CS140 Project 2: User Programs

Project Requirements

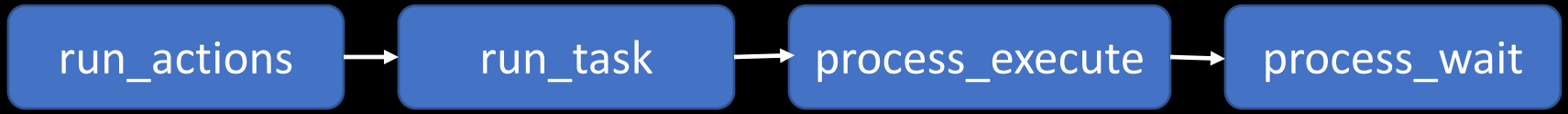
- Argument passing
- Safe Memory Access
- System calls
- Process exit message
- File systems
- Denying writes to executables
- Utilities

Project Requirements

- Allow **user programs** to run on top of OS
- Restrictions:
  - One thread per process (no multithreaded user program)
  - No malloc
  - Restricted filesystem
- Can have multiple processes at the same time

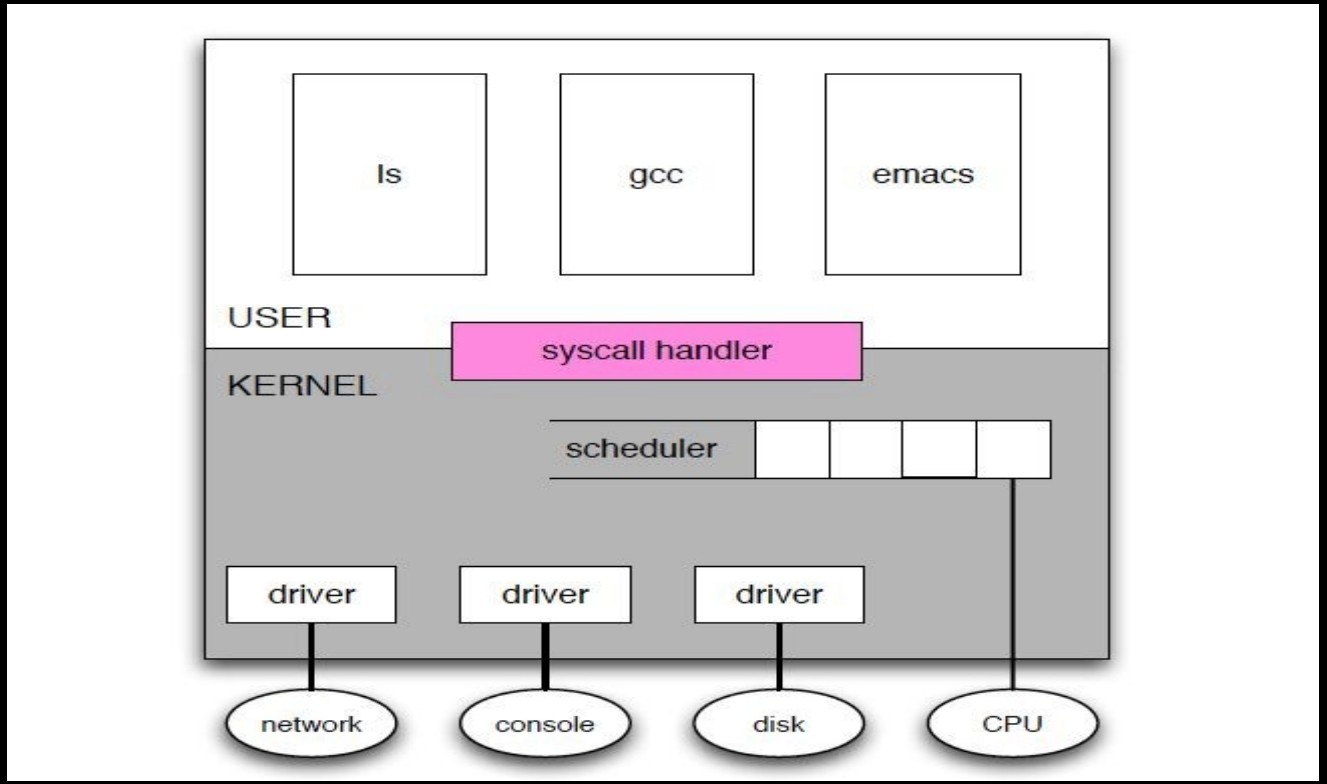
User Program Entry Point

- threads/init.c

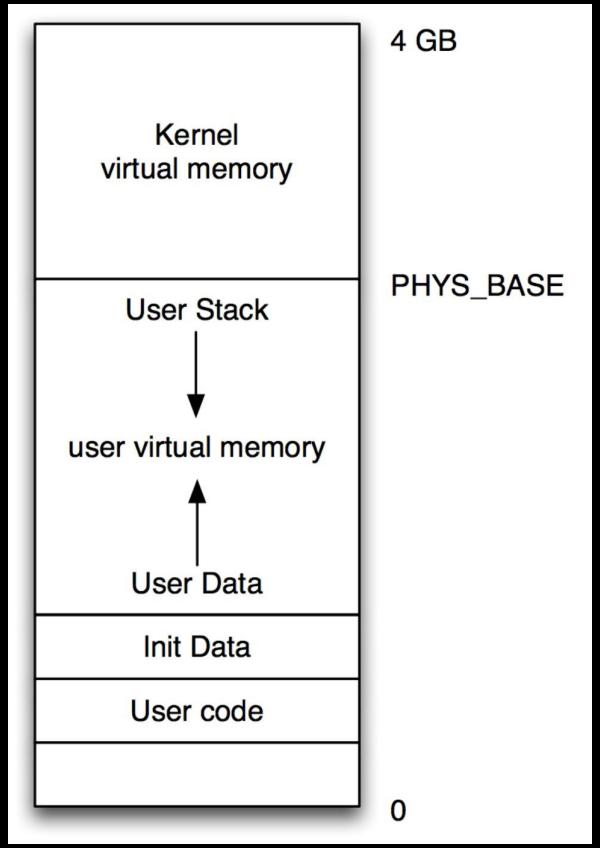


- userprog/process.c: process_execute()
  - creates thread running start_process()
  - thread loads executable file
  - sets up user virtual memory (stack, data, code)
  - starts executing user process (start address)

User vs Kernel Memory Space



Memory Layout



Argument Passing

- process_execute should allow multiple arguments to be passed in
- use strtok_r in lib/string.c to break commandline into args
- Pus the arguments onto the user stack
- Follow calling convention

Push arguments onto user stack (not the kernel stack)

Calling Convention

Program Startup Example

/bin/ls -l foo bar

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 Memory Access

- The kernel may access memory through user-provided pointers: buffers, strings, pointers
- Dangers
  - Null pointers
  - Pointers to unmapped virtual addresses or kernel addresses
- First validate address
- If invalid, then kill the process (free its resources, e.g. locks, memory)

Safe Memory Access cont.

- Approach 1: Verify every user pointer before dereferencing (recommended)
  - Ensure it is in user's address space (below PHYS_BASE)
  - Ensure mapped (userprog/pagedir.c:pagedir_get_page)
- Approach2: Modify page fault handler in userprog/exception.c
  - Ensure pointer (or buffer) is below PHYS_BASE and then dereference. Invalid pointers will trigger page faults

System Calls

- Allows user programs to invoke kernel functions
- Has a syscall number and possibly argument(s)
- To user programs, like normal function calls (args in stack)
- Execute internal interrupt (int 0x30)
  - userprog/syscall.c: syscall_handler(struct intr_frame *f)
- Stack pointer (f->esp) at syscall number
- Return value just like functions (f->eax). This is how you can communicate the
  - value back to user space.

System Calls

- **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)
  - Validate pointers and buffers!
  - syscall numbers defined in **lib/syscall-nr.h**

System Calls to Implement

halt
exec
exit
wait

create
remove
open
filesize
read
write
seek
tell
close

System Calls: 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: wait

- int wait (pid_t pid)
  - Waits for a child process *pid,* retrieves child's exit status
  - Parent must block until child process pid exits (or is terminated by the kernel)
  - Returns exit status of the child
  - If terminated by the kernel, return -1
  - Must work if child has ALREADY exited
  - Must fail if it has already been called on child before
  - (most time consuming system call to implement!)

System Calls: exit

- **void exit (int status)**
  - Exit with status and free resources
  - You must print process termination message
  - Parent must be able to retrieve status via wait

Pintos File System

- Simple filesys impl is provided: filesys.h, file.h
- No need to modify it, but familiarize yourself
- 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: read from keyboard and writing to console
  - write(STDOUT_FILENO, ...) use putbuf or putchar
  - read(STDIN_FILENO, ...) use input_getc

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 open file
  - Closing file re-enables writes
  - Keep executable open as long as the process is running

Using GDB for User Programs

- You can use GDB to debug user code
- Start GDB as usual, then do:
  - **(gdb) loadusersymbols <userprog.o>**
- User symbols will not override kernel symbol. 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

- 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 could be useful for Projects 3 and 4
  - Not necessary however

Getting Started: Setting up the file system

- Create a simulated disk called “filesys.dsk” with a 2MB Pintos file system partition
  - pintos-mkdisk filesys.dsk --filesys-size=2
  - Then format pintos -f –q
- Copy simple programs to your simulated file system
  - pintos -p ../../examples/echo -a echo -- -q
- Run
  - pintos -q run ‘echo x’

Getting Started: Implement this first!

- **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 “fd 1” (system console)
- **Temporarily change process_wait into an infinite loop so that Pintos doesn’t immediately power off**

*esp = PHYS_BASE - 12;