NEU CS 3650 Computer Systems

Instructor: Dr. Ziming Zhao

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Alden Jackson, Ben Weintraub, Gene Cooperman, Peter Desnoyers' lecture slides for the same course.

Process

- The concept of process
- ELF and other executable file format
- How is an ELF loaded to memory?
- Virtualizing CPU with time-sharing
- The mechanism to switch processes

Diving into the Operating Systems

- So far, we have been preparing for our further exploration:
  - CPU, Memory, Assembly, C

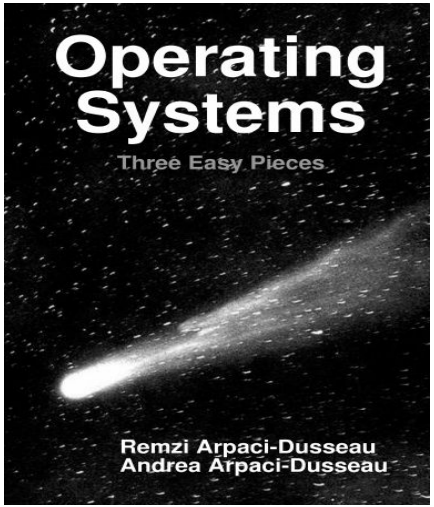
- Today we will dive into the OS itself. What we learned will be useful
  - Registers and instruction concepts
  - Memory as a linear array and ways to work with memory addresses
  - C is at the core of many common OSes

OS: Virtualization + Abstraction

- The OS is a (software) land of magic and illusions
- OS makes a computer “easy” to use
- OS hides overwhelming complexities of hardware behind an API
  - This is **abstraction**
- OS creates the illusion of an ideal, general, and powerful machine
  - This is **virtualization**
- We will start by looking at how OSes provides CPU virtualization at the process level.
- Other abstractions the OS uses

Required Reading

- The OSTEP book: up to Ch. 3-6
- Online: https://pages.cs.wisc.edu/~remzi/OSTEP/
- Hard copy: Lulu or Amazon



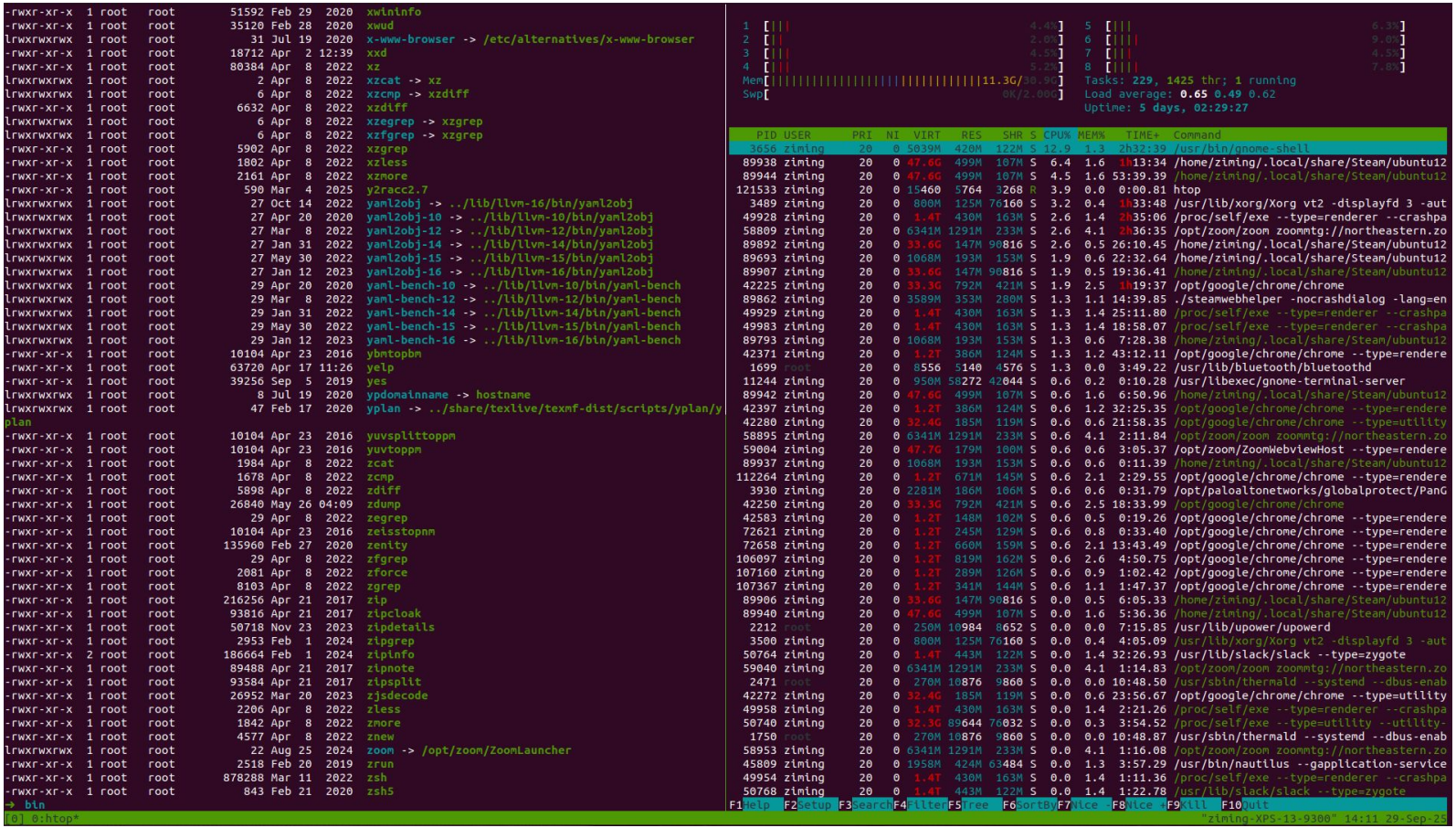
Process

- Basic function of an OS is to execute and manage code dynamically:

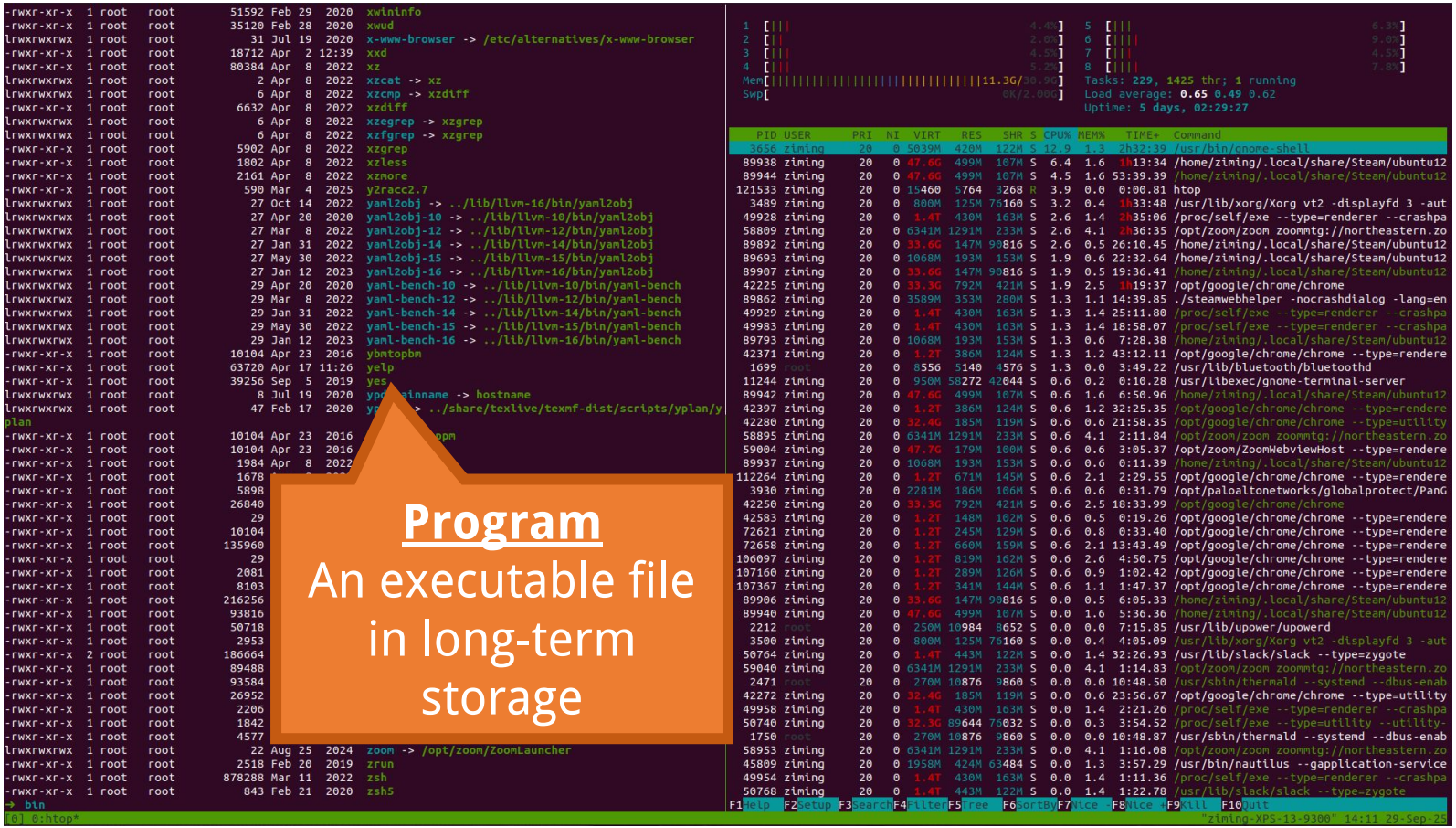
  for example,
  - A command issued at a command line terminal
  - An icon double clicked from the desktop
  - Jobs/tasks run as part of a batch system

- A process is the basic unit of a program in execution. On other words, process is a running program.

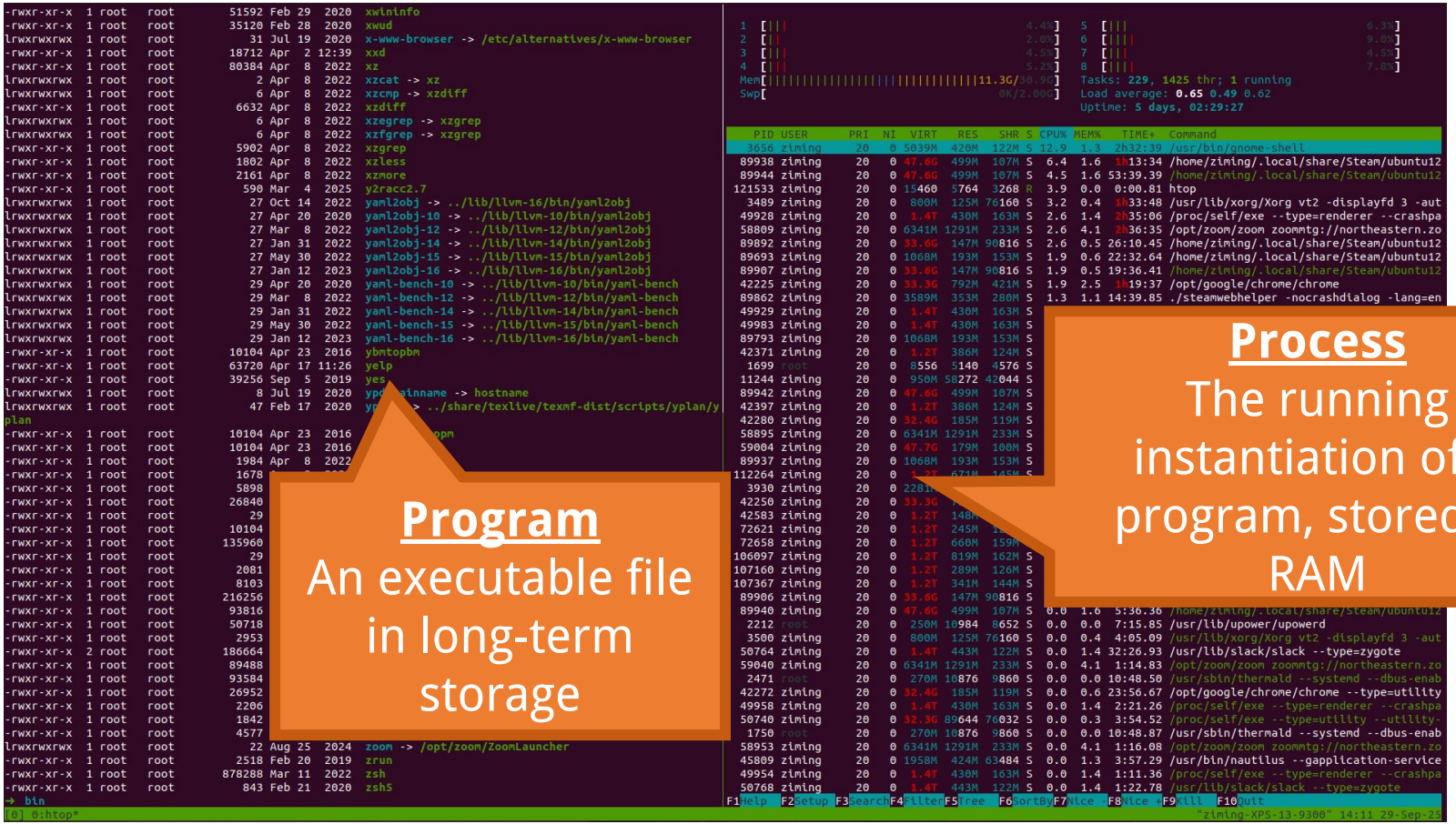
Programs and Processes (Commands: ls ps htop)



Programs and Processes



Programs and Processes



Programs and Processes

```
ziming     50731  0.0  0.4 34498188 146668 ?     Sl   Sep25   1:13 /usr/lib/slack/slack --type=zy
ziming     50734  0.0  0.2 33900548 89724 ?      Sl   Sep25   4:07 /proc/self/exe --type=utility --utilit
ziming     50764  0.5  1.4 1461505468 462016 ?   Sl   Sep25  32:27 /usr/lib/slack/slack --type=zygote
ziming     50784  0.0  0.1 34136220 61784 ?      Sl   Sep25   0:06 /proc/self/exe --type=utility --utilit
ziming     52373  0.0  0.1 1067900 57816 ?       Sl   Sep25   0:01 /usr/bin/gnome-calendar --gapplication
ziming     58804  0.0  0.0   7708  1968 ?        S    Sep25   0:00 /usr/bin/zoom zoommtg://northeastern.z
ziming     58809  2.9  4.0 6493848 1322692 ?     SLl  Sep25 156:39 /opt/zoom/zoom zoommtg://northeastern.
ziming     58823  0.0  0.6 2400544 209692 ?      Sl   Sep25   0:12 /opt/zoom/ZoomWebviewHost
ziming     58830  0.0  0.2 34173764 71188 ?      S    Sep25   0:00 /opt/zoom/ZoomWebviewHost --type=zygot
ziming     58831  0.0  0.2 34173752 68888 ?      S    Sep25   0:00 /opt/zoom/ZoomWebviewHost --type=zygot
ziming     58833  0.0  0.0 34173776 17440 ?      S    Sep25   0:00 /opt/zoom/ZoomWebviewHost --type=zygot
ziming     58858  0.0  0.3 35602488 119860 ?     Sl   Sep25   0:11 /opt/zoom/ZoomWebviewHost --type=gpu-p
ziming     58883  0.0  0.3 1095732 98028 ?       Sl   Sep25   0:01 /opt/zoom/ZoomWebviewHost --type=utili
ziming     58966  0.0  0.1 34545108 45116 ?      Sl   Sep25   0:00 /opt/zoom/ZoomWebviewHost --type=utili
ziming     58973  0.0  0.4 49749360 160980 ?     Sl   Sep25   0:02 /opt/zoom/ZoomWebviewHost --type=rende
ziming     59004  0.0  0.5 49977456 183792 ?     Sl   Sep25   3:05 /opt/zoom/ZoomWebviewHost --type=rende
ziming     59010  0.0  0.6 49751972 195264 ?     Sl   Sep25   0:04 /opt/zoom/ZoomWebviewHost --type=rende
ziming     68676  0.0  0.5 1283861032 184084 ?   Sl   Sep26   0:23 /opt/google/chrome/chrome --type=rende
ziming     72319  0.0  0.6 1285596768 201684 ?   Sl   Sep26   0:18 /opt/google/chrome/chrome --type=rende
ziming     72621  0.0  0.7 1283877664 251512 ?   Sl   Sep26   0:33 /opt/google/chrome/chrome --type=rende
ziming     72630  0.0  0.4 1283855656 142184 ?   Sl   Sep26   0:13 /opt/google/chrome/chrome --type=rende
ziming     72647  0.0  0.4 1283855580 134700 ?   Sl   Sep26   0:12 /opt/google/chrome/chrome --type=rende
ziming     72658  0.3  2.1 1289104872 690636 ?   Sl   Sep26  13:43 /opt/google/chrome/chrome --type=rende
ziming     74032  0.0  0.1 471484 46088 ?        SLl  Sep26   0:01 /usr/bin/seahorse --gapplication-servi
ziming     89520  0.0  0.0  12972  3620 ?        S    Sep26   0:00 bash /home/ziming/.local/share/Steam/s
ziming     89535  0.0  0.0  29684  4440 ?        S    Sep26   0:00 /home/ziming/.local/share/Steam/ubuntu
ziming     89693  0.5  0.6 1094244 198576 ?      Sl   Sep26  22:35 /home/ziming/.local/share/Steam/ubuntu
ziming     89716  0.0  0.0   4632   676 ?        Ss   Sep26   0:00 /home/ziming/.local/share/Steam/steamr
ziming     89729  0.0  0.0  27404  3068 ?        S    Sep26   0:00 /home/ziming/.local/share/Steam/ubuntu
ziming     89817  0.0  0.0 322500  4884 ?        Sl   Sep26   0:00 steam-runtime-launcher-service --along
ziming     89838  0.0  0.0  32276  4808 ?        Ss   Sep26   0:00 /usr/lib/pressure-vessel/from-host/lib
ziming     89862  0.3  1.1 3671232 362312 ?      Sl   Sep26  14:41 ./steamwebhelper -nocrashdialog -lang=
ziming     89865  0.0  0.1 419800 55896 ?        Sl   Sep26   0:00 /home/ziming/.local/share/Steam/ubuntu
ziming     89870  0.0  0.2 34190312 67572 ?      S    Sep26   0:00 /home/ziming/.local/share/Steam/ubuntu
ziming     89871  0.0  0.2 34190300 67452 ?      S    Sep26   0:00 /home/ziming/.local/share/Steam/ubuntu
ziming     89873  0.0  0.0 34190324 16936 ?      S    Sep26   0:00 /home/ziming/.local/share/Steam/ubuntu
ziming     89892  0.6  0.4 35278576 150892 ?     Sl   Sep26  26:13 /home/ziming/.local/share/Steam/ubuntu
ziming     89917  0.0  0.2 1040808 92332 ?       Sl   Sep26   0:01 /proc/self/exe --type=utility --utilit
ziming     89926  0.0  0.1 34562112 41236 ?      Sl   Sep26   0:00 /home/ziming/.local/share/Steam/ubuntu
ziming     89938  1.9  1.5 49949696 507136 ?     Sl   Sep26  73:43 /home/ziming/.local/share/Steam/ubuntu
ziming     90103  0.0  0.2 49758268 87720 ?      Sl   Sep26   0:00 /home/ziming/.local/share/Steam/ubuntu
ziming    105305  0.0  0.0  43996  6576 ?        Ss   Sep28   0:00 /usr/lib/bluetooth/obexd
ziming    105348  0.0  0.6 1283876196 209712 ?   Sl   Sep28   0:12 /opt/google/chrome/chrome --type
ziming    105385  0.0  0.4 1283872056 150432 ?   Sl   Sep28   0:07 /opt/google/chrome/chrome --type=rende
ziming    105886  0.0  0.5 1283873736 172316 ?   Sl   Sep28   0:05 /opt/google/chrome/chrome --type=rende
ziming    106097  0.2  2.5 1285679540 837956 ?   Sl   Sep28   4:50 /opt/google/chrome/chrome --type=rende
ziming    106178  0.0  0.6 1285596948 213200 ?   Sl   Sep28   0:21 /opt/google/chrome/chrome --type=rende
ziming    106213  0.0  0.4 1283863960 144060 ?   Sl   Sep28   0:05 /opt/google/chrome/chrome --type=rende
ziming    106265  0.0  1.0 1295892516 356152 ?   Sl   Sep28   0:41 /opt/google/chrome/chrome --type=rende
ziming    106417  0.0  0.5 1283860776 180696 ?   Sl   Sep28   0:04 /opt/google/chrome/chrome --type=rende
ziming    106653  0.0  0.3 776052 121148 ?       Sl   Sep28   0:03 /opt/sublime_text/sublime_text --detac
ziming    106664  0.0  0.0 152684  1140 ?        Sl   Sep28   0:00 /opt/sublime_text/crash_handler --no-r
ziming    106707  0.0  0.0  57032 16188 ?        Sl   Sep28   0:00 /opt/sublime_text/plugin_host-3.3 1066
ziming    106710  0.0  0.0  73760 25668 ?        Sl   Sep28   0:00 /opt/sublime_text/plugin_host-3.8 1066
```

One-to-many relationship between program and processes

Why so many Chrome processes?

To shield the browser from attacks, Google Chrome developers eyed three key problems.

BY CHARLES REIS, ADAM BARTH, AND CARLOS PIZANO

Browser Security: Lessons from Google Chrome

THE WEB HAS become one of the primary ways people interact with their computers, connecting people with a diverse landscape of content, services, and applications. Users can find new and interesting content on the Web easily, but this presents a security challenge: malicious Web site operators can attack

users through their Web browsers. Browsers face the challenge of keeping their users safe while providing a rich platform for Web applications.

Browsers are an appealing target for attackers because they have a large and complex trusted computing base with a wide network-visible interface. Historically, every browser at some point has contained a bug that let a malicious Web site operator circumvent the browser's security policy and compromise the user's computer. Even after these vulnerabilities are patched, many users continue to run older, vulnerable versions.[1] When these users visit malicious Web sites, they run the risk of having their computers compromised.

Generally speaking, the danger posed to users comes from three factors, and browser vendors can help keep their users safe by addressing each of these factors:

▸ *The severity of vulnerabilities.* By sandboxing their rendering engine, browsers can reduce the severity of vulnerabilities. Sandboxes limit the damage that can be caused by an attacker who exploits a vulnerability in the rendering engine.

https://dl.acm.org/doi/pdf/10.1145/1536616.1536634

How to Run a Program?

- How does the OS turn an executable file into a process?

- What information must an executable file contain to run as a program?

Program Formats

- Programs obey specific file formats
  - Unix/Linux: Executable and Linkable Format (ELF)
  - Mac OSX: Mach object file format (Mach-O)
  - Windows Portable Executable (PE, PE32+) (*.exe)
    - Modified version of Unix COFF executable format
  - PE files start with an MZ header.

  - CP/M (control program monitor) and DOS (disk operating system) : COM executables (*.com)
  - DOS: MZ executables (*.exe)
    - Named after Mark Zbikowski, a DOS developer

Program Formats

Format	Platform	Spec availability	Ownership
ELF	Linux, BSD, Unix	Open System V ABI	Community / Linux Foundation
PE	Windows	Public (Microsoft PE/COFF spec)	Microsoft
Mach-O	macOS, iOS	Public headers/docs by Apple	Apple

"**System V**" (often written **SysV**) was a line of commercial UNIX from AT&T in the 1980s. **System V ABI** is a set of open specifications defining: (1) the **Executable and Linkable Format (ELF)** for object files and executables. (2) Calling conventions, symbol tables, relocation formats, dynamic linking rules, etc.

The docs were published and made freely available, not proprietary or locked. That's why Linux, BSD, Solaris, and many other Unix-like OSes could adopt ELF with no licensing restrictions.

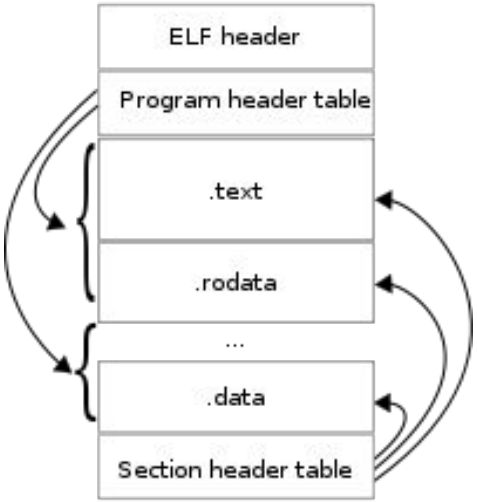
ELF File Format

- Spec: https://refspecs.linuxfoundation.org/elf/elf.pdf

- 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

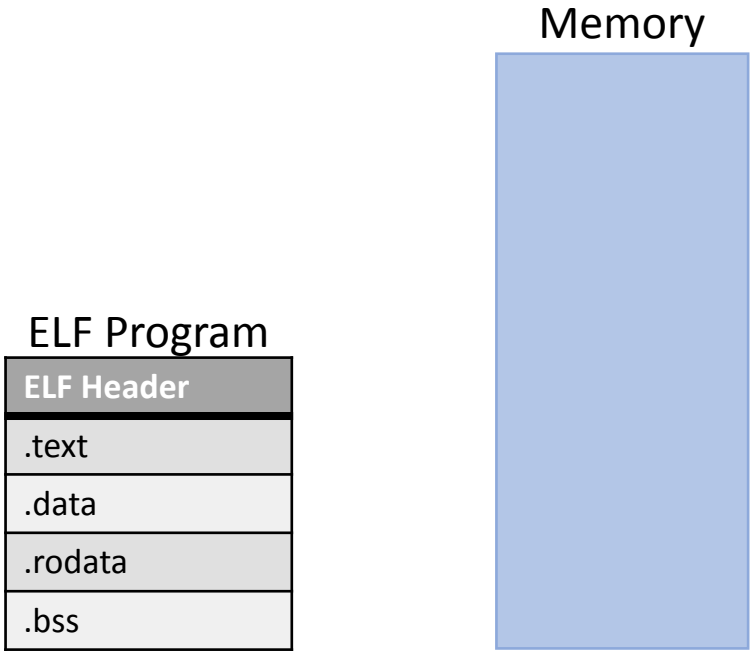


ELF Header Example

```
@zzm7000 → /workspaces/assignment-3-zzm7000 (main) $ readelf -a ./a.out
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:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1040
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13928 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
```

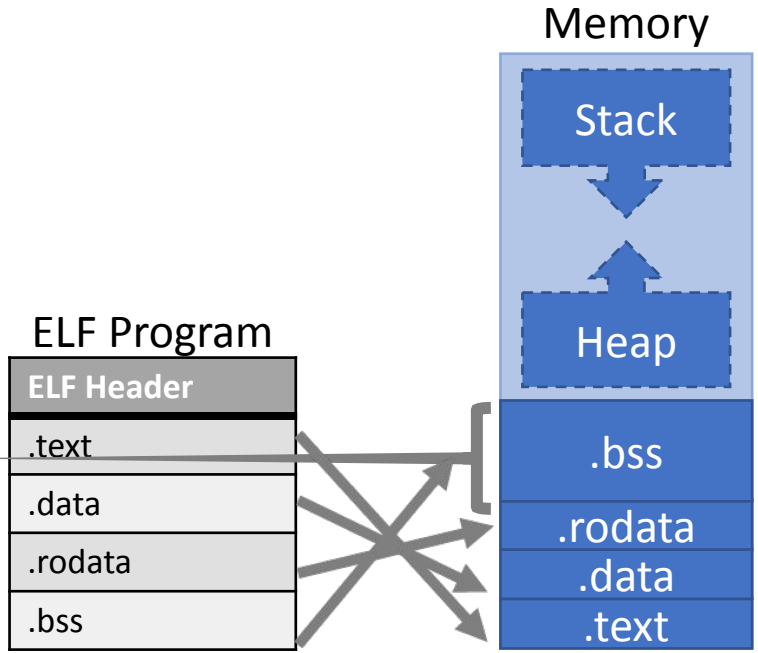
The Program Loader

- OS functionality that loads programs into memory, creates processes
  - Places segments into memory
  - Loads necessary dynamic libraries
  - Performs relocation
  - Allocated the initial stack frame
  - Sets EIP to the programs entry point
- Process is a live program execution context or basic unit of execution



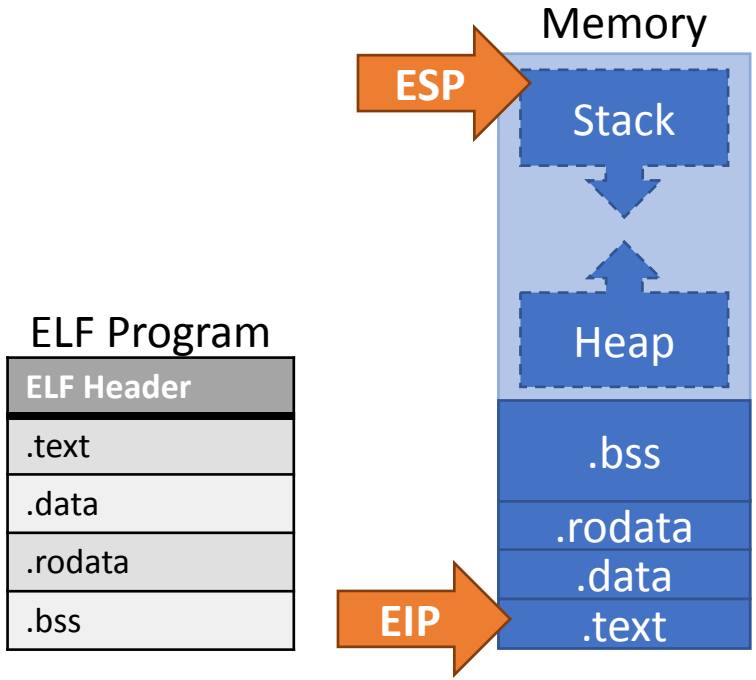
The Program Loader

- OS functionality that loads programs into memory, creates processes
  - Places segments into memory
  - Loads necessary dynamic libraries
  - Performs relocation
  - Allocated the initial stack frame
  - Sets EIP to the programs entry point
- Process is a live program execution context or basic unit of execution



The Program Loader

- OS functionality that loads programs into memory, creates processes
  - Places segments into memory
  - Loads necessary dynamic libraries
  - Performs relocation
  - Allocated the initial stack frame
  - Sets EIP to the programs entry point
- Process is a live program execution context or basic unit of execution



xv6 as an example

Background: xv6 programs are statically linked. All the user programs you compile for xv6 (like ls, sh, etc.) are **fully statically linked** into one ELF file — no shared libraries. Kernel itself is the loader.

In exec.c (exec() in kernel/exec.c):

- The kernel opens the ELF file from the file system.
- Reads the ELF header (struct elfhdr) and each program header (struct proghdr).
- For each loadable segment, it allocates memory, maps it into the new process's address space, and copies the data from the file.
- Sets up the stack and the initial instruction pointer to the ELF's entry address.

https://github.com/mit-pdos/xv6-public/blob/master/exec.c

xv6 as an example

myproc() *(kernel/proc.c)*
Returns a pointer to the **current process** (struct proc *) running on this CPU. It looks up the per-CPU structure and fetches its proc field.

setupkvm() *(kernel/vm.c)*
Builds a **fresh kernel page table** (kernel virtual memory only): maps kernel text/data, devices, I/O space, etc. Used when creating a new address space for a process (before loading user pages).

allocuvm(pagetable, oldsz, newsz) *(kernel/vm.c)*
Grows a process's user address space from oldsz to newsz. Allocates physical pages and maps them into pagetable with user permissions. Returns the new size (or 0 on failure).

loaduvm(pagetable, va, ip, offset, sz) *(kernel/vm.c / kernel/exec.c)*
Loads program bytes from disk into user memory: reads sz bytes from inode ip at file offset and copies them into the user virtual region starting at virtual address va (assumes those pages are already allocated/mapped). Used by exec() to load ELF segments.

switchuvm(p) *(kernel/vm.c / kernel/proc.c)*
Switches the hardware MMU to a process's page table (and does any CPU/TSS setup on x86). Called when the scheduler is about to run process p, or when a process's memory image has just changed (e.g., after exec).

freevm(pagetable) *(kernel/vm.c)*
Frees a process's user page table and all user pages it points to (tears down the address space). Used on process exit or on exec failure paths.

Linux as an example

How it starts:

1. You run an ELF file => kernel's execve() system call reads its ELF headers.

2. If the file has an INTERP section, the kernel:
   - Loads that interpreter binary (e.g., ld-linux-x86-64.so.2) into memory.
   - Jumps to its entry point **in user space**.

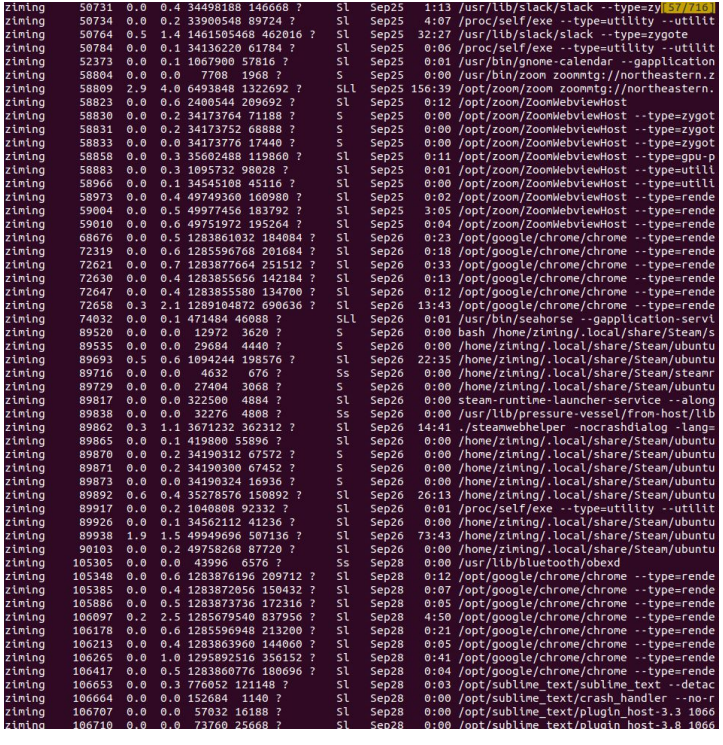
3. The interpreter (now running in ring 3) maps your program's segments and shared libraries, relocates symbols, and finally transfers control to your program's _start.

```
ldd /usr/bin/ls
linux-vdso.so.1 (0x00007ffc3ed1f000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9ec75ce000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9ec73dc000)
libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007f9ec734b000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9ec7345000)
/lib64/ld-linux-x86-64.so.2 (0x00007f9ec764c000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9ec7322000)
```

Warmup

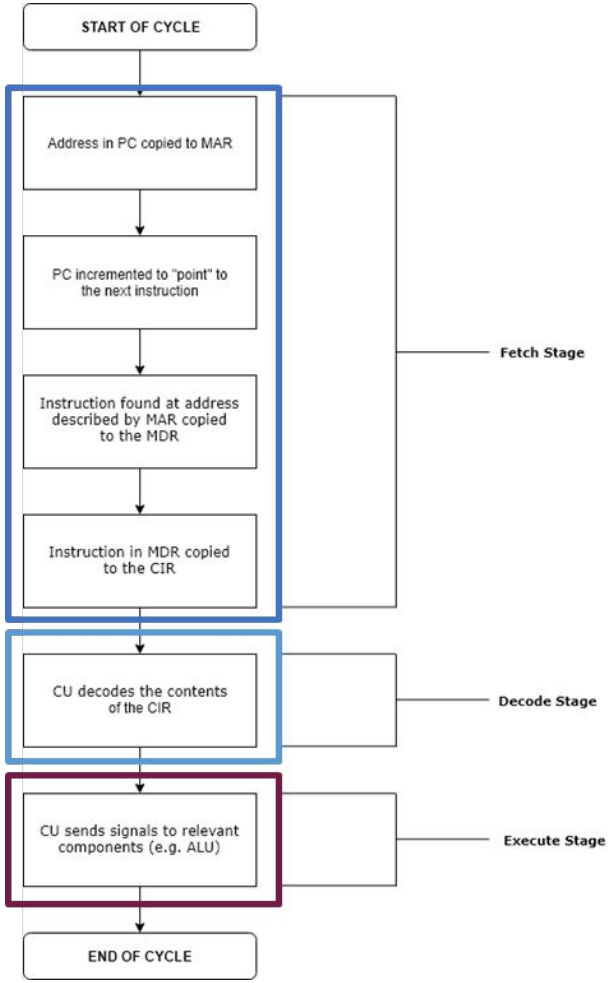
- How many processes do you have open at any given time?
  - 10s, 100s? More!? :)

ps -aux | wc -l



First: Instruction Execution

- Code in an executable is a sequence of instructions
- A CPU/core runs an instruction at a time
- This is done in a fetch-decode-execute cycle
- If you have **4 cores**, your processor can do **4 FDE cycles** at a time
- But how do we see ~100s of programs running on 4 cores?
- What about a single core CPU?

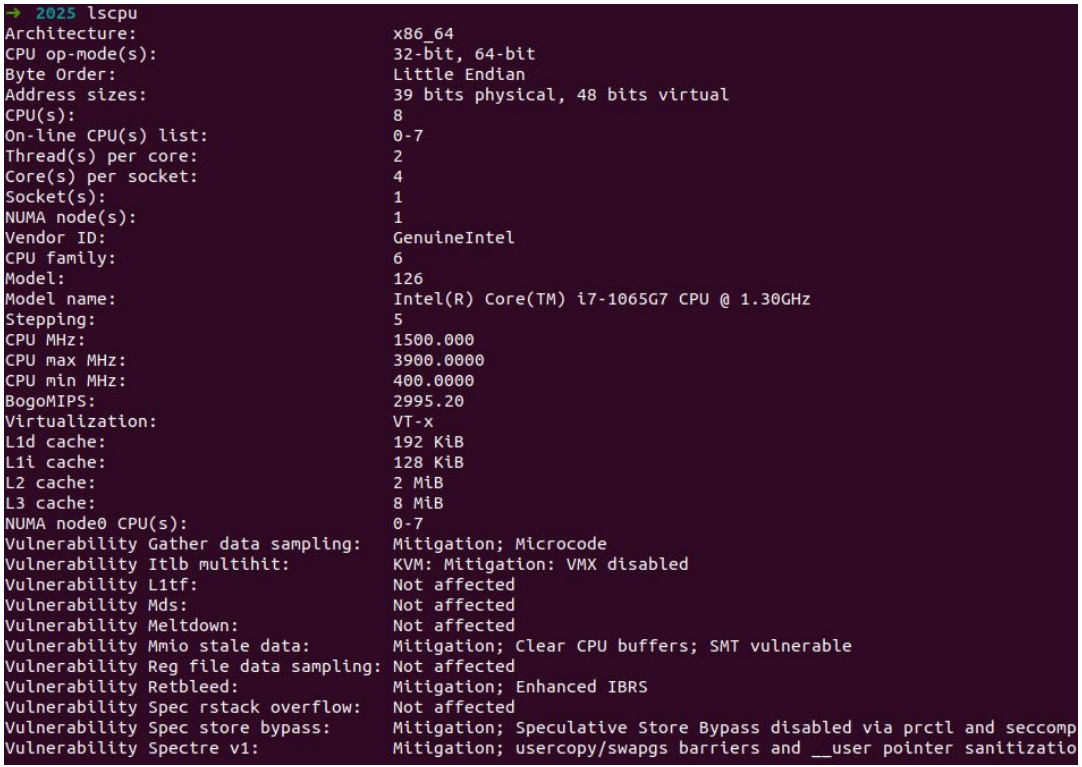


Internal CPU registers that implement the fetch–decode–execute cycle:
Memory Address Register: holds address of current instruction
Memory Data Register: holds contents of address in MAR
Current Instruction Register: stores current instruction, so not overwritten by additional fetches to MBR/MDR

From the warm up

- Many programs are running, but only 8 CPUs that do the work

lscpu



From the warm up

- Many programs are running, but only 8 CPUs that do the work

The Problem: So how does our Operating System provide the illusion of hundreds of processes running at once?

Virtualization with time sharing

- If one CPU, the Operating System runs one process at a time,
  - That executes one instruction a time
    - After some amount of time the process stops or finishes
    - Then the OS starts another process
    - Eventually the same process will run again and continue where it left off
    - Repeat

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process_1 now done

Virtualization with time sharing

- If one CPU, the Operating System runs one process at a time,
  - That executes one instruction a time
    - After some amount of time the process stops or finishes
    - Then the OS starts another process
    - Eventually the same process will run again and continue where it left off
    - Repeat

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

  - This concept is known as time sharing
  - Are the two states, **Running** and **Ready**, enough?

Process States

- What if the process needs to read/write to disk or perform a network request? Any problems?

Process States

- What if the process needs to read/write to disk or perform a network request? Any problems?
  - These operations take (comparatively) long to complete

Process States

- What if the process needs to read/write to disk or perform a network request? Any problems?
  - These operations take (comparatively) long to complete
  - Keeping process state to **Running?**
  - Keeping process state to **Ready?**

Process States

- What if the process needs to read/write to disk or perform a network request? Any problems?
  - These operations take (comparatively) long to complete
  - Keeping process state to **Running?**
    - Hogs the CPU just waiting for disk/network access to complete
  - Keeping process state to **Ready?**

Process States

- What if the process needs to read/write to disk or perform a network request? Any problems?
  - These operations take (comparatively) long to complete
  - Keeping process state to **Running?**
    - Hogs the CPU just waiting for disk/network access to complete
  - Keeping process state to **Ready?**
    - Might not be ready to run when its turn comes
    - Asking it to run may be waste of time

Process States

- What if the process needs to read/write to disk or perform a network request? Any problems?
  - These operations take (comparatively) long to complete
  - Keeping process state to **Running?**
    - Hogs the CPU just waiting for disk/network access to complete
  - Keeping process state to **Ready?**
    - Might not be ready to run when its turn comes
    - Asking it to run may be waste of time
- Solution?

Process States

- What if the process needs to read/write to disk or perform a network request? Any problems?
  - These operations take (comparatively) long to complete
  - Keeping process state to **Running?**
    - Hogs the CPU just waiting for disk/network access to complete
  - Keeping process state to **Ready?**
    - Might not be ready to run when its turn comes
    - Asking it to run may be waste of time
- Solution?
  - Introduce a 3rd state, **Blocked**
    - Meaning: the process requested some I/O operation and cannot run until that operation is completed

Process States

- Each process can be in one of several states
- The OS schedules the state the process is in
- Typically, these are:
  - Running: the process is executing on the CPU
  - Ready: the process is ready to execute, but the OS did not choose to run it
  - Blocked - the process issued some blocking operation
    - I/O is a common blocking operation

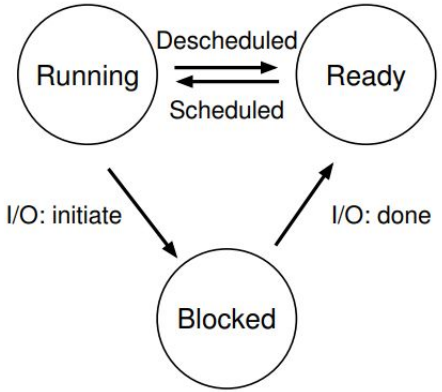


Figure 4.2: **Process: State Transitions**

xv6 as an example

- **UNUSED** – empty slot in the process table; no process here.
- **EMBRYO** (x86) / **USED** (RISC-V) – slot allocated and being initialized (after allocproc()), but not runnable yet.
- **SLEEPING** – blocked, waiting on some event (a “channel”), e.g., disk I/O to finish.
- **RUNNABLE** – ready to run; waiting for CPU.
- **RUNNING** – currently executing on a CPU.
- **ZOMBIE** – finished execution (exit() called), but parent hasn’t wait()ed yet; holds exit status/resources to be reaped.

https://github.com/mit-pdos/xv6-public/blob/master/proc.h

Why I Don’t Feel Apps Lagging Even Though They Time-Share the CPU?

Then how does OS switch processes?

OS Challenges to Virtualization

- Performance
  - How to implement virtualization without excessive overhead?

- Control
  - How to run multiple processes without losing control over the CPU?
  - Without OS control, a process
    - could occupy the CPU and run forever
    - access memory it does not have access impacting safety and security

Switching between processes

- Switching between processes is a challenge, because

  If the CPU is running a program, then the OS is not running

Switching between processes

- Switching between processes is a challenge, because

  If the CPU is running a program, then the OS is not running

- If OS is not running, then how can it switch out/in processes?
  - Think about how you would design the OS!

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:

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()

Voluntary Yielding

- Idea: processes must voluntary give up control by calling an OS API, e.g. thread_yield()

- Problems?

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
    e.g., while (1) { //do something without yielding }
  - No guarantee that apps will yield in a reasonable amount of time
  - Waste 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

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?

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 call OS APIs
  - Some normal apps don't use OS APIs for long periods of time
    - E.g. a long, CPU intensive matrix calculation

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

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?

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 Switching

- So far, processes will not switch to another until an action is taken
  - e.g. an API call or an I/O interrupt

- Idea: use a timer to force context switching at set intervals
  - Timer is running 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

Preemptive Switching

- So far, processes will not switch to another until an action is taken
  - e.g. an API call or an I/O interrupt

- Idea: use a timer to force context switching at set intervals
  - Timer is running 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? Who will trigger the timer

Preemptive Switching

- So far, processes will not switch to another until an action is taken
  - e.g. an API call or an I/O interrupt

- Idea: use a timer to force context switching at set intervals
  - Timer is running 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? Who will trigger the timer
  - Requires hardware support (a programmable timer)
    - Thankfully, this is built-in to most modern CPUs

Why I Don’t Feel Apps Lagging Even Though They Time-Share the CPU?

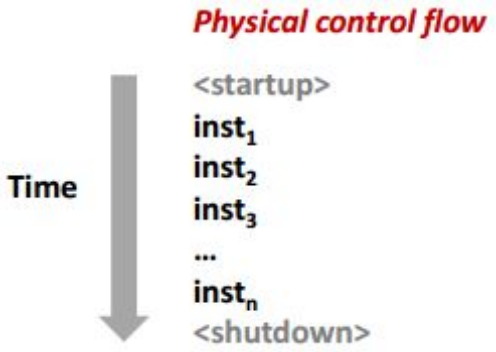
Linux as an Example

On most desktop Linux systems, the default scheduler “tick” runs **250 Hz or 1000 Hz** — meaning the kernel timer fires every **4 ms or 1 ms**. This timer tick alone lets the scheduler run up to **1000 times per second per CPU**, checking whether to switch processes. On top of this, Linux can also reschedule when events happen (I/O completes, a higher-priority process wakes up), so the system can respond even faster when needed.

Mechanisms for switching: Exceptional Control Flow

Normal Control Flow or Regular Execution

- Computers only really do one thing, they execute one instruction one after another
  - This is based on the execution in your program.
  - Your programs follow some control flow based on jumps and branches (and calls and returns)
    - This is based on your programs state.



Reasons to break normal control flow

However, sometimes we want to break normal control flow

- Multitasking. Stop one so others can run.
- The program needs to take care of user input. E.g., you hit Ctrl+C on the keyboard in your terminal and execution stops.
- Program can make a system call.

Exceptional Control Flow Mechanisms

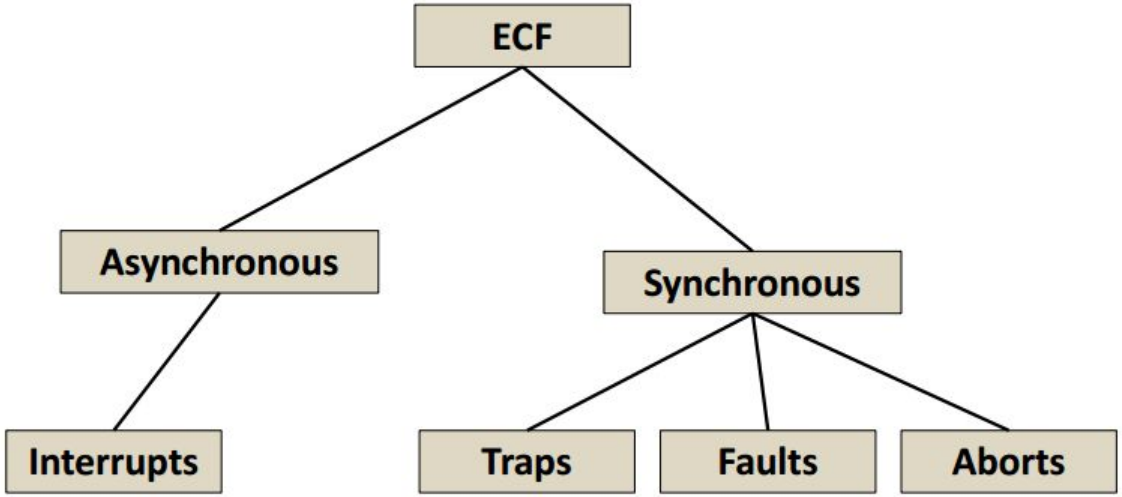
- Low level mechanism
  - ***Exceptions***
    - Change in control flow in response to a system event.
    - This is implemented in hardware and OS software

Exceptional Control Flow Mechanisms

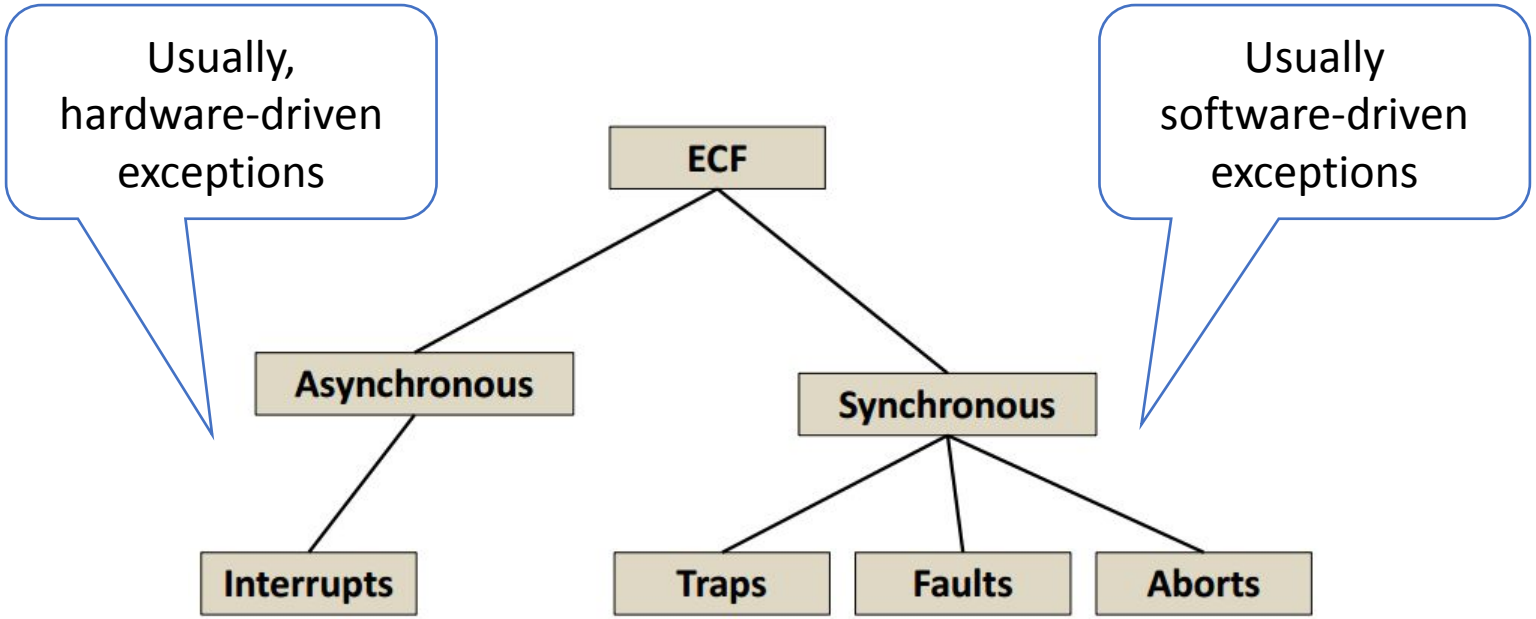
- High level mechanisms
  - Process context switch
    - e.g. It appears that multiple programs are running at once on your OS, but remember only one instruction at a time.
    - Context switches provide this illusion
  - Signals
    - Implemented by OS software

Exceptional Control Flow Taxonomy

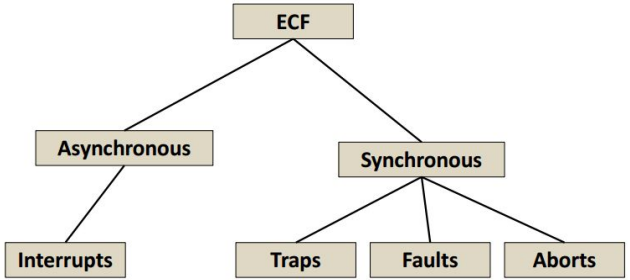
Exception (broad sense): any event that causes the CPU to stop its current normal execution path and transfer control to special handler code.



Exceptional Control Flow Taxonomy



Asynchronous Exceptions (Interrupts)



- Caused by events external to processor
  - I.e., not from the result of an instruction the user wrote
  - E.g.
    - Timer interrupts scheduled to happen every few milliseconds
      - A kernel can use this to take back control from a program/user
    - Some network data arrives (I/O)
    - A nice example is while reading from disk
      - The processor can start reading, then hop over and perform some other tasks until memory is actually fetched.

```
→ CactiLab.github.io git:(master) cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
  1:          0          0          0          0          0          0          0     130731  IR-IO-APIC    1-edge      i8042
  8:          0          0          0          0          0          0          0          0  IR-IO-APIC    8-edge      rtc0
  9:          0    4232726          0          0          0          0          0          0  IR-IO-APIC    9-fasteoi   acpi
 12:          0          0          0          0          0          0        140          0  IR-IO-APIC   12-edge      i8042
 14:          0          0     261613          0          0          0          0          0  IR-IO-APIC   14-fasteoi   INT3455:00
 16:          0          0          0    2043581          0          0          0          0  IR-IO-APIC   16-fasteoi   intel_ish_ipc, idma64.0, i801_smbus, i2c_designware.0
 17:          0          0       1514     114950   16686610      12166          0          0  IR-IO-APIC   17-fasteoi   idma64.1, i2c_designware.1
 20:          0          0          0          0          0          0          0          0  IR-IO-APIC   20-fasteoi   idma64.2
120:          0          0          0          0          0          0          0          0  DMAR-MSI    0-edge      dmar0
121:          0          0          0          0          0          0          0          0  DMAR-MSI    1-edge      dmar1
122:          0          0          0          0          0          0          0          0  IR-PCI-MSI 114688-edge      PCIe PME, pciehp
123:          0          0          0          0          0          0          0          0  IR-PCI-MSI 118784-edge      PCIe PME, pciehp
124:          0          0          0          0          0          0          0          0  IR-PCI-MSI 475136-edge      PCIe PME
125:          0          0          0          0          0          0          0          0  IR-PCI-MSI 489472-edge      PCIe PME, pciehp
126:          0          0          0          0          0        733          0          0  IR-PCI-MSI 217088-edge      thunderbolt
127:          0          0          0          0          0          0       1017          0  IR-PCI-MSI 217089-edge      thunderbolt
142:      20318     286966        625      28130      78172       2829          7       3437  IR-PCI-MSI 212992-edge      xhci_hcd
143:          0          0          0          0          0          0          0        106  IR-PCI-MSI 46137344-edge      rtsx_pci
144:    2233079    6412488    2024743   13565716      88127    1357648    2781621     385438  IR-PCI-MSI 327680-edge      xhci_hcd
145:          1         28          0          0          3          0         72          0  IR-PCI-MSI 45613056-edge      nvme0q0
146:     452940          0          0          0          0          0          0          0  IR-PCI-MSI 45613057-edge      nvme0q1
147:          0     460347          0          0          0          0          0          0  IR-PCI-MSI 45613058-edge      nvme0q2
148:          0          0     435105          0          0          0          0          0  IR-PCI-MSI 45613059-edge      nvme0q3
149:          0          0          0     452592          0          0          0          0  IR-PCI-MSI 45613060-edge      nvme0q4
150:          0          0          0          0     448577          0          0          0  IR-PCI-MSI 45613061-edge      nvme0q5
151:          0          0          0          0          0     477954          0          0  IR-PCI-MSI 45613062-edge      nvme0q6
152:          0          0          0          0          0          0     462865          0  IR-PCI-MSI 45613063-edge      nvme0q7
153:          0          0          0          0          0          0          0     452517  IR-PCI-MSI 45613064-edge      nvme0q8
154:          0          0     261613          0          0          0          0          0  INT3455:00   35  DLL096D:01
155:    5064963      17310    3206231     220145    4504975    5283354   58368619     499374  IR-PCI-MSI 32768-edge      i915
156:          0          0          0          0          0        733          0          0  IR-PCI-MSI 219136-edge      thunderbolt
157:          0          0          0          0          0          0       1025          0  IR-PCI-MSI 219137-edge      thunderbolt
172:         48          0          0          0          0          0          0          0  IR-PCI-MSI 360448-edge      mei_me
173:     322800    1008195    1919846    1051926    1934430     241138       6366    1054598  IR-PCI-MSI 333824-edge      iwlwifi:default_queue
174:      77855      68860      52230     149625      85835      89668     194793     115748  IR-PCI-MSI 333825-edge      iwlwifi:queue_1
175:      55587      48889     168899      79574      96700     157103      44309      82157  IR-PCI-MSI 333826-edge      iwlwifi:queue_2
176:     109035      47673      42471     586984      93652     115404      39459      80553  IR-PCI-MSI 333827-edge      iwlwifi:queue_3
177:      50653      64542      36201      78944      81029      89647      47000      86400  IR-PCI-MSI 333828-edge      iwlwifi:queue_4
178:      74801      37065      36703      88397     190921      84557      37344     192138  IR-PCI-MSI 333829-edge      iwlwifi:queue_5
179:      52167      76304     391103     169174     130707     429014      49444      73248  IR-PCI-MSI 333830-edge      iwlwifi:queue_6
180:      78602      57998      97479     112693      69675      80179      86634     321517  IR-PCI-MSI 333831-edge      iwlwifi:queue_7
181:      79443      83061     177047      98436      76296     110730      44851      71342  IR-PCI-MSI 333832-edge      iwlwifi:queue_8
182:          1          3          0          0          1          0          3         47  IR-PCI-MSI 333833-edge      iwlwifi:exception
183:          0          0          0          0          0          0          0          0  als-dev0       als_consumer0
185:        406          0          0          0          0          0          0          0  IR-PCI-MSI 514048-edge      snd_hda_intel:card0
NMI:          0          0          0          0          0          0          0          0   Non-maskable interrupts
LOC:   65631155   64596281   63676227   64238073   64059082   64735579   67949340   63804742   Local timer interrupts
SPU:          0          0          0          0          0          0          0          0   Spurious interrupts
PMI:          0          0          0          0          0          0          0          0   Performance monitoring interrupts
IWI:    3121548     610104    2096845     704034    2787303    3280945   30305026     917879   IRQ work interrupts
RTR:          2          0          0          0          0          0          0          0   APIC ICR read retries
RES:     933823     907409     925545     917141     956591     955761    1183767     943247   Rescheduling interrupts
CAL:   14480409   13290398   12253091   11837881   11574765   11343450   11072357   11242028   Function call interrupts
TLB:    7137555    7250981    7202742    7093647    7180035    7198626    6894040    7193845   TLB shootdowns
TRM:        978        978        978        978        978        978        978        978   Thermal event interrupts
THR:          0          0          0          0          0          0          0          0   Threshold APIC interrupts
DFR:          0          0          0          0          0          0          0          0   Deferred Error APIC interrupts
MCE:          0          0          0          0          0          0          0          0   Machine check exceptions
MCP:        858        859        859        859        859        859        859        859   Machine check polls
ERR:          0
MIS:          0
PIN:          0          0          0          0          0          0          0          0   Posted-interrupt notification event
NPI:          0          0          0          0          0          0          0          0   Nested posted-interrupt event
PIW:          0          0          0          0          0          0          0          0   Posted-interrupt wakeup event
```

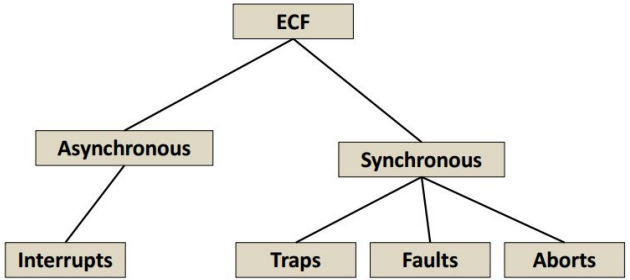
On Linux

cat /proc/interrupts

grep '^CONFIG_HZ=' /boot/config-$(uname -r)

65,631,155/250 ≈ 262,500 seconds ≈ 3 days of active ticks

Synchronous Exceptions



- Events caused by executing an instruction
  - Traps
    - Intentionally done by the user
      - e.g. system calls, breakpoints (like in gdb)
    - Returns control to the next instruction
  - Faults
    - Unintentional, but possibly recoverable
      - e.g. page faults (we'll learn more about soon), floating point exceptions
    - Handled by re-executing current instruction or aborting execution
  - Aborts
    - Unintentional and unrecoverable
      - e.g. illegal instruction executed, parity error

```
→ CactiLab.github.io git:(master) grep pgfault /proc/vmstat
pgfault 1432038924
→ CactiLab.github.io git:(master) grep pgmajfault /proc/vmstat
pgmajfault 34497
```

What is a page fault?

The CPU tried to access a virtual page not currently mapped.

Two types

- **Minor fault:** Page not mapped yet but data already in RAM; kernel just sets up page tables (fast).
- **Major fault:** Page must be read from disk or swap (slow).

Why so many?

- Linux uses **demand paging** — memory isn't backed until first use.
- Shared libraries and file mappings are faulted in on demand.
- Growing stacks and heap also cause minor faults.

Key metrics

- pgfault — total page faults (mostly minor).
- pgmajfault — major faults (performance cost).

Software breakpoint — int3 / #BP

The debugger overwrite the first byte of the target instruction with the opcode 0xCC (INT3).

When the CPU fetches and executes that byte, it **generates #BP (Breakpoint Exception, vector 3)**.

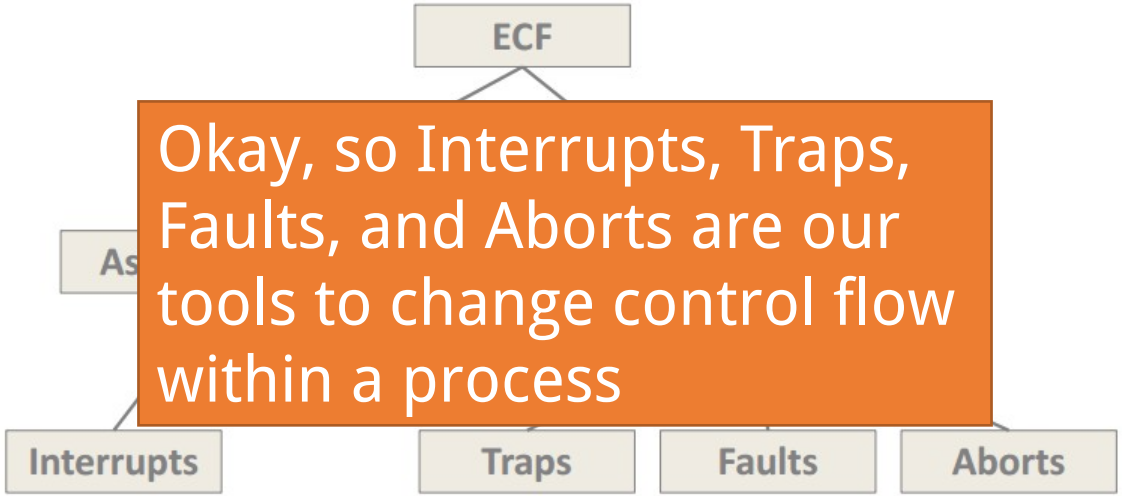
The OS delivers this exception back to the debugger (e.g., via ptrace on Linux).

After stopping, the debugger typically restores the original byte so the instruction can run if you continue.

No debug registers involved.

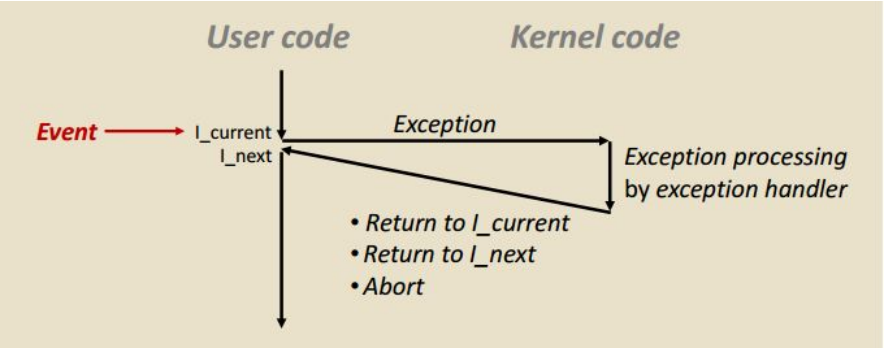
In 8086 assembly, `INT 3` is a special one-byte instruction (opcode `0xCC`)

Exceptional Control Flow Taxonomy



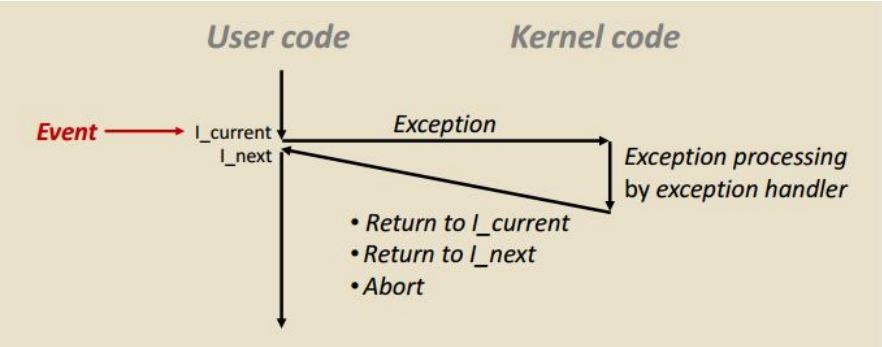
Exceptions

- An exception is a transfer of control to the OS kernel
  - The kernel is the memory-resident part of the OS
    - Meaning OS lives in memory forever: we do not modify this!
- Examples of exceptions we may be familiar with:
  - Divide by 0, arithmetic overflow, or typing Ctrl+C



Exceptions

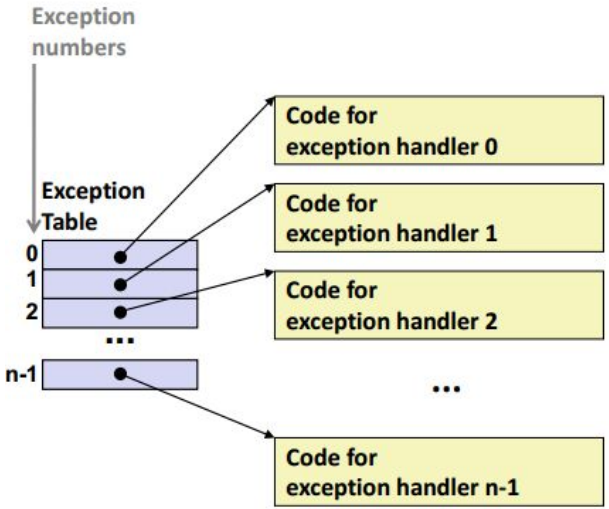
- An exception is a transfer of control to the OS kernel
  - The kernel is the memory-resident part of the OS
    - Meaning OS lives in memory forever: we do not modify this!
- Examples of exceptions we may be familiar with:
  - Divide by 0, arithmetic overflow, or typing Ctrl+C



- How does the OS know how to handle the exception?

Exception Tables

- Somewhere in the OS, a table exists with different exceptions.
  - Think of it like a giant switch or many if else-if statements.
- This is part of a kernel that you cannot modify.
  - This code is in a **“protected region”** of memory
- For each exception, there is one way to handle it
  - (We call these **“exception handlers”**)

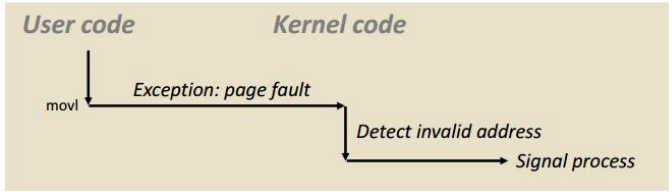


Our favorite: Invalid Memory Reference

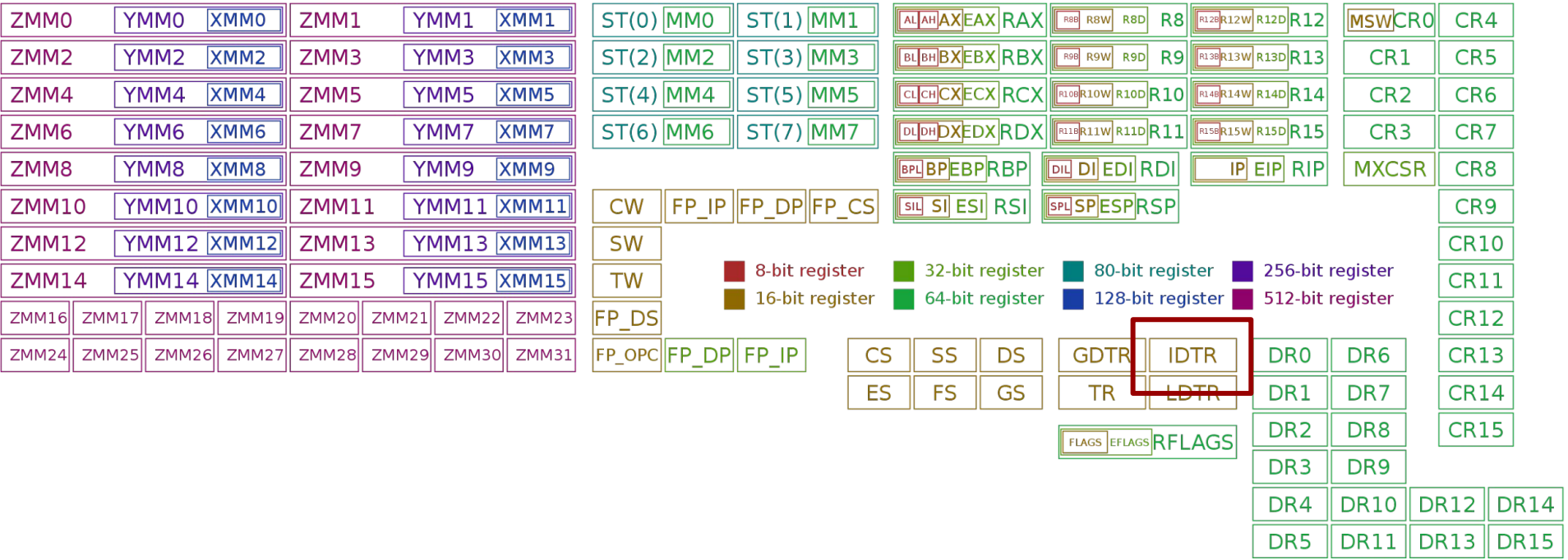
- That is, the segmentation fault
  - OS sends signal SIGSEGV to our user process
  - This time the program gets terminated.

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:    c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```

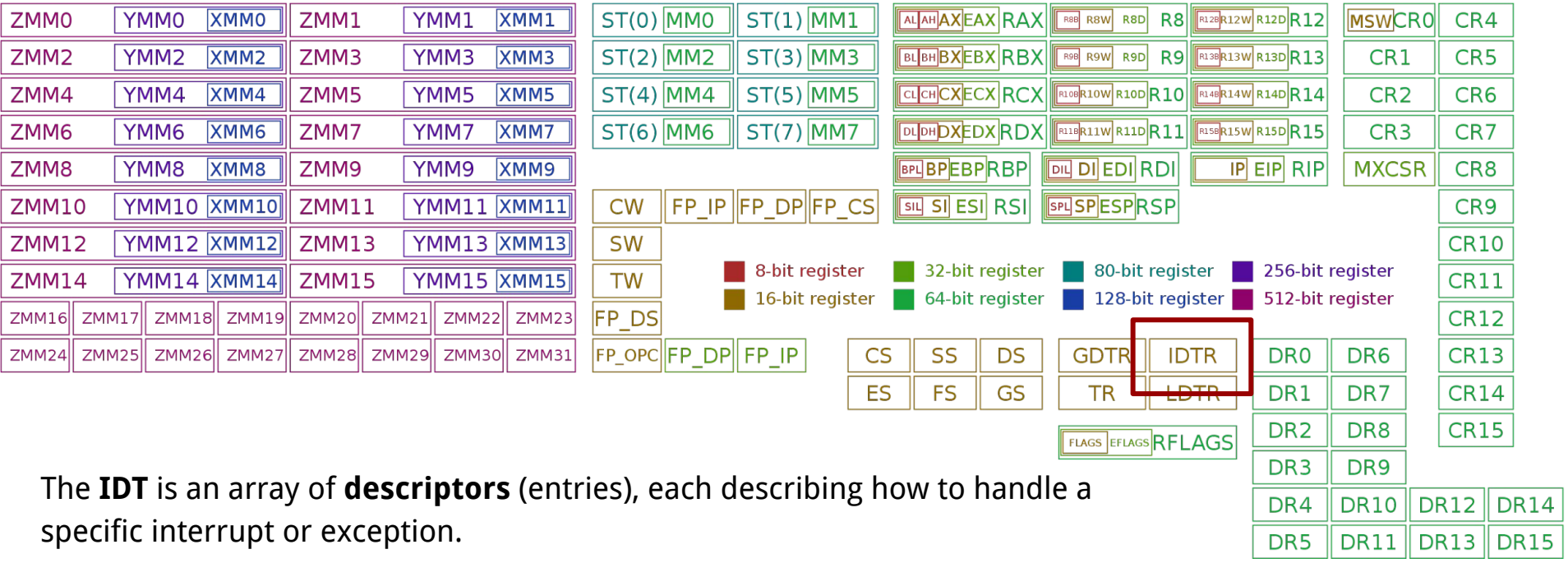


x86 and xv6 as an example



In **x86 processors**, the **Interrupt Descriptor Table (IDT)** is a special data structure the CPU uses to look up what code to run when an **interrupt**, **exception**, or **trap** happens.

x86 and xv6 as an example



The **IDT** is an array of **descriptors** (entries), each describing how to handle a specific interrupt or exception.

Each entry tells the CPU:

- The **address** of the interrupt/trap handler function (in kernel space).
- The **segment selector** to use (usually the kernel code segment).
- Attributes such as privilege level and gate type.

x86 and xv6 as an example

https://cdrdv2.intel.com/v1/dl/getContent/671447

x86 and xv6 as an example

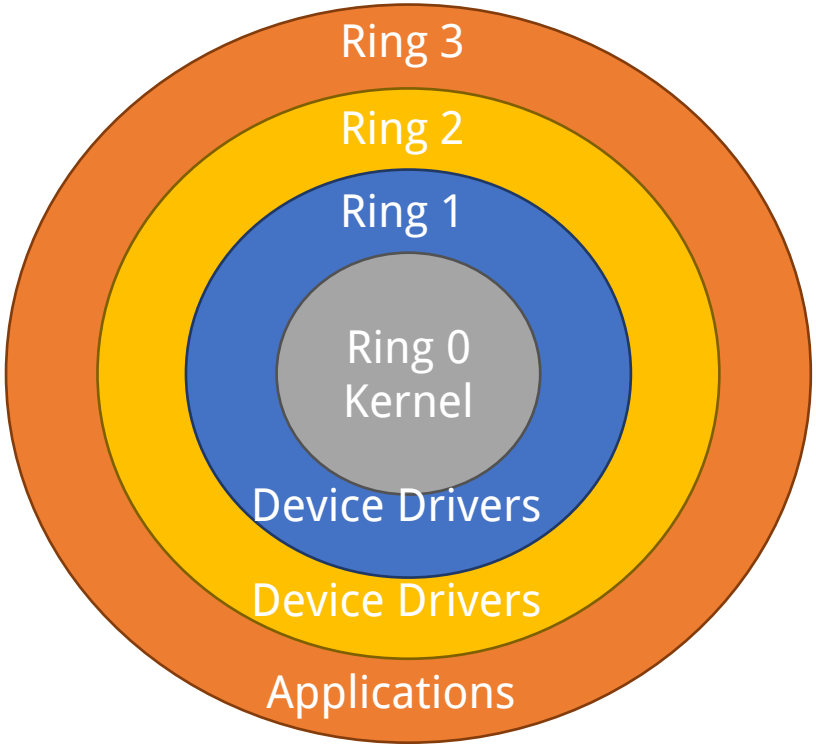
```
// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;

void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}

void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

https://github.com/mit-pdos/xv6-public/blob/master/trap.c

System Calls

Different privilege levels

- Most modern CPUs support protected mode
- x86 CPUs support three rings with different privileges
  - Ring 0: OS kernel. Most privileged. Full hardware access.
  - Ring 1, 2: device drivers
  - Ring 3: userland. Restricted.
- Most OSes only use rings 0 and 3



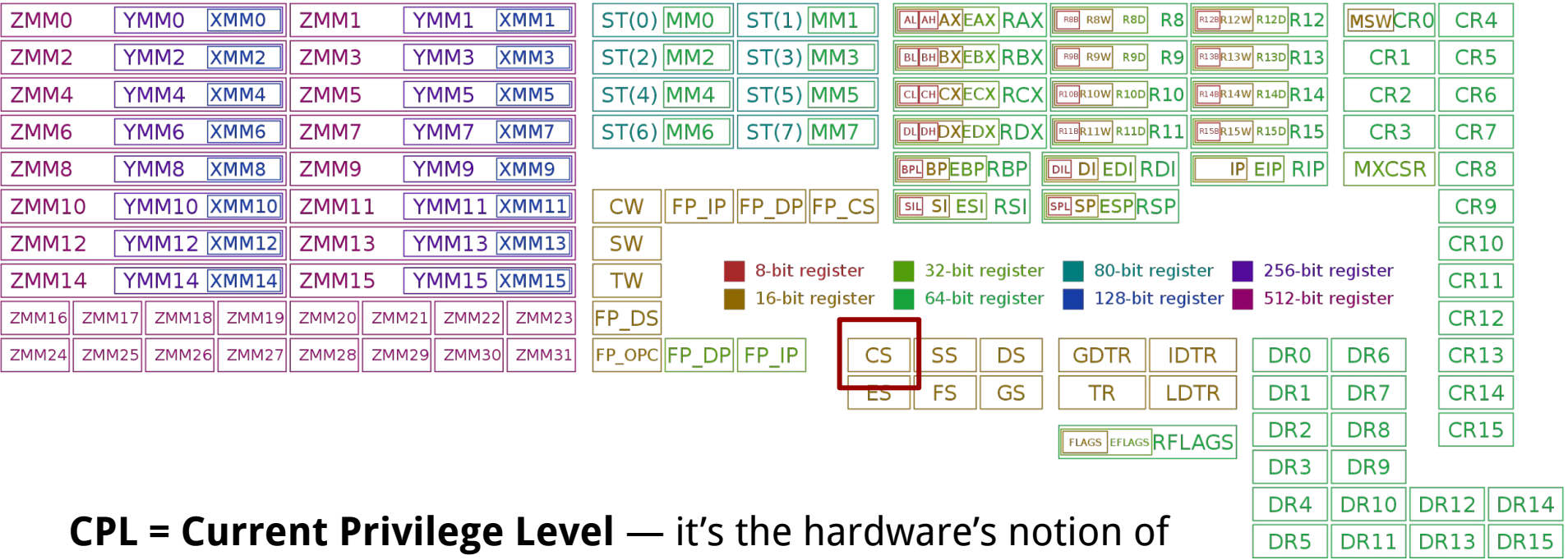
Dual-Mode Operation

- 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

x86-64 CPL - Current Privilege Level



CPL = Current Privilege Level — it's the hardware's notion of "what ring am I running in right now."

CPL is stored in the **lowest two bits of the CS (Code Segment) selector register** on x86/x86-64.

Key Differences: Ring 0 vs Ring 3

Feature	Ring 0 (Kernel)	Ring 3 (User)
Privilege level	Highest (CPL=0)	Lowest (CPL=3)
Access to hardware	Direct (I/O ports, control registers)	No direct hardware access
Memory access	Can map & modify page tables	Limited to user space virtual memory
Instructions allowed	All x86 privileged instructions	Non-privileged subset only
System calls	**Not needed (already privileged)**	**Must trap to kernel via syscalls / int ?? /syscall**

What are System Calls?

When a process needs to invoke a kernel service, it invokes a procedure call in the operating system interface using special instructions (not a **call** instruction in x86; but **int** or **syscall**). Such a procedure is called a system call.

The system call enters the kernel; the kernel performs the service and returns. Thus a process alternates between executing in user space and kernel space.

System calls are generally not invoked directly by a program, but rather via wrapper functions in glibc (or perhaps some other library).

System Calls

- Syscall is the lowest level of interaction with an operating system from a C programmer
- A user program can ask the OS for services that the OS manages
  - You may have used ‘_exit’ in your assignment
  - Anything else you can think of?

System Calls

- Syscall is the lowest level of interaction with an operating system from a C programmer
- A user program can ask the OS for services that the OS manages
  - You may have used '_exit' in your assignment
  - Anything else you can think of?

Number	Name	Description
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

Popular System Call

On Unix, Unix-like and other POSIX-compliant operating systems, popular system calls are open, read, write, close, wait, exec, fork, exit, and kill.

Many modern operating systems have hundreds of system calls. For example, Linux and OpenBSD each have over 300 different calls, FreeBSD has over 500, Windows 7 has close to 700.

Glibc interfaces

Often, but not always, the name of the wrapper function is the same as the name of the system call that it invokes.

For example, glibc contains a function chdir() which invokes the underlying "chdir" system call.

System Calls change the CPU Mode (CPL)

- 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

Tools: strace

```
→ paper-auto-glitching git:(main) strace ls
execve("/usr/bin/ls", ["ls"], 0x7fff83ec4e70 /* 54 vars */) = 0
brk(NULL)                               = 0x55b54b6a0000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe8074a7d0) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=179304, ...}) = 0
mmap(NULL, 179304, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7c7fba9000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@p\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=163200, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7c7fba7000
mmap(NULL, 174600, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7c7fb7c000
mprotect(0x7f7c7fb82000, 135168, PROT_NONE) = 0
mmap(0x7f7c7fb82000, 102400, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x6000) = 0x7f7c7fb82000
mmap(0x7f7c7fb9b000, 28672, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1f000) = 0x7f7c7fb9b000
mmap(0x7f7c7fba3000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f7c7fba3000
mmap(0x7f7c7fba5000, 6664, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f7c7fba5000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300A\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0W\222s/x1X\306o\264\363udX\312$"..., 68, 880) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0W\222s/x1X\306o\264\363udX\312$"..., 68, 880) = 68
mmap(NULL, 2037344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7c7f98a000
mmap(0x7f7c7f9ac000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7f7c7f9ac000
mmap(0x7f7c7fb24000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19a000) = 0x7f7c7fb24000
mmap(0x7f7c7fb72000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7f7c7fb72000
mmap(0x7f7c7fb78000, 13920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f7c7fb78000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpcre2-8.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\340\"\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=588488, ...}) = 0
mmap(NULL, 590632, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7c7f8f9000
mmap(0x7f7c7f8fb000, 413696, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f7c7f8fb000
mmap(0x7f7c7f960000, 163840, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x67000) = 0x7f7c7f960000
mmap(0x7f7c7f988000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x8e000) = 0x7f7c7f988000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0 \22\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=18848, ...}) = 0
mmap(NULL, 20752, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7c7f8f3000
mmap(0x7f7c7f8f4000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f7c7f8f4000
mmap(0x7f7c7f8f6000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x3000) = 0x7f7c7f8f6000
mmap(0x7f7c7f8f7000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x3000) = 0x7f7c7f8f7000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220q\0\0\0\0\0\0"..., 832) = 832
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\227Sr\5\2W;\227\333\254Y[a\375r\302"..., 68, 824) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=157224, ...}) = 0
```

strace ls

Use “man 2 syscall_name” to check out its usage

Making a System Call in x86/64 Assembly

On x86/x86-64, most system calls rely on the software interrupt.

A software interrupt is caused either by an exceptional condition in the processor itself, or a special instruction (the **int 0x80** instruction or **syscall** instruction).

For example: a divide-by-zero exception will be thrown if the processor's arithmetic logic unit is commanded to divide a number by zero as this instruction is in error and impossible.

Making a System Call in x86 Assembly (INT 0x80)

x86 (32-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)	arg3 (%esi)	arg4 (%edi)	arg5 (%ebp)
0	restart_syscall	man/ cs/	0x00	-	-	-	-	-	-
1	exit	man/ cs/	0x01	int error_code	-	-	-	-	-
2	fork	man/ cs/	0x02	-	-	-	-	-	-
3	read	man/ cs/	0x03	unsigned int fd	char *buf	size_t count	-	-	-
4	write	man/ cs/	0x04	unsigned int fd	const char *buf	size_t count	-	-	-
5	open	man/ cs/	0x05	const char *filename	int flags	umode_t mode	-	-	-
6	close	man/ cs/	0x06	unsigned int fd	-	-	-	-	-
7	waitpid	man/ cs/	0x07	pid_t pid	int *stat_addr	int options	-	-	-
8	creat	man/ cs/	0x08	const char *pathname	umode_t mode	-	-	-	-
9	link	man/ cs/	0x09	const char *oldname	const char *newname	-	-	-	-
10	unlink	man/ cs/	0x0a	const char *pathname	-	-	-	-	-
11	execve	man/ cs/	0x0b	const char *filename	const char *const *argv	const char *const *envp	-	-	-
12	chdir	man/ cs/	0x0c	const char *filename	-	-	-	-	-
13	time	man/ cs/	0x0d	time_t *tloc	-	-	-	-	-
14	mknod	man/ cs/	0x0e	const char *filename	umode_t mode	unsigned dev	-	-	-
15	chmod	man/ cs/	0x0f	const char *filename	umode_t mode	-	-	-	-
16	lchown	man/ cs/	0x10	const char	uid_t user	gid_t group			

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```

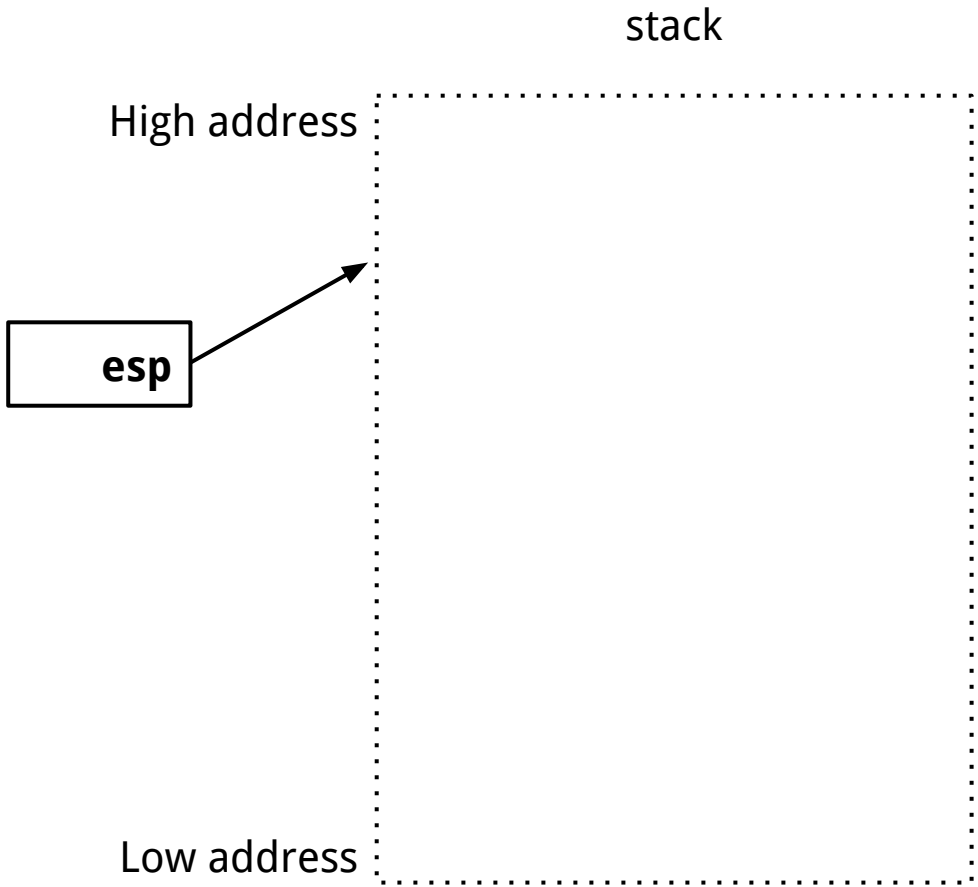
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

http://shell-storm.org/shellcode/files/shellcode-827.php

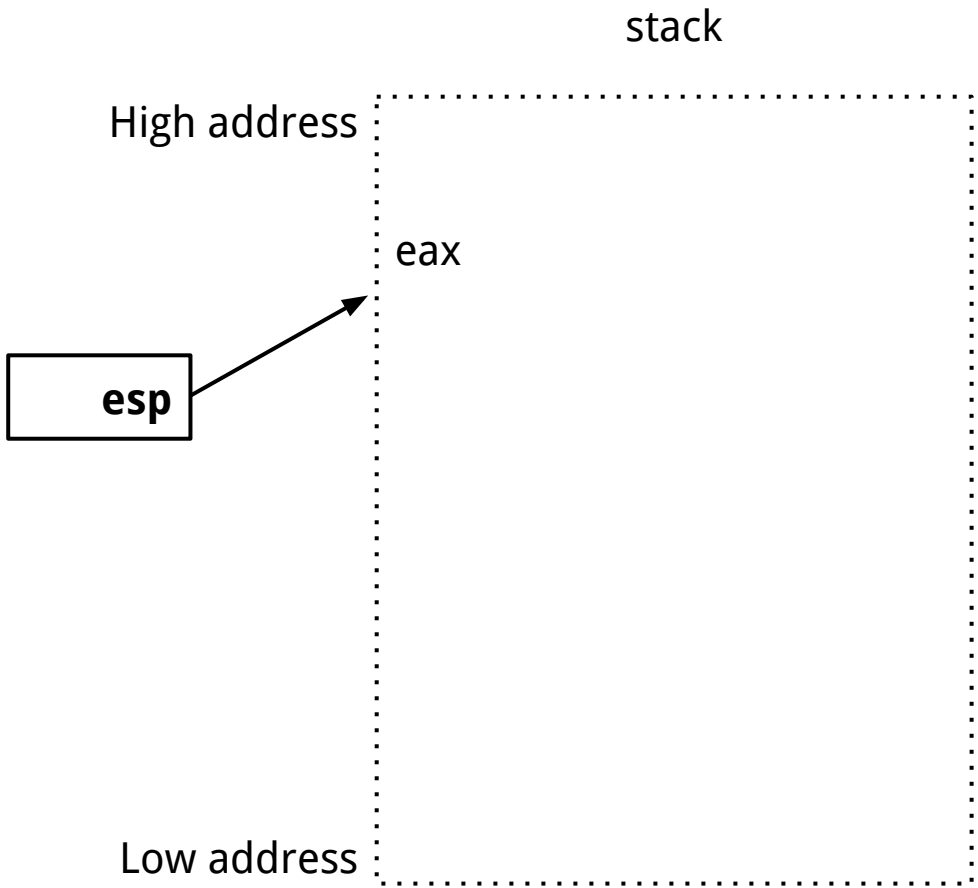
Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```



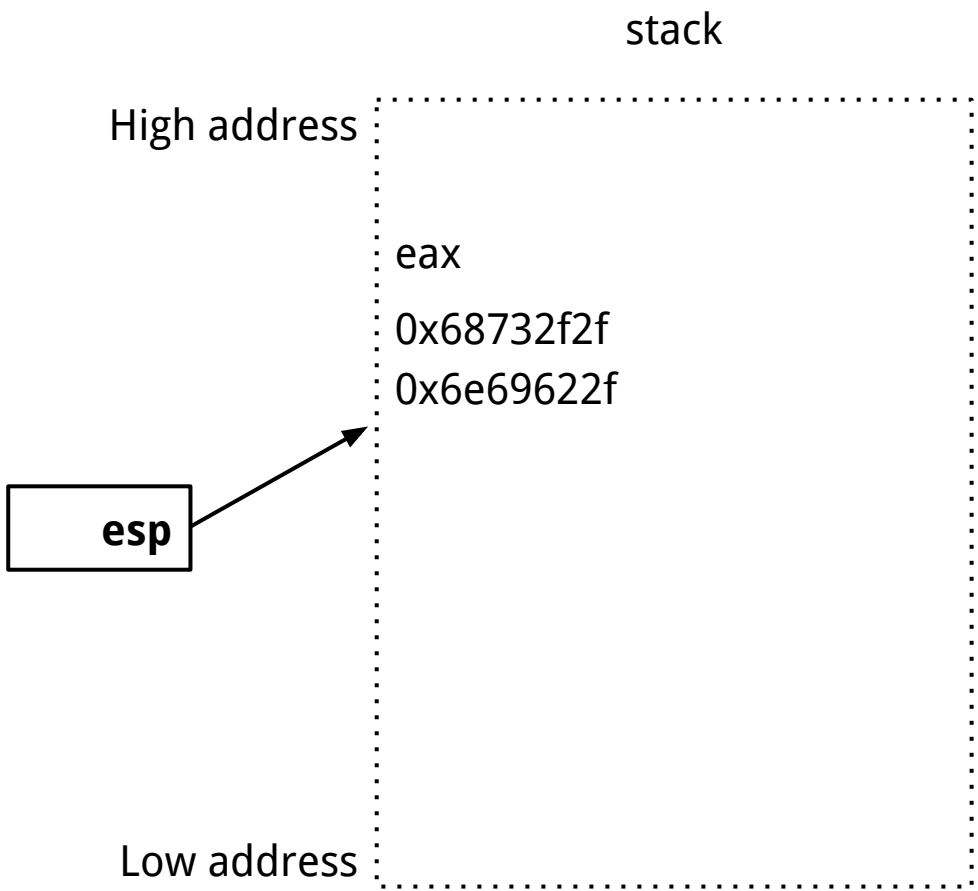
Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```



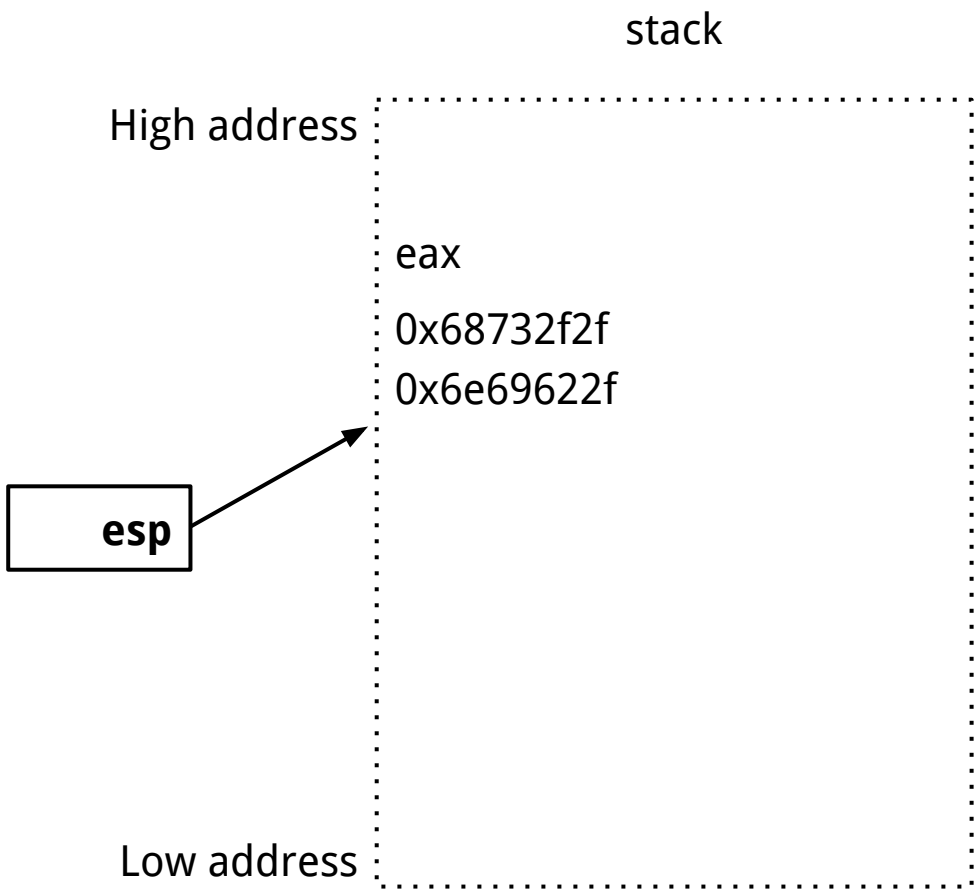
Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```

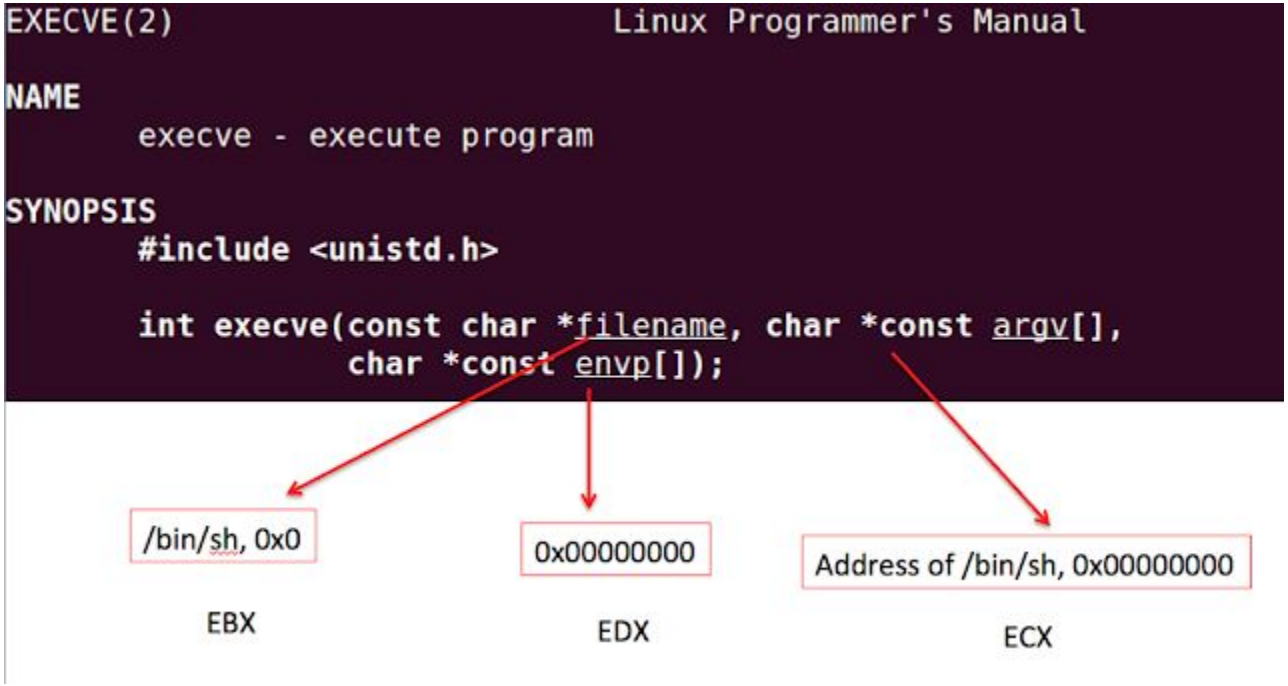


Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```



Making a System Call in x86 Assembly



execve(“/bin/sh”, address of string “/bin/sh”, 0)

Making a System Call in x86_64 (64-bit) Assembly

x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
2	open	man/ cs/	0x02	const char *filename	int flags	umode_t mode	-	-	-
3	close	man/ cs/	0x03	unsigned int fd	-	-	-	-	-
4	stat	man/ cs/	0x04	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
5	fstat	man/ cs/	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	-	-	-	-
6	lstat	man/ cs/	0x06	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
7	poll	man/ cs/	0x07	struct pollfd *ufds	unsigned int nfds	int timeout	-	-	-
8	lseek	man/ cs/	0x08	unsigned int fd	off_t offset	unsigned int whence	-	-	-
9	mmap	man/ cs/	0x09	?	?	?	?	?	?
10	mprotect	man/ cs/	0x0a	unsigned long start	size_t len	unsigned long prot	-	-	-
11	munmap	man/ cs/	0x0b	unsigned long addr	size_t len	-	-	-	-
12	brk	man/ cs/	0x0c	unsigned long brk	-	-	-	-	-
13	rt_sigaction	man/ cs/	0x0d	int	const struct sigaction *	struct sigaction *	size_t	-	-

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

Making a System Call in x86_64 (64-bit) Assembly

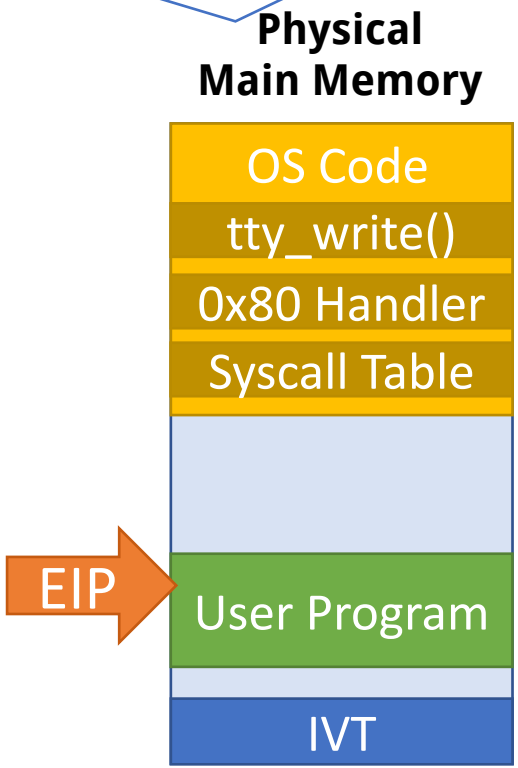
NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
59	execve	man/ cs/	0x3b	const char *filename	const char *const *argv	const char *const *envp	-	-	-

```
push rax
xor rdx, rdx
xor rsi, rsi
mov rbx,'/bin//sh'
push rbx
push rsp
pop rdi
mov al, 59
syscall
```

System Call Example

1. Software executes int 0x80
   - Pushes EIP, CS, and EFLAGS

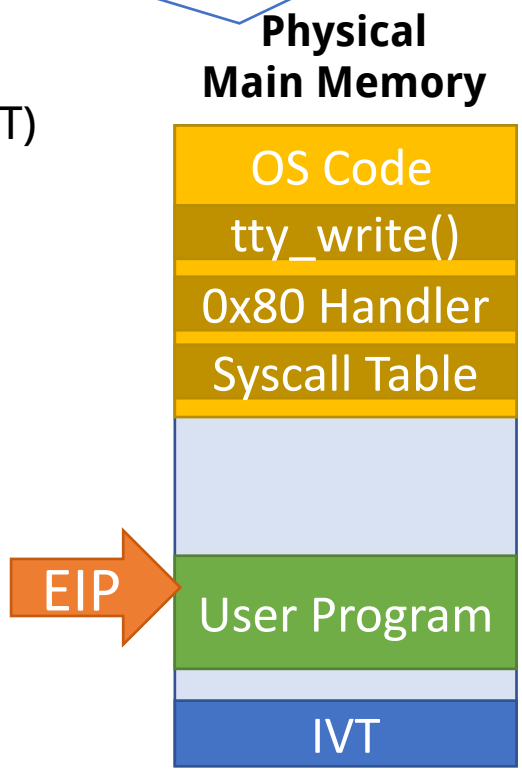
Note: this shows a physical memory layout. The user program thinks it owns the entire memory space (the diagram that we saw in previous lectures).



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 Interrupt Vector Table (IVT)
   - Switch from ring 3 to 0

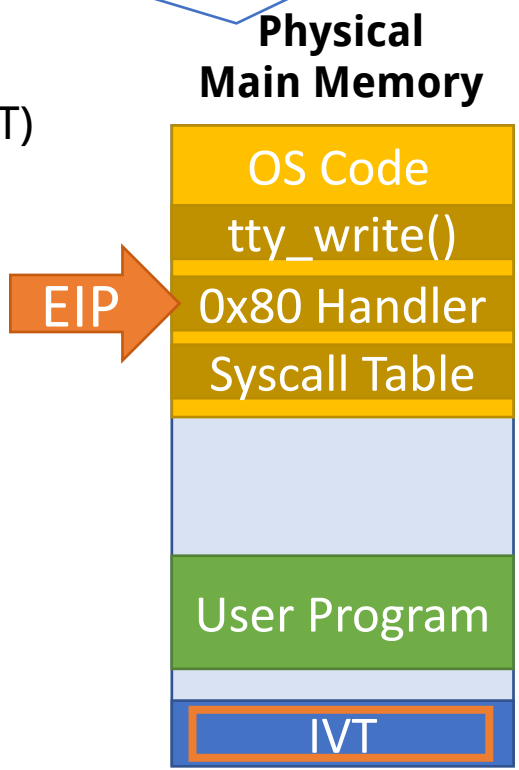
Note: this shows a physical memory layout. The user program thinks it owns the entire memory space (the diagram that we saw in previous lectures).



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 Interrupt Vector Table (IVT)
   - Switch from ring 3 to 0

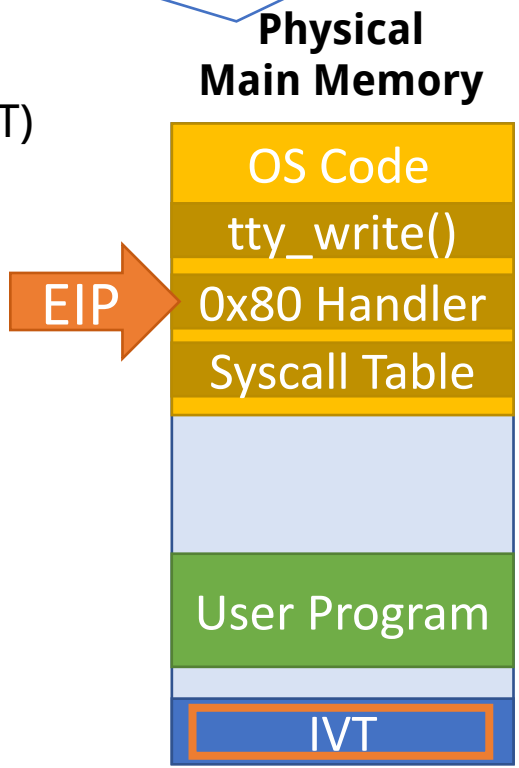
Note: this shows a physical memory layout. The user program thinks it owns the entire memory space (the diagram that we saw in previous lectures).



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 Interrupt Vector Table (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

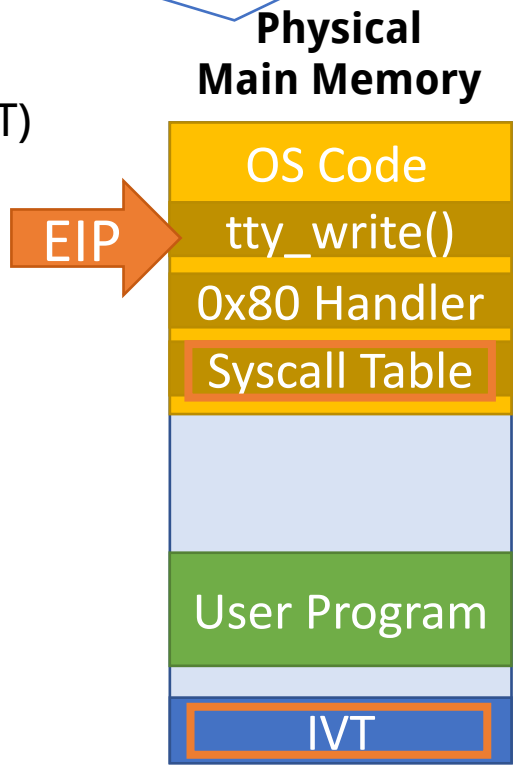
Note: this shows a physical memory layout. The user program thinks it owns the entire memory space (the diagram that we saw in previous lectures).



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 Interrupt Vector Table (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

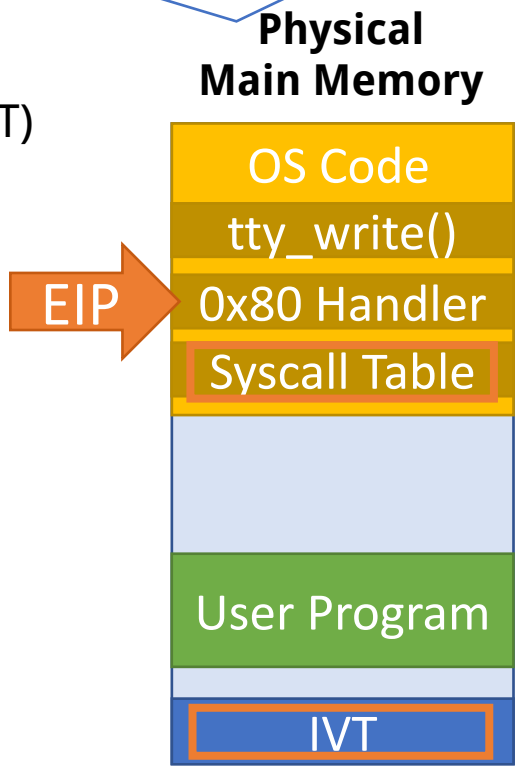
Note: this shows a physical memory layout. The user program thinks it owns the entire memory space (the diagram that we saw in previous lectures).



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 Interrupt Vector Table (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

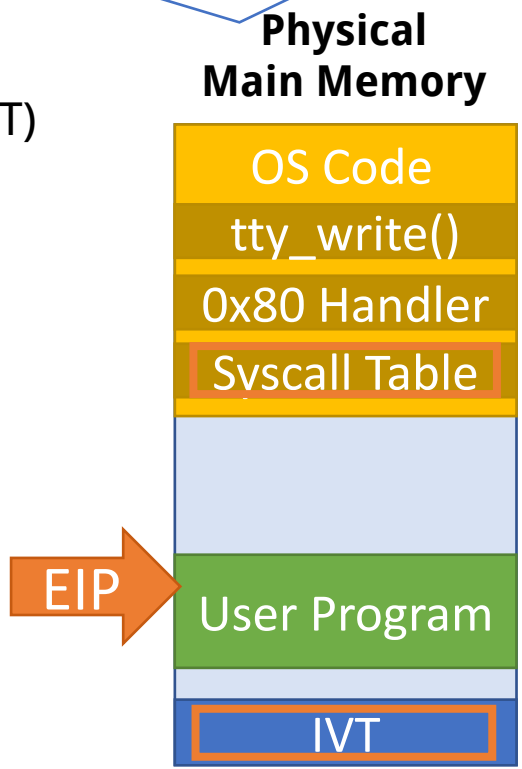
Note: this shows a physical memory layout. The user program thinks it owns the entire memory space (the diagram that we saw in previous lectures).



System Call Example

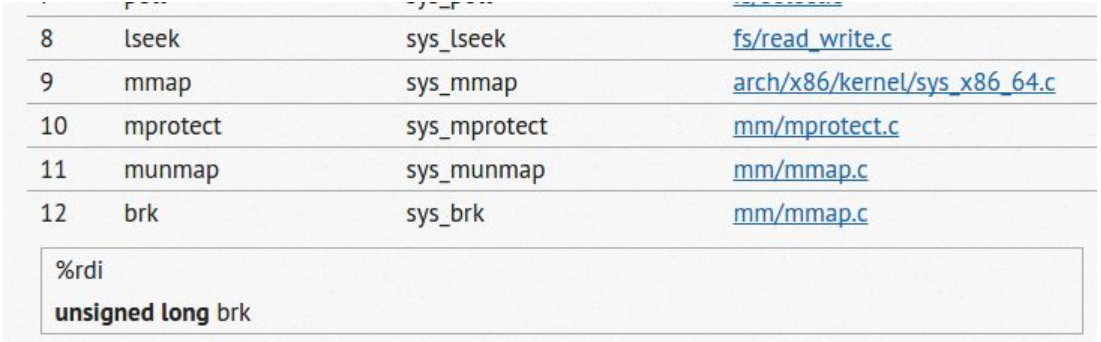
1. Software executes int 0x80
   - Pushes EIP, CS, and EFLAGS
2. CPU transfers execution to the OS handler
   - Look up the handler in the Interrupt Vector Table (IVT)
   - Switch from ring 3 to 0
3. OS executes the system call
   - Save the processes state
   - Use EAX to locate the system call
   - 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

Note: this shows a physical memory layout. The user program thinks it owns the entire memory space (the diagram that we saw in previous lectures).



System Calls and arguments

- Helpful webpage with syscalls and arguments
  - https://filippo.io/linux-syscall-table/



8	lseek	sys_lseek	fs/read_write.c
9	mmap	sys_mmap	arch/x86/kernel/sys_x86_64.c
10	mprotect	sys_mprotect	mm/mprotect.c
11	munmap	sys_munmap	mm/mmap.c
12	brk	sys_brk	mm/mmap.c

%rdi

unsigned long brk

Opening a File

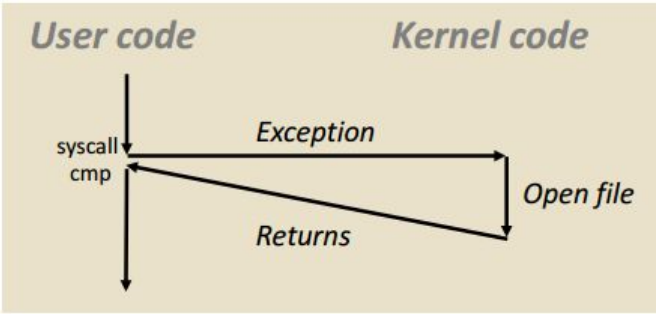
- rax holds the system call # that we want to pass.
  - Other arguments accessed as follows

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c

%rdi	%rsi	%rdx
const char __user * filename	**int** flags	**umode_t** mode

Opening a File | Illustration

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



x86 and xv6 as an example

```
8   #include "traps.h"
9   #include "spinlock.h"
10
11  // Interrupt descriptor table (shared by all CPUs).
12  struct gatedesc idt[256];
13  extern uint vectors[];  // in vectors.S: array of 256 entry pointers
14  struct spinlock tickslock;
15  uint ticks;
16
17  void
18  tvinit(void)
19  {
20    int i;
21
22    for(i = 0; i < 256; i++)
23      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
24    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
25
26    initlock(&tickslock, "time");
27  }
28
```

https://github.com/mit-pdos/xv6-public/blob/master/trap.c
https://github.com/mit-pdos/xv6-public/blob/master/syscall.c

Piping and Redirection

Channels of Communication for Linux Process

Every process in Linux has three initial, standard channels of communication:

- Standard Input (stdin, fd=0) is the channel through which the process takes input. For example, your shell uses Standard Input to read the commands that you input.
- Standard Output (stdout, fd=1) is the channel through which processes output normal data, such as the flag when it is printed to you in previous challenges or the output of utilities such as *ls*.
- Standard Error (stderr, fd=2) is the channel through which processes output error details. For example, if you mistype a command, the shell will output, over standard error, that this command does not exist.

Examples

Redirecting output > or 1>

echo hi > asdf *echo hi 1> asdf*

ls > files.txt

Appending output >>

echo hi >> asdf

Redirecting errors 2>

/challenge/run 2> errors.log

Redirecting input <

rev < messagefile

sort < names.txt

Pipe

The | (pipe) operator. Standard output from the command to the left of the pipe will be connected to (piped into) the standard input of the command to the right of the pipe.

echo hello-world | wc -c

How to use the fork syscall?

What it does

- Creates a **new process** by duplicating the calling process.
- The new process is called the **child**; the original is the **parent**.

Key details

- **Both processes continue execution from the point after the fork() call.**
- Return values:
  - Parent gets the child's **PID** (positive number).
  - Child gets **0**.
  - If fork fails, parent gets **-1**.
- Child initially gets a copy of:
  - Parent's **address space** (code, data, stack).
  - **File descriptors**.
  - **Environment** and **signal dispositions**.

How to use the fork syscall?

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        printf("I am the child! PID = %d\n", getpid());
    } else if (pid > 0) {
        // Parent process
        printf("I am the parent! Child PID = %d\n", pid);
    } else {
        // Error
        perror("fork failed");
    }
    return 0;
}
```

```
strace -f ./a.out
```

How many processes?

```
#include <stdio.h>
#include <unistd.h>

int main() {
    fork();    // First fork
    fork();    // Second fork
    fork();    // Third fork

    printf("Hello from PID
%d\n", getpid());
    return 0;
}
```

How to use the dup and dup2 syscall?

Both: create a duplicate of an existing file descriptor (FD)
An FD is a handle the OS gives to open files, pipes, sockets, etc.

dup(oldfd)

- Creates a new FD that refers to the same file/pipe/socket as oldfd.
- Returns **lowest-numbered free FD** (you don't choose the number).
- Leaves oldfd unchanged.

```
int fd2 = dup(fd1);  // fd2 is some free number
```

dup2(oldfd, newfd)

- Makes newfd refer to the same thing as oldfd.
- If newfd is already open, it's closed first.
- Guarantees the new FD number is exactly newfd.

```
dup2(fd1, STDOUT_FILENO); // redirect stdout to fd1
```

How to use the wait syscall?

Purpose

- Allows a **parent process** to wait for one of its **child processes** to finish.
- Collects the child's exit status to avoid creating a **zombie** process.

pid_t wait(int *status);

Behavior:

- Suspends the calling (parent) process until one of its children exits.
- Returns the **PID** of the terminated child.
- If status is not NULL, stores the child's exit code.

Returns **-1** if no children or on error.

How to use the wait syscall?

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child: exiting\n");
        _exit(42);
    } else {
        int status;
        pid_t child = wait(&status);
        printf("Parent: child %d exited with code %d\n",
               child, WEXITSTATUS(status));
    }
    return 0;
}
```

What does this program do?

```
pid_t pid = fork();
    if (pid == 0) {
        // Child process
        int fd = open("out.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
        if (fd < 0) { perror("open"); exit(1); }

        // Redirect stdout to the file
        dup2(fd, STDOUT_FILENO);
        close(fd);

        // Run command
        execlp("ls", "ls", NULL);
        perror("execlp"); // only runs if execlp fails
        exit(1);
    } else if (pid > 0) {
        wait(NULL);
    }
```

What does this program do?

```
pid = fork();
    if (pid == 0) {
        int fd = open("in.txt", O_RDONLY);
        if (fd < 0) { perror("open"); exit(1); }

        // Redirect stdin from the file
        dup2(fd, STDIN_FILENO);
        close(fd);

        execlp("sort", "sort", NULL);
        perror("execlp");
        exit(1);
    } else if (pid > 0) {
        wait(NULL);
    }
    return 0;
```

Announcements

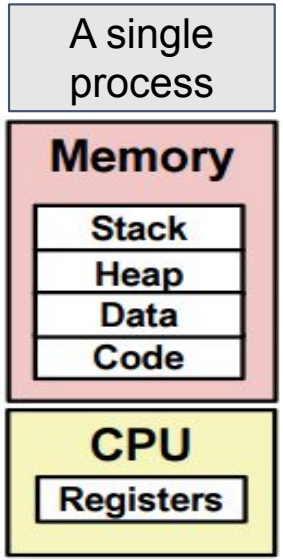
1. The class scheduled for Tuesday will be delivered online. I will post the exact meeting time on Canvas later. Attendance is optional, and students are welcome to join if they wish. The recording of the session will be uploaded to YouTube afterward for anyone who prefers to watch it later.
2. Midterm exam on Thursday. If you've done the readings and the assignments for the first 4 weeks, then you should be fine. Open book/notes, no electronic devices, no substantial coding questions.

Processes

The Process

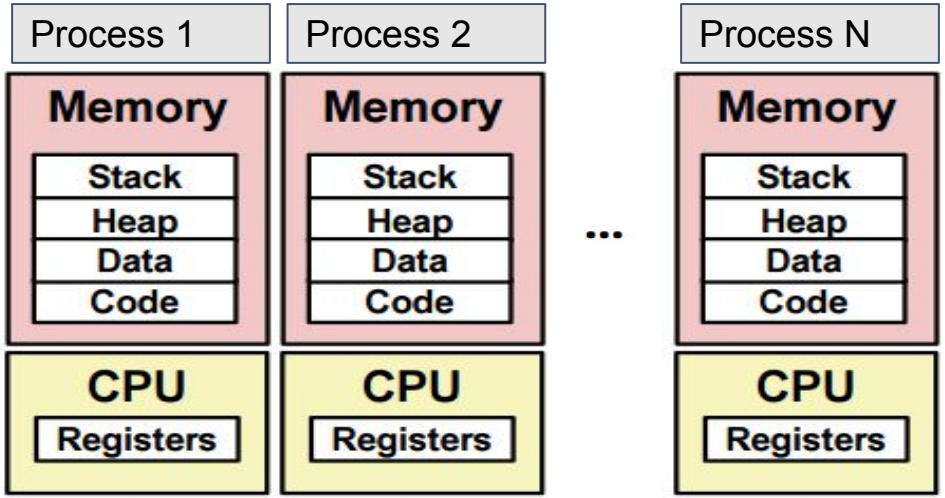
- A process is alive, a program is dead. A program is just the code.

- Processes are organized by the OS using two key abstractions
  - Logical Control Flow
    - Programs “appear” to have exclusive control over the CPU
    - Done by “context switching”
  - Private Address Space
    - Each program “appears” to have exclusive use of main memory
    - Provided by mechanism called virtual memory



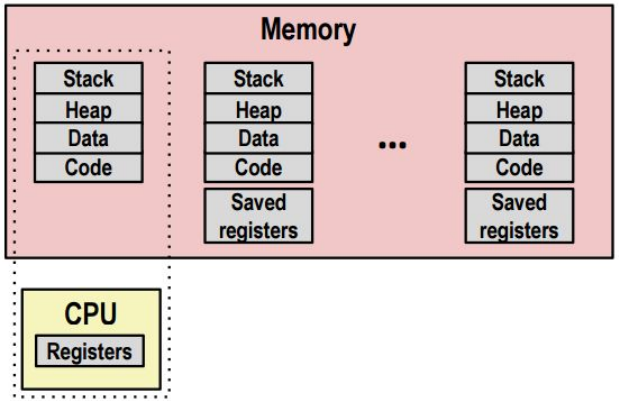
Multiprocessing: Illusion

- When running processes, it appears that we are running many different tasks at the same time
- It also appears that our memory is neatly organized.
  - Note from this diagram we see every process has its own
    - stack
    - heap
    - data
    - code
    - registers

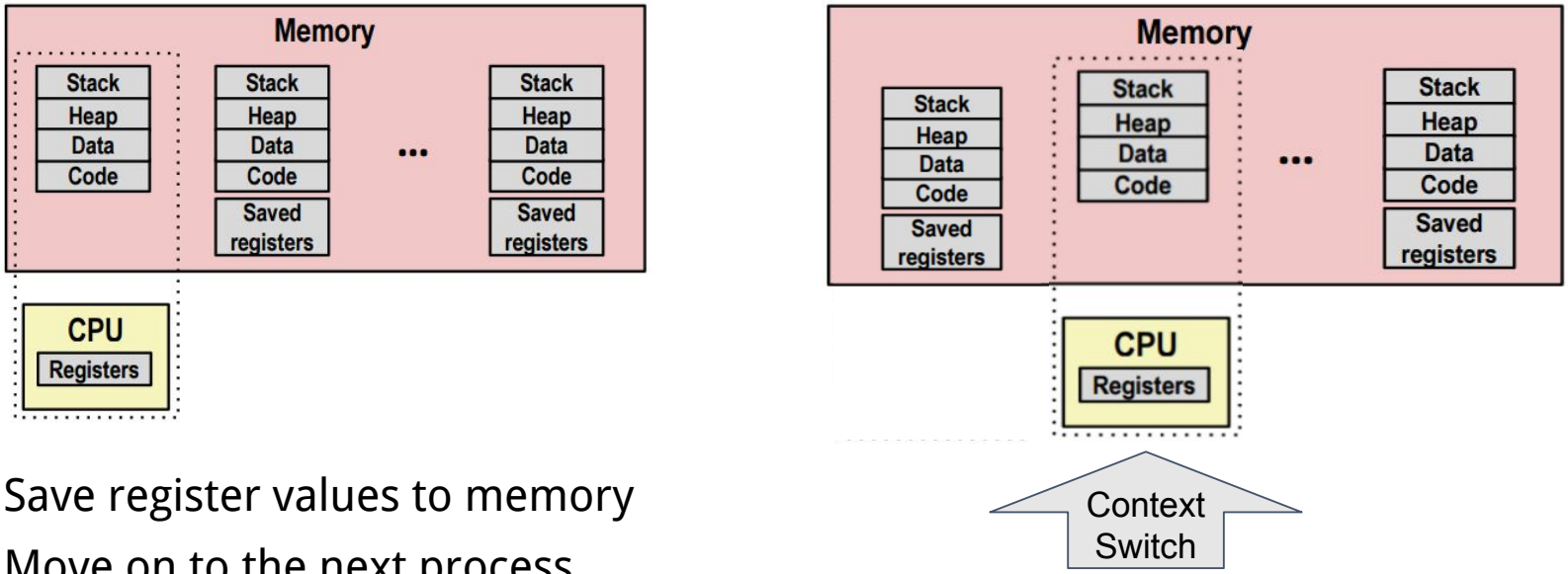


Multiprocessing: Reality

- Remember, at any time, only one processor is really running code
- Program execution is interleaved
- OS manages memory addresses in virtual memory
- OS stores the saved registers for different programs.
  - (At some point in this class, you probably figured 16 registers is not enough for all of the processes that you were running.)
- When we switch which process is executing: this is a context switch



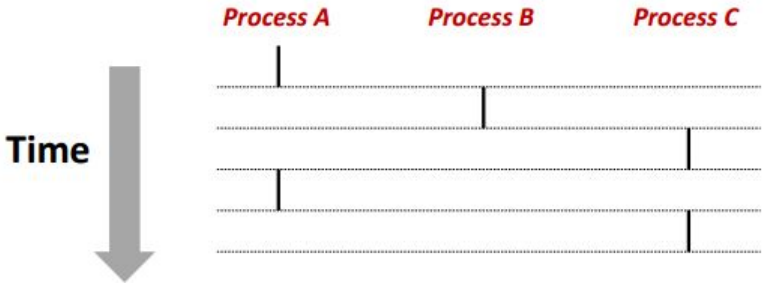
Context switch: a high-level view



- Save register values to memory
- Move on to the next process
  - Point to the stack of the next process
  - Restore saved register values
- Start running executing the next process

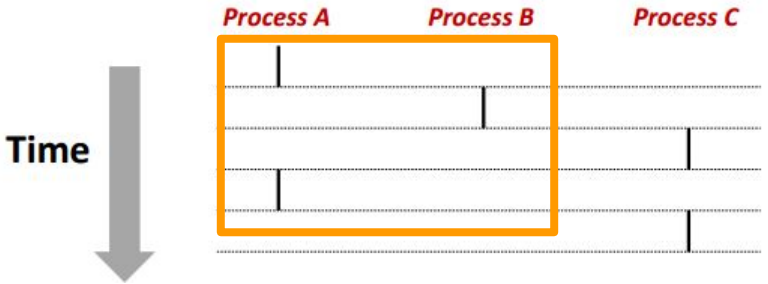
Concurrent Processing

- Each process running has its own control flow
- If they overlap in their lifetime, then they are running concurrently
  - otherwise they are sequential
- Remember only 1 process at a time can execute
  - On a single core, which processes here are concurrent to each other?
    - Concurrent:
  - Which are sequential?
    - Sequential:



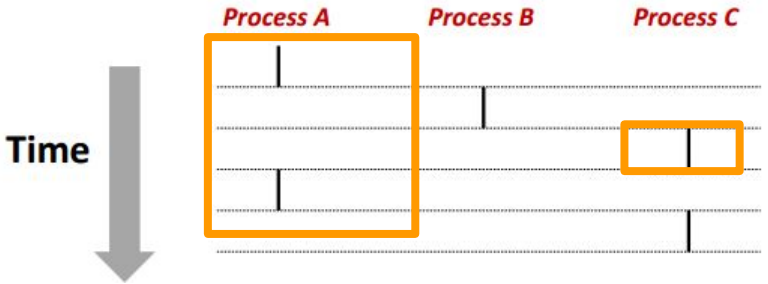
Concurrent Processing

- Each process running has its own control flow
- If they overlap in their lifetime, then they are running concurrently
  - otherwise they are sequential
- Remember only 1 process at a time can execute
  - On a single core, which processes here are concurrent to each other?
    - Concurrent: A&B
  - Which are sequential?
    - Sequential:



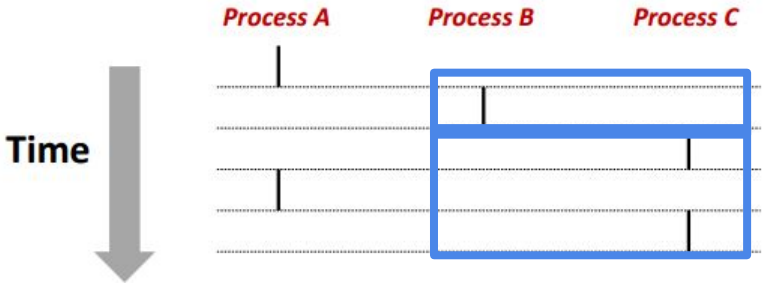
Concurrent Processing

- Each process running has its own control flow
- If they overlap in their lifetime, then they are running concurrently
  - otherwise they are sequential
- Remember only 1 process at a time can execute
  - On a single core, which processes here are concurrent to each other?
    - Concurrent: A&B, A&C
  - Which are sequential?
    - Sequential:



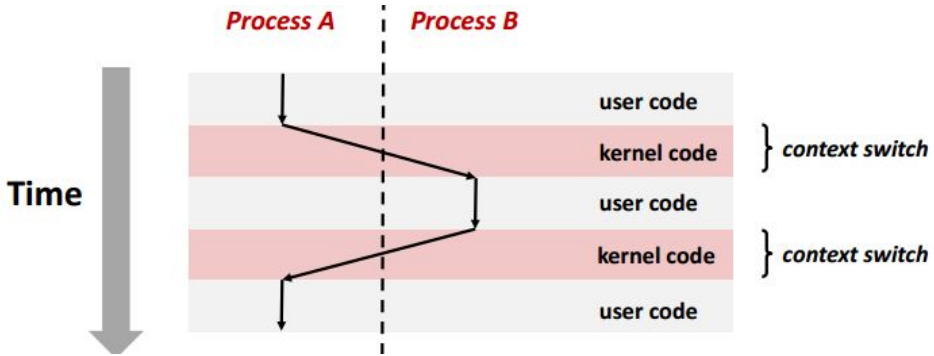
Concurrent Processing

- Each process running has its own control flow
- If they overlap in their lifetime, then they are running concurrently
  - otherwise they are sequential
- Remember only 1 process at a time can execute
  - On a single core, which processes here are concurrent to each other?
    - Concurrent: A&B, A&C
  - Which are sequential?
    - Sequential: B &C



Context Switching Illustration

- Processes are managed by a shared chunk of memory-resident OS code called the **kernel**
  - The kernel is not a separate process itself, but runs as part of other existing processes
- Context Switches pass the control flow from one process to another
  - Note how going from A to B (and B to A) requires some kernel code to be executed



An example of a context switch: there can be different implementations

Process 1's Code

EIP

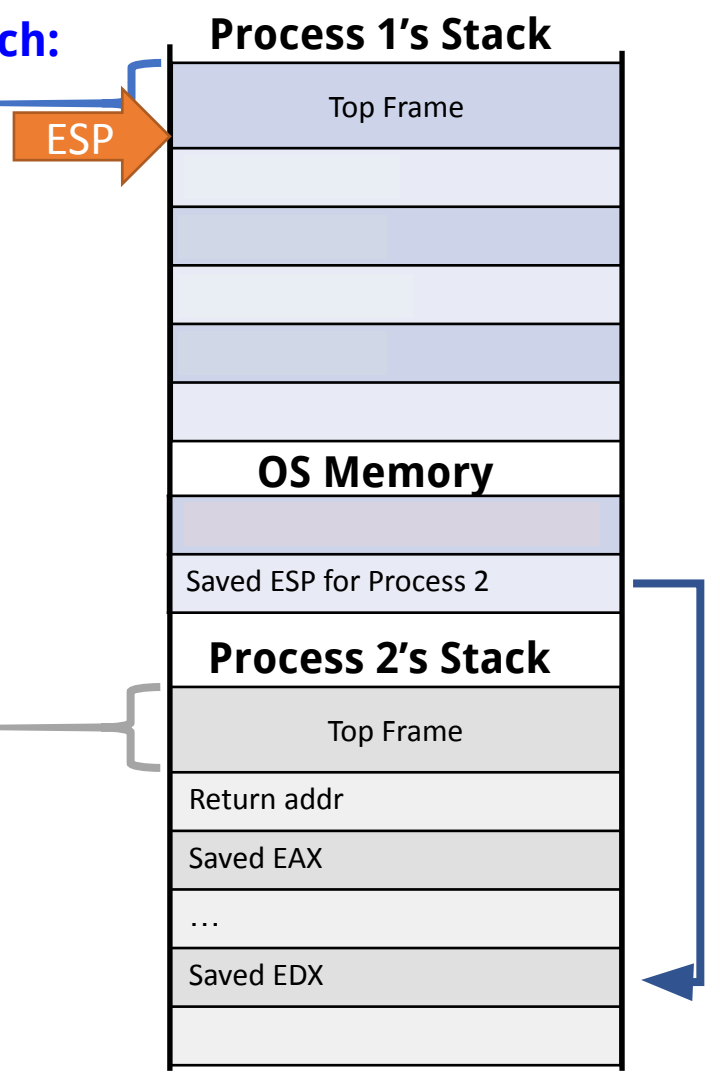
```
a = b + 1;
switch();
b--;
```

Process 2's Code

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

OS Code

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



An example of a context switch: there can be different implementations

Process 1's Code

EIP

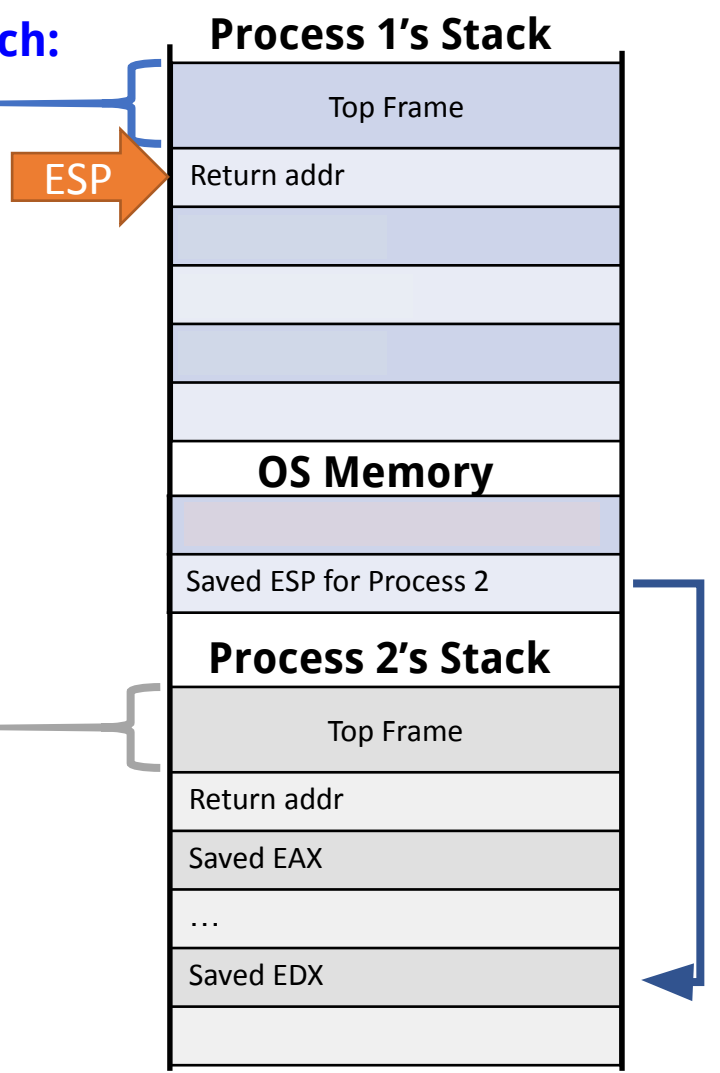
```
a = b + 1;
switch();
b--;
```

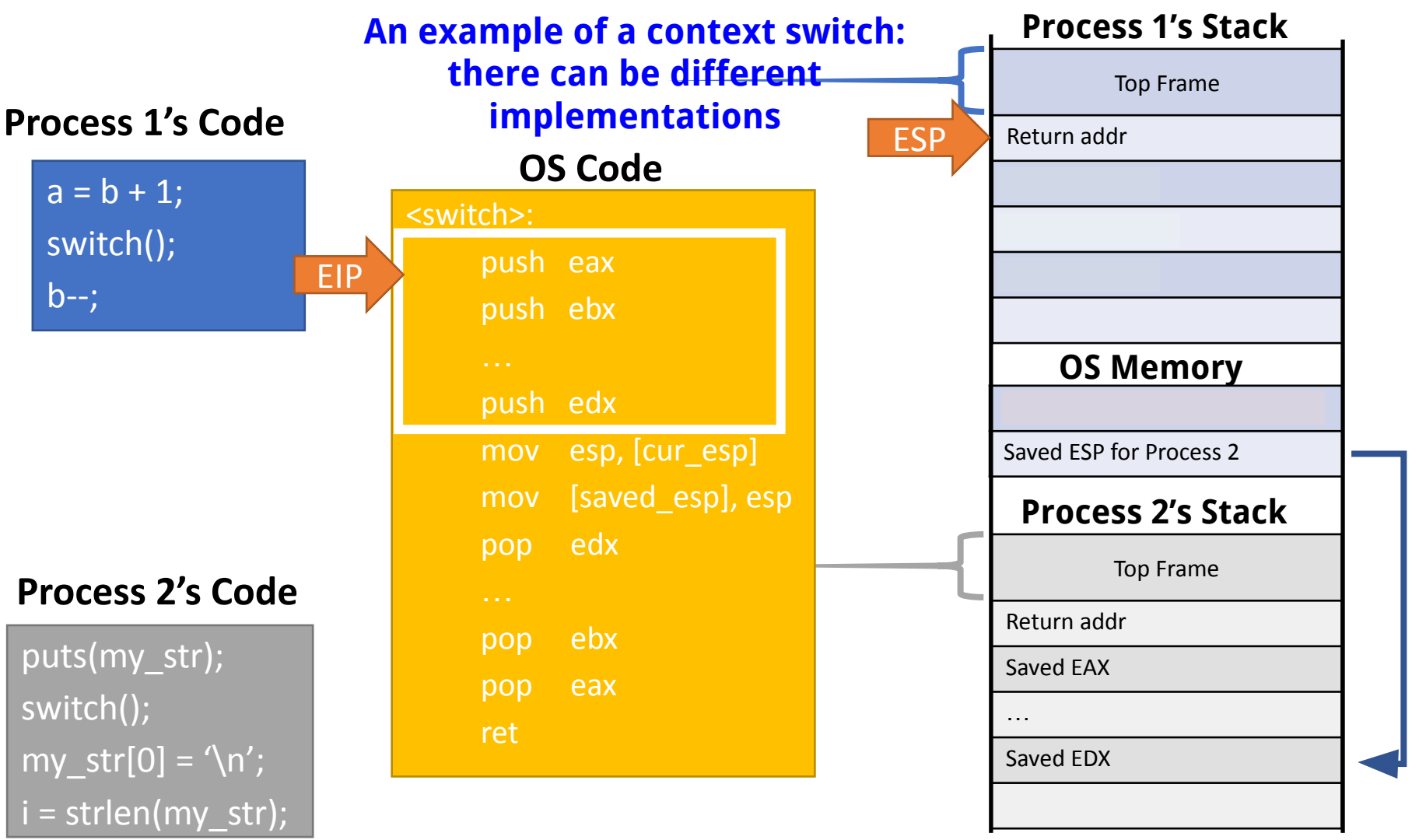
Process 2's Code

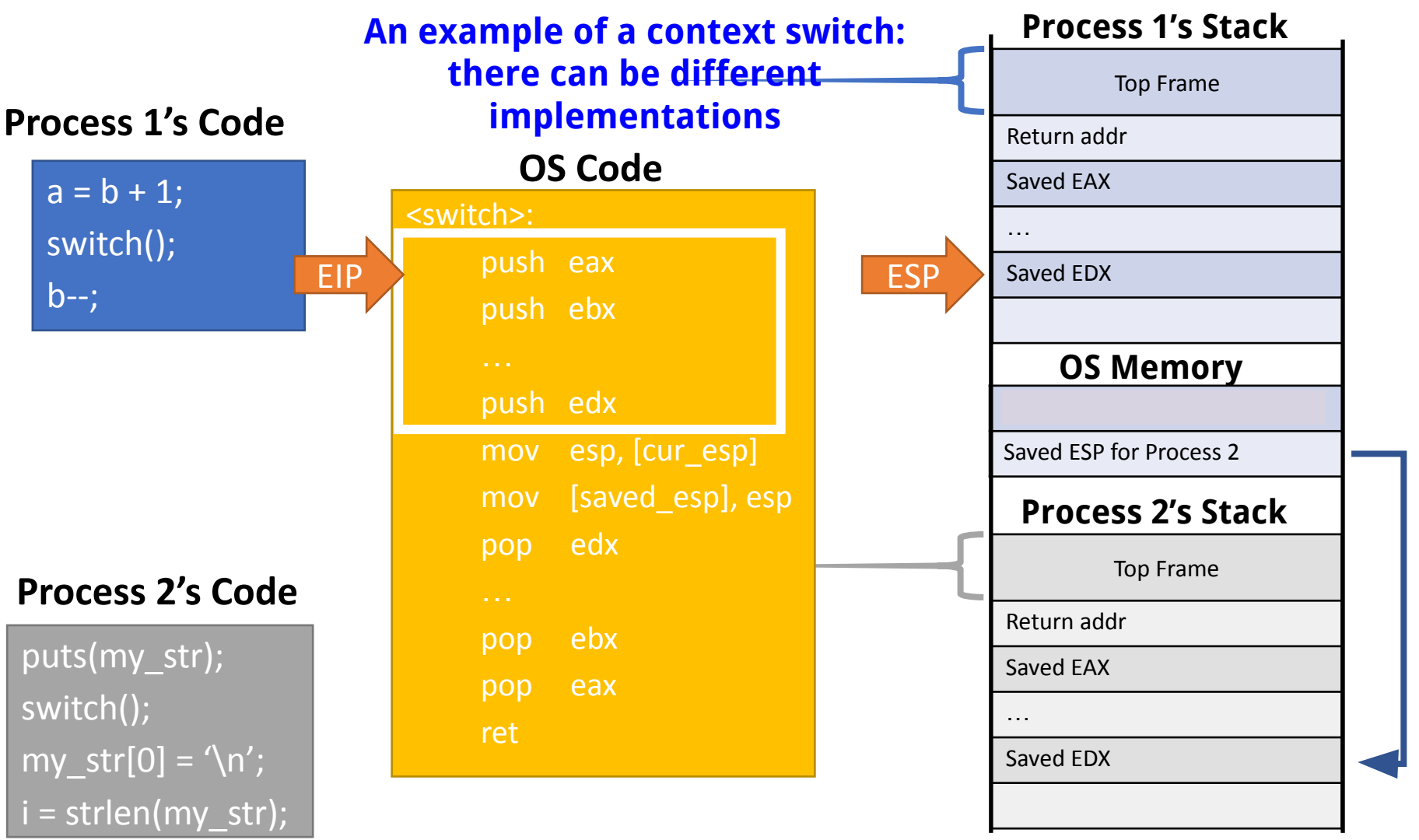
```
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
```

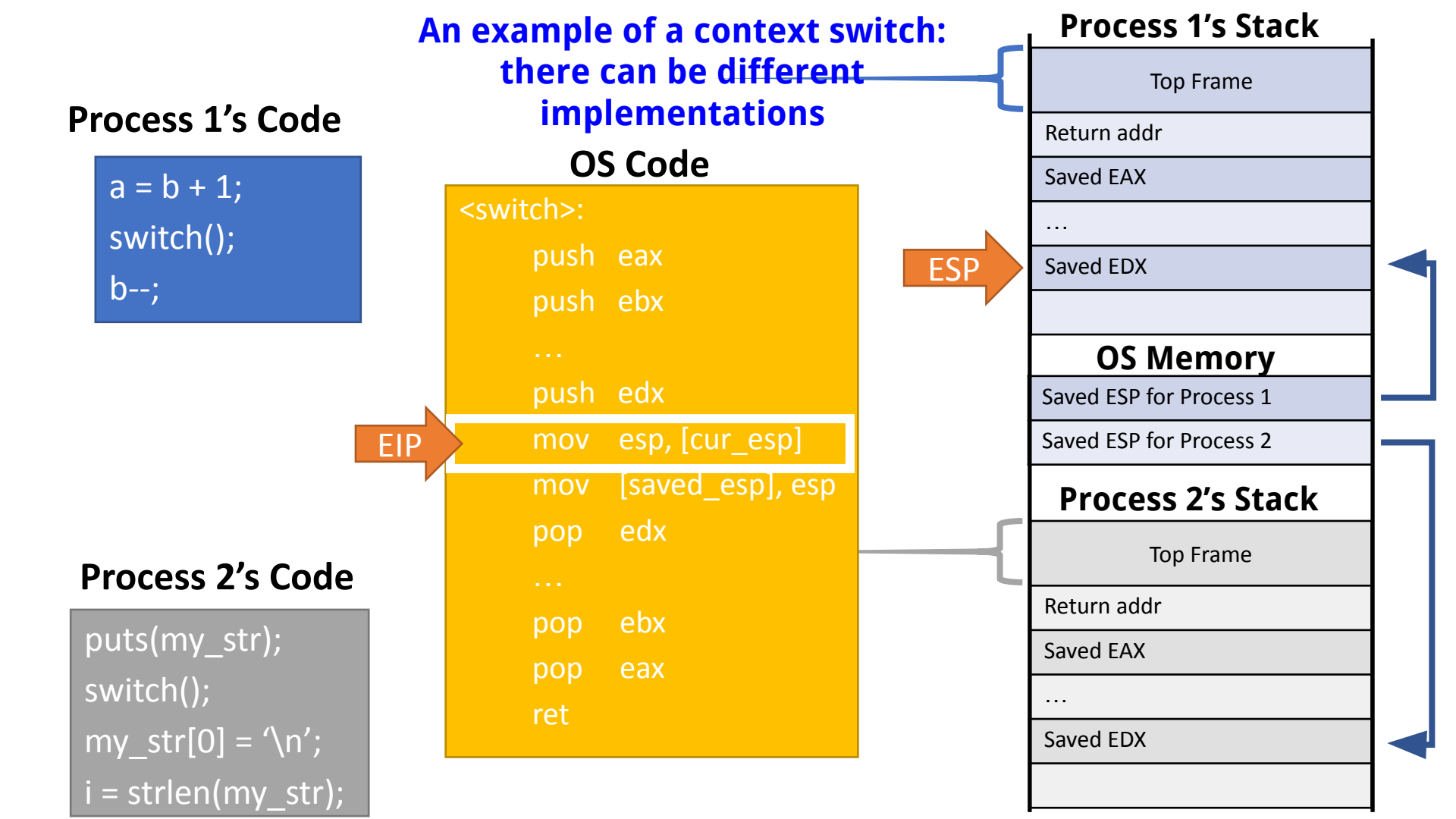
OS Code

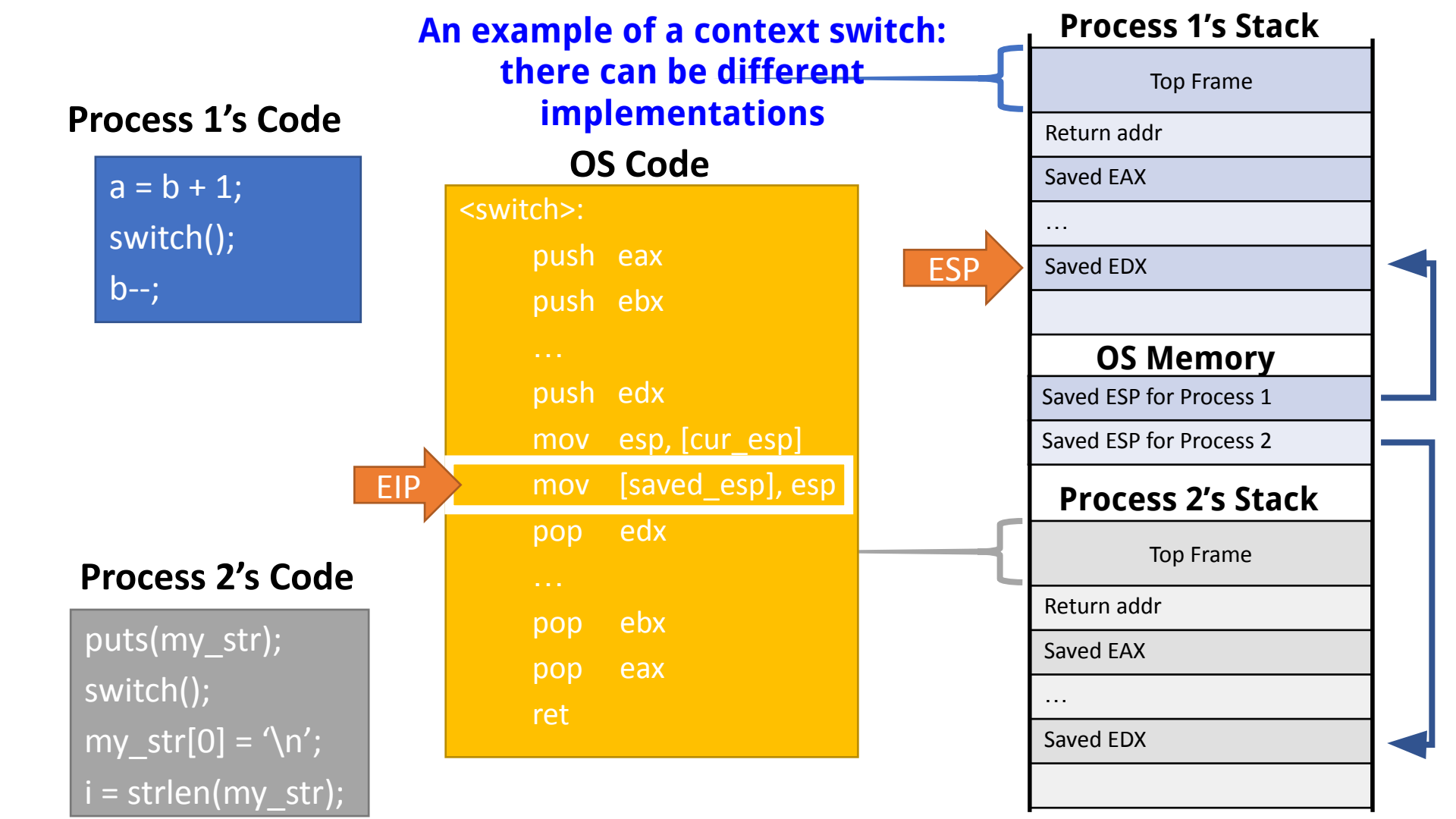
```
<switch>:
        push   eax
        push   ebx
        …
        push   edx
        mov    esp, [cur_esp]
        mov    [saved_esp], esp
        pop    edx
        …
        pop    ebx
        pop    eax
        ret
```

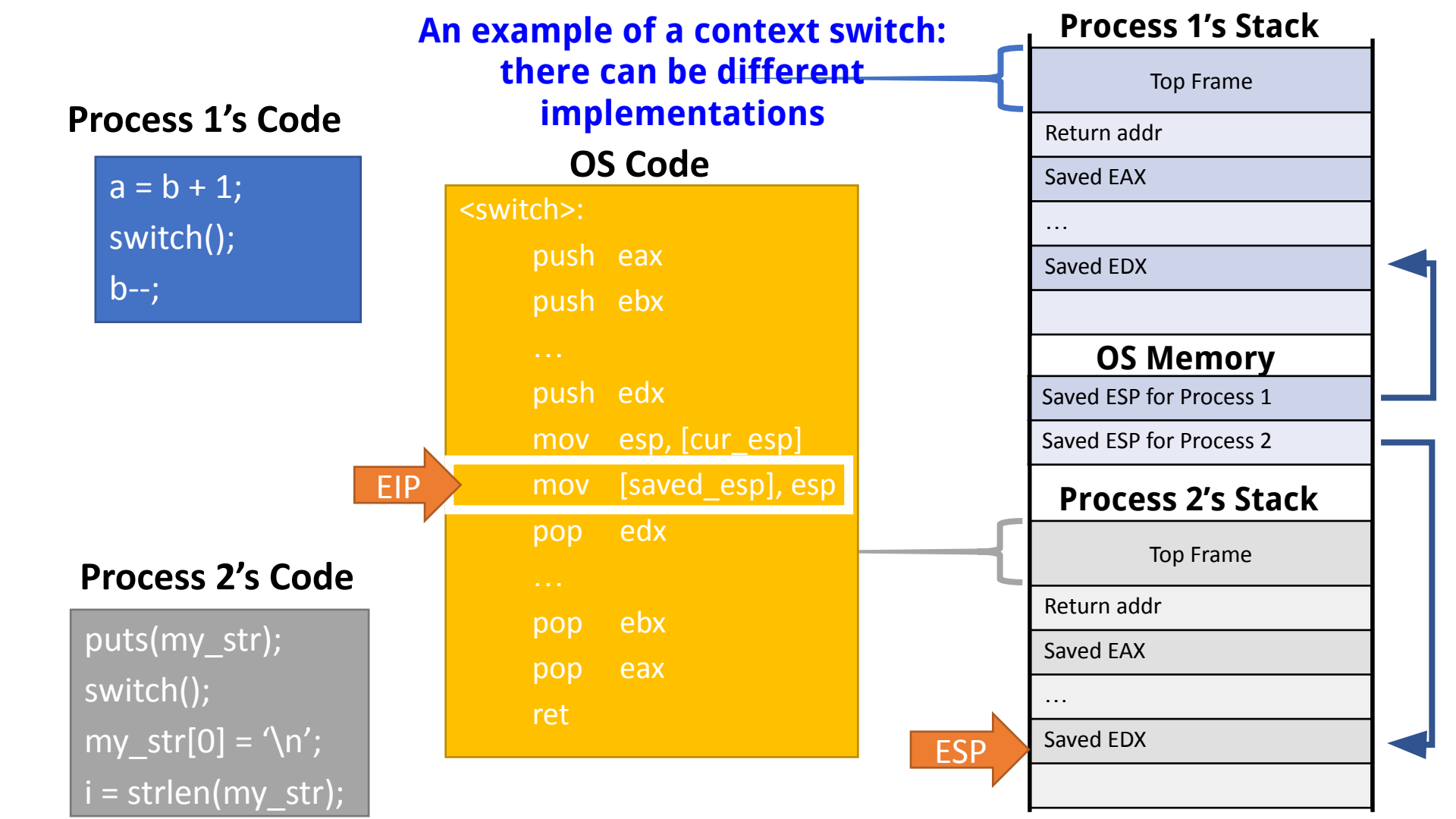


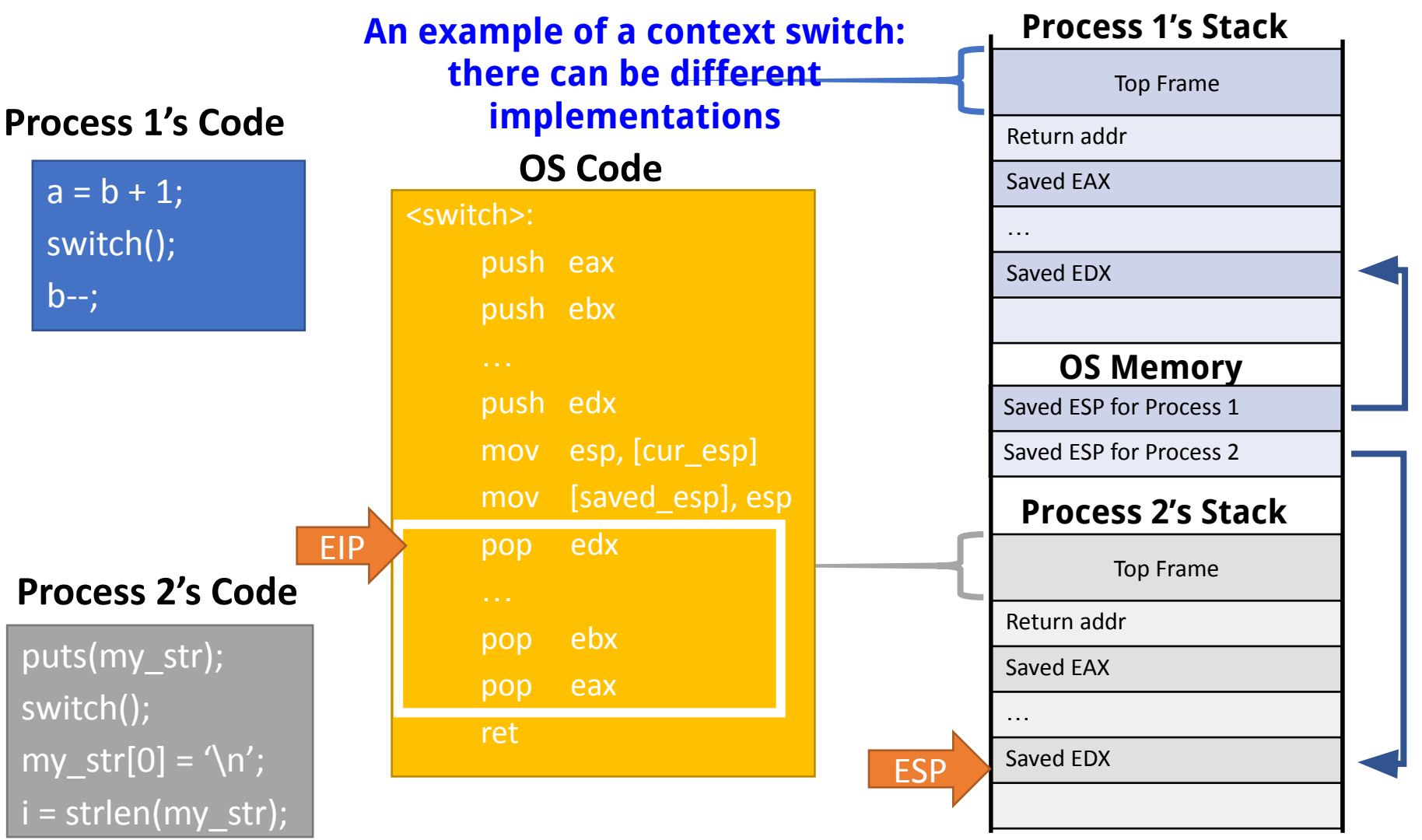

An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
EIP
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
ESP
Process 1's Stack
Top Frame
Return addr
OS Memory
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX

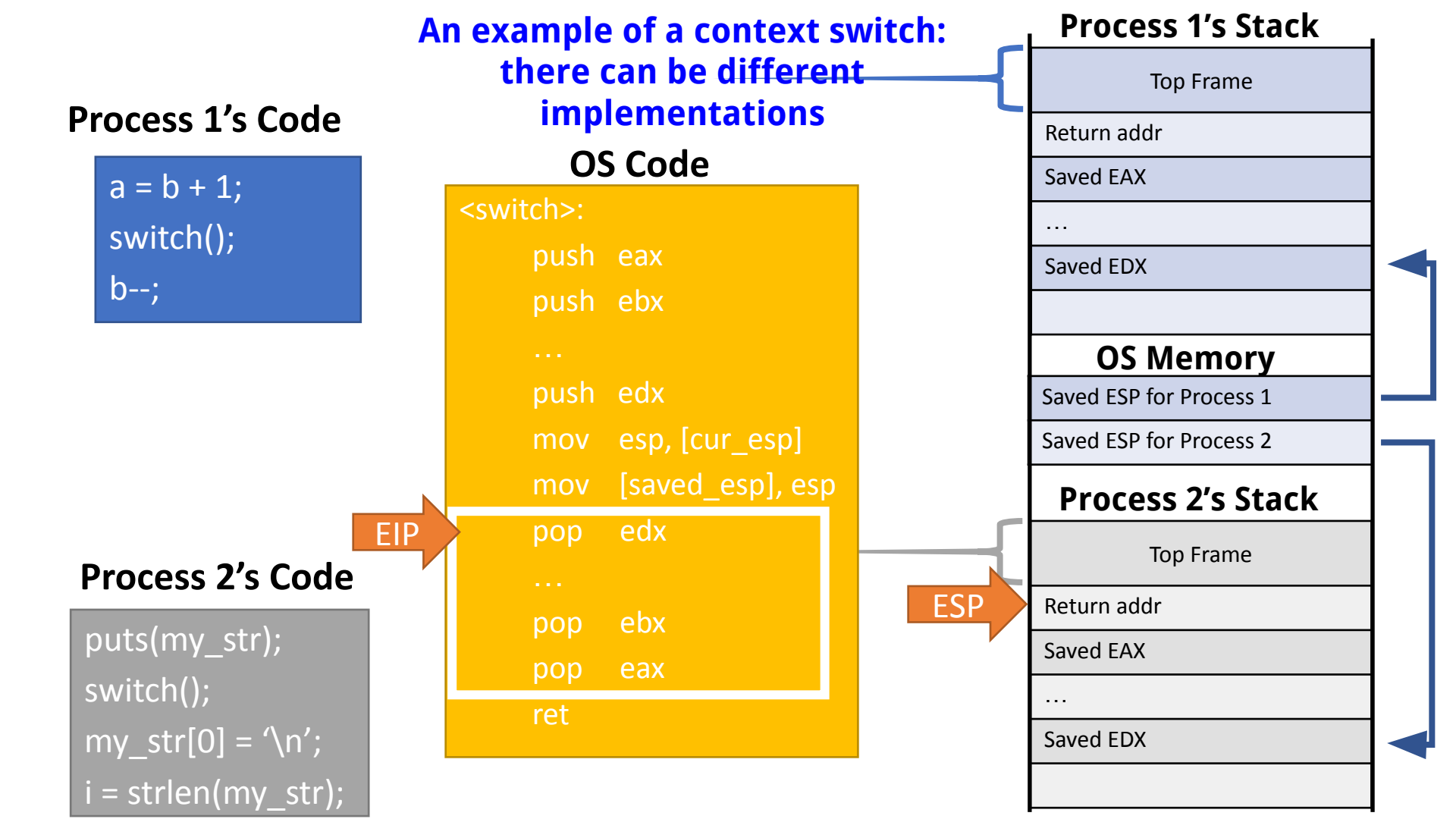

An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
ESP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX

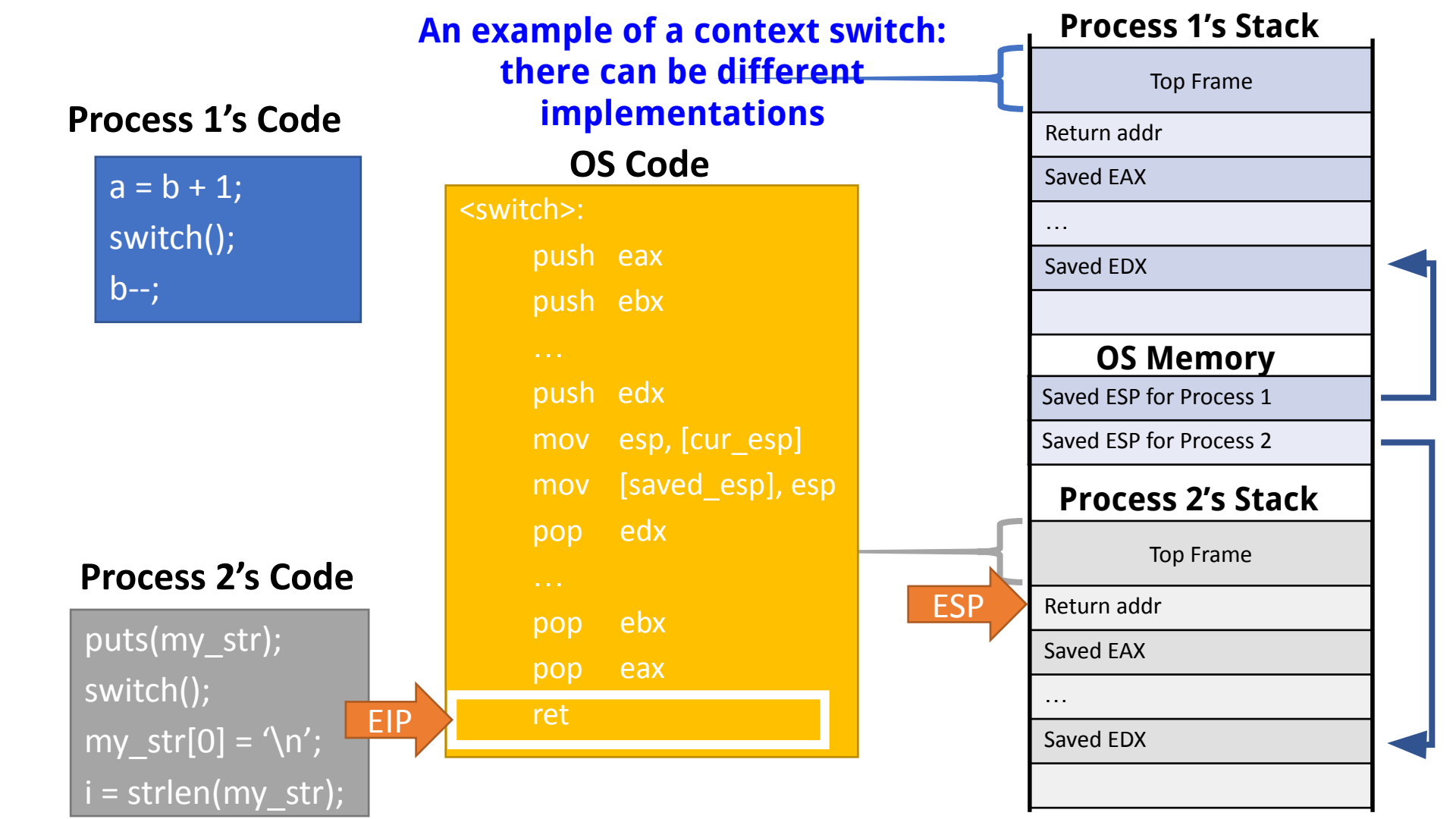

An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
ESP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX


An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
ESP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX


An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
ESP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX


An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
ESP


An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
ESP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX


An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
EIP
ESP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX

An example of a context switch: there can be different implementations

Process 1's Code

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

Process 2's Code

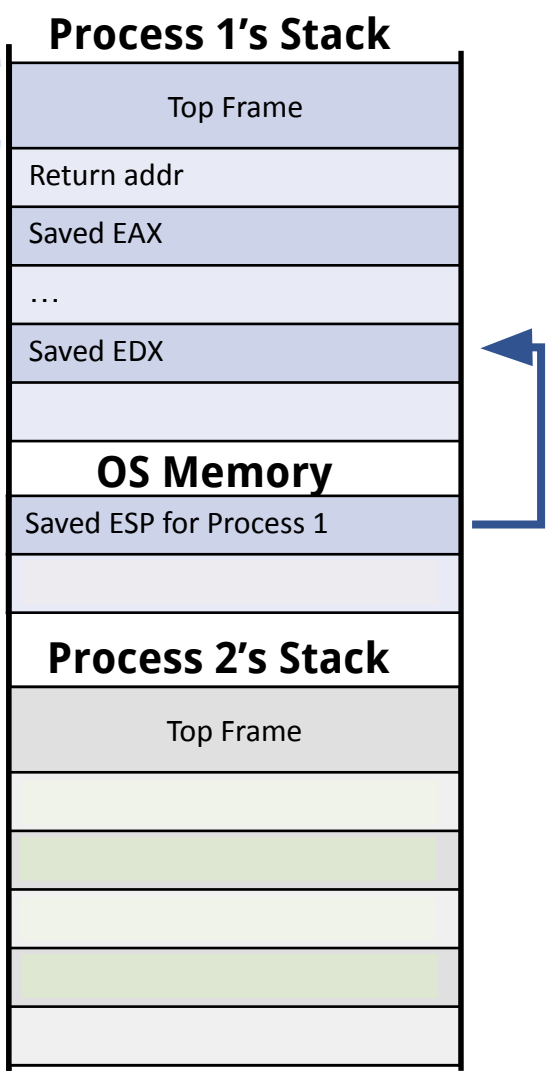
```
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
```

EIP

OS Code

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

ESP



An example of a context switch: there can be different implementations

Process 1's Code

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

Process 2's Code

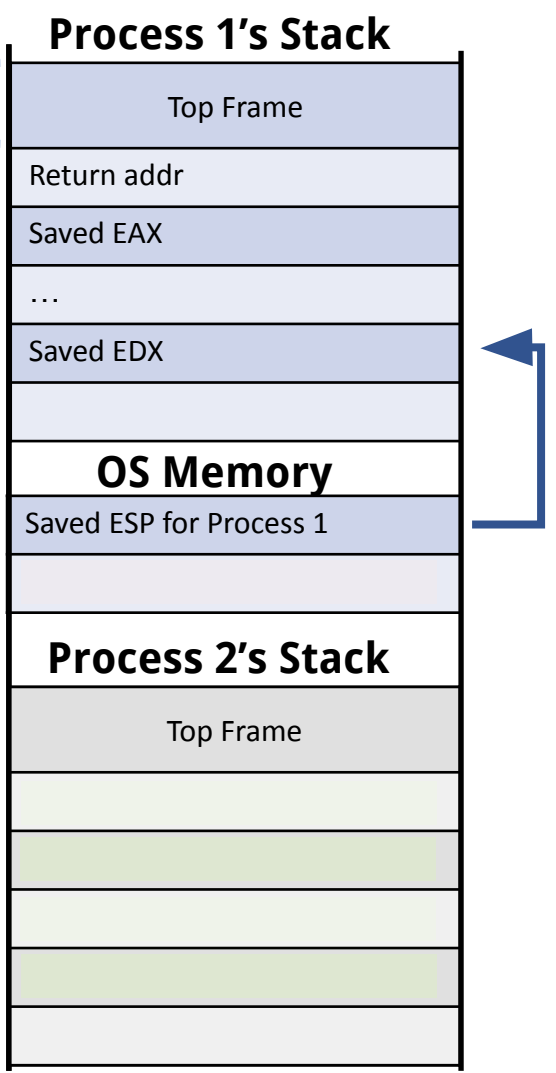
```
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
```

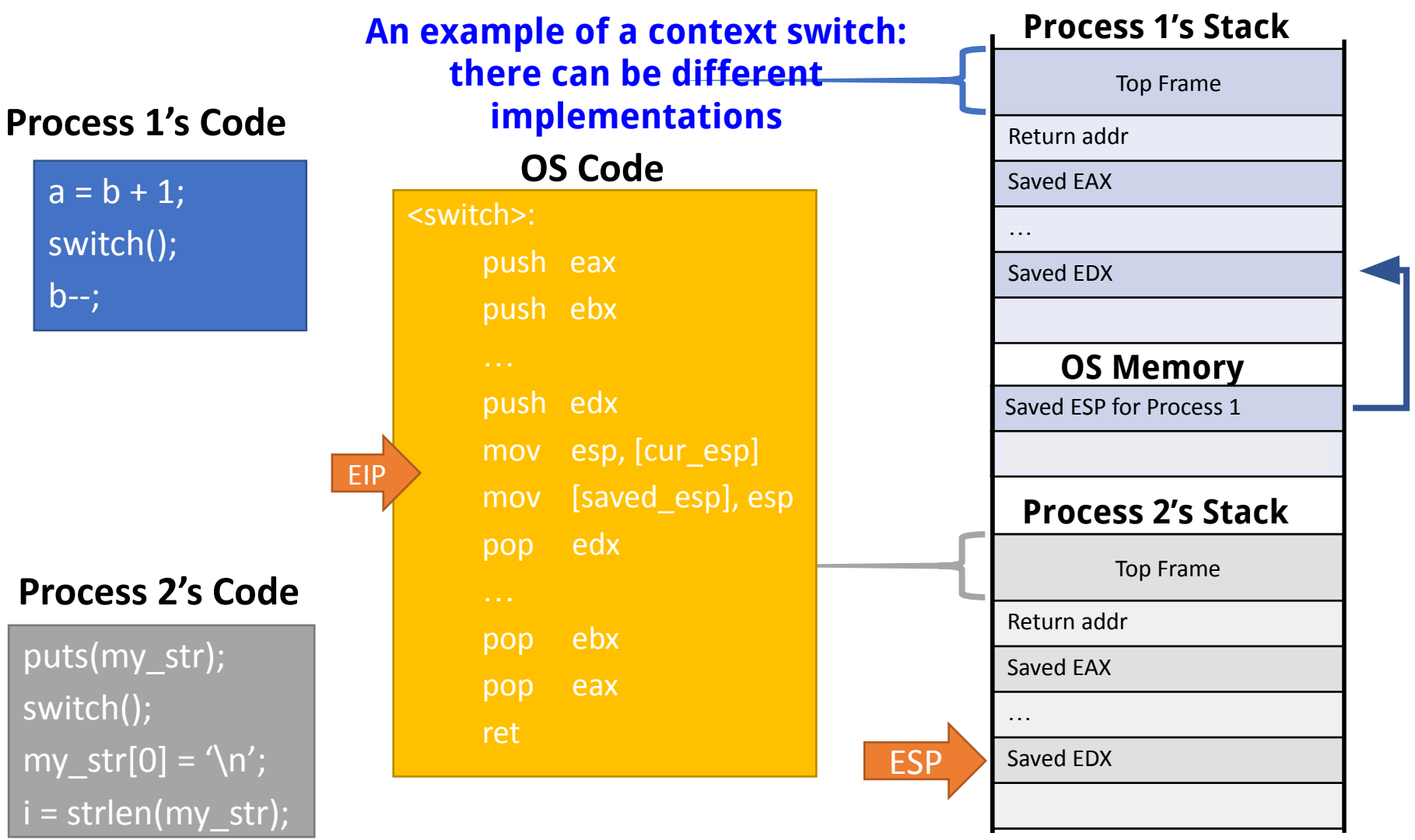
EIP

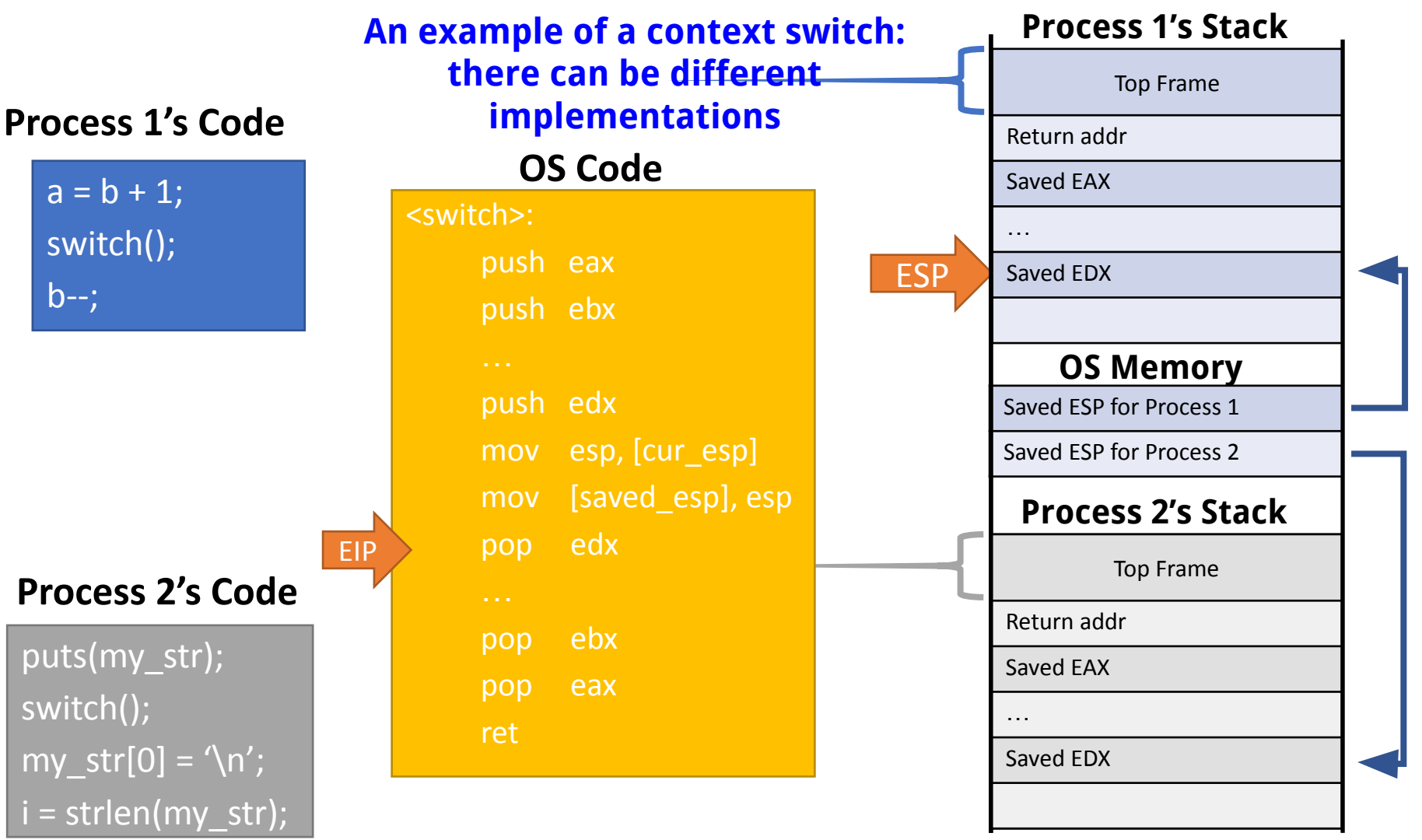
OS Code

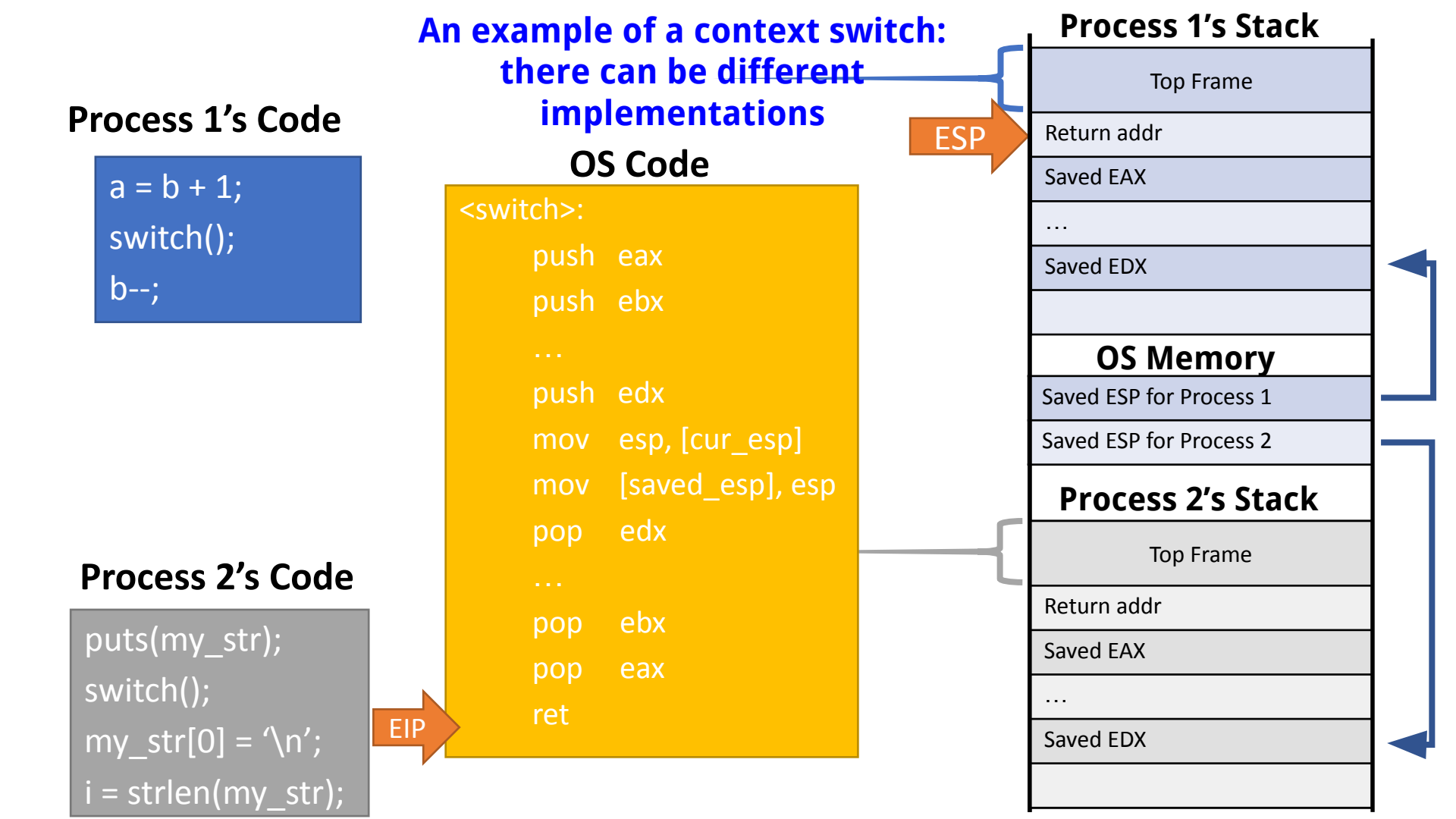
```
<switch>:
        push   eax
        push   ebx
        …
        push   edx
        mov    esp, [cur_esp]
        mov    [saved_esp], esp
        pop    edx
        …
        pop    ebx
        pop    eax
        ret
```

ESP




An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
ESP


An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
ESP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX


An example of a context switch: there can be different implementations
Process 1's Code
a = b + 1;
switch();
b--;
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
EIP
ESP
Process 1's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX
OS Memory
Saved ESP for Process 1
Saved ESP for Process 2
Process 2's Stack
Top Frame
Return addr
Saved EAX
…
Saved EDX

An example of a context switch: there can be different implementations

Process 1's Code

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

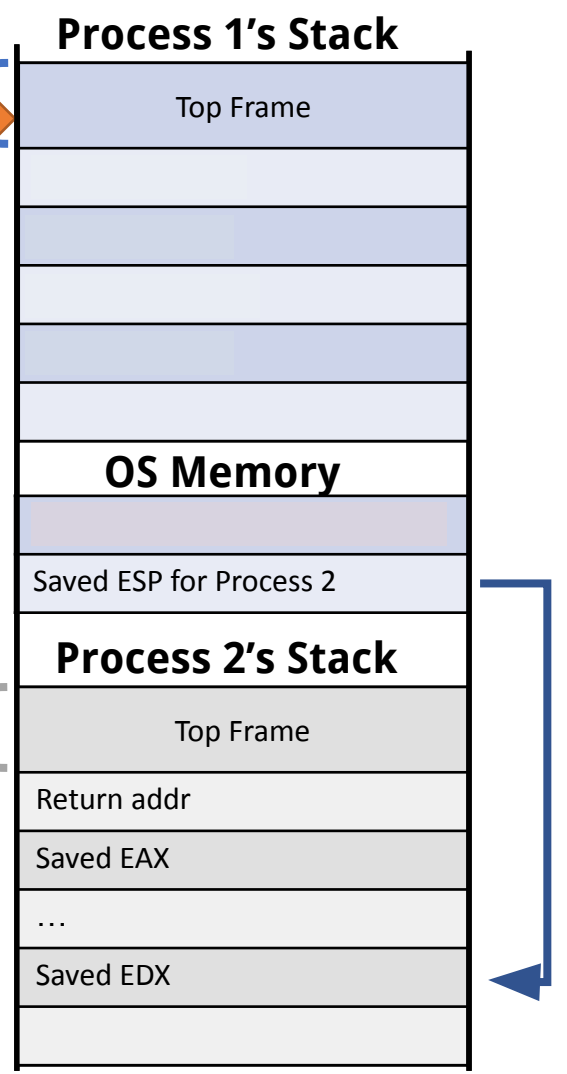
EIP

Process 2's Code

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

OS Code

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



xv6 as an example

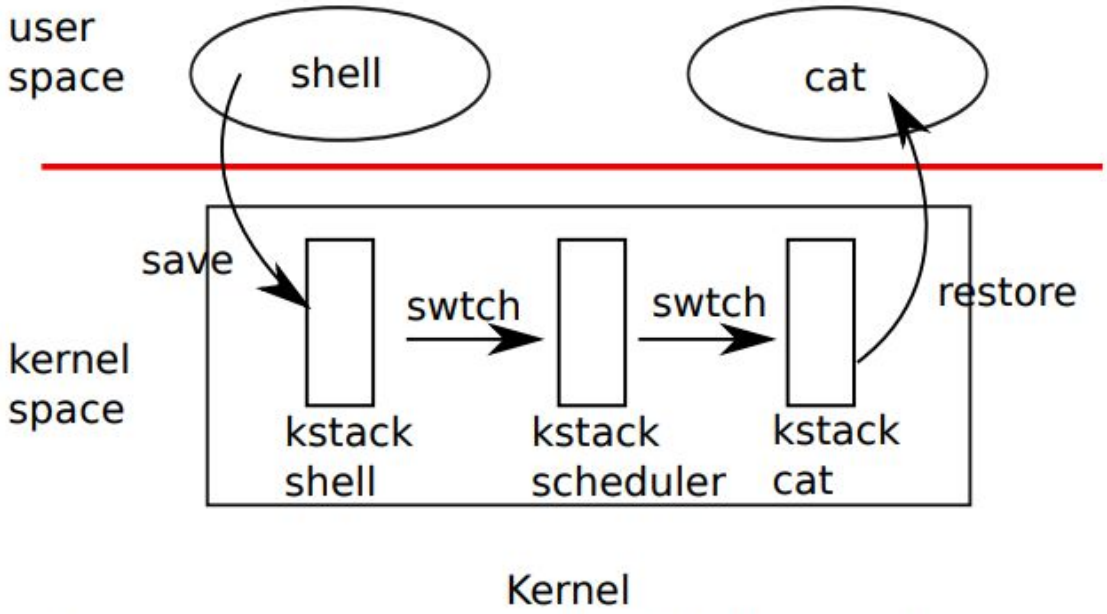


Figure 5-1. Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

xv6 as an example: process A running in user mode

```
37  // Per-process state
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52  };
53
```

https://github.com/mit-pdos/xv6-public/blob/master/proc.h

xv6 as an example: timer interrupt fires

```
1   #include "mmu.h"
2
3     # vectors.S sends all traps here.
4   .globl alltraps
5   alltraps:
6     # Build trap frame.
7     pushl %ds
8     pushl %es
9     pushl %fs
10    pushl %gs
11    pushal
12
13    # Set up data segments.
14    movw $(SEG_KDATA<<3), %ax
15    movw %ax, %ds
16    movw %ax, %es
17
18    # Call trap(tf), where tf=%esp
19    pushl %esp
20    call trap
21    addl $4, %esp
22
23    # Return falls through to trapret...
24  .globl trapret
25  trapret:
26    popal
27    popl %gs
28    popl %fs
29    popl %es
30    popl %ds
31    addl $0x8, %esp  # trapno and errcode
32    iret
```

Hardware (on privilege change) switches to **A's kstack** (from TSS.esp0)

Hardware **pushes**: SS, ESP, EFLAGS, CS, EIP (old user context) onto A's **kstack**

alltraps then saves the rest:

pushal # eax,ecx,edx,ebx,esp,ebp,esi,edi

https://github.com/mit-pdos/xv6-public/blob/master/trapasm.S

xv6 as an example: timer interrupt fires

trapframe

https://github.com/mit-pdos/xv6-public/blob/master/x86.h

xv6 as an example: trap()

```
35    //PAGEBREAK: 41
36    void
37    trap(struct trapframe *tf)
38    {
39      if(tf->trapno == T_SYSCALL){
40        if(myproc()->killed)
41          exit();
42        myproc()->tf = tf;
43        syscall();
44        if(myproc()->killed)
45          exit();
46        return;
47      }
48
49      switch(tf->trapno){
50      case T_IRQ0 + IRQ_TIMER:
51        if(cpuid() == 0){
52          acquire(&tickslock);
53          ticks++;
54          wakeup(&ticks);
55          release(&tickslock);
56        }
57        lapiceoi();
58        break;
```

```
103       // Force process to give up CPU on clock tick.
104       // If interrupts were on while locks held, would need to check nlock.
105       if(myproc() && myproc()->state == RUNNING &&
106          tf->trapno == T_IRQ0+IRQ_TIMER)
107         yield();
```

https://github.com/mit-pdos/xv6-public/blob/master/trap.c

xv6 as an example: yield() from A to scheduler

```
384     // Give up the CPU for one scheduling round.
385     void
386     yield(void)
387     {
388       acquire(&ptable.lock);  //DOC: yieldlock
389       myproc()->state = RUNNABLE;
390       sched();
391       release(&ptable.lock);
392     }
```

https://github.com/mit-pdos/xv6-public/blob/master/proc.c

xv6 as an example: sched()

```
358  // Enter scheduler.  Must hold only ptable.lock
359  // and have changed proc->state. Saves and restores
360  // intena because intena is a property of this
361  // kernel thread, not this CPU. It should
362  // be proc->intena and proc->ncli, but that would
363  // break in the few places where a lock is held but
364  // there's no process.
365  void
366  sched(void)
367  {
368    int intena;
369    struct proc *p = myproc();
370
371    if(!holding(&ptable.lock))
372      panic("sched ptable.lock");
373    if(mycpu()->ncli != 1)
374      panic("sched locks");
375    if(p->state == RUNNING)
376      panic("sched running");
377    if(readeflags()&FL_IF)
378      panic("sched interruptible");
379    intena = mycpu()->intena;
380    swtch(&p->context, mycpu()->scheduler);
381    mycpu()->intena = intena;
382  }
```

sched() calls **swtch(&A->context, cpu->scheduler)**:

- saves **A's kernel context** (callee-saved regs & return EIP) into A->context
- loads the CPU's scheduler context
- jumps to it

https://github.com/mit-pdos/xv6-public/blob/master/proc.c

xv6 as an example: swtch

```
1   # Context switch
2   #
3   #   void swtch(struct context **old, struct context *new);
4   #
5   # Save the current registers on the stack, creating
6   # a struct context, and save its address in *old.
7   # Switch stacks to new and pop previously-saved registers.
8
9   .globl swtch
10  swtch:
11    movl 4(%esp), %eax
12    movl 8(%esp), %edx
13
14    # Save old callee-saved registers
15    pushl %ebp
16    pushl %ebx
17    pushl %esi
18    pushl %edi
19
20    # Switch stacks
21    movl %esp, (%eax)
22    movl %edx, %esp
23
24    # Load new callee-saved registers
25    popl %edi
26    popl %esi
27    popl %ebx
28    popl %ebp
29    ret
```

the kernel context of the **CPU's scheduler** (the code that runs scheduler()

Save the current process's kernel registers into p->context,

and restore the scheduler's registers from cpu->scheduler.

https://github.com/mit-pdos/xv6-public/blob/master/swtch.S

xv6 as an example

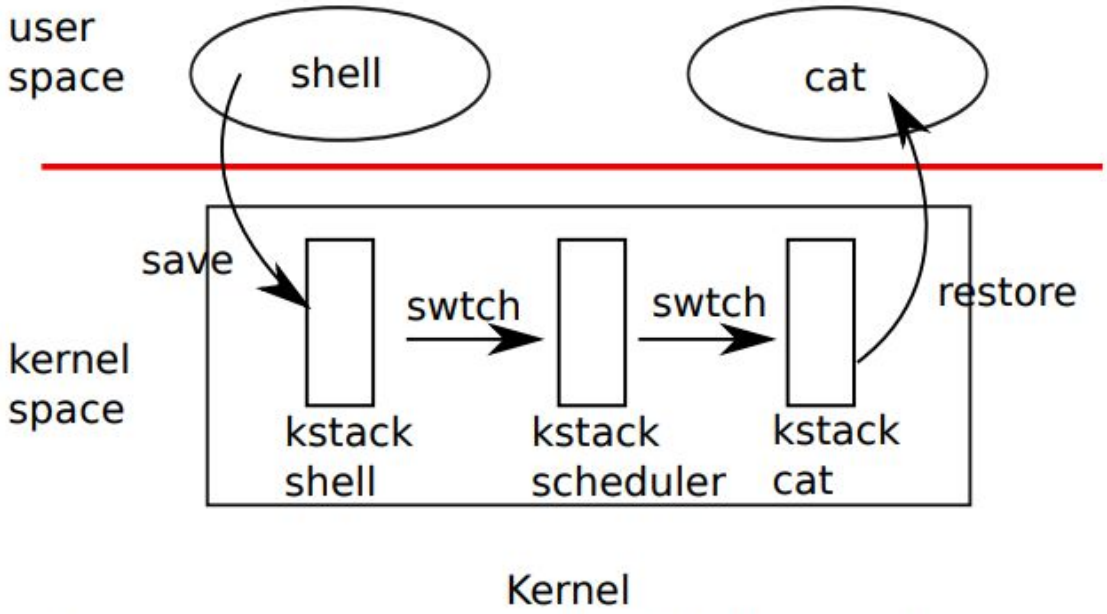


Figure 5-1. Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

xv6 as an example: CPU's scheduler loop picks B

```
314   //PAGEBREAK: 42
315   // Per-CPU process scheduler.
316   // Each CPU calls scheduler() after setting itself up.
317   // Scheduler never returns.  It loops, doing:
318   //  - choose a process to run
319   //  - swtch to start running that process
320   //  - eventually that process transfers control
321   //      via swtch back to the scheduler.
322   void
323   scheduler(void)
324   {
325     struct proc *p;
326     struct cpu *c = mycpu();
327     c->proc = 0;
328
329     for(;;){
330       // Enable interrupts on this processor.
331       sti();
332
333       // Loop over process table looking for process to run.
334       acquire(&ptable.lock);
335       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336         if(p->state != RUNNABLE)
337           continue;
338
339         // Switch to chosen process.  It is the process's job
340         // to release ptable.lock and then reacquire it
341         // before jumping back to us.
342         c->proc = p;
343         switchuvm(p);
344         p->state = RUNNING;
345
346         swtch(&(c->scheduler), p->context);
347         switchkvm();
348
349         // Process is done running for now.
350         // It should have changed its p->state before coming back.
```

The scheduler scans for a **RUNNABLE** process (here, **B**).

switchuvm(B):

- **loads CR3** with B's page directory (B's address space)
- sets the CPU's **TSS.esp0** = top of **B's kstack** (for future user→kernel entries)

swtch(&cpu->scheduler, B->context):

- saves scheduler's context
- **restores B's kernel context**
- **jumps** to B->context->eip (resuming B's kernel thread where it last yielded/slept) - ret instruction in swtch

https://github.com/mit-pdos/xv6-public/proc.c