Lab2: User Programs

TA Session

TA: Yongtong Wu
PPT Credit : Yinmin Zhong

Some announcements:

- Lab 2 Code will due

> Code Due: Thursday 04/23 11:59 pm
>
> Design Doc Due: Sunday 04/26 11:59 pm

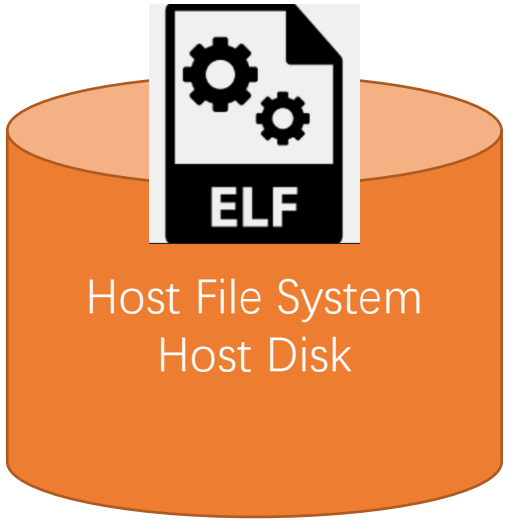
- No grace day
- Start early, Start early, Start early
- You can complete Lab2 from a clean codebase

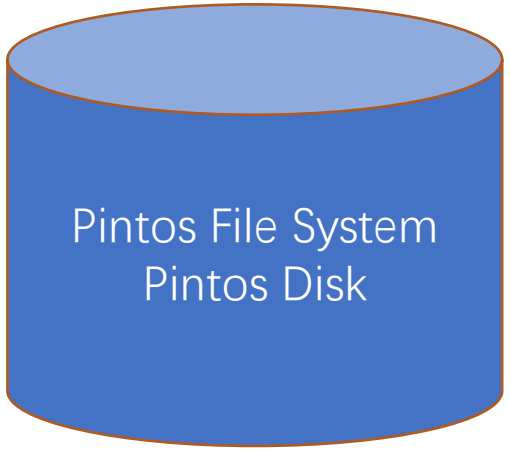
Contents

- Pintos Disk and File System
- System Call
- Interrupt Handling
- Lab2 tasks and suggestions

Where are the User Programs?

- Source files are under /src/examples/ directory
- Run `make` under /src/examples/ → ELF files

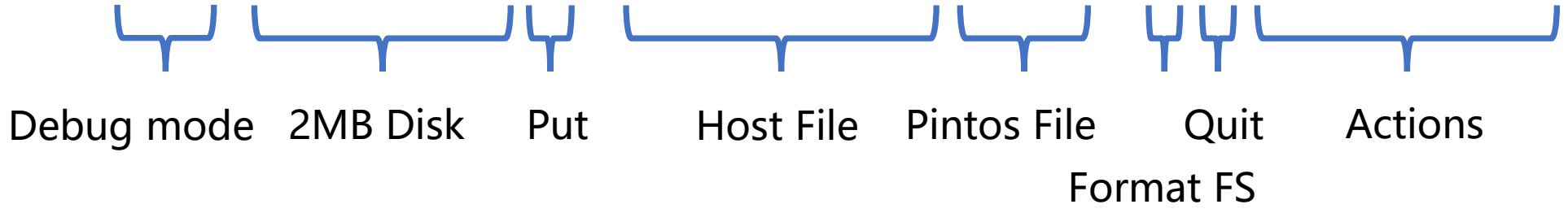




Debug example:

- Run `make && cd build` under /src/userprog/ directory

- pintos --gdb --filesys-size=2 -p ../../examples/echo -a echo -- -f -q run 'echo iloveos'



- Details on Lab Document

Overview

Always Remember the Goal: reach main().
It is quite complex… be patient!
LLM is a wonderful tool for codebase understanding!

1. BIOS 引导 → 加载器进入保护模式 → 跳转内核入口 threads/loader.S
2. pintos_init() 初始化所有子系统（内存、中断、文件系统等） threads/init.c:77
3. 解析内核命令行，-f 格式化文件系统，-q 设置运行后关机 threads/init.c:233
4. run_actions() 依次执行命令行中的 action threads/init.c:302
5. fsutil_extract() 从 scratch 分区读 ustar 归档，将 echo 写入 Pintos 文件系统 filesys/fsutil.c:73
6. run_task() 调用 process_execute("echo iloveos") threads/init.c:286
7. process_execute() 复制文件名，调用 thread_create() 创建新内核线程，入口函数为 start_process userprog/process.c:29
8. thread_create() 在新线程内核栈上压入 switch_entry → kernel_thread → start_process 的调用帧，然后放入 ready_list threads/thread.c:166
9. 调度器选中新线程，上下文切换过来 threads/thread.c:553
10. kernel_thread() 开中断，调用 start_process(fn_copy) threads/thread.c:419
11. start_process() 构造伪中断帧（段寄存器设为用户态），调用 load() 加载 ELF userprog/process.c:51
12. load() 创建新页目录、写入 CR3、解析 ELF、逐段分配物理页并映射、建立用户栈，设置 eip=_start, esp=PHYS_BASE userprog/process.c:209
13. start_process() 将 %esp 指向伪中断帧，jmp intr_exit userprog/process.c:75
14. intr_exit 执行 popal 恢复寄存器，iret 弹出 eip/cs/esp/ss，CPU 从 Ring 0 切到 Ring 3 threads/intr-stubs.S:51
15. 用户态 _start(argc, argv) 开始执行，调用 main(argc, argv) lib/user/entry.c:7

Parsing

pintos_init():

```
/* Break command line into arguments and parse options. */
argv = read_command_line ();
argv = parse_options (argv);
```

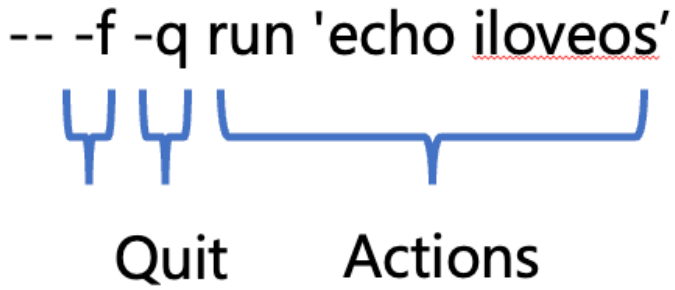
```
printf ("Boot complete.\n");

if (*argv != NULL) {
  /* Run actions specified on kernel command line. */
  run_actions (argv);
} else {
  // TODO: no command line passed to kernel. Run interactively
}
```

Lab2 User Program

Lab0 Shell

Still parsing…



run_actions():

```c
/* An action. */
struct action
  {
    char *name;
    int argc;
    void (*function) (char **argv);
  };
```

```c
  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
      {"cat", 2, fsutil_cat},
      {"rm", 2, fsutil_rm},
      {"extract", 1, fsutil_extract},
      {"append", 2, fsutil_append},
#endif
      {NULL, 0, NULL},
    };
```

Actually we have a stage named **extract** before **run**: It moves ELF into Pintos FS. Figure it out yourself!

The Journey Starts from process_execute()

run_task():

```
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

Q1: What is ARGV[0] ?

ARGV[0] = run

Q2: What does run_test() do ?

run_test() run the test cases in lab1

After process_execute(): process_wait()

```c
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

Now: Return Immediately !!
Your Task: Make It Work

```c
int
process_wait (tid_t child_tid UNUSED)
{
  return -1;
}
```

A Closer Look to process_execute

It just adds a new thread into the ready_list.

调用链

```
process_execute("echo iloveos")
  → thread_create("echo iloveos", PRI_DEFAULT, start_process, fn_copy)
      → 新线程被创建，入口函数设为 start_process
      → thread_unblock(t)  // 放入 ready_list
  → 返回 tid

// ... 调度器某一时刻切换到新线程 ...

新线程开始执行 → kernel_thread() → start_process(fn_copy)
```

```
// ... 调度器某一时刻切换到新线程 ...
新线程开始执行 → kernel_thread() → start_process(fn_copy)
```

start_process:

```
/* Initialize interrupt frame and load executable. */
memset (&if_, 0, sizeof if_);
if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
if_.cs = SEL_UCSEG;
if_.eflags = FLAG_IF | FLAG_MBS;
success = load (file_name, &if_.eip, &if_.esp);
```

- Load the ELF file from Disk into memory
- We are still in the kernel !!
- Initialize **Interrupt Frame** (eip, esp, segment registers, eflags)
- Start the user process by **simulating a return from an interrupt**

```
asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
```

https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

How to load elf: You must have a pagetable first.

核心就三步：**建房子 → 搬进去 → 装修**

1. 建房子： `pagedir_create()`

分配一页内存做新的页目录，然后把内核的页目录 `init_page_dir` 整个拷贝过来。

这样新页目录**天生就有内核映射**（`0xc0000000` 以上），但用户空间（`0x00000000 ~ 0xbfffffff`）还是空的。

2. 搬进去： `process_activate()`

把新页目录的物理地址写入 CPU 的 **CR3 寄存器**。从此 CPU 按新页目录翻译地址。

因为内核映射已经拷贝过来了，所以切换后内核代码照常跑。

3. 装修： `load_segment()` + `setup_stack()`

逐页把 ELF 文件的内容加载到用户空间：

对每一页：
  分配物理页 → 读入 ELF 数据 → 在页目录中建立映射（用户虚拟地址 → 物理页）

最后 `setup_stack()` 在 `0xc0000000` 正下方映射一页做用户栈。

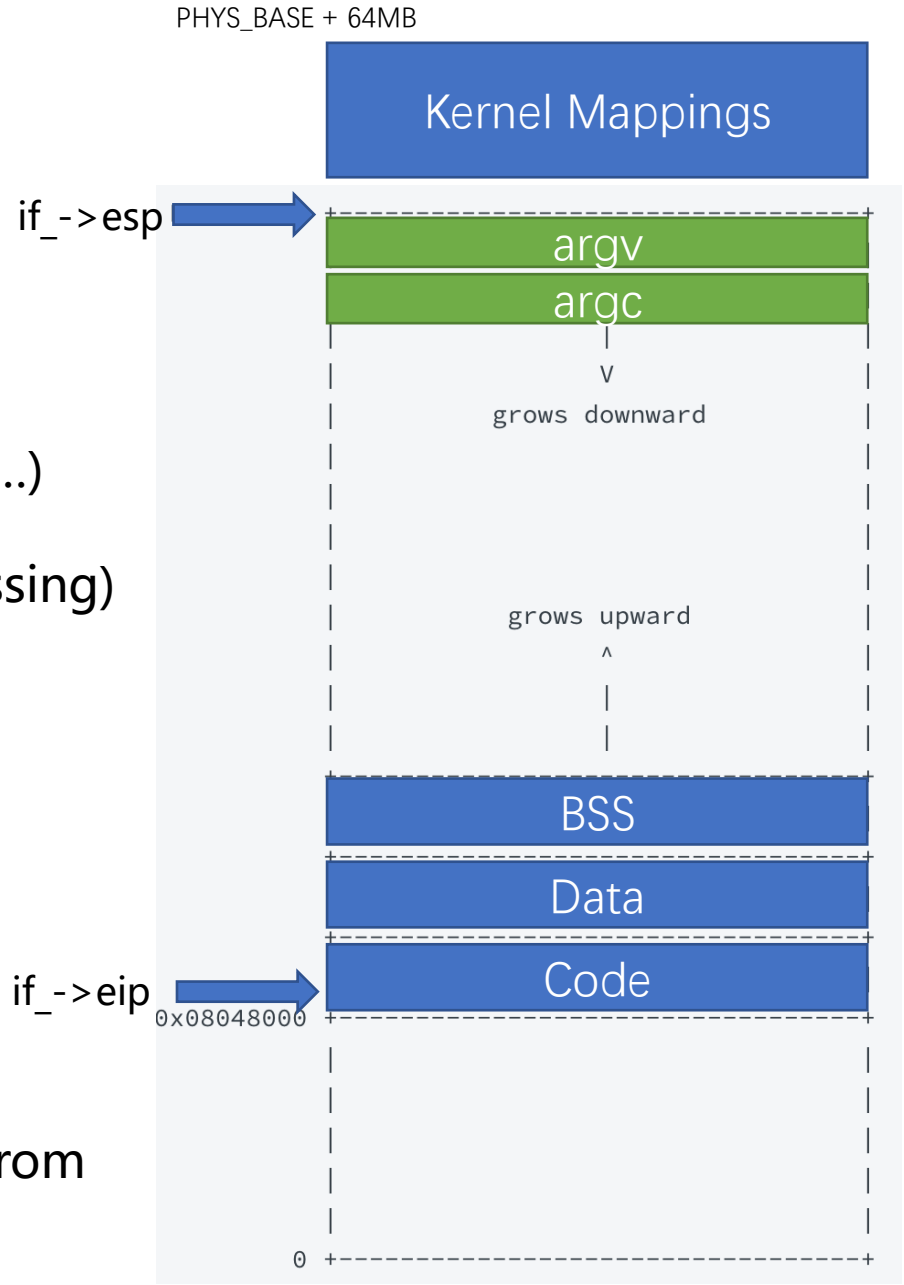
load:

- Read and verify ELF executable header
- Read ELF program header and load segments (code, data ……)
- Set up if_->stack (You will fix this in Exercise2: Argument Passing)
- Set up if_->eip with the entry point in executable header

/src/lib/user/entry.c:

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

- After loading, start the user process by simulating a return from an interrupt with interrupt frame if_



Virtual Address Space

Wow, your process is running in User Space!

- But, we want system call support !!

/src/lib/user/syscall.h:

```
void
halt (void)
{
  syscall0 (SYS_HALT);
  NOT_REACHED ();
}
```

/src/lib/user/syscall.c:

```
/* Invokes syscall NUMBER, passing no arguments, and returns the
   return value as an `int'. */
#define syscall0(NUMBER)                                        \
        ({                                                      \
          int retval;                                           \
          asm volatile                                          \
            ("pushl %[number]; int $0x30; addl $4, %%esp"       \
               : "=a" (retval)                                  \
               : [number] "i" (NUMBER)                          \
               : "memory");                                     \
          retval;                                               \
        })
```

Wow, your process is running in User Space!

- But, we want system call support !!

/src/lib/user/syscall.h:

```
void
exit (int status)
{
  syscall1 (SYS_EXIT, status);
  NOT_REACHED ();
}
```

/src/lib/user/syscall.c:

```
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)                                           \
        ({                                                               \
          int retval;                                                    \
          asm volatile                                                   \
            ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
               : "=a" (retval)                                           \
               : [number] "i" (NUMBER),                                  \
                 [arg0] "g" (ARG0)                                       \
               : "memory");                                              \
          retval;                                                        \
        })
```

System Call Numbers:

/src/lib/syscall-nr.h

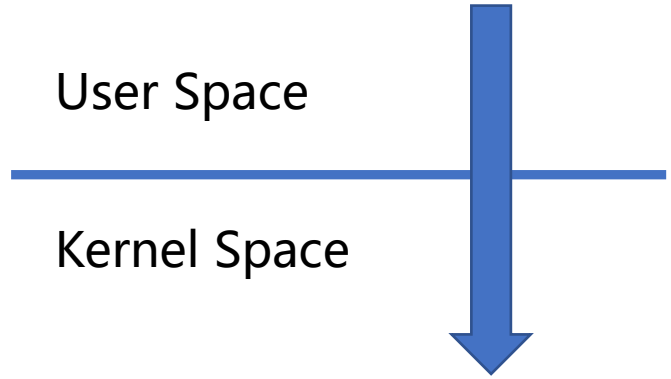
```
/* System call numbers. */
enum
  {
    /* Projects 2 and later. */
    SYS_HALT,                   /* Halt the operating system. */
    SYS_EXIT,                   /* Terminate this process. */
    SYS_EXEC,                   /* Start another process. */
    SYS_WAIT,                   /* Wait for a child process to die. */
    SYS_CREATE,                 /* Create a file. */
    SYS_REMOVE,                 /* Delete a file. */
    SYS_OPEN,                   /* Open a file. */
    SYS_FILESIZE,               /* Obtain a file's size. */
    SYS_READ,                   /* Read from a file. */
    SYS_WRITE,                  /* Write to a file. */
    SYS_SEEK,                   /* Change position in a file. */
    SYS_TELL,                   /* Report current position in a file. */
    SYS_CLOSE,                  /* Close a file. */

    /* Project 3 and optionally project 4. */
    SYS_MMAP,                   /* Map a file into memory. */
    SYS_MUNMAP,                 /* Remove a memory mapping. */

    /* Project 4 only. */
    SYS_CHDIR,                  /* Change the current directory. */
    SYS_MKDIR,                  /* Create a directory. */
    SYS_READDIR,                /* Reads a directory entry. */
    SYS_ISDIR,                  /* Tests if a fd represents a directory. */
    SYS_INUMBER                 /* Returns the inode number for a fd. */
  };
```

Now, all the magic is behind `int 0x30`

```
/* Invokes syscall NUMBER, passing no arguments, and returns the
   return value as an `int'. */
#define syscall0(NUMBER)                                        \
        ({                                                      \
          int retval;                                           \
          asm volatile                                          \
            ("pushl %[number]; int $0x30; addl $4, %%esp"       \
               : "=a" (retval)                                  \
               : [number] "i" (NUMBER)                          \
               : "memory");                                     \
          retval;                                               \
        })
```



Interrupt Handler

- save the context of the interrupted

/src/threads/interrupt.c:

```
void
intr_handler (struct intr_frame *frame)
```

```
/* Invoke the interrupt's handler. */
handler = intr_handlers[frame->vec_no];
if (handler != NULL)
  handler (frame);
```

/src/user

```
void
syscall_init
{
  intr_regist
}
```

```
static void
syscall_hand
{
  printf ("sy
  thread_exi
}
```

Implemen

```
/* Interrupt stack frame. */
struct intr_frame
  {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t edi;               /* Saved EDI. */
    uint32_t esi;               /* Saved ESI. */
    uint32_t ebp;               /* Saved EBP. */
    uint32_t esp_dummy;         /* Not used. */
    uint32_t ebx;               /* Saved EBX. */
    uint32_t edx;               /* Saved EDX. */
    uint32_t ecx;               /* Saved ECX. */
    uint32_t eax;               /* Saved EAX. */
    uint16_t gs, :16;           /* Saved GS segment register. */
    uint16_t fs, :16;           /* Saved FS segment register. */
    uint16_t es, :16;           /* Saved ES segment register. */
    uint16_t ds, :16;           /* Saved DS segment register. */

    /* Pushed by intrNN_stub in intr-stubs.S. */
    uint32_t vec_no;            /* Interrupt vector number. */

    /* Sometimes pushed by the CPU,
       otherwise for consistency pushed as 0 by intrNN_stub.
       The CPU puts it just under `eip', but we move it here. */
    uint32_t error_code;        /* Error code. */

    /* Pushed by intrNN_stub in intr-stubs.S.
       This frame pointer eases interpretation of backtraces. */
    void *frame_pointer;        /* Saved EBP (frame pointer). */

    /* Pushed by the CPU.
       These are the interrupted task's saved registers. */
    void (*eip) (void);         /* Next instruction to execute. */
    uint16_t cs, :16;           /* Code segment for eip. */
    uint32_t eflags;            /* Saved CPU flags. */
    void *esp;                  /* Saved stack pointer. */
    uint16_t ss, :16;           /* Data segment for esp. */
  };
```

Contents

- Pintos Disk and File System
- System Call
- Interrupt Handling
- Lab2 tasks and suggestions

Some useful tips:

- Pintos exec == Unix fork + exec
- You can use malloc in kernel (#include “threads/malloc.h”)
- Useful GDB command: loadusersymbols
- Reference to xv6 implementation
- multi-oom testcase will take some time, be patient

Suggested Order of Implementation:

- 1. **Argument Passing** — push argc/argv onto user stack after load(); pass args-xxx tests
- 2. **Halt System Call** — build syscall_handler() dispatch infrastructure
- 3. **Temporal Workarounds** — bare exit + write(fd=1) + process_wait() infinite loop → programs can run & output
- 4. **Accessing User Memory** — validate user pointers before use; kill process on bad access, don't crash kernel
- 5. **Process Control Syscalls** — exit/exec/wait, design together (shared data structures)
- 6. **File System Syscalls** — per-process fd table, global file lock; read file.c & filesys.c interfaces
- 7. **Deny Writes to Executables** — file_deny_write() in load(), keep file open until process exits

- Refer to https://echostone.gitbook.io/pintos/project-description/lab2-user-programs/suggestions

Enjoy Your Pintos Journey ~~

Any Problem ?