CS140 Help Session
Homework 2
UserProg

Megan Wachs
User vs. Kernel Code

- Tests in src/userprog/ are now user code
- More user programs to play with are in src/examples
- You can write and compile your own, with some limitations
- You don’t NEED to write any user code
User vs. Kernel Code

- OLD: `run_test` ran code directly in the main thread
- NEW: 
  
  ```
  process_execute("cp -r pintos .")
  ```
  
  calls `create_thread(...)`
  
  calls `execute_thread(...)`
  
  Runs "cp" in user space.
User vs. Kernel Memory

User code can not access addresses above PHYS_BASE.

The user can only access mapped addresses.
User Virtual Memory

If the user tries to access an unmapped address, it will page fault.

Even in kernel mode you can page fault if you try to access an unmapped user address.
Enabling User Code

1. Start it running
   • Set up the stack
Enabling User Code

1. Start it running
   • Set up the stack

2. Allow it to do things
   • Interact with the file system
   • Communicate with other processes
Enabling User Code

1. Start it running
   • Set up the stack

2. Allow it to do things
   • Interact with the file system
   • Interact with the user
   • Communicate with other processes

3. Don’t let it crash the kernel!
   • Check all user pointers
Setting Up the Stack

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

Your code must set up the “main” function call by hand.
Setting Up the Stack

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

Your code must set up the “main” function call by hand.

```
argv
argc
Return value
```

Stack Pointer
Setting Up the Stack

`cp -r pintos .`

`argc = 4`
`argv[0] = "cp"
argv[1] = "-r"
argv[2] = "pintos"
argv[3] = "."
argv[4] = 0`
System Calls

- File System Related
- Process Related
- more to come in HW 3
System Calls

```
int fd = fopen("foo.c");
```

```
syscall_handler(struct intr_frame *f);
```

```
File * file = filesystem_open("foo.c");
```

src/lib/user/syscall.h

userprog/syscall.c

filesystem/filesys.h
System Calls

```c
int fd = fopen("foo.c");

syscall_handler(struct intr_frame *f);

File * file = filesys_open("foo.c");

Your code
```
System Calls - File System

• You do NOT need to change the file system for this project.
• Users deal with file descriptors (ints). The file system uses struct file *.
  You need to enforce a mapping.
• The file system is not thread-safe (yet!) so 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 **exits**
  - Must return the status value of the child
  - Must work if child has already exited
  - Must fail if it has already been called on the child.

- **void exit (int status)**
  - Exit w/ the specified status & free resources
  - Required: print out the exit status
  - Need synchronization with **wait** so the parent can retrieve your status if desired.
System Calls - Processes

• `pid_t exec (const char *cmd_line)`
  – Create a new child process
  – This **MUST** not return until the new process has successfully been created (or has failed)

Design these three well! They are the most time-consuming.
System Calls - Paranoia

• In a system call, the user will pass you all kinds of addresses (buffers, strings, stack pointers, etc).
• Trust no one. Check anything the user passes to you.
  – Is the passed-in address in user memory?
  – You can use pagedir_get_page(…) to check if the address is mapped for the user program.
• Kill the child if it passes an illegal address.
  – Free any locks and resources.
Utilities –Making Disks

- User code must be on a virtual hard drive

```
cd pintos/src/userprog
make
pintos-mkdisk fs.dsk 2 /*create a 2MB disk*/
pintos -f -q /*format the disk*/
pintos -p ../examples/echo -a echo -- -q /*put a program on the disk and rename it*/
pintos -q run 'echo x' /* run the program*/
```
Utilities – Making Disks

• Recommend making a backup disk with programs loaded in case your disk gets trashed
Getting Started

• Make a disk and add some simple programs
  – run `make` in src/examples

• Temporarily set up the stack to avoid page faulting immediately.
  \[ \text{esp} = \text{esp} - 12; \]

• Enable reading from user memory addresses

• Handle `write()` syscall for STDOUT_FILENO

• Change `process_wait` to an infinite loop so that pintos doesn’t power off
Utilities – Debugging User Code

• Start pintos-gdb as usual
• `add-symbol-file program`
• Set breakpoints, etc, in user code
  • Kernel names will take precedence over user code
  • Change this by doing `pintos-gdb userprog.o`, then
    `add-symbol-file kernel.o`