CS202 (003): Operating Systems
Putting Everything Together

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last time

Loading and Executing Programs

What happens when a program executes the following code?

```
char *argv[];
char *envp[];
// Initialize argv and envp
// ...

if (fork() == 0) {
  // Executed in the child process.
  execve("hello", argv, envp);
}
```

Loading and Executing Programs

What happens when a program executes the following code?

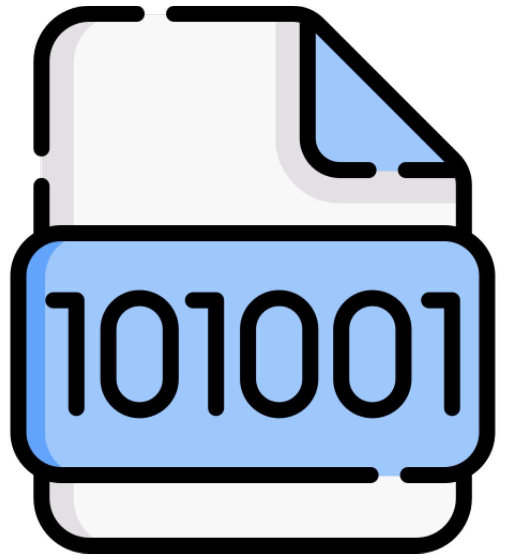
```c
char *argv[];
char *envp[];
// Initialize argv and envp
// ...

if (fork() == 0) {
  // Executed in the child process.
  execve("hello", argv, envp);
}
```

Source Code

statically-linked!

`gcc hello.c -o hello static`



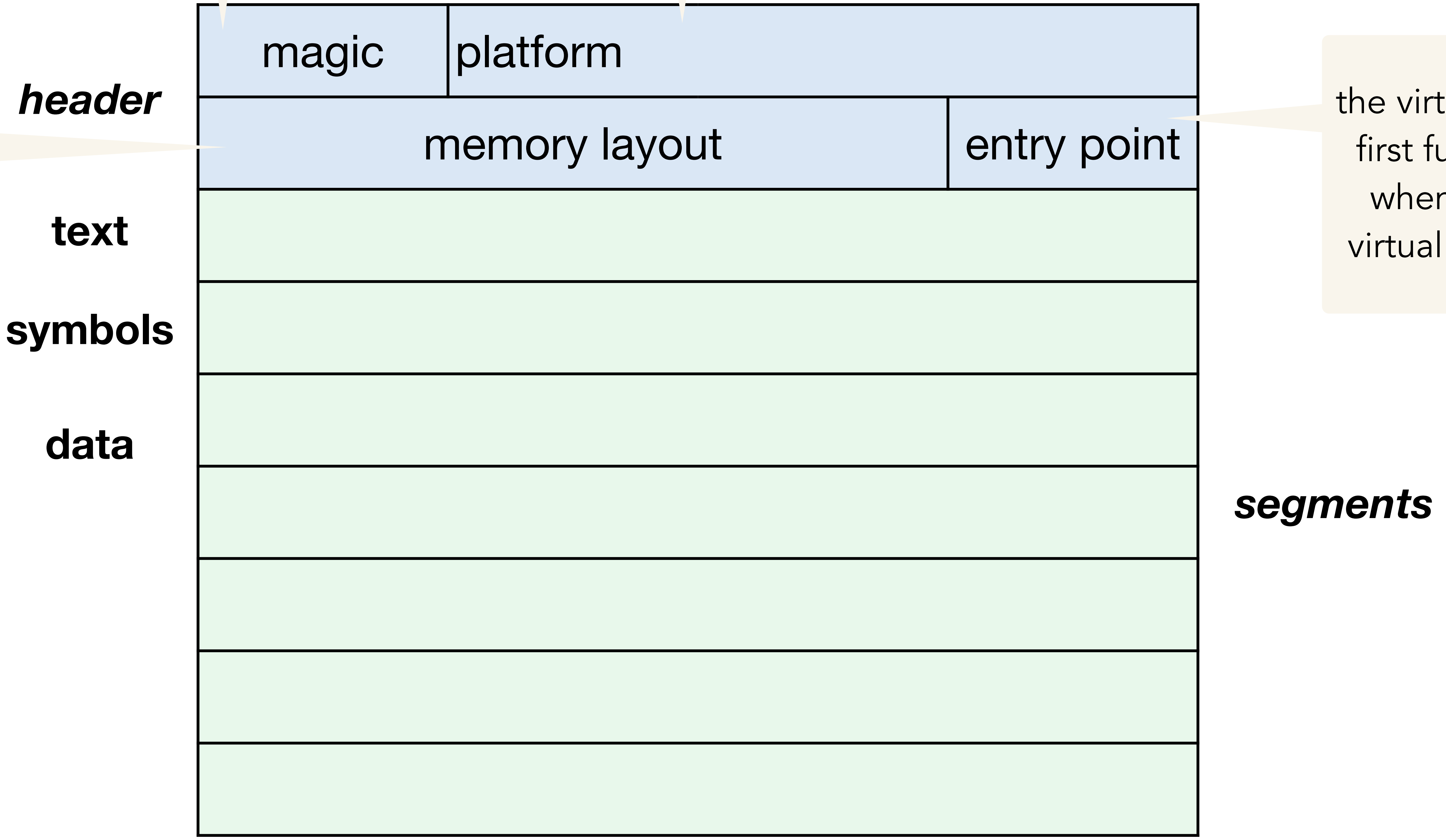
Linux Executable: Executable and Linkable Format (ELF)
Windows Executable:Portable Executable (PE)

What is inside an executable?

a known byte sequence at the beginning of the file that identifies the file as an executable.

The platform (operating system and machine architecture) on which the executable can be run.

An array of sections.
Each section is a struct specifying:
- The virtual address at which the section should be placed. The compiler **assumes** that it knows the virtual address for instructions and variables when compiling.
- Whether the contents of the memory should be read from the file or left uninitialized.
- The offset in the file where the contents of the section are stored.
- The memory protection bits that should be set for the section.



the virtual memory address at which the first function that should be executed when the program is run, that is the virtual address of the `_start` functions

More details on the execve

```
char *argv[];
char *envp[];
// Initialize argv and envp
// ...

if (fork() == 0) {
  // Executed in the child process.
  execve("hello", argv, envp);
}
```

- The loader checks that the file is readable and can be executed (permissions, magic number, header/platform)
- The loader calls `munmap` to `unmap` the process's memory.
- The loader reads through the executable's memory layout array and uses mmap to allocate and set up the processes memory layout:
  - Use the executable's file descriptor when allocating sections that need to be read from the executable file.
  - Use `MAP_ANONYMOUS` for any sections that do not need to be read from the executable.
  - Set protection bits based on information in the section.
- The loader uses `mmap` to allocate a stack, and sets `%RSP` and `%RBP` so they point to the top of the allocated stack. The loader then copies `argv` to the stack.
- The loader then sets `%RDI` to `argc` and `%RSI` to point to `argv`, and jumps to the entry point specified in the executable.

Power up to Terminal

Step 1: Power up

Step 2: Firmware

Step 3: OS bootloader to Kernel

Step 4: Kernel

Step 5: init

Step 6: login(1)

Step 1: Power up

Processor Initialization

- Zero out registers
- Set control registers to default values (Intel defaults in Software Development Manual)
- **Enter Real Mode:**
  - No paging - all addresses are physical
  - Up to 1MB physical memory access

Firmware Loading

- Processor copies executable from ROM to RAM
- Jumps to known offset (historically 0xFFFF0)
- **Modern Firmware:**
  - Stored on EEPROMs/Flash for upgrades
  - Settings stored in battery-backed CMOS

Step 2: Firmware

Firmware

Responsible for hardware initialization and providing a runtime for the kernel during early boot
On recent Intel machines, firmware can be broadly classified as either BIOS or **UEFI** firmwares.
Both are specifications, and many different implementations exist for both

UEFI Initialization Steps

- Switch to Long Mode
  - Enable paging & 64-bit addressing
  - Create identity mapped page table
  - Install IDT for interrupts
  - Initialize processor structures
- Initialize Devices
  - Disks, USB, Display, Input devices
  - Network cards and peripherals
- Mount VFAT partition & load OS bootloader

Key Components

- UEFI Services
  - Network communication
  - File operations
  - Display & input handling
- Device Tree (CONFIGURATION_TABLE)
  - Lists all connected devices
  - Specifies I/O methods & addresses
  - Maps interrupt routing
-

Note: UEFI (Unified Extensible Firmware Interface) handles hardware initialization and provides runtime for early kernel boot

Step 3: OS bootloader to Kernel

The UEFI firmware loads and executes the OS bootloader.

On recent Linux kernels the bootloader is the vmlinuz file with a stub for UEFI, we only consider this case here.

vmlinuz Structure

1

PE Header + EFI stub

2

ELF header + decompression stub

3

Compressed kernel data (bzip2/gzip)

Execution Flow

1. EFI Stub

Loads and executes decompression stub

2. Decompression

Uncompresses kernel data into memory

3. Kernel Launch

Executes kernel with CONFIGURATION_TABLE pointer

4. Firmware Exit

Kernel terminates UEFI firmware

Note: vmlinuz requires root directory device ID as argument (root=<device>)

Step 4: Kernel

Memory Management

Switch from identity-mapped virtual address space

Interrupt Handling

Rewrite interrupt descriptor table

Device Management

- Load and initialize device drivers
- Use CONFIGURATION_TABLE information
- Drivers run as part of kernel
- Communication via /dev files

Root Device

- Mount root device
- Run fsck if required

Final Step

Fork and launch init process

Step 5: init

Executed as root

System Initialization

- Device Configuration
  - Network IP (DHCP)
  - GPU resolution
  - Power management
- Communication via /dev

Daemon Management

- Launch system services (sshd, httpd)
- Handle port binding (1-1024)
- Manage privileges
- Monitor & restart on failures

Session Management

- Launch login manager (login(1))
- Handle user sessions
- Restart on user logout

Note: Most distributions now use systemd, but this represents a simpler init.rc-like implementation

Step 6: login(1)

Must run as root

Authentication

- Prompt for username and password
- Verify against:
  - /etc/passwd
  - /etc/shadow

Login Sequence

1. Fork new process

   Parent waits for child exit
2. Set user permissions
   - setuid(2) for user ID
   - setgid(2) for group ID
3. Change to user's home directory
4. Launch login shell (e.g., bash)

Logout Process

1. User kills shell process → Parent login process exits
2. init detects exit (via wait) → Starts new login process

Remarks

Two operating systems

Firmware: a simple operating system providing a few services. It does not support multiple processes, and has only limited functionality.

Kernel: a richer set of functionality, including schedulers, etc.

Different architectures

Monolithic Kernels : Device drivers are a part of the kernel (like in Linux)

Microkernels: Device drivers and many other portions are run as independent processes

https://cs.nyu.edu/~mwalfish/classes/ut/s10-cs372h/ref/ast-torvalds.html

Final Exam Logistics

Happens on 5/9 12-1:50pm (110 mins) at WW 312

Closed book, 1 letter-sized double-sided cheat sheet allowed

(You must write/type the cheatsheet yourself)

Format similar to the midterm exam

Everything we covered in this semester might show up in exam

Bring your ID, Any electronics NOT allowed

Review session next Tuesday!