CS 5600
Computer Systems

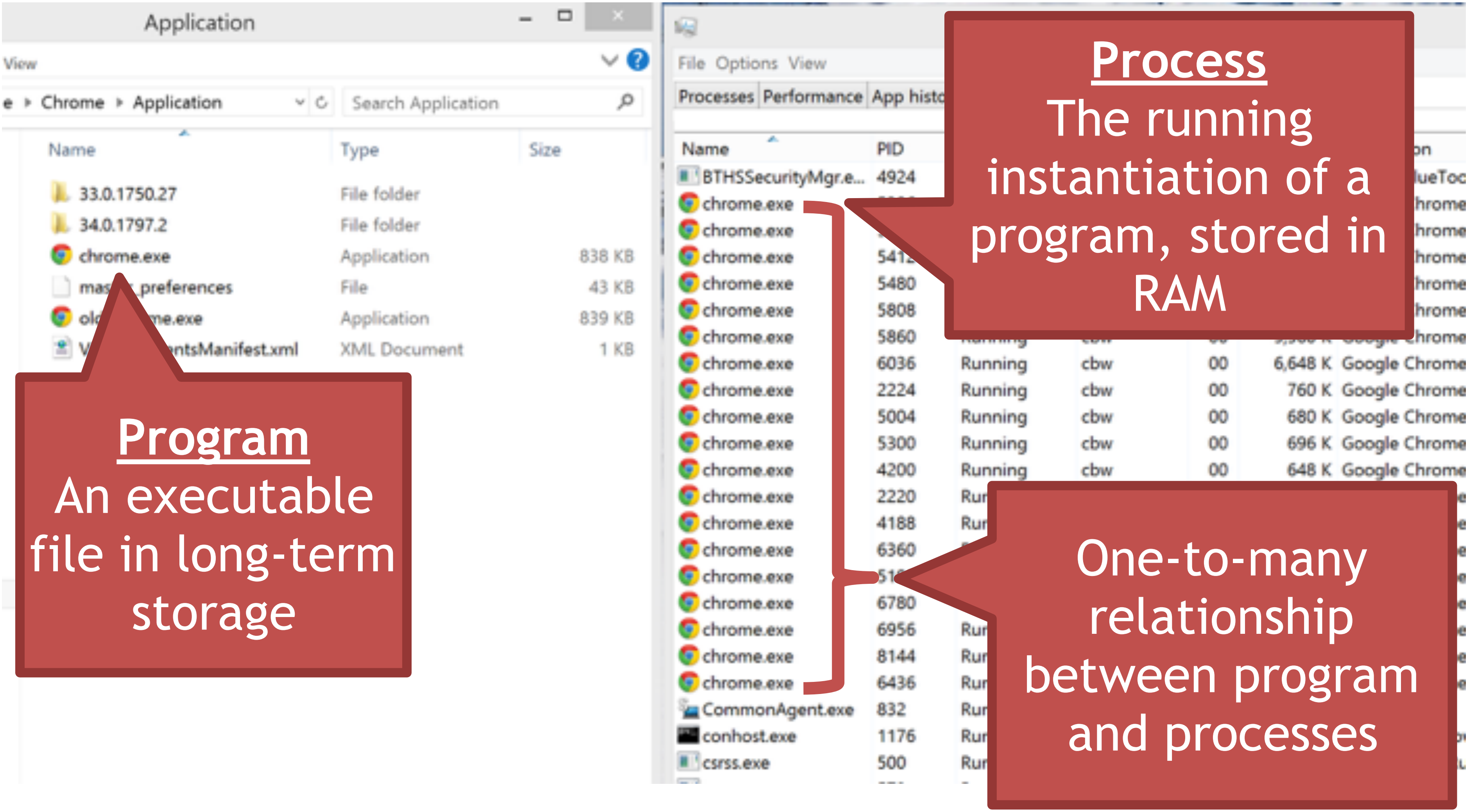
Lecture 4: Programs, Processes, and Threads

- Programs
- Processes
- Context Switching
- Protected Mode Execution
- Inter-process Communication
- Threads

Running Dynamic Code

- One basic function of an OS is to execute and manage code dynamically, e.g.:
  - A command issued at a command line terminal
  - An icon double clicked from the desktop
  - Jobs/tasks run as part of a batch system (MapReduce)
- A process is the basic unit of a program in execution

Programs and Processes



Name	Type	Size
33.0.1750.27	File folder	
34.0.1797.2	File folder	
chrome.exe	Application	838 KB
mas preferences	File	43 KB
old me.exe	Application	839 KB
V ntsManifest.xml	XML Document	1 KB

Name	PID					on
BTHSSecurityMgr.e...	4924					lueToo
chrome.exe						hrome
chrome.exe						hrome
chrome.exe	541					hrome
chrome.exe	5480					hrome
chrome.exe	5808					hrome
chrome.exe	5860					hrome
chrome.exe	6036	Running	cbw	00	6,648 K	Google Chrome
chrome.exe	2224	Running	cbw	00	760 K	Google Chrome
chrome.exe	5004	Running	cbw	00	680 K	Google Chrome
chrome.exe	5300	Running	cbw	00	696 K	Google Chrome
chrome.exe	4200	Running	cbw	00	648 K	Google Chrome
chrome.exe	2220	Ru				
chrome.exe	4188	Ru				
chrome.exe	6360					
chrome.exe	51					
chrome.exe	6780					
chrome.exe	6956	Ru				
chrome.exe	8144	Ru				
chrome.exe	6436	Ru				
CommonAgent.exe	832	Ru				
conhost.exe	1176	Ru				
csrss.exe	500	Ru				

Process
The running instantiation of a program, stored in RAM

Program
An executable file in long-term storage

One-to-many relationship between program and processes

How to Run a Program?

- When you double-click on an .exe, how does the OS turn the file on disk into a process?

- What information must the .exe file contain in order to run as a program?

Program Formats

- Programs obey specific file formats
  - CP/M and DOS: COM executables (*.com)
  - DOS: MZ executables (*.exe)
    - Named after Mark Zbikowski, a DOS developer
  - Windows Portable Executable (PE, PE32+) (*.exe)
    - Modified version of Unix COFF executable format
    - PE files start with an MZ header.
  - Mac OSX: Mach object file format (Mach-O)
  - Unix/Linux: Executable and Linkable Format (ELF)
    - designed to be flexible and extensible
    - all you need to know to load and start execution regardless of architecture

ABI - Application Binary Interface

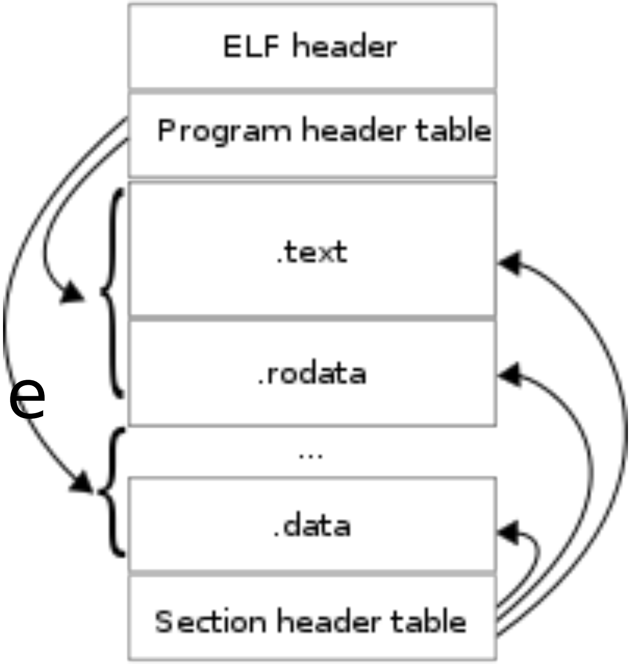
- interface between 2 programs at the binary (machine code) level
  - informally, similar to API but on bits and bytes
- Calling conventions
  - where are args and results stored
- Binary format info to be passed from one program to another
- Compiler and OS take care of this
  - binaries created from different compiler-OS pair will not always run on your machine!

test.c

```c
#include <stdio.h>

int big_big_array[10 * 1024 * 1024];
char *a_string = "Hello, World!";
int a_var_with_value = 100;

int main(void) {
    big_big_array[0] = 100;
    printf("%s\n", a_string);
    a_var_with_value += 20;

    printf("main is : %p\n", &main);
    return 0;
}
```

ELF File Format

- ELF Header
  - Contains compatibility info
  - Entry point of the executable code
- Program header table
  - Lists all the segments in the file
  - Used to load and execute the program
- Section header table
  - Used by the linker



ELF Header Format

```
typedef struct {
        unsigned char e_ident[EI_NIDENT];
        Elf32_Half e_type;
        Elf32_Half e_machine;
        Elf32_Word e_version;
        Elf32_Addr e_entry;
        Elf32_Off e_phoff;
        Elf32_Off e_shoff;
        Elf32_Word e_flags;
        Elf32_Half e_ehsize;
        Elf32_Half e_phentsize;
        Elf32_Half e_phnum;
        Elf32_Half e_shentsize;
        Elf32_Half e_shnum;
        Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

ISA of executable code

- Entry point of executable code
- What should EIP be set to initially?

# of program headers

# of section headers

ELF Header Example

```
$ gcc –g –o test test.c
$ readelf --header test
ELF Header:
 Magic:                          7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                         ELF64
  Data:                          2's complement, little endian
  Version:                       1 (current)
  OS/ABI:                        UNIX - System V
  ABI Version:                   0
  Type:                          EXEC (Executable file)
  Machine:                       Advanced Micro Devices X86-64
  Version:                       0x1
  Entry point address:                     0x400460
  Start of program headers:                64 (bytes into file)
  Start of section headers:                5216 (bytes into file)
  Flags:                         0x0
  Size of this header:             64 (bytes)
  Size of program headers:                 56 (bytes)
  Number of program headers:               9
  Size of section headers:                 64 (bytes)
  Number of section headers:               36
  Section header string table index:       33
```

Investigating the Entry Point

```
int main(void) {
        ...
            printf("main is : %p\n", &main);
            return 0;
}
```

```
$ gcc -g -o test test.c
$ readelf --headers ./test | grep Entry
   Entry point address:               0x400460
$ ./test
  Hello World!
  main is : 0x400544
```

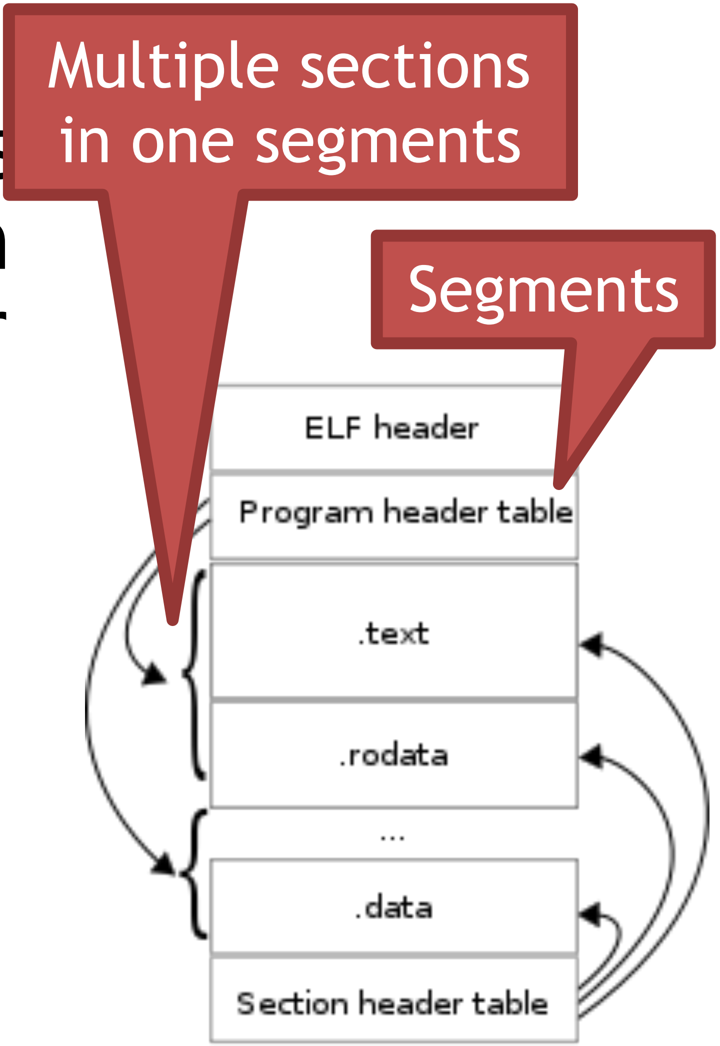
Entry point != &main

- Most compilers insert extra code into compiled programs
- This code typically runs before and after main()

```
$ ./test
  Hello World!
  main is : 0x400544
$ readelf --headers ./test | grep Entry
   Entry point address:              0x400460
$ objdump --disassemble –M intel ./test
…
0000000000400460 <_start>:
 400460: 31 ed                  xor    ebp,ebp
 400462: 49 89 d1               mov    r9,rdx
 400465: 5e                     pop    rsi
 400466: 48 89 e2               mov    rdx,rsp
 400469: 48 83 e4 f0            and    rsp,0xfffffffffffffff0
 40046d: 50                     push   rax
 40046e: 54                     push   rsp
 40046f: 49 c7 c0 20 06 40 00       mov    r8,0x400620
 400476: 48 c7 c1 90 05 40 00       mov    rcx,0x400590
 40047d: 48 c7 c7 44 05 40 00       mov    rdi,0x400544
 400484: e8 c7 ff ff ff         call   400450 <__libc_start_main@plt>
…
```

Sections and Segments

- Sections are the various pieces of code and data that get linked together by the compiler
- Each segment contains one or more sections
  - Each segment contains sections that are related
    - E.g. all code sections
  - Segments are the basic units for the loader



Multiple sections in one segments

Segments

Common Sections

- Sections are the various pieces of code and data that compose a program
- Key sections:
  - .text – Executable code
  - .bss – Global variables initialized to zero
  - .data, .rodata – Initialized data and strings
  - .strtab – Names of functions and variables
  - .symtab – Debug symbols

Section Example

String variable → .data

Empty 10 MB array → .bss

```
int big_big_array[10*1024*1024];
char *a_string = "Hello, World!";
int a_var_with_value = 0x100;

int main(void) {
        big_big_array[0] = 100;
        printf("%s\n", a_string);
        a_var_with_value += 20;
        ...
}
```

Initialized global variable → .data

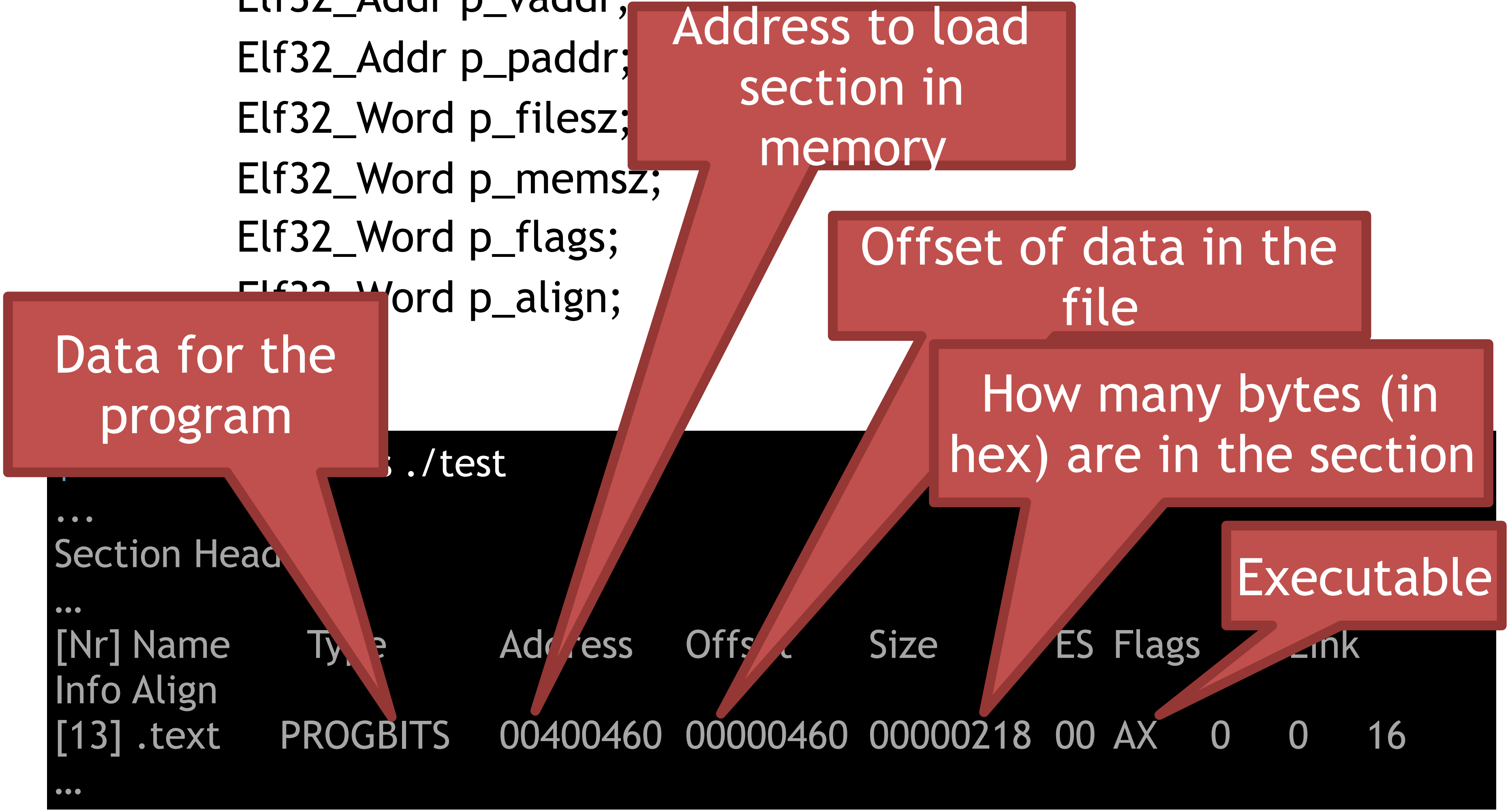
String constant → .rodata

Code → .text

```
$ readelf --headers ./test
...
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-
id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini
.rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .ctors .dtors .jcr .dynamic .got
...
There are 36 section headers, starting at offset 0x1460:
Section Headers:
[Nr] Name              Type         Address  Offset   Size     ES  Flags Link Info Align
[ 0]                   NULL         00000000 00000000 00000000 00        0    0    0
[ 1] .interp           PROGBITS     00400238 00000238 0000001c 00  A     0    0    1
[ 2] .note.ABI-tag     NOTE         00400254 00000254 00000020 00  A     0    0    4
[ 3] .note.gnu.build-I              NOTE     00400274 00000274 0000002400  A    0       0
     4
[ 4] .gnu.hash         GNU_HASH     00400298 00000298 0000001c  00 A     5    0    8
[ 5] .dynsym           DYNSYM       004002b8 000002b8 00000078 18  A     6    1    8
[ 6] .dynstr           STRTAB       00400330 00000330 00000044 00  A     0    0    1
[ 7] .gnu.version      VERSYM       00400374 00000374 0000000a 02  A     5    0    2
```

.text Example Header

```
typedef struct {
            Elf32_Word p_type;
            Elf32_Off  p_offset;
            Elf32_Addr p_vaddr;
            Elf32_Addr p_paddr;
            Elf32_Word p_filesz;
            Elf32_Word p_memsz;
            Elf32_Word p_flags;
            Elf32_Word p_align;
```



Address to load section in memory

Offset of data in the file

Data for the program

How many bytes (in hex) are in the section

Executable

```
                  ./test
...
Section Head
...
[Nr] Name       Ty  e        Ad   ess    Offs  t      Size       ES  Flags        ink
Info Align
[13] .text      PROGBITS     00400460  00000460  00000218  00  AX     0    0    16
...
```

.bss Example Header

```
int big_bi
```

```
ypedef struct {
        Elf32_Word p_type;
        Elf32_Off  p_offset;
        Elf32_Addr p_vaddr;
        Elf32_Addr p_paddr;
        Elf32_Word p_filesz;
        Elf32_Word p_memsz;
        Elf32_Word p_flags;
        Elf32_Word p_align;
```

Offset of data in the file

Address to load section in memory

Contains no data

hex(4*10*1024*1024) = 0x2800020

Writable

```
$ readelf -  ctions ./test
...
Section Heade  :
...
[Nr]  Name       Type       Address    Offse     Size       ES  Flags      Link  Info  Align
[25]  .bss        NOBITS    00601040   00001034  02800020   00  WA     0     0     32
[26]  .comment   PROGBITS   00000000   00001034  000002a    01  MS     0     0     1
...
```

Segments

- Each segment contains one or more sections
  - All of the sections in a segment are related, e.g.:
    - All sections contain compiled code
    - Or, all sections contain initialized data
    - Or, all sections contain debug information
    - … etc…
- Segments are used by the loader to:
  - Place data and code in memory
  - Determine memory permissions (read/write/execute)

Segment Header

```
typedef struct {
        Elf32_Word p_type
        Elf32_Off  p_offset;
        Elf32_Addr p_vaddr;
        Elf32_Addr p_paddr;
        Elf32_Word p_filesz;
        Elf32_Word p_memsz;
        Elf32_Word p_flags;
        Elf32_Word p_ali
}
```

Type of segment

Offset within the ELF

f

Location to load the segment into memory

Size of the segment in memory

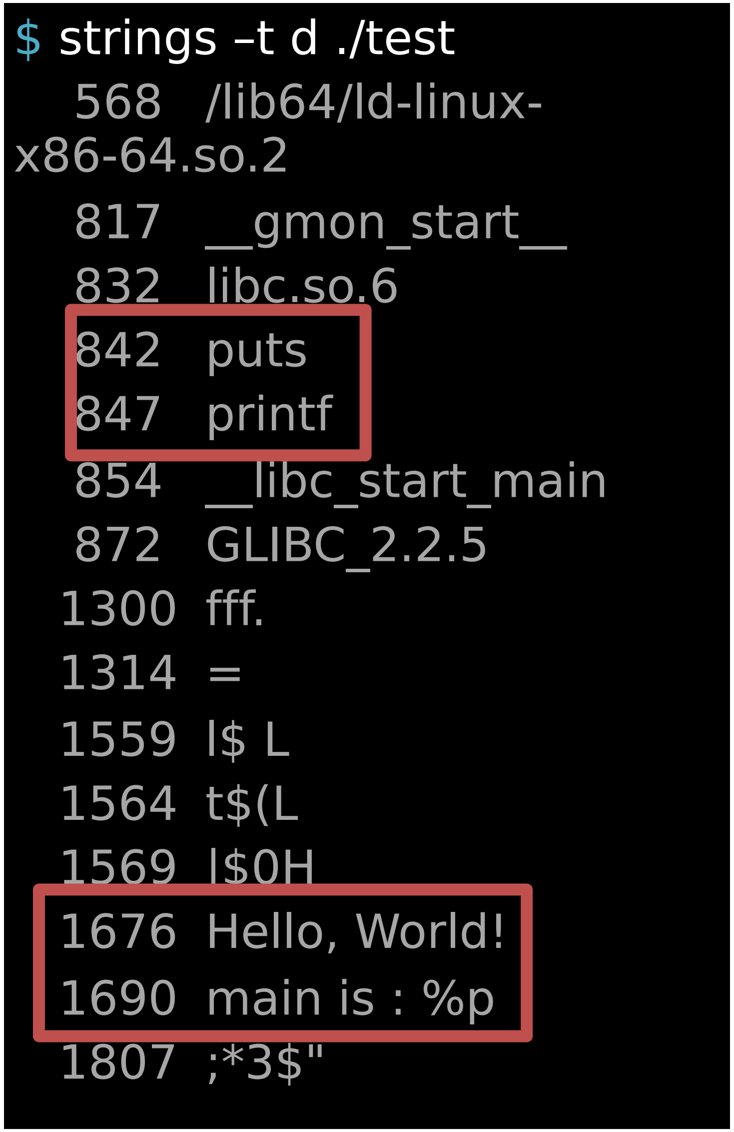
- Flags describing the section data
- Examples: executable, read-only

Executable

```
$ readelf --segments ./test
Elf file type is EXEC (Executable file)
Entry point 0x400460
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags   lign
  PHDR           0x00000040 0x00400040 0x00400040 0x000001f8 0x000001f8 R E     8
  INTERP         0x00000238 0x00400238 0x00400238 0x0000001c 0x0000001c R       1
  LOAD           0x00000000 0x00400000 0x00400000 0x0000077c 0x0000077c R E     200000
  LOAD           0x00000e28 0x00600e28 0x00600e28 0x0000020c 0x02800238 RW      200000
  DYNAMIC        0x00000e50 0x00600e50 0x00600e50 0x00000190 0x00000190 RW      8
  NOTE           0x00000254 0x00400254 0x00400254 0x00000044 0x00000044 R       4
  GNU_EH_FRAME   0x000006a8 0x004006a8 0x004006a8 0x0000002c 0x0000002c R       4
  GNU_STACK      0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 RW      8
  GNU_RELRO      0x00000e28 0x00600e28 0x00600e28 0x000001d8 0x000001d8 R       1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-
id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .e
h_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
...
```

What About Static Data?

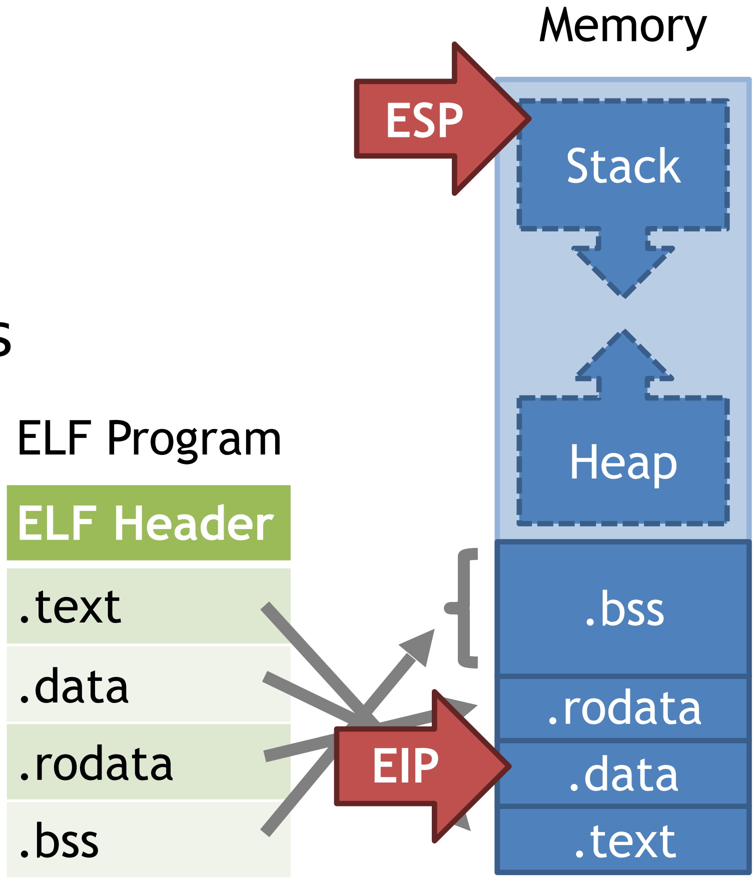
```c
#include <stdio.h>

int big_big_array[10 * 1024 * 1024];
char *a_string = "Hello, World!";
int a_var_with_value = 100;

int main(void) {
    big_big_array[0] = 100;
    printf("%s\n", a_string);
    a_var_with_value += 20;

    printf("main is : %p\n", &main);
    return 0;
}
```



```
$ strings –t d ./test
   568   /lib64/ld-linux-x86-64.so.2
   817   __gmon_start__
   832   libc.so.6
   842   puts
   847   printf
   854   __libc_start_main
   872   GLIBC_2.2.5
  1300  fff.
  1314  =
  1559  l$ L
  1564  t$(L
  1569  l$0H
  1676  Hello, World!
  1690  main is : %p
  1807  ;*3$"
```

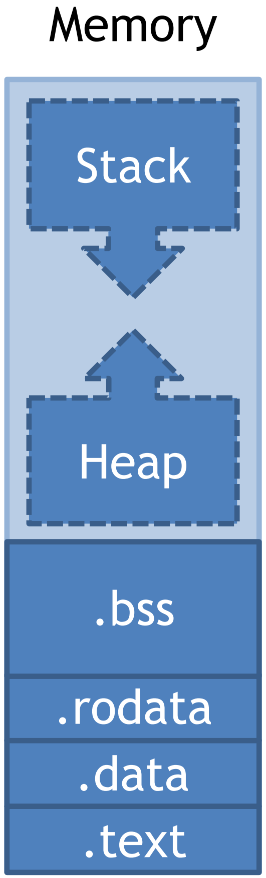
The Program Loader

- OS functionality that loads programs into memory, creates processes
  - Places segments into memory
    - Expands segments like .bss
  - Loads necessary dynamic libraries
  - Performs relocation
  - Allocates the initial stack frame
  - Sets EIP to the programs entry point



Single-Process Address Apace

- The stack is used for local variables and function calls
  - Grows downwards
- Heap is allocated dynamically (malloc/new)
  - Grows upwards
- When the stack and heap meet, there is no more memory left in the process
  - Process will probably crash
- Static data and global variables are fixed at compile time



Problem: Pointers in Programs

- Consider the following code:

```
int foo(int a, int b) { return a * b – a / b; }
int main(void) { return foo(10, 12); }
```

- Compiled, it might look like this:

```
000FE4D8 <foo>:
000FE4D8:  mov eax, [esp+4]
000FE4DB:  mov ebx, [esp+8]
000FE4DF:  mul eax, ebx
...
000FE21A:  push eax
000FE21D:  push ebx
000FE21F:  call 0x000FE4D8
```

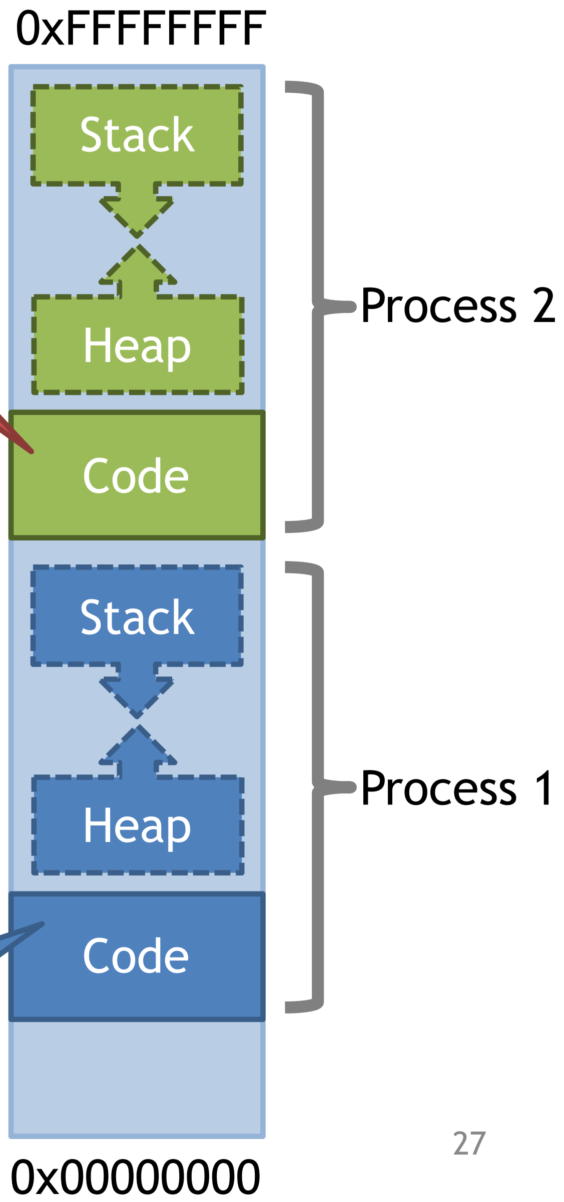
- … but this assembly assumes foo() is at address 0x000FE4D8

Program Load Addresses

- Loader must place each process in memory
- Program may not be placed at the correct location!
  - Example: two copies of the same prog

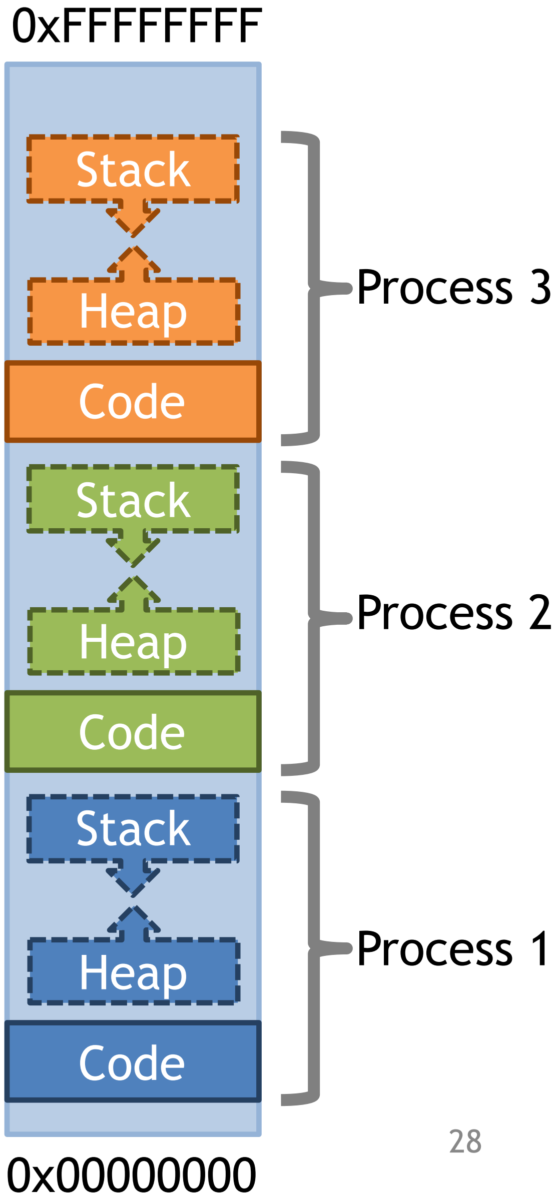
Addr of foo(): 0x0DEB49A3

Addr of foo(): 0x000FE4D8



Address Spaces for Multiple Processes

- Many features of processes depend on pointers
  - Addresses of functions
  - Addresses of strings, data
  - Etc.
- For multiple processes to run together, they all have to fit into memory together
- However, a process may not always be loaded into the same memory location



Address Spaces for Multiple Processes

- There are several methods for configuring address spaces for multiple processes
  1. Fixed address compilation
  2. Load-time fixup
  3. Position independent code
  4. Hardware support

Fixed-Address Compilation

Single Copy of Each Program

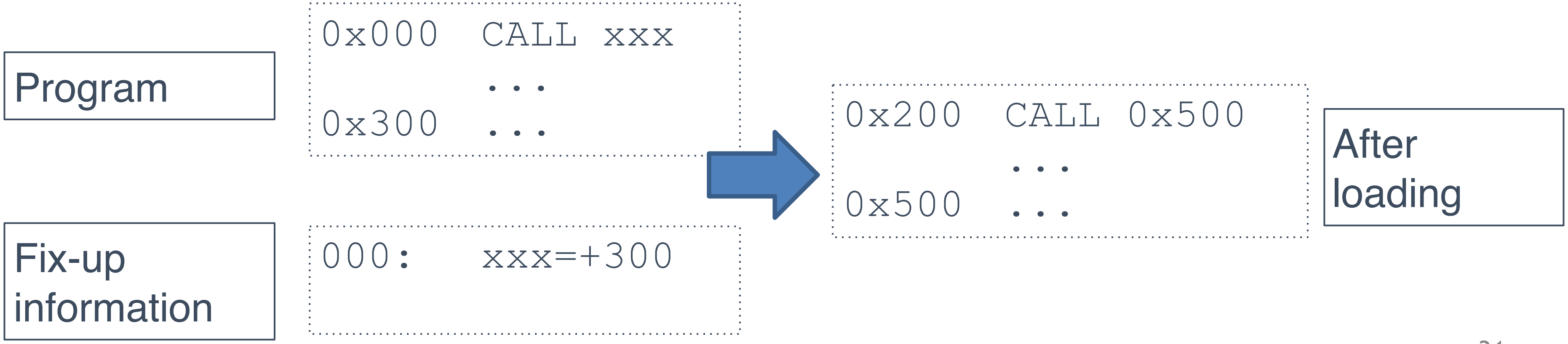
- Compile each program once, with fixed addresses
- OS may only load program at the specified offset in memory
- Typically, only one process may be run at any time
- Example: MS-DOS 1.0

Multiple Copies of Each Program

- Compile each program multiple times
- Once for each possible starting address
- Load the appropriate compiled program when the user starts the program
- Bad idea
  - Multiple copies of the same program

Load-Time Fixup

- Calculate addresses at load-time instead of compile-time
- The program contains a list of locations that must be modified at startup
  - All relative to some starting address
- Used in some OSes that run on low-end microcontrollers without virtual memory hardware



Program

```
0x000  CALL xxx
            ...
0x300  ...
```

Fix-up information

```
000:   xxx=+300
```

After loading

```
0x200  CALL 0x500
            ...
0x500  ...
```

Position-Independent Code

- Compiles programs in a way that is independent of their starting address
  - PC-relative address
- Slightly less efficient than absolute addresses
- Commonly used today for security reasons

Absolute addressing

```
0x200  CALL 0x500
       ...
0x500  ...
```

PC-relative addressing

```
0x200  CALL PC+0x300
       ...
0x500 ...
```

Hardware Support

- Hardware address translation
- Most popular way of sharing memory between multiple processes
  - Linux
  - OS X
  - Windows
- Program is compiled to run at a fixed location in virtual memory
- The OS uses the MMU to map these locations to physical memory

MMU and Virtual Memory

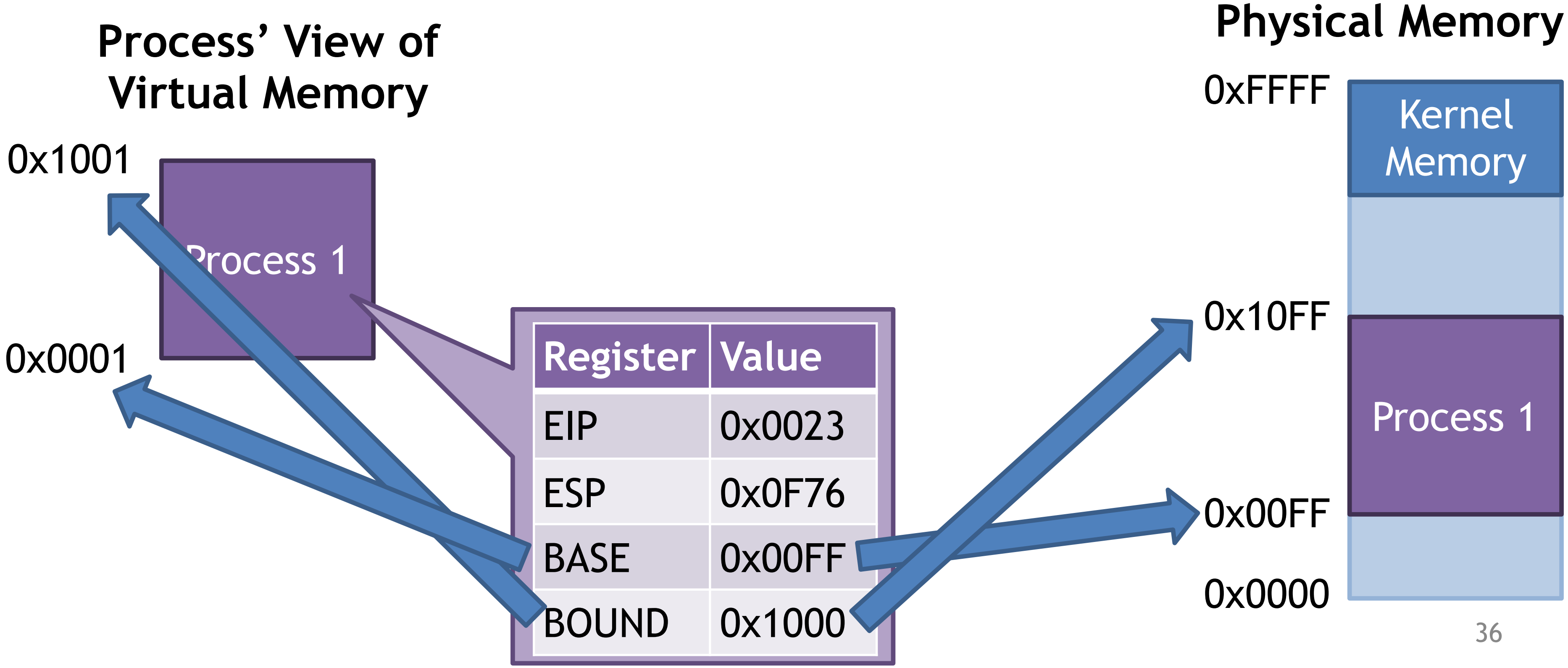
- The Memory Management Unit (MMU) translates between virtual addresses and physical addresses
  - Process uses virtual address for calls and data load/store
  - MMU translates virtual addresses to physical addresses
  - The physical addresses are the true locations of code and data in RAM

Advantages of Virtual Memory

- Flexible memory sharing
  - Simplifies the OS’s job of allocating memory to different programs
- Simplifies program writing and compilations
  - Each program gets access to 4GB of RAM (on a 32-bit CPU)
- Security
  - Can be used to prevent one process from accessing the address of another process
- Robustness
  - Can be used to prevent writing to addresses belonging to the OS (which may cause the OS to crash)

Virtual Memory - Base and Bounds Registers

- A simple mechanism for address translation
- Maps a contiguous virtual address region to a contiguous physical address region

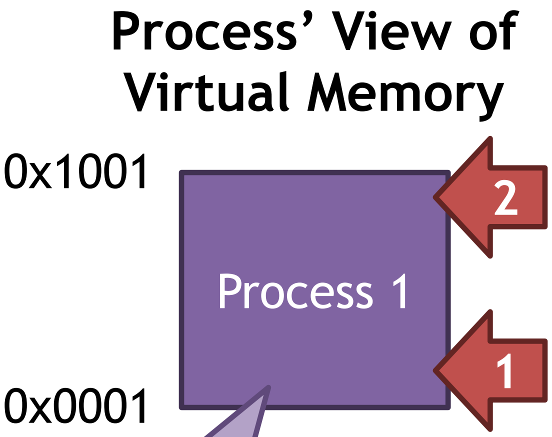


Process' View of Virtual Memory

Physical Memory

Register	Value
EIP	0x0023
ESP	0x0F76
BASE	0x00FF
BOUND	0x1000

Base and Bounds Example



Register	Value
EIP	0x0023
ESP	0x0F76
BASE	0x00FF
BOUND	0x1000

`0x0023 mov eax, [esp]`

1) Fetch instruction

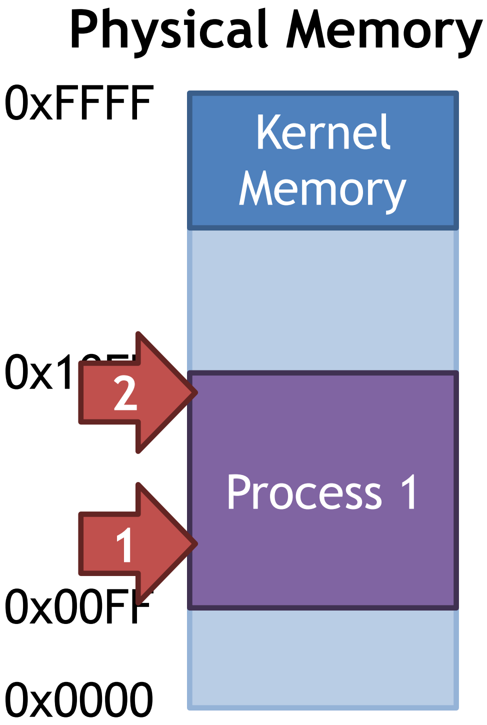
0x0023 + 0x00FF = 0x0122

2) Translate memory access

0x0F76 + 0x00FF = 0x1075

3) Move value to register

[0x1075] → eax



Confused About Virtual Memory?

- For now, focus on the goal that Virtual Memory's goal
- We will discuss virtual memory at great length later in the semester
- In project 3, you will implement virtual memory in Pintos

- Programs
- Processes
- Context Switching
- Protected Mode Execution
- Inter-process Communication
- Threads

From the Loader to the Kernel

- Once a program is loaded, the kernel must manage this new process
- Program Control Block (PCB): kernel data structure representing a process
  - Has at least one thread (possibly more…)
  - Keeps track of the memory used by the process
    - Code segments
    - Data segments (stack and heap)
  - Keeps runtime state of the process
    - CPU register values
    - EIP

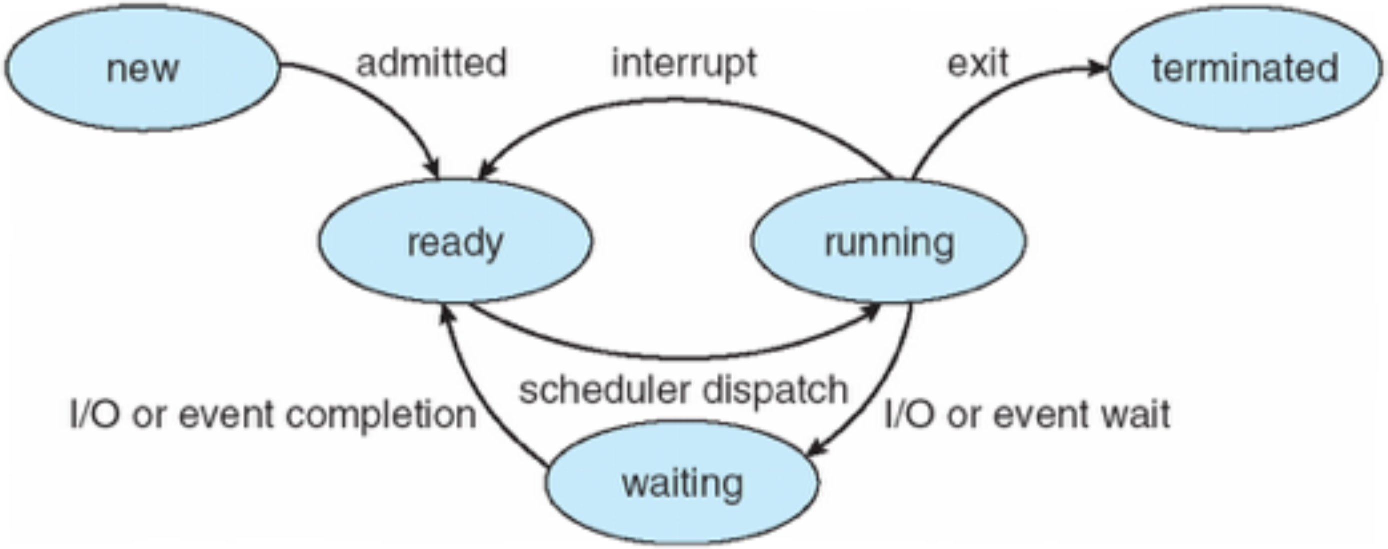
Program Control Block (PCB)

- OS structure that represents a process in memory
- Created for each process by the loader
- Managed by the kernel

```
struct task_struct {                              // Typical Unix PCB
        pid t_pid;                        // process identifier
        long state;                       // state of the process
        unsigned int time_slice;  //scheduling information
        struct task_struct *parent; // this process’s parent
        struct list_head children;// this process’s children
        struct files_struct *files;  // list of open files
        struct mm_struct *mm; // address space of this
process
};
```

Process States

- As a process, P, executes, it changes **state**
  - **new**: P is being created
  - **running**: P’s instructions are being executed
  - **waiting:** P is waiting for some event to occur
  - **ready**: P is waiting to be assigned to a processor
  - **terminated**: P has finished execution

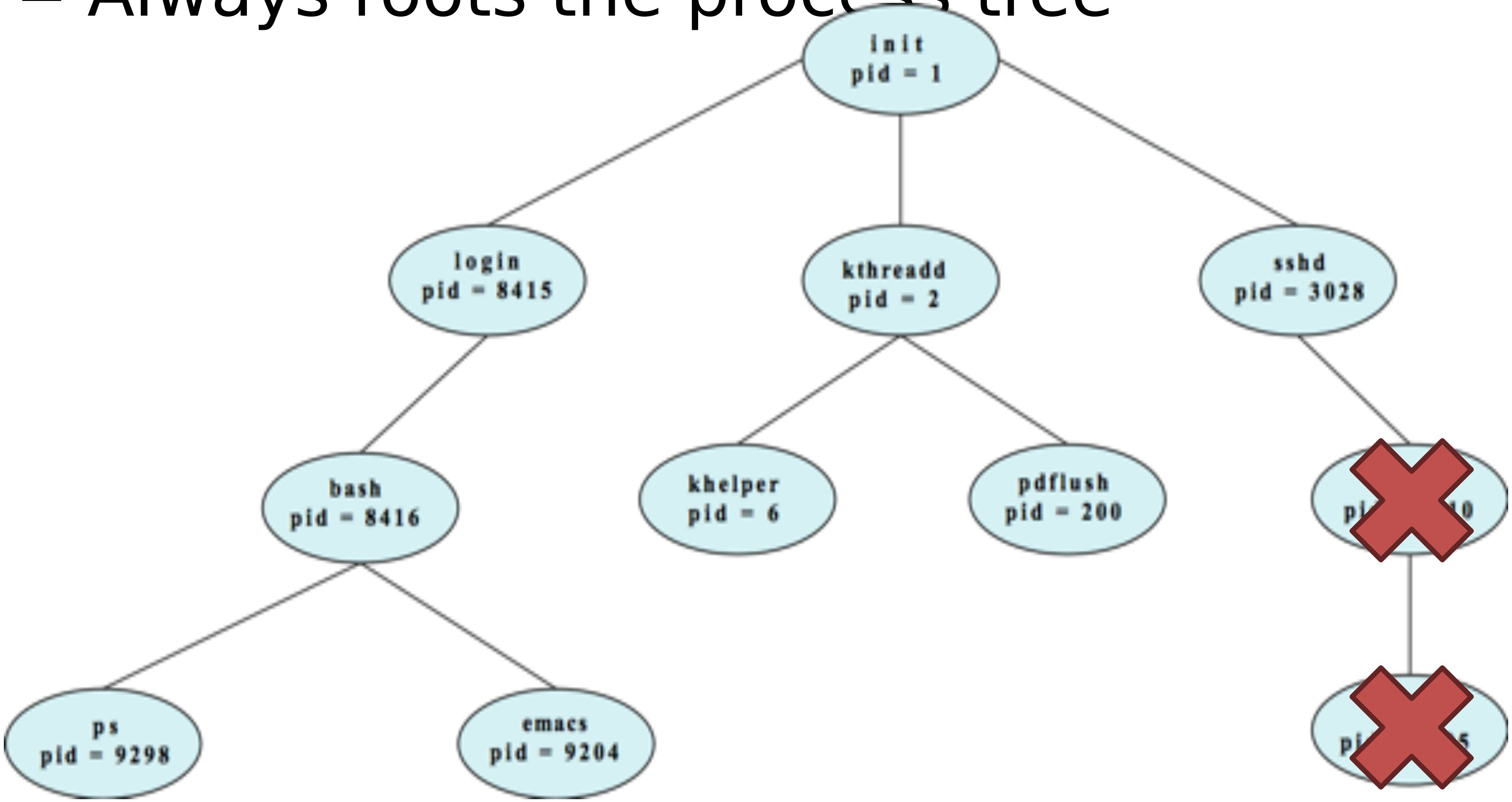


Parents and Children

- On Unix/Linux, all processes have parents
  - i.e. which process executed this new process?
- If a process spawns other processes, they become it's children
  - This creates a tree of processes
- If a parent exits before its children, the children become orphans
- If a child exits before the parent calls wait(), the child becomes a zombie

Process Tree

- init is a special process started by the kernel
  - Always roots the process tree



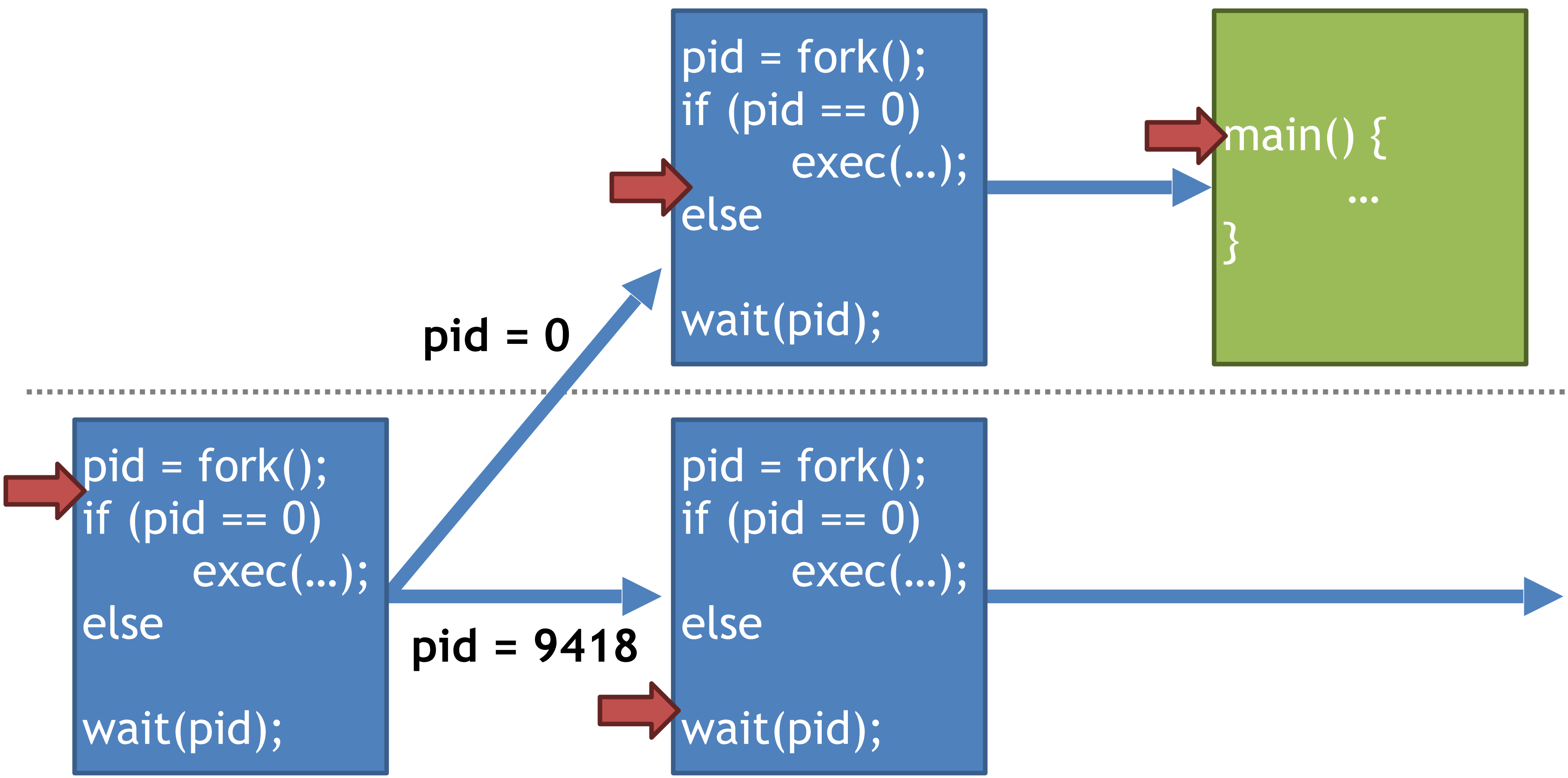
Additional Execution Context

- File descriptors
  - stdin, stdout, stderr
  - Files on disck
  - Sockets
  - Pipes
- Permissions
  - User and group
  - Access to specific APIs
  - Memory protection
- Environment
  - $PATH
- Shared Resources
  - Locks
  - Mutexes
  - Shared Memory

UNIX Process Management

- fork() – system call to create a copy of the current process, and start it running
  - No arguments!
- exec() – system call to change the program being run by the current process
- wait() – system call to wait for a process to finish
- signal() – system call to send a notification to another process

UNIX Process Management



Child Process

```
pid = fork();
if (pid == 0)
        exec(…);
else

wait(pid);
```

```
main() {
        …
}
```

pid = 0

pid = 9418

```
pid = fork();
if (pid == 0)
        exec(…);
else

wait(pid);
```

```
pid = fork();
if (pid == 0)
        exec(…);
else

wait(pid);
```

Original Process

Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {   // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                // I'm the parent process
    printf("I am parent of process #%d\n",
   child_pid);
    return 0;
}
```

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

Implementing UNIX fork()

- Steps to implement UNIX fork()
  1. Create and initialize the process control block (PCB) in the kernel
  2. Create a new address space
  3. Initialize the address space with a copy of the entire contents of the address space of the parent
  4. Inherit the execution context of the parent (e.g., any open files)
  5. Inform the scheduler that the new process is ready to run

Implementing UNIX exec()

- Steps to implement UNIX exec()
  1. Load the new program into the *current address space*
  2. Copy command line arguments into memory in the address space
  3. Initialize the hardware context to start execution
     - EIP = Entry point in the ELF header
     - ESP = A newly allocated stack

Process Termination

- Typically, a process will wait(pid) until its child process(es) complete
- abort(pid) can be used to immediately end a child process

- Programs
- Processes
- Context Switching
- Protected Mode Execution
- Inter-process Communication
- Threads

The Story So Far…

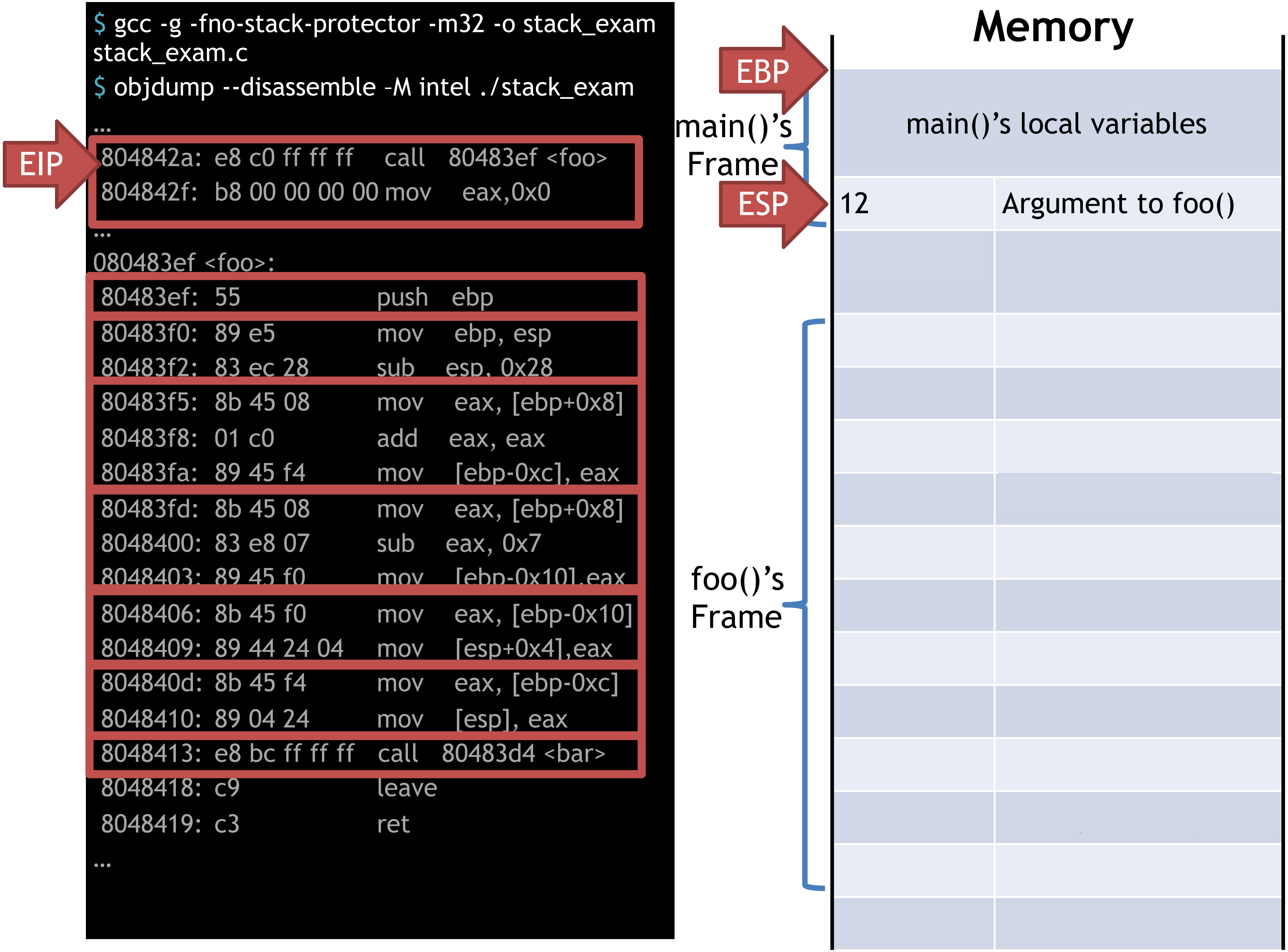
- At this point, we have gone over how the OS:
  - Turns programs into processes
  - Represents and manages running process
- Next step: context switching
  - How does a process access OS APIs?
    - i.e. System calls
  - How does the OS share the CPU between several programs?
    - Multiprocessing

Context Switching

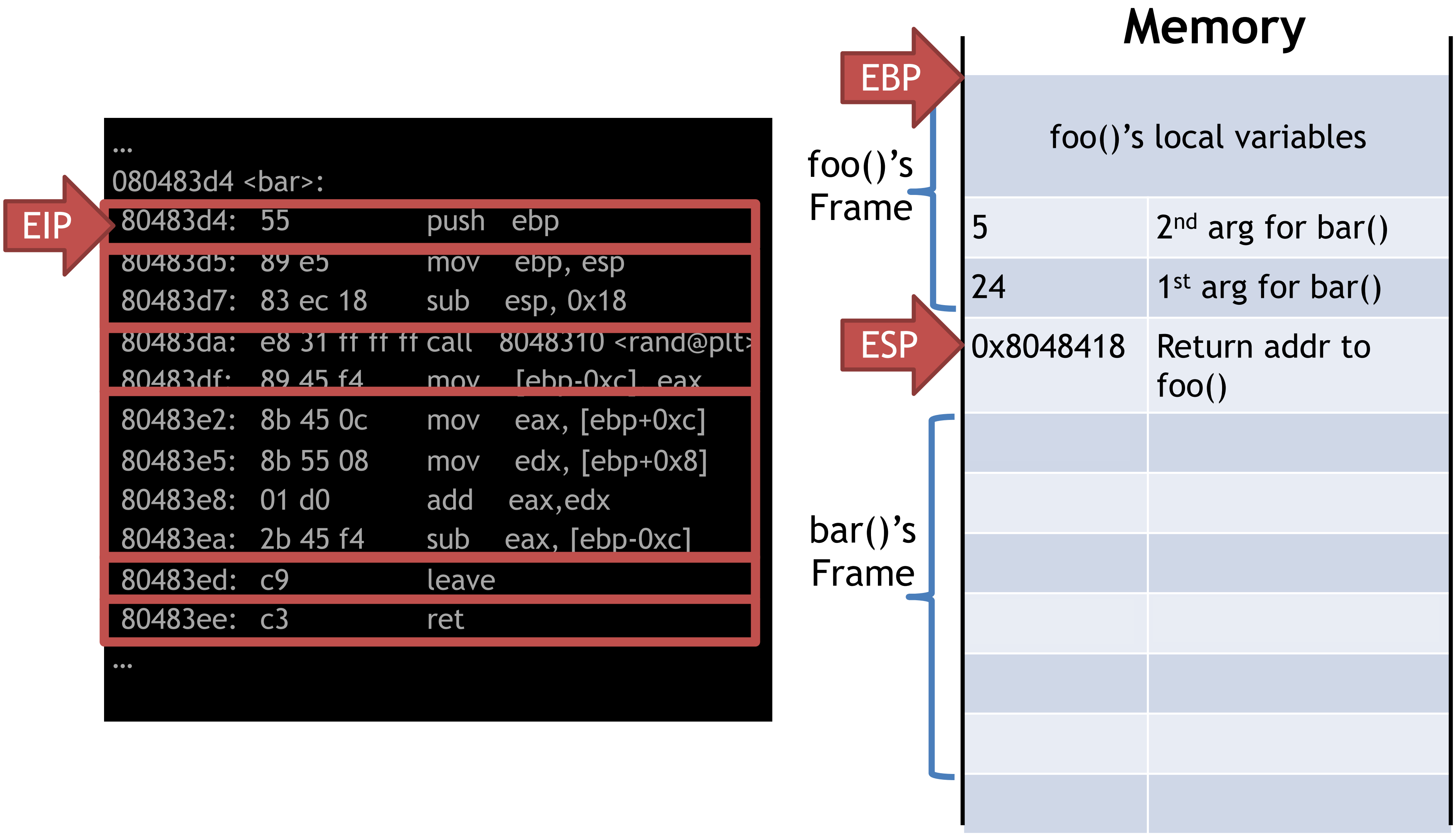
- Context switching
  - Saves state of a process before a switching to another process
  - Restores original process state when switching back
- Simple concept, but:
  - How do you save the state of a process?
  - How do you stop execution of a process?
  - How do you restart the execution of process that has been switched out?

The Process Stack

- Each process has a stack in memory that stores:
  - Local variables
  - Arguments to functions
  - Return addresses from functions
- On x86:
  - The stack grows downwards
  - ESP (Stack Pointer register) points to the bottom of the stack (i.e. the newest data)
    - EBP (Base Pointer) points to the base of the current frame
  - Instructions like push, pop, call, ret, int, and iret all modify the stack



```
$ gcc -g -fno-stack-protector -m32 -o stack_exam stack_exam.c
$ objdump --disassemble –M intel ./stack_exam
…
804842a: e8 c0 ff ff ff    call   80483ef <foo>
804842f: b8 00 00 00 00 mov    eax,0x0
…
080483ef <foo>:
80483ef:  55                push   ebp
80483f0:  89 e5             mov    ebp, esp
80483f2:  83 ec 28          sub    esp, 0x28
80483f5:  8b 45 08          mov    eax, [ebp+0x8]
80483f8:  01 c0             add    eax, eax
80483fa:  89 45 f4          mov    [ebp-0xc], eax
80483fd:  8b 45 08          mov    eax, [ebp+0x8]
8048400:  83 e8 07          sub    eax, 0x7
8048403:  89 45 f0          mov    [ebp-0x10],eax
8048406:  8b 45 f0          mov    eax, [ebp-0x10]
8048409:  89 44 24 04       mov    [esp+0x4],eax
804840d:  8b 45 f4          mov    eax, [ebp-0xc]
8048410:  89 04 24          mov    [esp], eax
8048413:  e8 bc ff ff ff    call   80483d4 <bar>
8048418:  c9                leave
8048419:  c3                ret
…
```



```
…
080483d4 <bar>:
 80483d4:   55              push   ebp
 80483d5:   89 e5           mov    ebp, esp
 80483d7:   83 ec 18        sub    esp, 0x18
 80483da:   e8 31 ff ff ff  call   8048310 <rand@plt
 80483df:   89 45 f4        mov    [ebp-0xc], eax
 80483e2:   8b 45 0c        mov    eax, [ebp+0xc]
 80483e5:   8b 55 08        mov    edx, [ebp+0x8]
 80483e8:   01 d0           add    eax,edx
 80483ea:   2b 45 f4        sub    eax, [ebp-0xc]
 80483ed:   c9              leave
 80483ee:   c3              ret
…
```

Memory

foo()'s local variables	
5	2nd arg for bar()
24	1st arg for bar()
0x8048418	Return addr to foo()

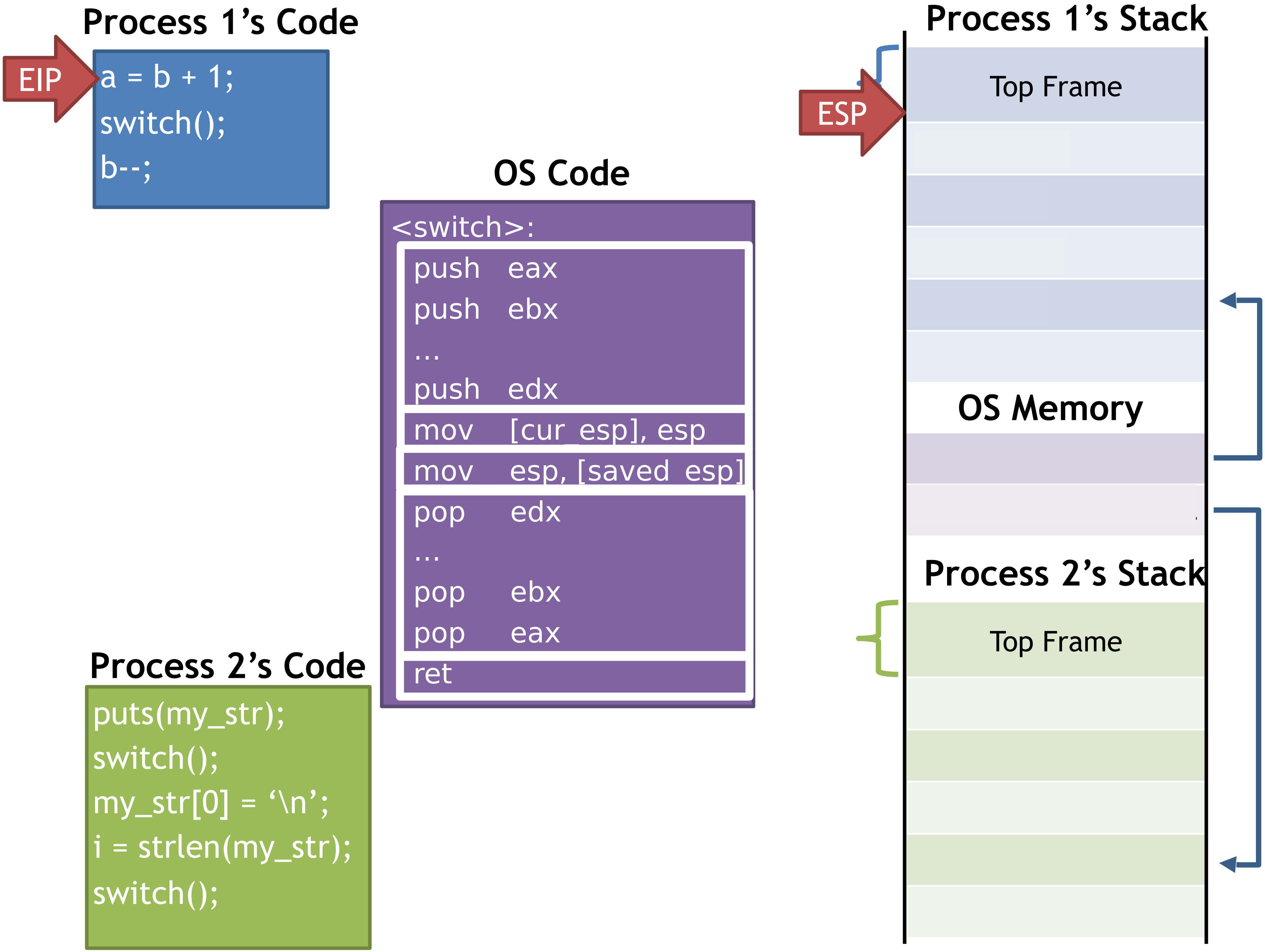
- leave → mov esp, ebp; pop ebp;
- Return value is placed in EAX

Stack Switching

- We’ve seen that the stack holds
  - Local variables
  - Arguments to functions
  - Return addresses
  - … basically, the state of a running program
- Crucially, a process’ control flow is stored on the stack
- If you modify the stack, you also modify control flow
  - Stack switching is effectively process switching

Switching Between Processes

1. Process 1 calls into `switch()` routine
2. CPU registers are pushed onto the stack
3. The stack pointer is saved into memory
4. The stack pointer for process 2 is loaded
5. CPU registers are restored
6. `switch()` returns back to process 2



Process 1's Code

```
a = b + 1;
switch();
b--;
```

OS Code

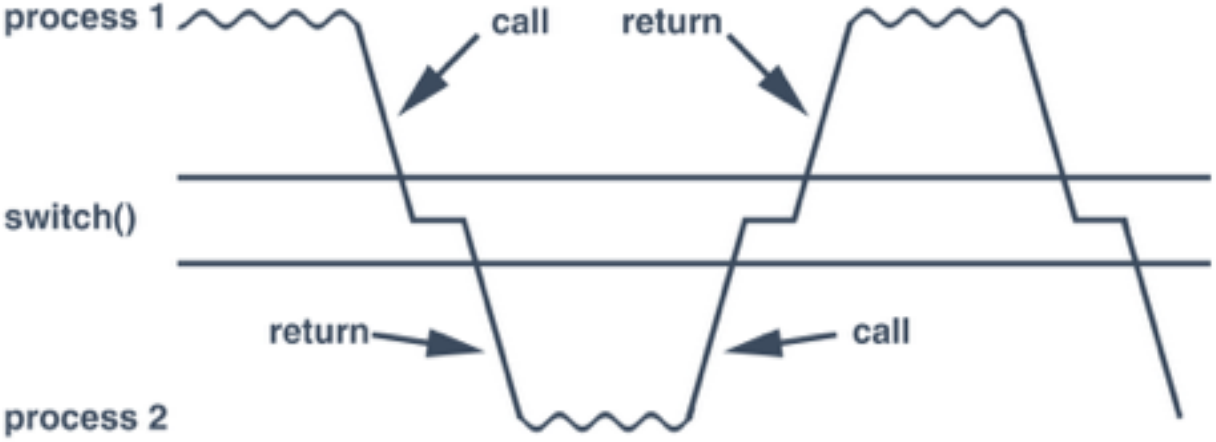
```
<switch>:
push   eax
push   ebx
…
push   edx
mov    [cur_esp], esp
mov    esp, [saved_esp]
pop    edx
…
pop    ebx
pop    eax
ret
```

Process 2's Code

```
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
switch();
```

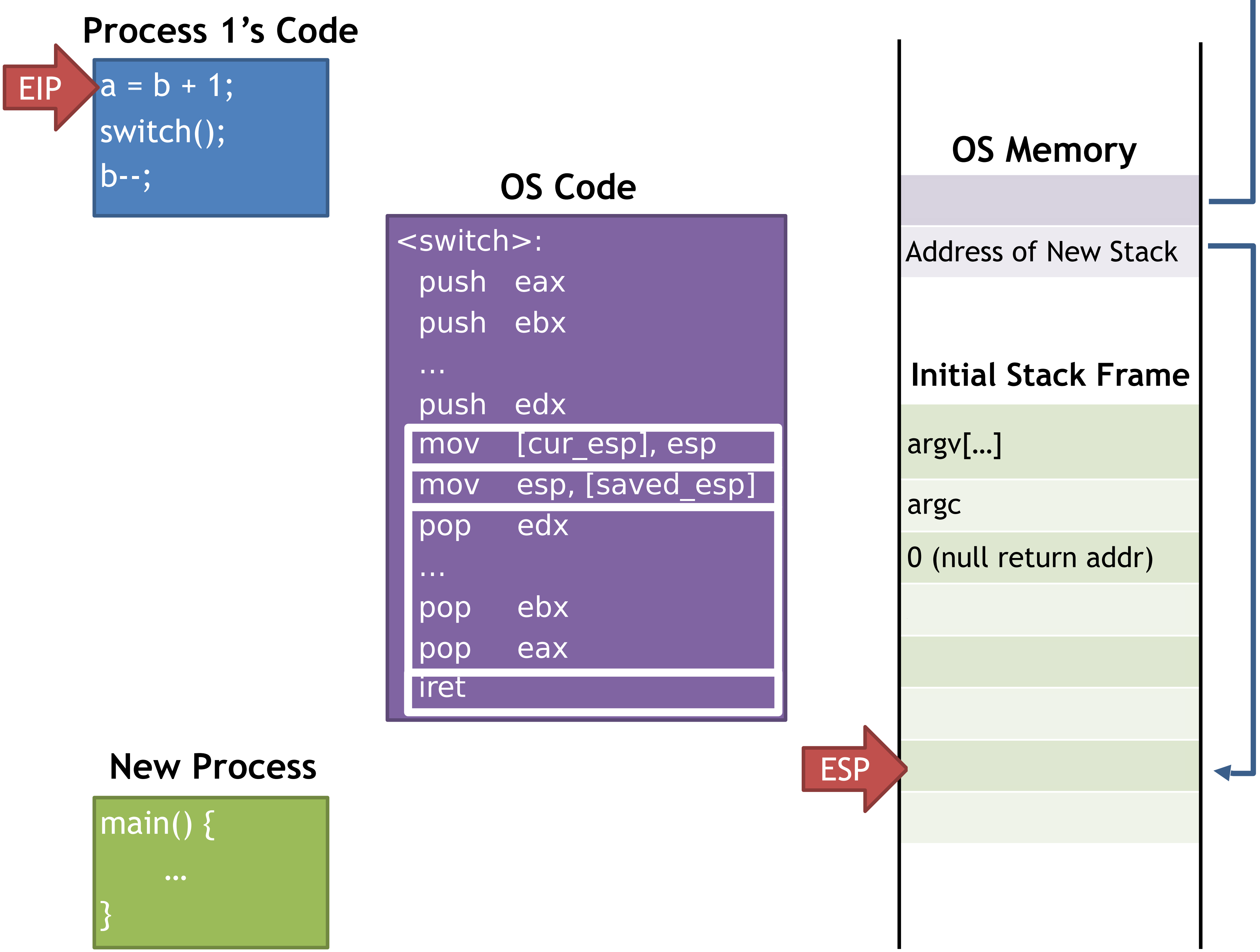
Abusing Call and Return

- Context switching uses function call and return mechanisms
  - Switches **into** a process by **returning** from a function
  - Switches **out** of a process by **calling** into a function



What About New Processes?

- But how do you start a process in the first place?
  - A new process doesn't have a stack…
  - … and it never called into switch()
- Pretend that there *was* a previous call
  - Build a fake initial stack frame
  - This frame looks exactly like the instruction just before main() called into switch()
  - When switch() returns, it'll allow main() to run from the beginning



Process 1's Code

```
a = b + 1;
switch();
b--;
```

OS Code

```
<switch>:
  push   eax
  push   ebx
  …
  push   edx
  mov    [cur_esp], esp
  mov    esp, [saved_esp]
  pop    edx
  …
  pop    ebx
  pop    eax
  iret
```

New Process

```
main() {
      …
}
```

OS Memory

Address of New Stack

Initial Stack Frame

argv[…]

argc

0 (null return addr)

When Do You Switch Processes?

- To share CPU between multiple processes, control must eventually return to the OS
  - When should this happen?
  - What mechanisms implements the switch from user process back to the OS?
- Four approaches:
  1. Voluntary yielding
  2. Switch during API calls to the OS
  3. Switch on I/O
  4. Switch based on a timer interrupt

Voluntary Yielding

- Idea: processes must voluntary give up control by calling an OS API, e.g. thread_yield()
- Problems:
  - Misbehaving or buggy apps may never yield
  - No guarantee that apps will yield in a reasonable amount of time
  - Wasteful of CPU resources, i.e. what if a process is idle-waiting on I/O?

Interjection on OS APIs

- Idea: whenever a process calls an OS API, this gives the OS an opportunity to context switch
  - E.g. printf(), fopen(), socket(), etc…
- The original Apple Macintosh used this approach
  - Cooperative multi-tasking
- Problems:
  - Misbehaving or buggy apps may never yield
  - Some normal apps don't use OS APIs for long periods of time
    - E.g. a long, CPU intensive matrix calculation

I/O Context Switch Example

- What’s happening here?

```
struct terminal {
     queue<char> keystrokes;  /* buffered keystrokes - array or list */
     process *waiting;        /* process waiting for input */
     ...
};
process *current;             /* the currently running process */
queue<process *> active;      /* linked list of other processes ready to run */

char get_char(terminal *term) {
     if (term->keystrokes.empty()) {
          term->waiting = current;       /* sleep waiting for input */
          switch_to(active.pop_head());  /* and switch to next active process */
     }
     return term->keystrokes.pop_head();
}

void interrupt(terminal *term, char key) {
     term->keystrokes.push_tail(key);      /* add keystroke to buffer */
     if (term->waiting) {
          active.push_tail(term->waiting); /* and wake up sleeping process */
          term->waiting = NULL;
     }
}
```

Context Switching on I/O

- Idea: when one process is waiting on I/O, switch to another process
  - I/O APIs already go through the OS, so context switching is easy
- Problems:
  - Some apps don't have any I/O for long periods of time

Preemptive Context Switching

- So far, our processes will not switch to another process until some action is taken
  - e.g. an API call or an I/O interrupt
- Idea: use a timer interrupt to force context switching at set intervals
  - Interrupt handler runs at a fixed frequency to measure how long a process has been running
  - If it’s been running for some max duration (scheduling quantum), the handler switches to the next process
- Problems:
  - Requires hardware support (a programmable timer)
    - Thankfully, this is built-in to most modern CPUs

- Programs
- Processes
- Context Switching
- Protected Mode Execution
- Inter-process Communication
- Threads

Process Isolation

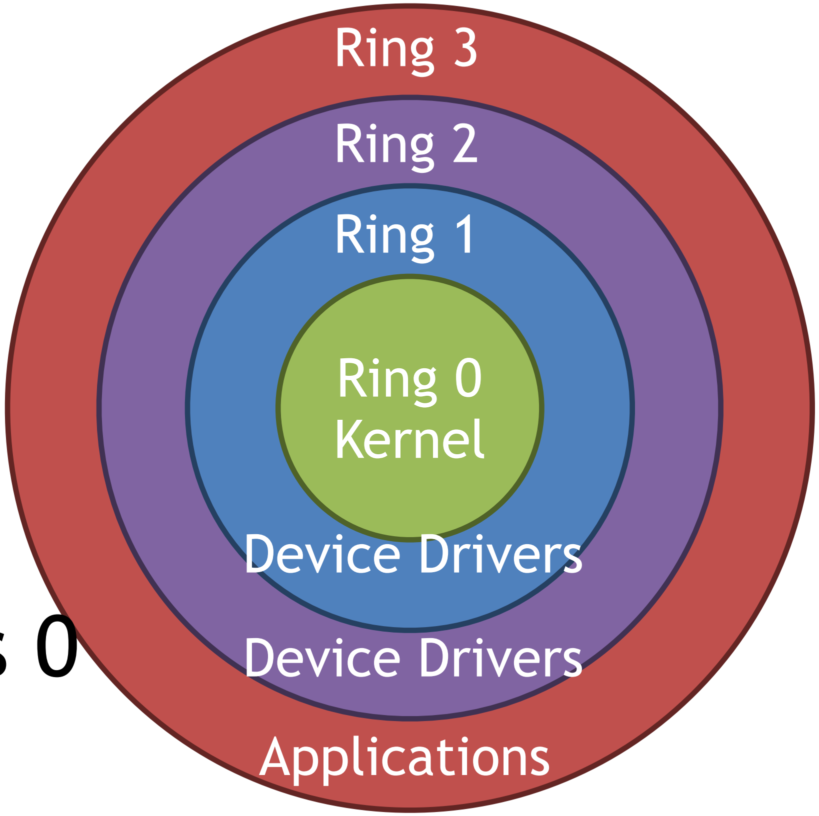
- At this point, we can execute multiple processes concurrently
- Problem: how do you stop processes from behaving badly?
  - Overwriting kernel memory
  - Reading/writing data from other processes
  - Disabling interrupts
  - Crashing the whole computer
  - Etc.

Thought Experiment

- How can we implement execution with limited privilege?
  - Use an interpreter or a simulator
    - Execute each program instruction in a simulator
    - If the instruction is permitted, do the instruction
    - Otherwise, stop the process
    - Basic model in Javascript, Java, …
- However, interpreters and simulators are slow
- How do we go faster?
  - Run the unprivileged code directly on the CPU

Protected Mode

- Most modern CPUs support protected mode
- x86 CPUs support three rings with different privileges
  - Ring 0: OS kernel
  - Ring 1, 2: device drivers
  - Ring 3: userland
- Most OSes only use rings 0 and 3
- What about hypervisors?



Real vs. Protected

- On startup, the CPU starts in 16-bit real mode
  - Protected mode is disabled
  - Assumes segment:offset addressing
- Typically, bootloader switches CPU to protected mode

```
mov eax, cr0
or eax, 1  ; set bit 1 of CR0 to 1
           ; enables pmode
mov cr0, eax
```

Dual-Mode Operation

- Ring 0: kernel/supervisor mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/ read any packet
- Ring 3: user mode or “userland”
  - Limited privileges
  - Only those granted by the operating system kernel

Protected Features

- What system features are impacted by protection?
  - Privileged instructions
    - Only available to the kernel
  - Limits on memory accesses
    - Prevents user code from overwriting the kernel
  - Access to hardware
    - Only the kernel may directly interact with peripherals
  - Programmable Timer Interrupt
    - May only be set by the kernel
    - Used to force context switches between processes

Privileged Instructions

- Examples?
  - sti/cli – Enable and disable interrupts
  - Any instruction that modifies the CR0 register
    - Controls whether protected mode is enabled
  - hlt – Halts the CPU
- What should happen if a user program attempts to execute a privileged instruction?
  - General protection (GP) exception gets thrown by the CPU
  - Control is transferred to the OS's exception handler

Changing Modes

- Applications often need to access the OS
  - i.e. system calls
  - Writing files, displaying on the screen, receiving data from the network, etc...
- But the OS is ring 0, and apps are ring 3
- How do apps get access to the OS?
  - Apps invoke system calls with an interrupt
    - E.g. `int 0x80`
  - `int` causes a mode transfer from ring 3 to ring 0

Mode Transfer

Userland

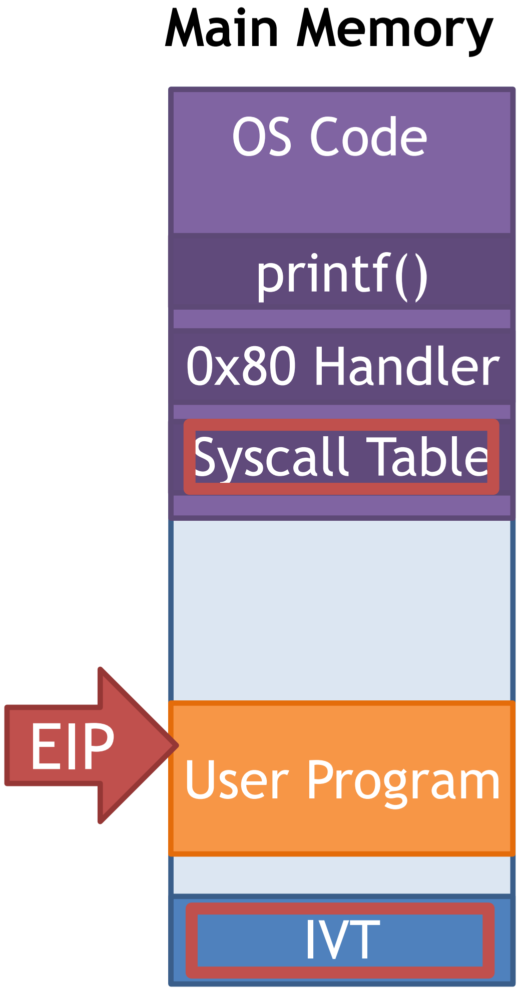
1. Application executes trap (int) instruction
   - EIP, CS, and EFLAGS get pushed onto the stack
   - Mode switches from ring 3 to ring 0

Kernel Mode

2. Save the state of the current process
   - Push EAX, EBX, …, etc. onto the stack
3. Locate and execute the correct syscall handler
4. Restore the state of process
   - Pop EAX, EBX, … etc.
5. Place the return value in EAX
6. Use iret to return to the process
   - Switches back to the original mode (typically 3)

System Call Example

1. Software executes int 0x80
   - Pushes EIP, CS, and EFLAGS
2. CPU transfers execution to the OS handler
   - Look up the handler in the IVT
   - Switch from ring 3 to 0
3. OS executes the system call
   - Save the processes state
   - Use EAX to locate the system call
   - Execute the system call
   - Restore the processes state
   - Put the return value in EAX
4. Return to the process with iret
   - Pops EIP, CS, and EFLAGS
   - Switches from ring 0 to 3



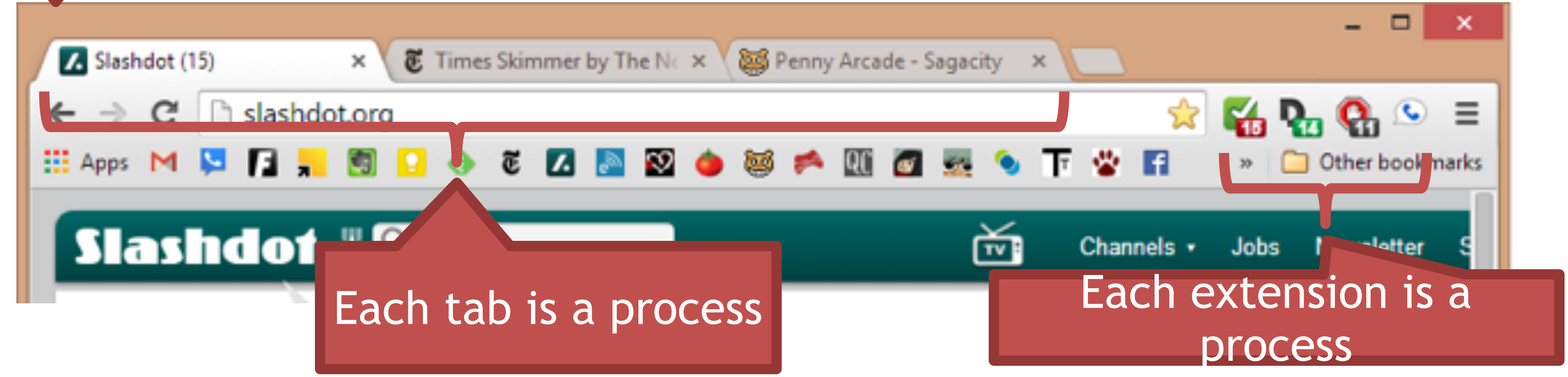
Alternative Syscall Mechanisms

- Thus far, all examples have used int/iret
- However, there are other syscall mechanisms on x86
  - sysenter/sysexit
  - syscall/sysret
- The sys* instructions are much faster than int/iret
  - Jump directly to OS code, rather than looking up handlers in the IVT
  - Used by modern OSes, including the Linux kernel

- Programs
- Processes
- Context Switching
- Protected Mode Execution
- Inter-process Communication (IPC)
- Threads

Processes are not Islands

- Thus far:
  - We can load programs as processes
  - We can context switch between processes
  - Processes are protected from each other
- if one or more processes want to communicate with each other?



Browser core is a process

Each tab is a process

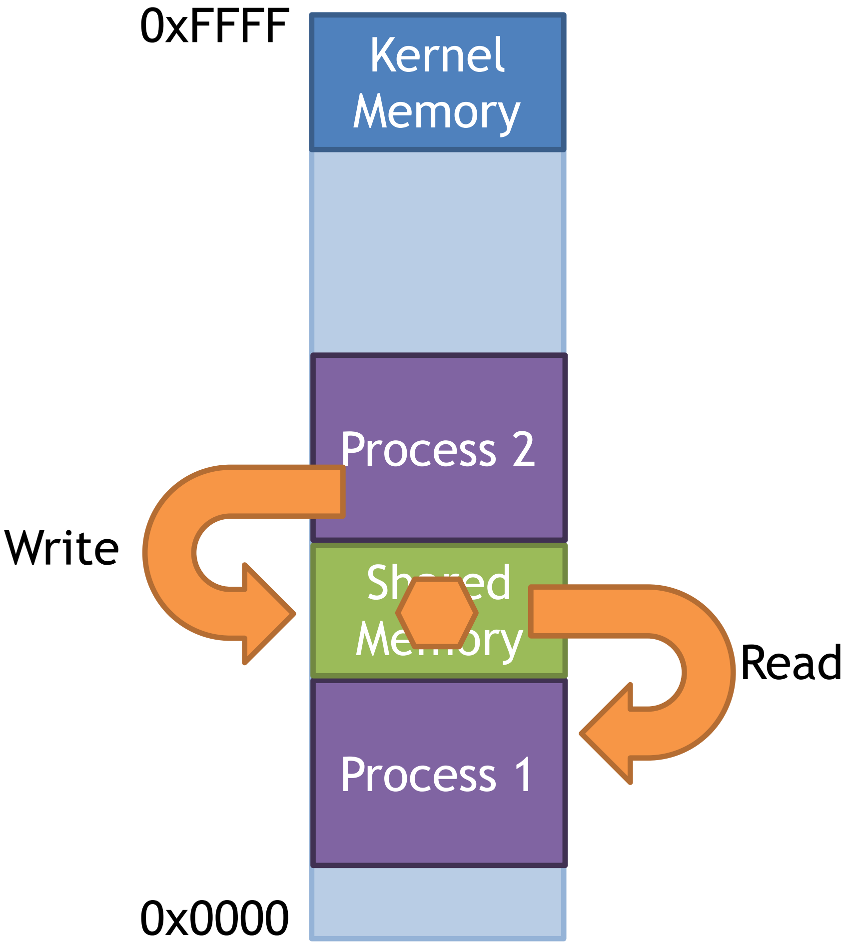
Each extension is a process

Mechanisms for IPC

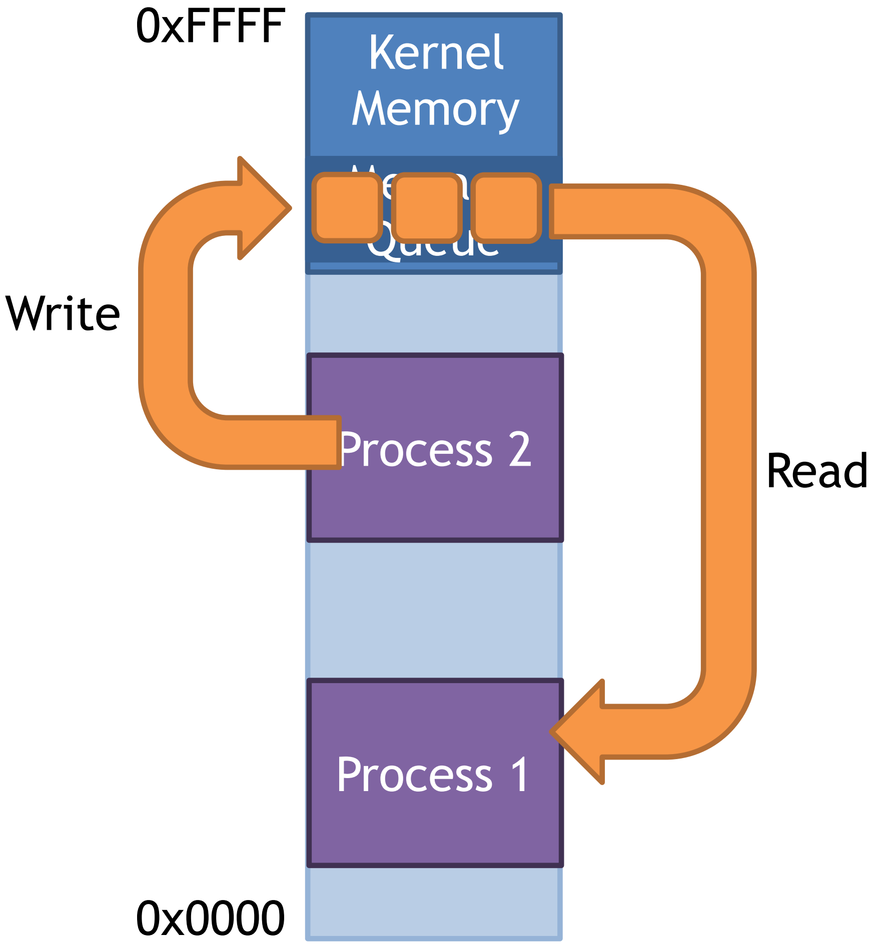
- Typcially, two ways of implementing IPC
  - Shared memory
    - A region of memory that many processes can all read/write
  - Message passing
    - Various OS-specific APIs
    - Pipes
    - Sockets
    - Signals

IPC Examples

Shared Memory



Message Passing



Posix Shared Memory API

- shm_open() – create and/or open a shared memory page
  - Returns a file descriptor for the shared page
- ltrunc() or ftruncate() – limit the size of the shared memory page
- mmap() – map the memory page into the processes address space
  - Now you can read/write the page using a pointer
- close() – close a file descriptor
- shm_unlink() – remove a shared page
  - Processes with open references may still access the page

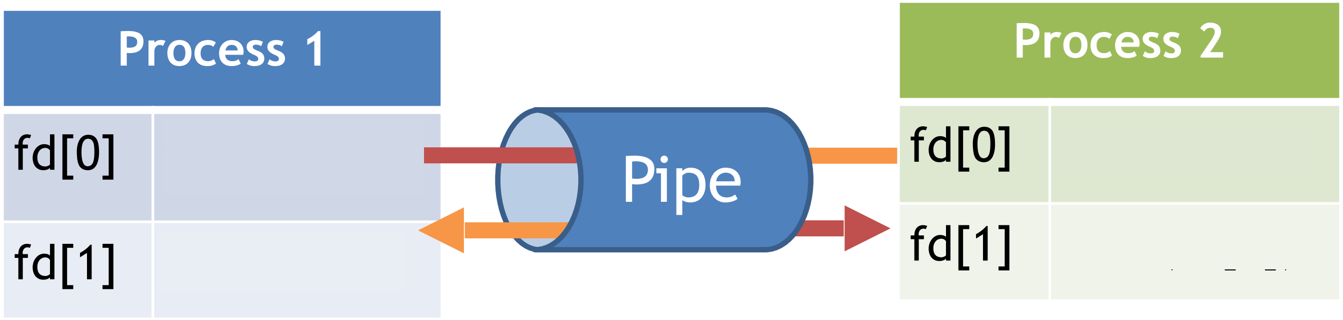
```
/* Program to write some data in shared memory */
int main() {
	const int SIZE = 4096; /* size of the shared page */
		/* name of the shared page */
	const char * NAME = “MY_PAGE”;
	const char * msg = “Hello World!”;
	int shm_fd;
	char * ptr;

	shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);
	ftruncate(shm_fd, SIZE);
	ptr = (char *) mmap(0, SIZE, PROT_WRITE,
					MAP_SHARED, shm_fd, 0);
	sprintf(ptr, “%s”, msg);
	close(shm_fd);
	return 0;
}
```

```
/* Program to read some data from shared memory */
int main() {
        const int SIZE = 4096; /* size of the shared page */
                /* name of the shared page */
        const char * NAME = “MY_PAGE”;
        int shm_fd;
        char * ptr;

        shm_fd = shm_open(name, O_RDONLY, 0666);
        ptr = (char *) mmap(0, SIZE, PROT_READ,
                                MAP_SHARED, shm_fd, 0);
        printf(“%s\n”, ptr);
        shm_unlink(shm_fd);
        return 0;
}
```

POSIX Message Queues

- Implementation of message passing
  - Producers add messages to shared FIFO queue
  - Consumer(s) remove messages
  - OS takes care of memory management, synchronization
- Posix API:
  - msgget() – creates a new message queue
  - msgsnd() – pushes a message onto the queue
  - msgrcv() – pops a message from the queue

Pipes



- File-like abstraction for sending data between processes
  - Can be read or written to, just like a file
  - Permissions controlled by the creating process
- Two types of pipes
  - Named pipe: any process can attach as long as it knows the name
    - Typically used for long lived IPC
  - Unnamed/anonymous pipe: only exists between a parent and its children
- Full or half-duplex
  - Can one or both ends of the pipe be read?
  - Can one or both ends of the pipe be written?

You’ve All Used Pipes

Pipe the output from one process
to the input of another process

```
$  ps x | grep ssh
 3299 ?        S      0:00 sshd: cbw@pts/0
```

```
int main() {  /* Program that passes a string to a child process through a pipe */
        int fd[2], nbytes;
        pid_t childpid;
        char string[] = "Hello, world!\n";
        char readbuffer[80];

        pipe(fd);
        if ((childpid = fork()) == -1) { perror("fork"); exit(1); }
        if (childpid == 0) {
                /* Child process closes up input side of pipe */
                close(fd[0]);
                /* Send "string" through the output side of pipe */
                write(fd[1], string, strlen(string) + 1);
        } else {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }
        return(0);
}
```

Sockets for IPC

- Yes, the same sockets you use for networking
- Server opens a listen socket, as usual
- Clients connect to this socket
    - The server can check the clients IP and drop connections from anyone other than 127.0.0.1
- Send and receive packets as usual

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- What is the capacity of each link?
- Are messages fixed size or variable size?
- Is the link unidirectional or bidirectional?
- Is the link synchronous or asynchronous?
- Does the API guarantee atomicity?
- What is the overhead of the API?

- Programs
- Processes
- Context Switching
- Protected Mode Execution
- Inter-process Communication
- Threads

Are Processes Enough?

- At this point, we have the ability to run processes
  - And processes can communicate with each other
- Is this enough functionality?
- Possible scenarios:
  - A large server with many clients
  - A powerful computer with many CPU cores

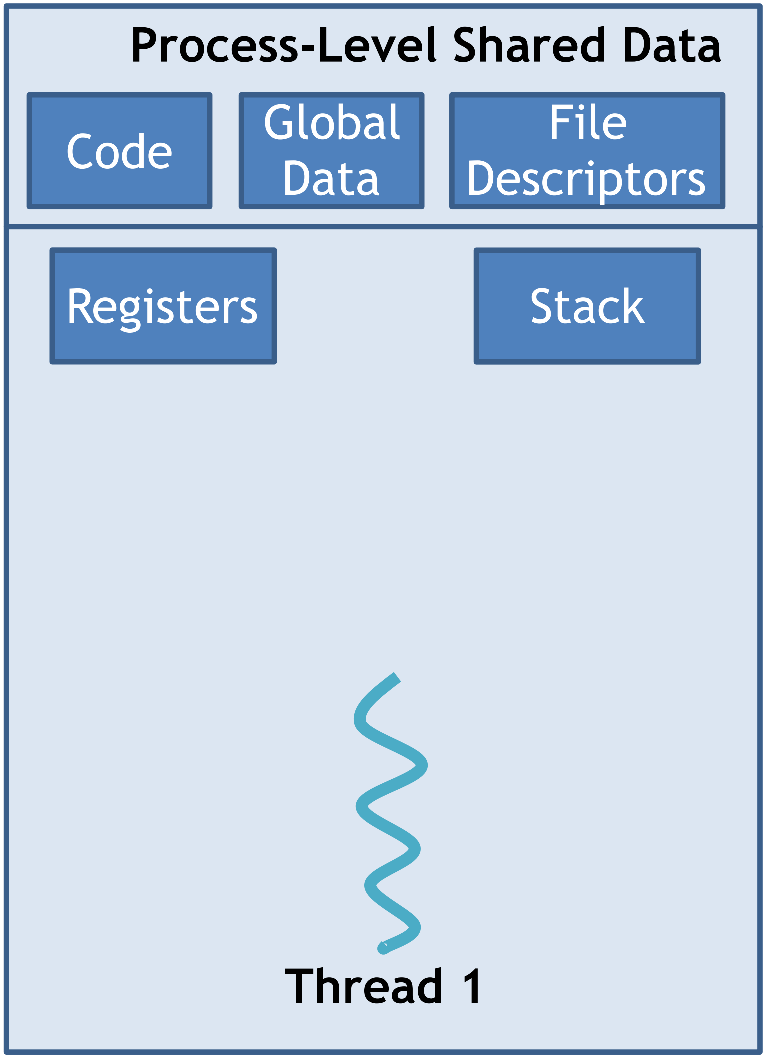
Problems with Processes

- Process creation is heavyweight (i.e. slow)
  - Space must be allocated for the new process
  - fork() copies all state of the parent to the child
- IPC mechanisms are cumbersome
  - Difficult to use fine-grained synchronization
  - Message passing is slow
    - Each message may have to go through the kernel

Threads

- Light-weight processes that share the same memory and state space
- Every process has at least one thread
- Benefits:
  - Resource sharing, no need for IPC
  - Economy: faster to create, faster to context switch
  - Scalability: simple to take advantage of multi-core CPUs

Single-Threaded Process



Process-Level Shared Data

Code

Global Data

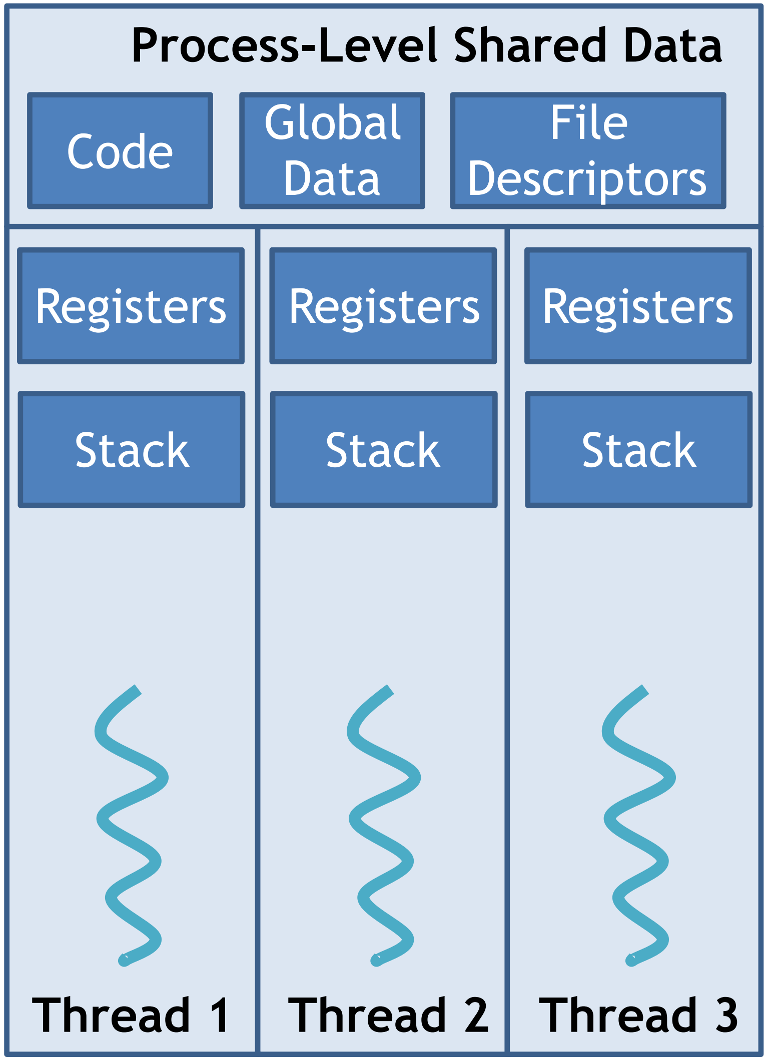
File Descriptors

Registers

Stack

Thread 1

Multi-Threaded Process



Process-Level Shared Data

Code

Global Data

File Descriptors

Registers

Stack

Thread 1

Registers

Stack

Thread 2

Registers

Stack

Thread 3

Thread Implementations

- Threads can be implemented in two ways:
  1. User threads
     - User-level library manages threads within a single process
  2. Kernel threads
     - Kernel manages threads for all processes

POSIX Pthreads

- POSIX standard API for thread creation
  - IEEE 1003.1c
  - *Specification*, not *implementation*
    - Defines the API and the expected behavior
    - … but not how it should be implemented
- Implementation is system dependent
  - On some platforms, user-level threads
  - On others, maps to kernel-level threads

Pthread API

- pthread_attr_init() – initialize the threading library
- pthread_create() – create a new thread
- pthread_exit() – exit the current thread
- pthread_join() – wait for another thread to exit
- Pthreads also contains a full range of synchronization primitives

Pthread Example

```
    pthread_t tid; // id of the child thread
    pthread_attr_t attr; // initialization data
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, 0);
    pthread_join(tid, 0);

void * runner(void * params) {
        …
        pthread_exit(0);
}
```

Linux Threads

- In the kernel, threads are just tasks
  - Remember the task_struct from earlier?
- New threads created using the clone() API
  - Sort of like fork()
  - Creates a new child task that copies the address space of the parent
    - Same code, same environment, etc.
    - New stack is allocated
    - No memory needs to be copied (unlike fork())

Thread Oddities

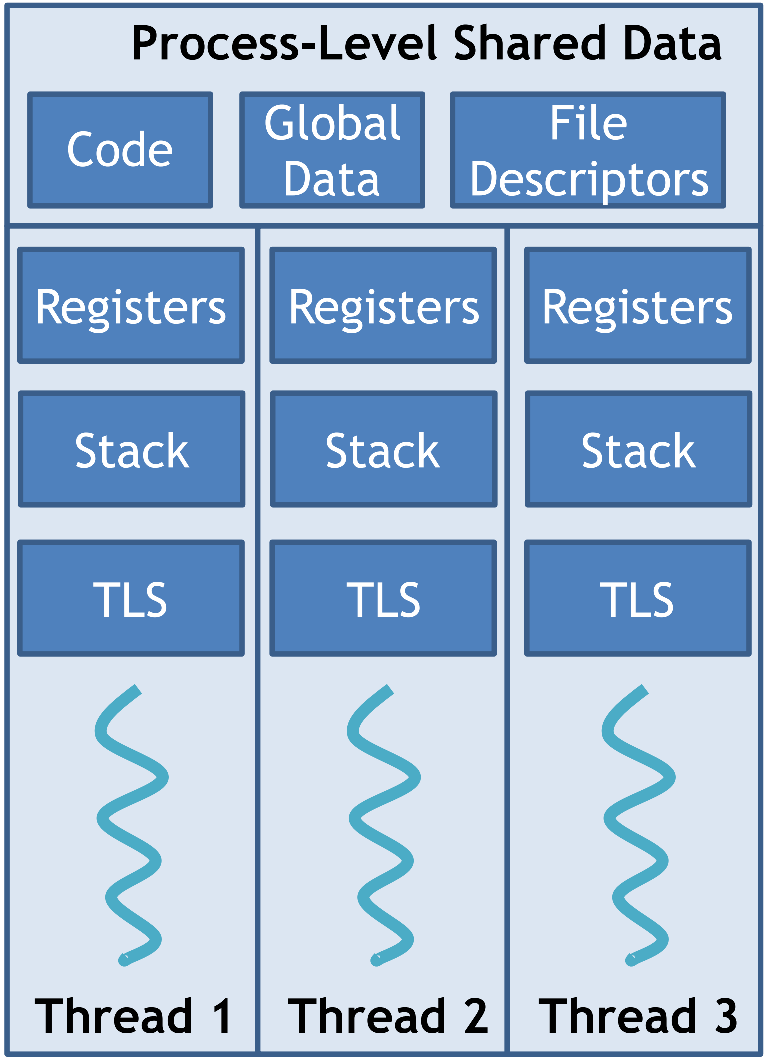
- What happens if you fork() a process that has multiple threads?
  - You get a child process with exactly one thread
  - Whichever thread called fork() survives
- What happens if you run exec() in a multi-threaded process?
  - All but one threads are killed
  - exec() gets run normally

Advanced Threading

- Thread pools:
  - Create many threads in advance
  - Dynamically give work to threads from the pool as it becomes available
- Advantages:
  - Cost of creating threads is handled up-front
  - Bounds the maximum number of threads in the process

Thread Local Storage

- Sometimes, you want each thread to have its own "global" data
  - Not global to all threads
  - Not local storage on the stack
- Thread local storage (TLS) allows each thread to have its own space for "global" variables
  - Similar to static variables



OpenMP

- Compiler extensions for C, C++ that adds native support for parallel programming
- Controlled with parallel regions
  - Automatically creates as many threads as there are cores

```
#include <omp.h>

int main() {
    int i, N = 20;
    #pragma omp parallel
    {
        printf(“I am a parallel region\n”);
    }

    # pragma omp parallel for
    for (i = 0; i < N; i++)
        printf(“This is a parallel for loop
\n”);

    return 0;
}
```

Processes vs. Threads

- Threads are better if:
  - You need to create new ones quickly, on-the-fly
  - You need to share lots of state

- Processes are better if:
  - You want protection
    - One process that crashes or freezes doesn’t impact the others
  - You need high security
    - Only way to move state is through well-defined, sanitized message passing interface