

- **many syscalls involve file system functionality**
- **simple filesys impl is provided:** `filesys.h, file.h`
  - no need to modify it, but familiarize yourself
- **file system is not thread-safe! (Project 4)**
  - 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: reading from keyboard and writing to console**
  - `write(STDOUT_FILENO, ...)` use `putbuf` or `putchar`
  - `read(STDIN_FILENO, ...)  use input_getc`

System Calls: Processes

- **Generally, these syscalls require the most design and implementation time.**
- `pid_t exec(const char *cmd_line)`
  - like UNIX `fork()` + `execve()`
  - creates a child process
  - must not return until new process has been created (or creation failed)

System Calls: Processes

- `int wait (pid_t pid)`
  - parent must block until child process `pid` `exits`
  - returns exit status of the child
  - must work if child has ALREADY exited
  - must fail if it has already been called on child before
- `void exit (int status)`
  - exit with `status` and free resources
  - process termination message
  - parent must be able to retrieve `status` via `wait`

System Calls: Security

- **How does system recover from null pointer segfault in user program?**
- **How does system recover from null pointer segfault in kernel?**

System Calls: Security

- **How does system recover from null pointer segfault in user program?**
  - kill user process, life goes on.
- **How does system recover from null pointer segfault in kernel?**
  - it (basically) doesn't!
- **Verify all user-passed memory references (pointers, buffers, strings)**
- **kill user program if passed illegal addresses.**

Denying writes to executables

- **Executables are files like any other.**
- **Pintos should not allow code that is currently running to be modified.**
  - use `file_deny_write()` to prevent writes to an open file
  - closing a file will re-enable writes
  - keep executable open as long as the process is running

Utilities: Making Disks

- **user executables must be on virtual hard disk**

```
cd pintos/src/userprog
make
pintos-mkdisk fs.dsk --filesys-size=2   /* Create 2MB disk */
pintos --disk=fs.dsk -- -f -q           /* format the disk */
pintos --disk=fs.dsk -p ../examples/echo -a echo -- -q
                                        /* copy it to disk */
pintos --disk=fs.dsk -- -q run "echo x" /* run the program */
```

- **user code examples in** `src/examples`
- **you can write your own code to test things**
  - but you don't need to.

Utilities: GDB

- **you can use GDB to debug user code**
- **start GDB as usual, then do:**

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

- **you can set breakpoints and inspect data as usual**
- **user symbols will not override kernel symbols**
  - work around duplicate symbols by inverting order
  - run gdb with:

    ```
    pintos-gdb <userprog.o>
    ```
  - then load the kernel symbols:

    ```
    (gdb) loadusersymbols kernel.o
    ```

Getting Started

- **Make a disk and add simple programs**
  - run `make` in `src/examples`
- **temporarily setup stack to avoid page faulting**
  - in `userprog/process.c:setup_stack()`
  - change: `*esp = PHYS_BASE`
  - to: `*esp = PHYS_BASE - 12`
  - this will allow running programs with no args
- **implement safe user memory access**

Getting Started

- **setup syscall dispatch**
- **implement** `exit`
- **implement** `write` **to** `STDOUT_FILENO`
  - no tests will pass until you can write to the console
- **change** `process_wait(...)` **to an infinite loop**
  - stub implementation exits immediately
  - Pintos will power off before any processes can run
- **Start early!**
- **Good luck!**