Project 2--User Programs

Jason Bau

CS140 Winter '09

Slides Acknowledgements to previous CS140 TAs

User Program/Process

- What happens in Unix shell when?

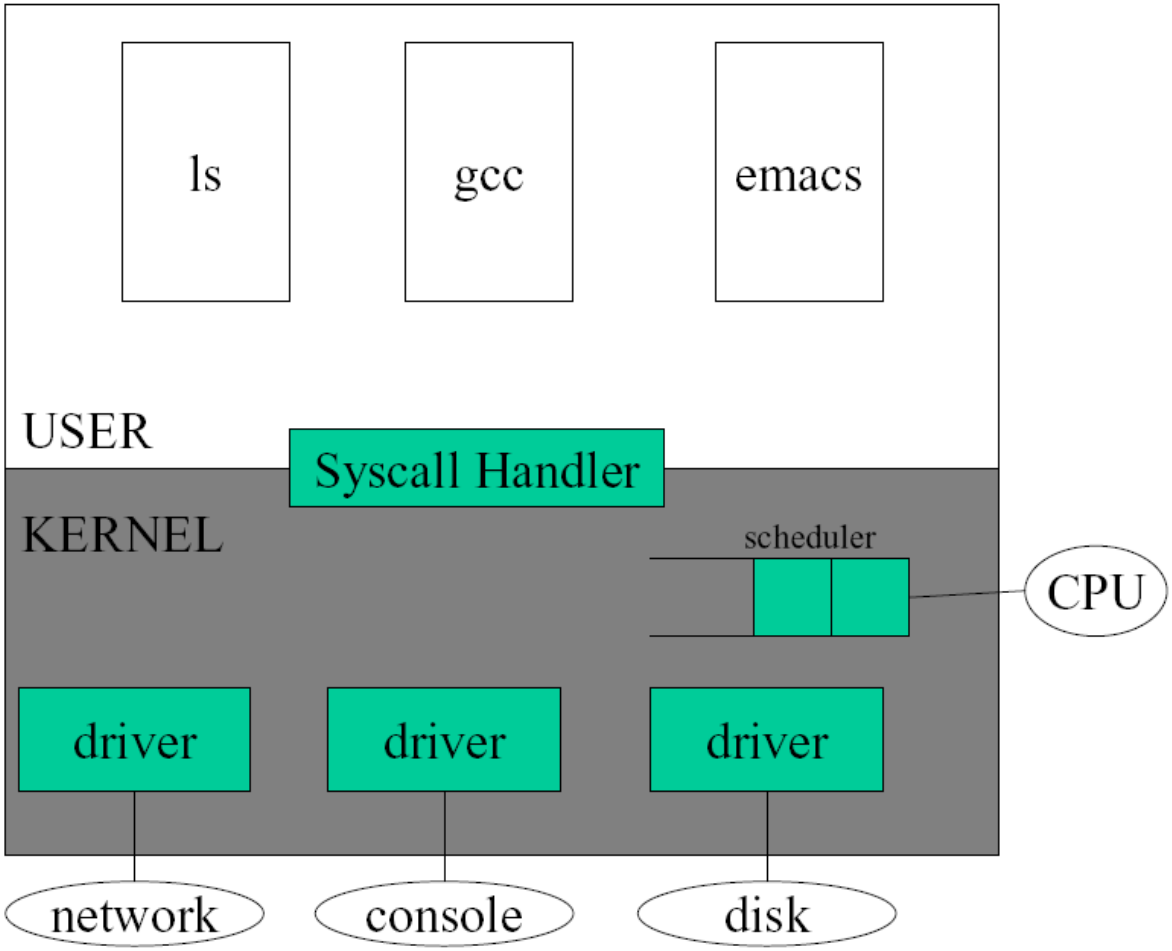
```
myth13:~/> cp -r pintos .
```

1. Shell handles user input
2. fork() and execve("cp","-r pintos .")
3. cp accesses file system to perform copy
4. cp prints messages (if any) to stdout
5. cp exits

Q: What is shell doing in the mean time?

Q: Which lines require system calls?

Kernel/User Differentiation



Pintos – without shell

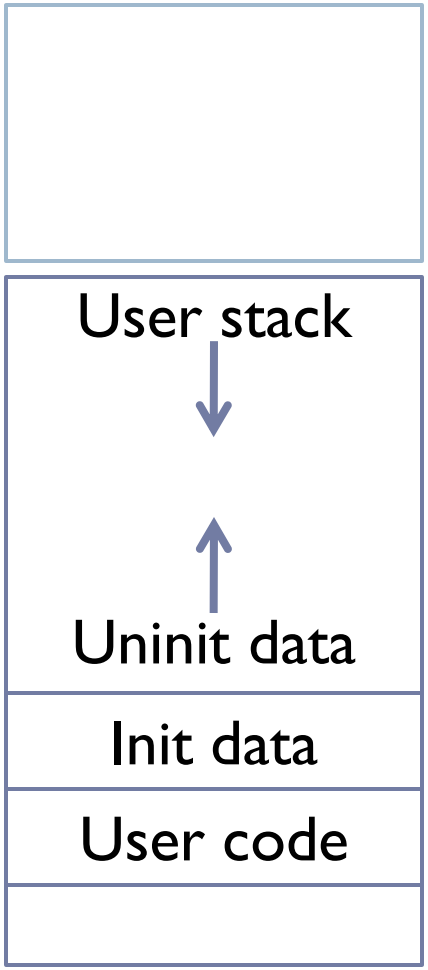
- Tests for proj2 in userprog are user processes

  How do they get run? On disk—details later
- threads/init.c
  - run_actions() -> 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)
  - starts executing user process @ _start (…)

User vs. Kernel Virtual Memory

Kernel virtual memory

User virtual memory



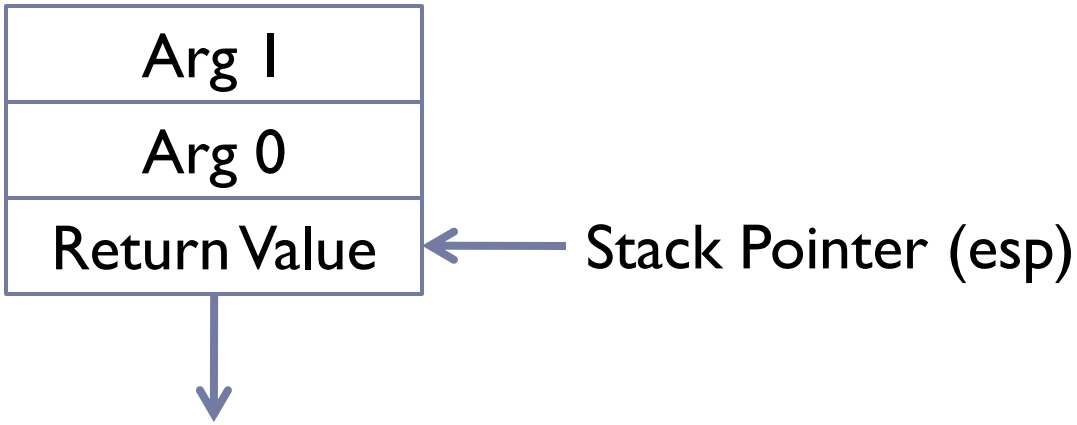
- User code cannot address above PHYS_BASE
- User can only access mapped addresses
- User access to unmapped address → page fault.
- Kernel can page fault if it accesses unmapped user address

Starting User Process

- lib/user/entry.c

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

- Pass process start arguments on user stack

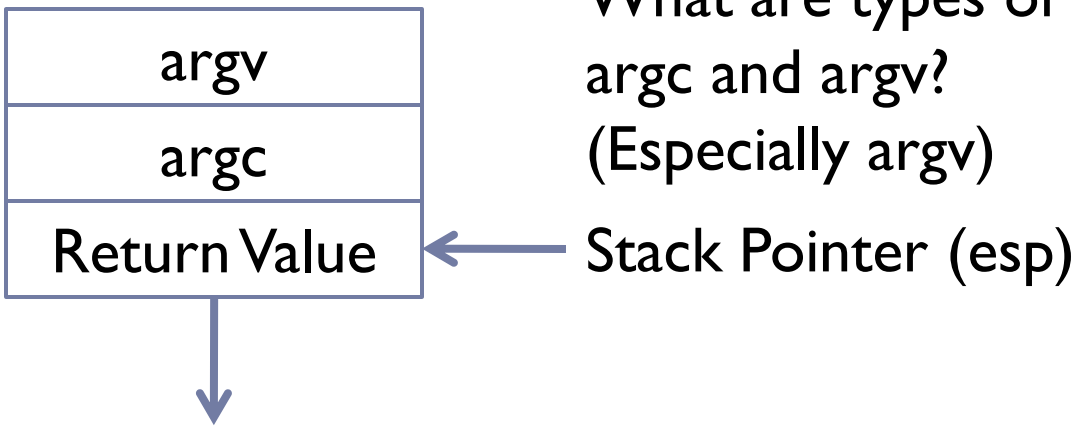


Starting User Process

- lib/user/entry.c

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

- Pass process start arguments on user stack

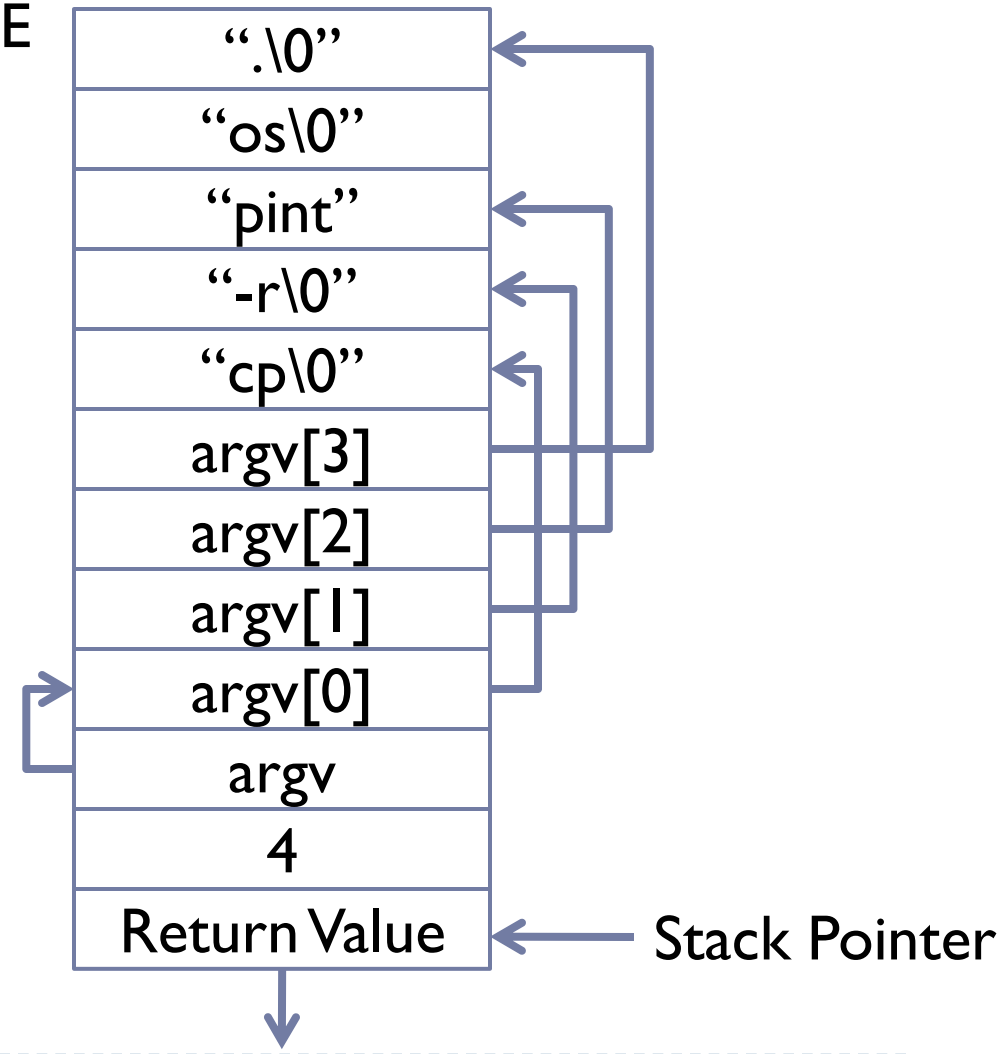


Setting up Starting Arguments

```
cp -r pintos .
```

argc = 4
argv[0]= “cp”
argv[1]= “-r”
argv[2]= “pintos”
argv[3]= “.”

Pictured without all alignment elements



Project 2 Assignment

- Argument passing
  - Already covered
- System calls
  - List in assignment page
  - We'll discuss shortly
- Process exit messages
- Denying writes to in-use executable files

System Calls

- Push args same as normal function calls
- Stack pointer at syscall number
- Execute internal interrupt
  - `int` instruction
- Calling thread data available
  - syscall_handler(struct intr_frame *f)
  - Use to pass args to handler, AND ???
  - To return value to user process

System Calls—File System

- You are writing syscall interface for this project
- You do NOT need to change Pintos file system code for this project
- Users deal with `file descriptors` (`int`s)
  - Pintos file system uses `struct file *`
  - You design mapping
- File system is not thread-safe (proj 4).
  - Use coarse synchronization to protect it

System Calls—File System

- Reading from the keyboard and writing to the console are special cases
- fd STDOUT_FILENO
  - Can use `putbuf(...)` or `putchar(...)`
  - In src/lib/kernel/console.c
- fd STDIN_FILENO
  - Can use `input_getc(...)`
  - In src/devices/input.h

System Calls—Processes

- int **wait** (pid_t *pid*)
  - Parent must block until the child process *pid* **exit**s
  - Returns exit status of the child
  - Must work if child has ALREADY exited
  - Must fail if it has already been called on child
- void **exit** (int *status*)
  - Exit with *status* and free resources
  - Process termination message
  - Communicate with **wait** so parent can retrieve your exit status

System Calls—Processes

- pid_t exec(const char *_cmd_line_)
  - Like unix `fork()` + `execve()`
  - Creates a child process
  - This must **not** return until new process has been successfully created (or has failed)

  Generally, these three syscalls require most design + implementation time. Do them well.

System Calls—Security

- How does system recover from null-pointer segfault in user program?
  - Kill user process, schedule others, and life goes on
- How does system recover from null-pointer segfault in kernel?
  - It (basically) doesn't!

Protecting the Kernel from Users

- Verify user-passed mem reference before use
  - Buffers
  - Strings
  - Pointers
- Check mem reference (two available techniques)
  - Is passed address in user memory?
  - Is it mapped?
    - pagedir_get_page() in userprog/pagedir.c
    - Modify page fault handler in userprog/exception.c
  - Size of reference a consideration?
- Kill the user program it passed illegal address
  - Remember to release any resources held

Utilities—Making Disks

- User code must be on virtual hard disk

```
cd pintos/src/userprog
make
pintos-mkdisk fs.dsk 2              /* Create 2MB disk*/
pintos -f -q                        /* Format the disk */
pintos -p ../examples/echo -a echo -- -q
                                    /* put a prog on the disk */
pintos -q run 'echo x'              /* run the program */
```

Utilities—Making Disks

- Recommend making a backup disk w/programs in case yours gets trashed
- User code examples in src/examples
- You can write your own user code for test,

  but don't NEED to.

Getting Started

- Make a disk and add some simple programs
  - Run make in src/examples
  - Maybe some of the first tests (args-*)
- Temporarily setup stack to avoid page faulting
  - esp = esp - 12;
- Basic syscall handler
  - Which syscall to dispatch
  - Reading from user memory address
- Skeleton exit system call body
- Handle write() syscall to STDOUT_FILENO
- Change process_wait() to infinite loop to instead of exit

Utilities—debugging user code

- Start `pintos-gdb` as usual
- `add-symbol-file` *`program.o`*
- Set breakpoints, etc, in user code
  - Kernel names take precedence over user code
  - To change:
    - `pintos-gdb userprog`
    - Then `add-symbol-file kernel.o`