CSCI 476: Computer Security

Lecture 3: Operating Systems, Processes, and `forking()`

Reese Pearsall

Fall 2023

*all images are stolen from the internet

Announcements

If you have an M1/M2 chip, or if you are still struggling with your VM, **check in with me this week**

Lab 0 due on Sunday 8/10 @ 11:59 PM

(All assignments will be due on Sundays)

We will start using our VM on Thursday

Might have to cancel a class next week

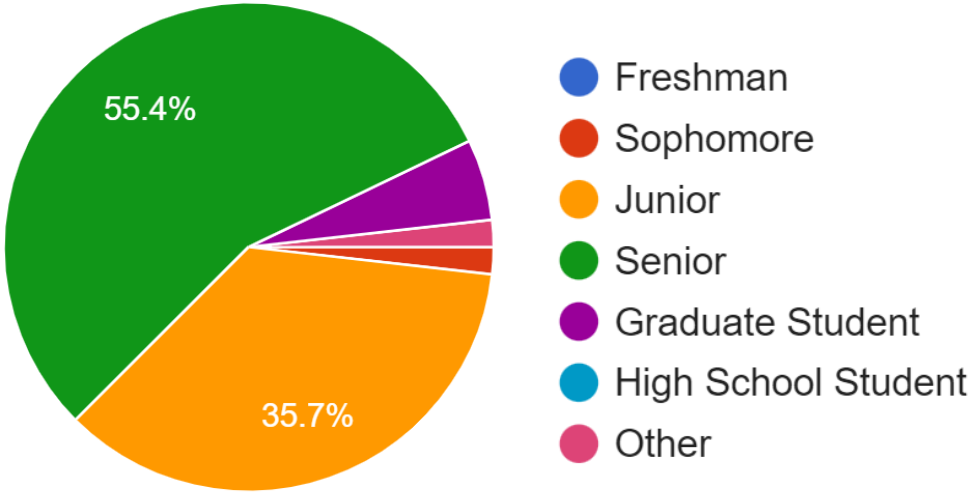
WELCOME HACKERCATS

Meeting Information
Wednesday 09/06 at 6 pm in NAH 137

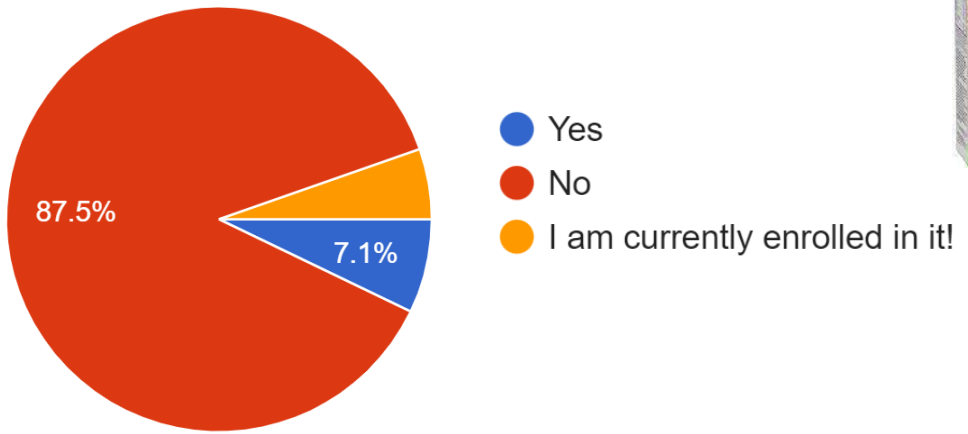
Topics

- Officer introductions
- Cybersecurity in a nutshell
- OSI interactive challenge

Course Questionnaire Results



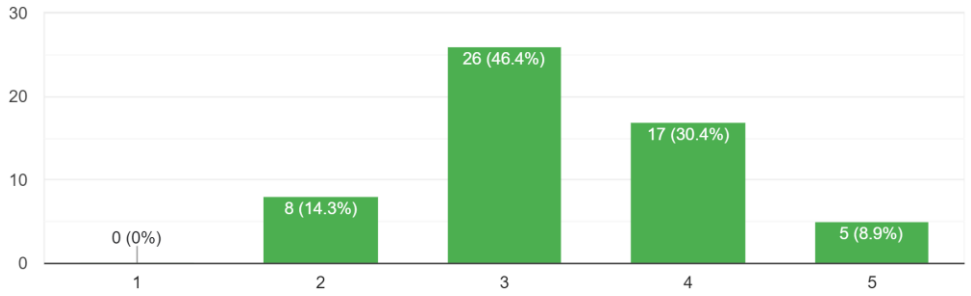
Have you taken Operating Systems (CSCI 460)



How comfortable are you C?

Copy

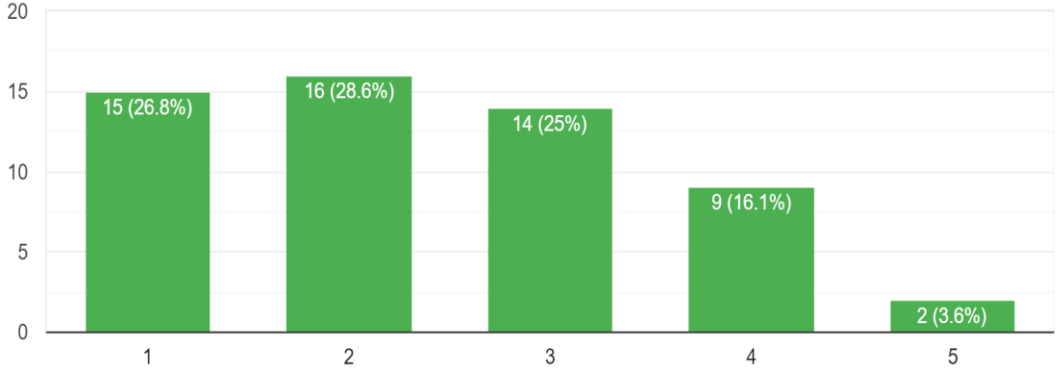
56 responses



How comfortable are you with reading assembly code?

Copy

56 responses



Course Questionnaire Results

"I am a big procrastinator"

Course Questionnaire Results

"I am a big procrastinator"

"This class seems relevant to my career path"

(and is required)

Course Questionnaire Results

"I am a big procrastinator"

"Im interested in learning about cryptography"

"This class seems relevant to my career path"

(and is required)

Course Questionnaire Results

"I am a big procrastinator"

"Im interested in learning about cryptography"

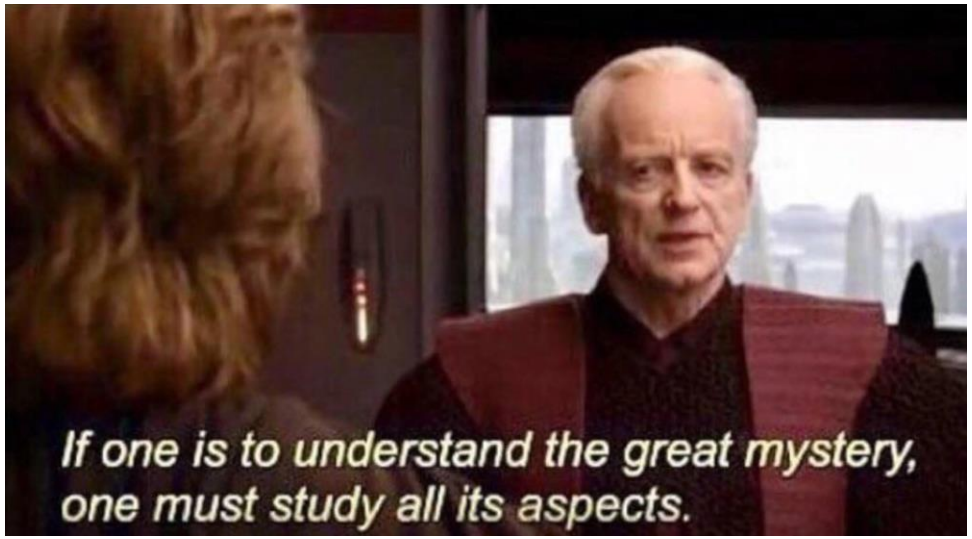
"This class seems relevant to my career path"

(and is required)

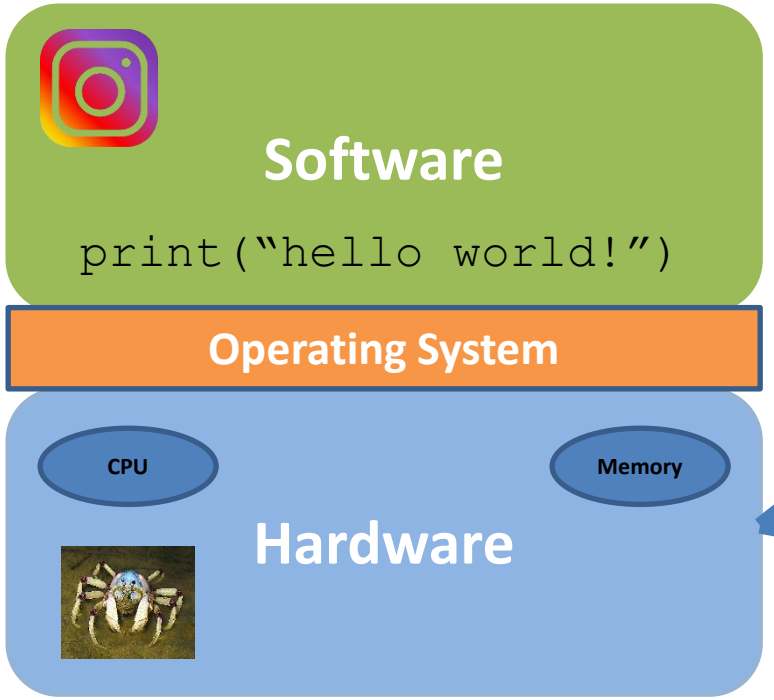
"The best cereal is *just milk*"

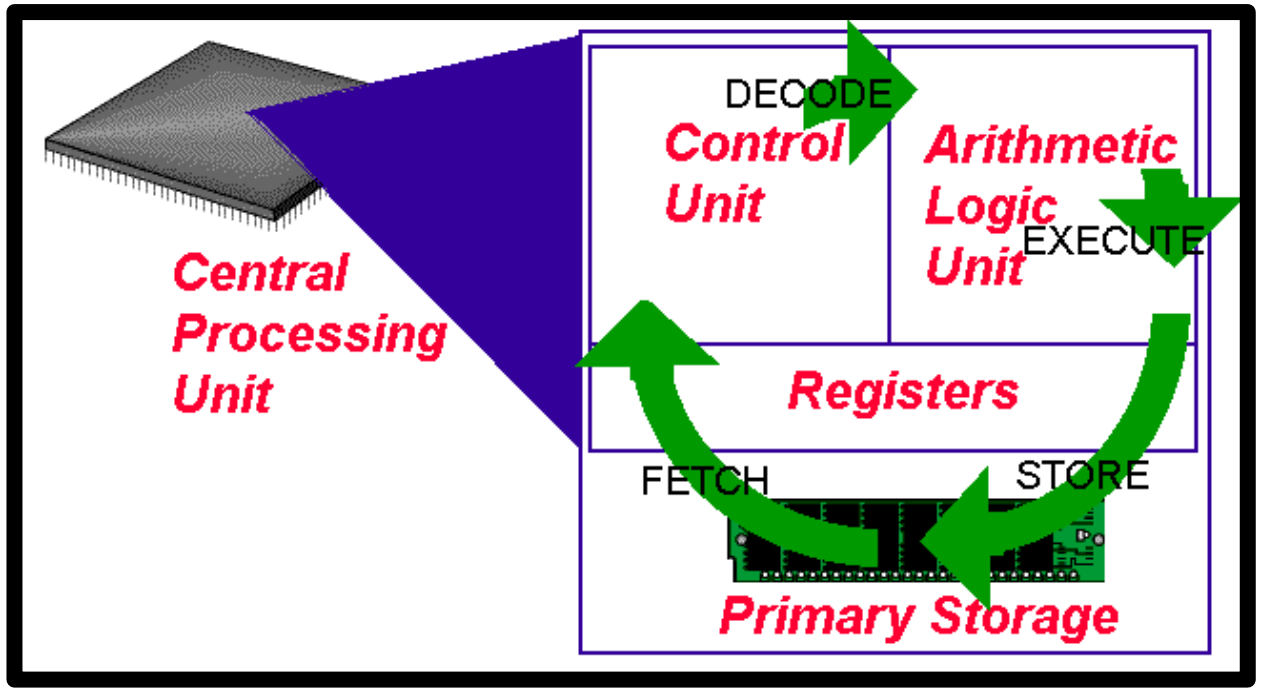
To understand the technical aspects of security, we must have a good understanding of how ~~computers~~ work

operating systems

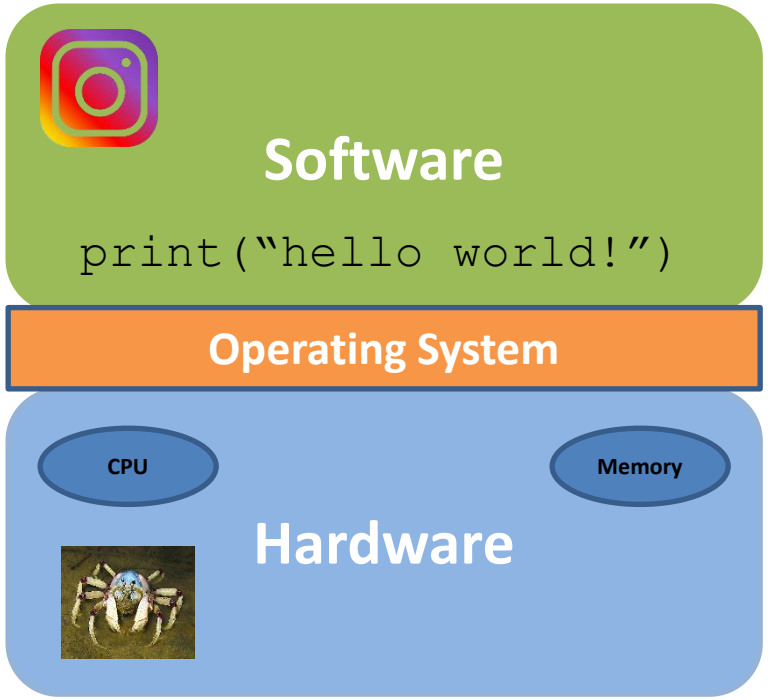


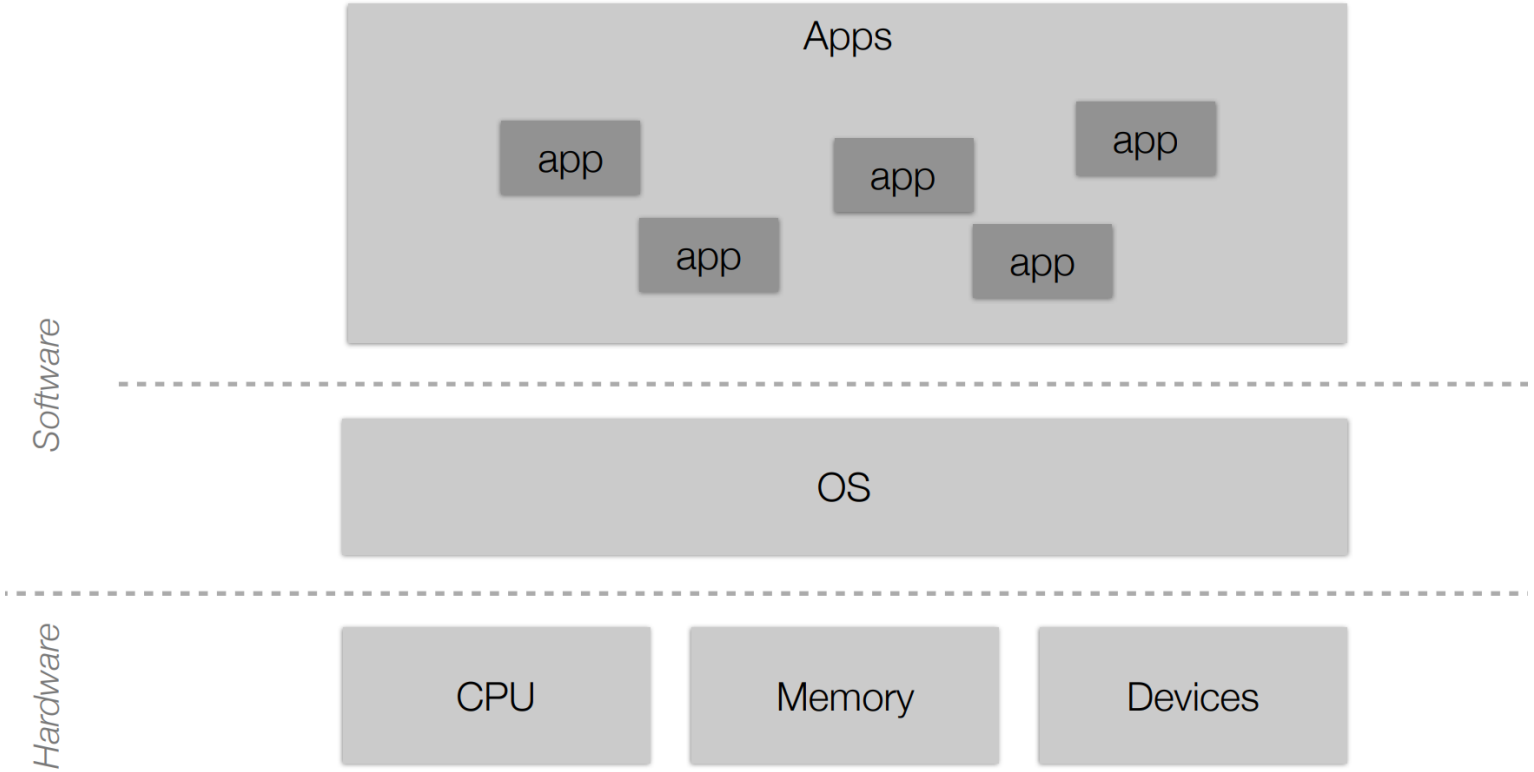
The Operating System





The Operating System





The jobs of an Operating System

1. **Process Manager**
   "The Coach"
2. **Interface Manager**
   "The Bouncer"
3. **Memory Manager**
   "The Farmer"
4. **Traffic Manager**
   "The Judge"
5. **Illusion Manager**
   "The Illusionist"

The jobs of an Operating System

1. **Process Manager**
   "The Coach"
2. **Interface Manager**
   "The Bouncer"
3. **Memory Manager**
   "The Farmer"
4. **Traffic Manager**
   "The Judge"
5. **Illusion Manager**
   "The Illusionist"

This will be the focus of the first half of lecture

Source code to binary

```c
#include <stdio.h>

int main() {
        printf("Hello WOrld! \n");


        int x = 0;
        int y = 3;

        int z = x + y;

        printf("%d  %d  %d \n",x,y,z);
        return 0;

}
```

Preprocessor

- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 10             sub    $0x10,%rsp
   c:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 13 <main+0x13>
  13:   e8 00 00 00 00          callq  18 <main+0x18>
  18:   c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%rbp)
  1f:   c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
  26:   8b 55 f4                mov    -0xc(%rbp),%edx
  29:   8b 45 f8                mov    -0x8(%rbp),%eax
  2c:   01 d0                   add    %edx,%eax
  2e:   89 45 fc                mov    %eax,-0x4(%rbp)
  31:   8b 4d fc                mov    -0x4(%rbp),%ecx
  34:   8b 55 f8                mov    -0x8(%rbp),%edx
  37:   8b 45 f4                mov    -0xc(%rbp),%eax
  3a:   89 c6                   mov    %eax,%esi
  3c:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 43 <main+0x43>
  43:   b8 00 00 00 00          mov    $0x0,%eax
  48:   e8 00 00 00 00          callq  4d <main+0x4d>
  4d:   b8 00 00 00 00          mov    $0x0,%eax
  52:   c9                      leaveq
  53:   c3                      retq
```

- Converted to assembly code
- .s file

Assembler

```
         00000000 00000100 0000000000000000
01011110 00001100 11000010 0000000000000010
         11101111 00010110 0000000000000101
         11101111 10011110 0000000000001011
11111000 10101101 11011111 0000000000010010
         01100010 11011111 0000000000010101
11101111 00000010 11111011 0000000000010111
11110100 10101101 11011111 0000000000011110
00000011 10100010 11011111 0000000000100001
11101111 00000010 11111011 0000000000100100
01111110 11110100 10101101
11111000 10101110 11000101 0000000000101011
00000110 10100010 11111011 0000000000110001
11101111 00000010 11111011 0000000000110100
         01010000 11010100 0000000000111011
                  00000100 0000000000111101
```

Linker

.exe

```
./hello_world
```

**THIS PROCESS IS NOT TRUE FOR EVERY LANGUAGE

What happens when we run `./hello_world`?

It gets turned into a **process**

A **process** is an instance of a <u>running</u> program on a computer

Processes | Performance | App history | Startup | Users | Details | Services

Name	Status	37% CPU	54% Memory	1% Disk	1% Network	17% GPU	GPU engine	Power usage	Power usage t...
Firefox (42)		6.5%	1,304.5 MB	0.5 MB/s	3.1 Mbps	9.0%	GPU 0 - Video Decode	High	Very low
Google Chrome (14)		0.8%	484.9 MB	0 MB/s	0 Mbps	0%		Low	Very low
Discord (32 bit) (6)		4.3%	328.8 MB	0 MB/s	8.7 Mbps	6.6%	GPU 0 - Video Encode	Moderate	Very low
Search		5.0%	185.9 MB	0.2 MB/s	0.8 Mbps	0%	GPU 0 - 3D	Moderate	Very low
Antimalware Service Executable		3.8%	178.2 MB	0.1 MB/s	0 Mbps	0%		Moderate	Very low
Google Chrome		0%	175.4 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Slack		0%	95.5 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Steam Client WebHelper		0%	89.1 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Google Chrome		0%	82.6 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Microsoft PowerPoint (32 bit) (2)		0.1%	69.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
SteelSeries GG Core		0.2%	67.7 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Steam Client WebHelper		0%	66.1 MB	0 MB/s	0 Mbps	0%		Very low	Very low

A **process** is an instance of a <u>running</u> program on a computer

All processes have the following data while they are running:

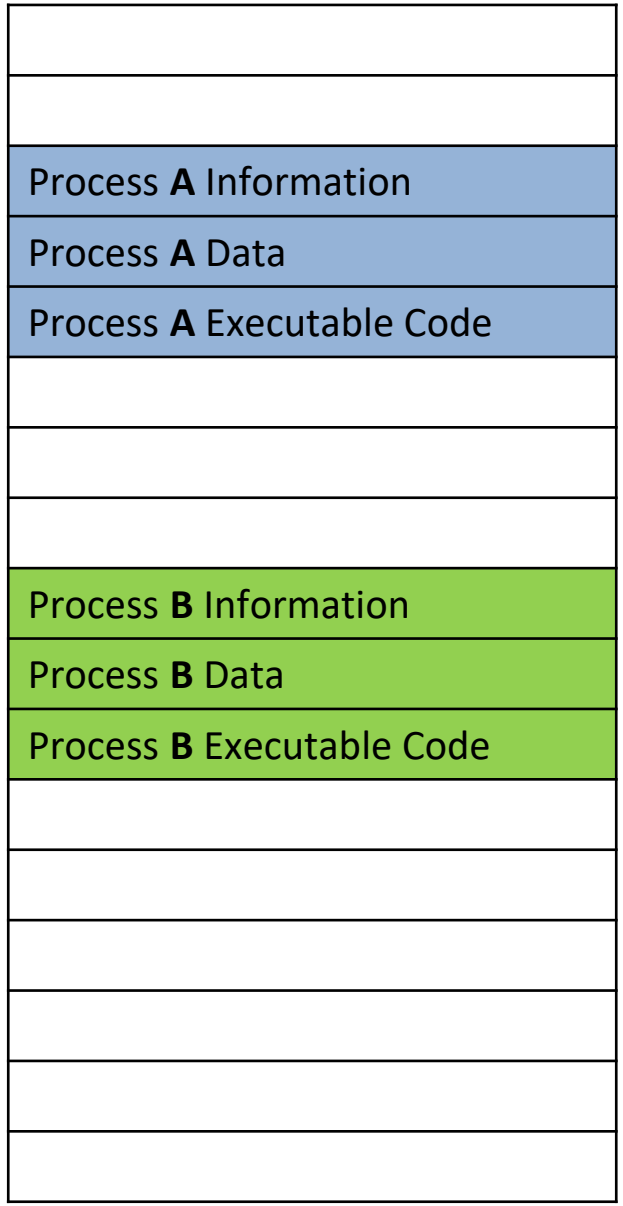
1. Executable Code

2. Associated Data

3. Execution Context/Bookkeeping information

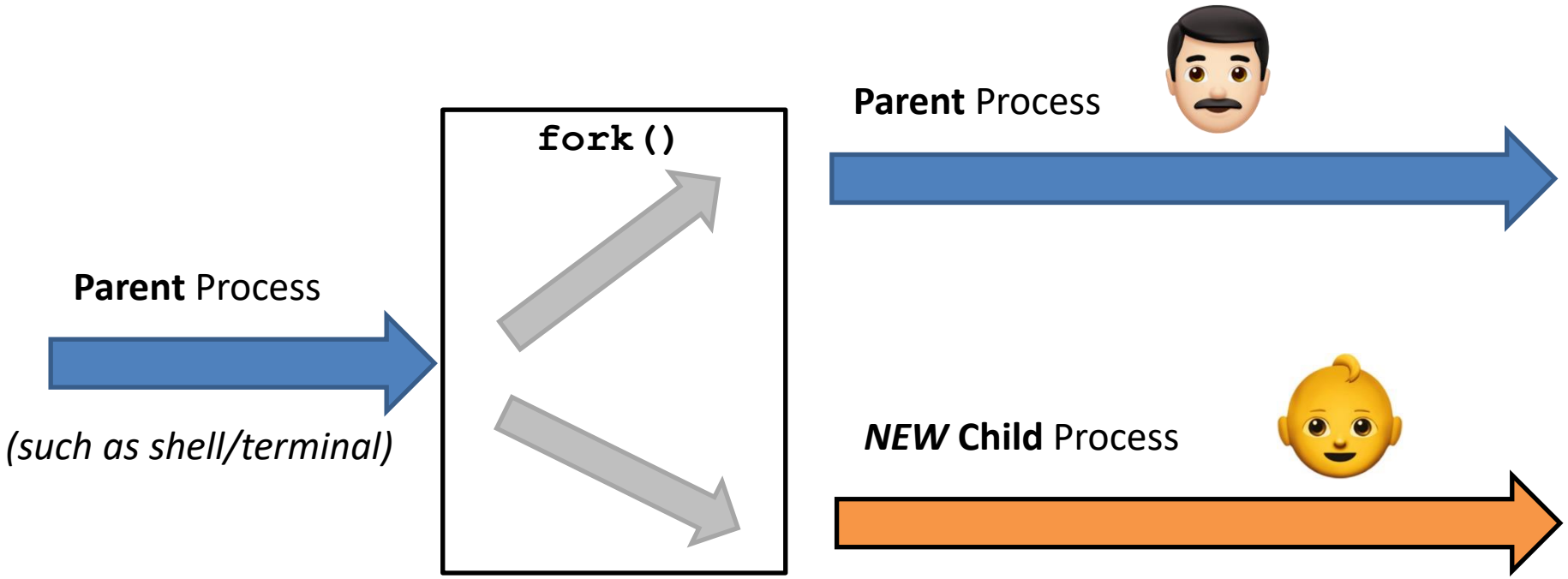
   (info that the OS needs to handle the process)

Main Memory



Ok, but how do we *actually* create a process?

- In the Unix family (and others), we use **`fork()`** to create a new process



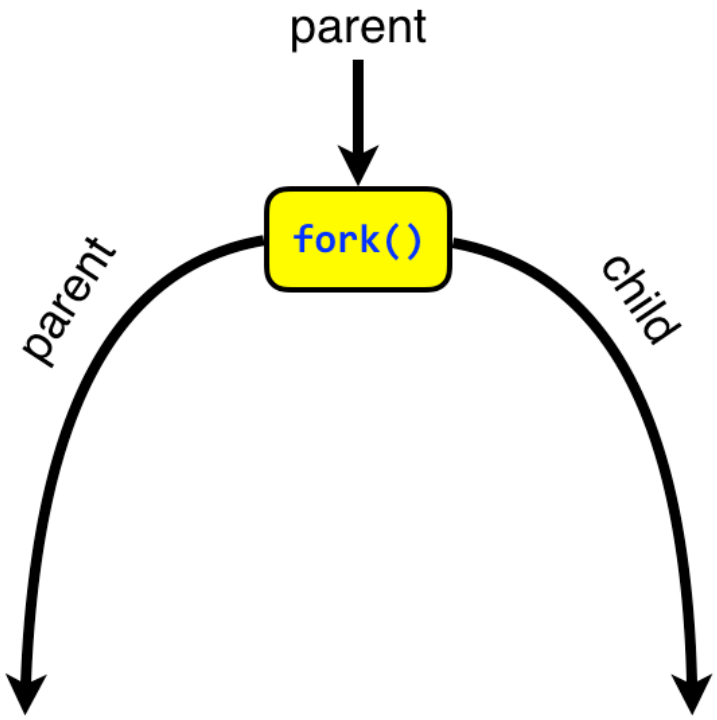
`fork()` duplicates a process so that instead of one process, you get two!

`fork()` duplicates a process so that instead of one process, you get two!

How can we tell the parent and child apart?

```
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child
        printf("Hi, I'm the child. \n");
    }

    sleep(1);
    printf("I'm the parent.);

    return 0;
}
```

We check the return value of **`fork()`**!



`fork()` duplicates a process so that instead of one process, you get two!

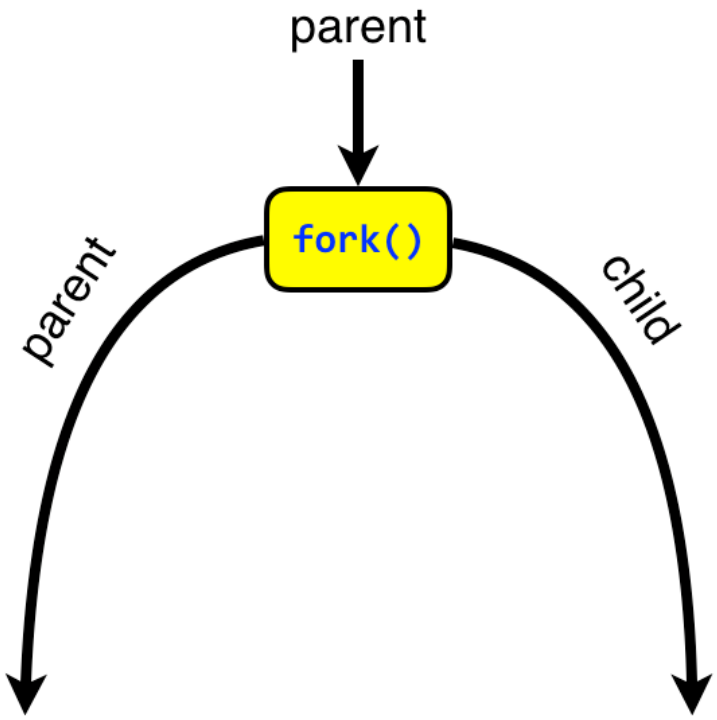
How can we tell the parent and child apart?

```
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child
        printf("Hi, I'm the child. \n");
    }

    sleep(1);
    printf("I'm the parent.);

    return 0;
}
```

parent

child

We check the return value of **`fork()`**!



1. Remember, **`fork()`** creates two process that are both actively running

`fork()` duplicates a process so that instead of one process, you get two!

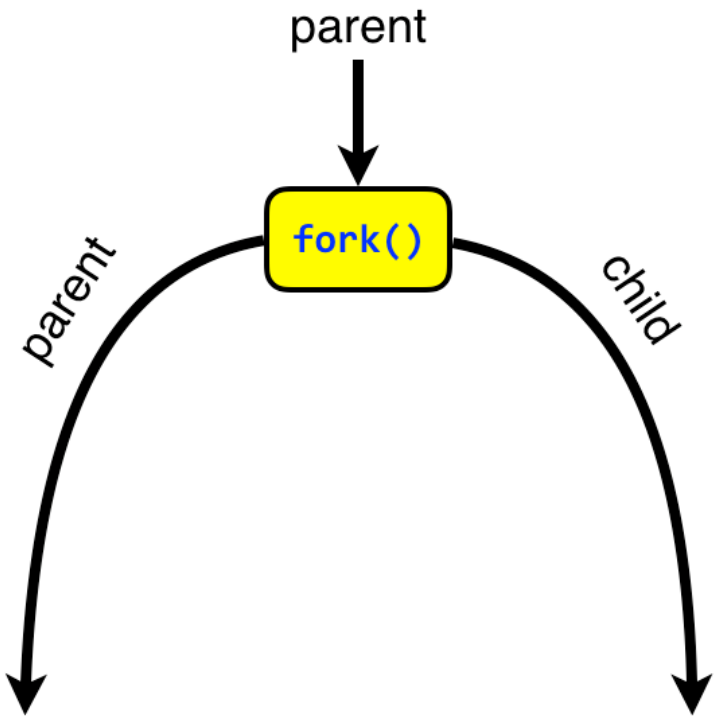
How can we tell the parent and child apart?

```
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child
        printf("Hi, I'm the child. \n");
    }

    sleep(1);
    printf("I'm the parent.);

    return 0;
}
```

We check the return value of **`fork()`**!

child

parent



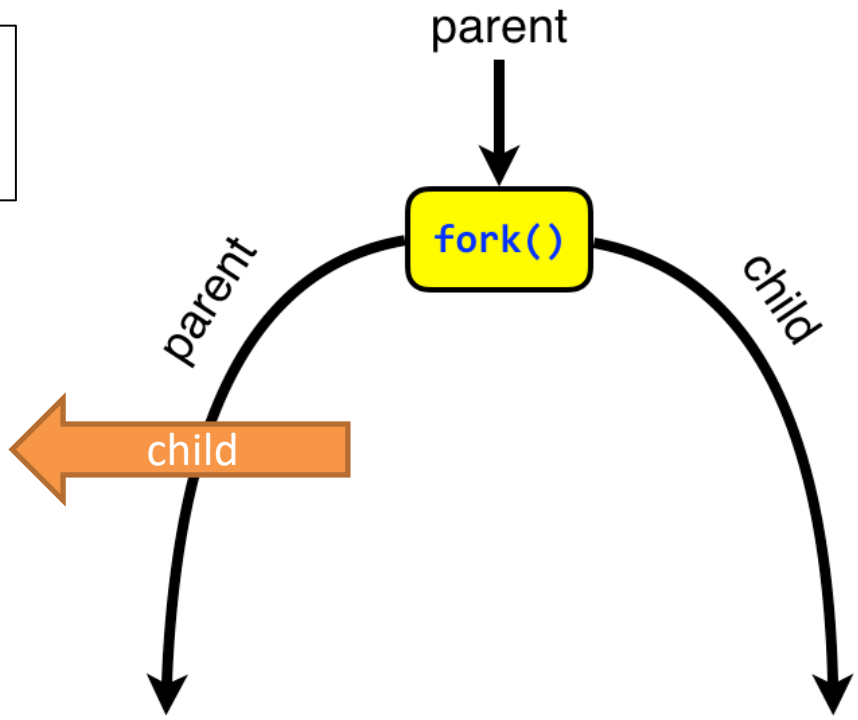
2. **`fork()`** always returns 0 for the child process, the parent process jumps to the code after the if statement

fork() duplicates a process so that instead of one process, you get two!

How can we tell the parent and child apart?

```
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child
        printf("Hi, I'm the child. \n");
    }

    sleep(1);
    printf("I'm the parent.);

    return 0;
}
```

We check the return value of **fork()**!



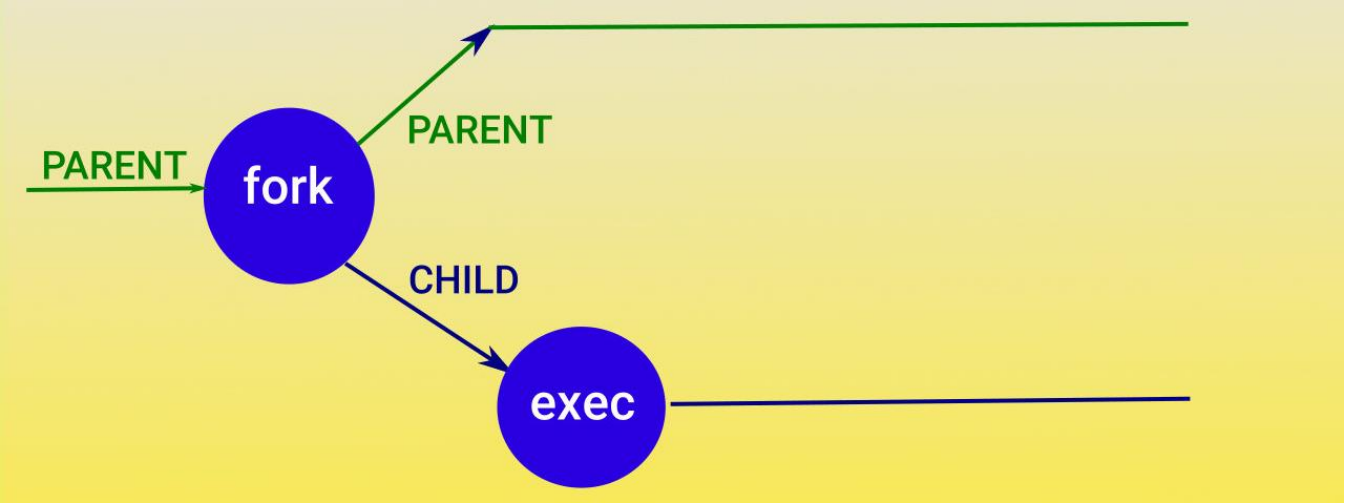
3. **fork()** always returns 0 for the child process, so the child process will execute the code in the if statement

Demo?

fork1.c

Issue: We want our child process to run an entirely new program (`hello_world` c program)

We use the **`exec()`** family of functions to execute a different program

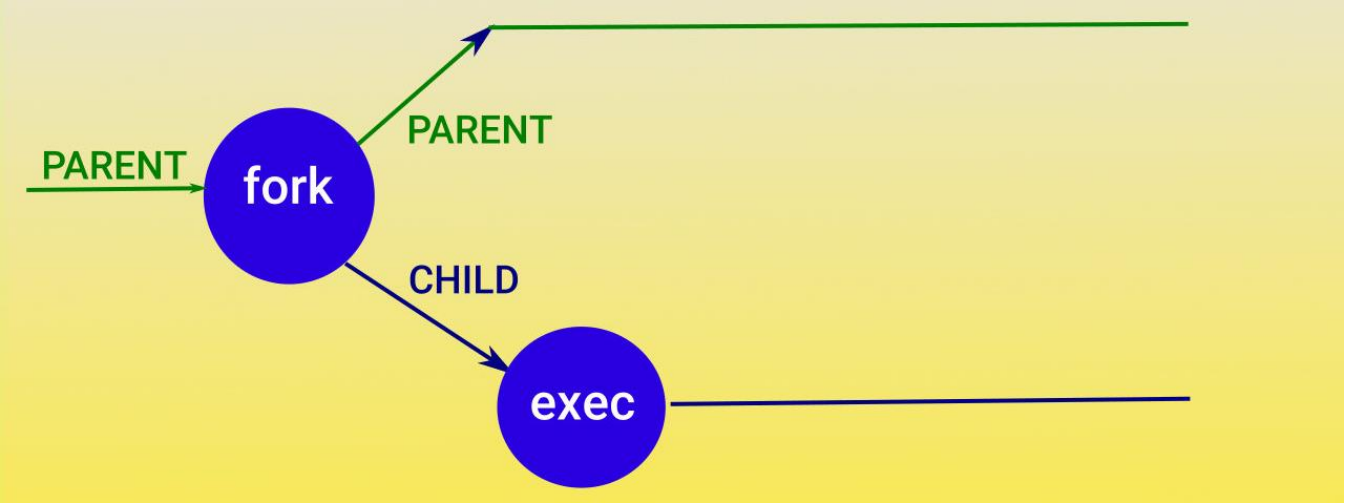


There are many different forms of the **`exec()`** function call

```
char *name[2];
name[0] = "./hello";
name[1] = NULL;
execve(name[0], name, NULL);
```

Issue: We want our child process to run an entirely new program (`hello_world` c program)

We use the **`exec()`** family of functions to execute a different program



There are many different forms of the **`exec()`** function call

```
char *name[2];
name[0] = "./hello";
name[1] = NULL;
execve(name[0], name, NULL);
```

This will invoke a program called `hello`

Fork() and Exec()

```
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child

        char *name[2];
        name[0] = "./hello";
        name[1] = NULL;
        execve(name[0], name, NULL);

        _exit(0);
    }
    sleep(1);
    printf("I'm the parent. My child has pid %d\n", pid);

    return 0;
}
```

Fork() and Exec()

```
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child

        char *name[2];
        name[0] = "./hello";
        name[1] = NULL;
        execve(name[0], name, NULL);

        _exit(0);
    }
    sleep(1);
    printf("I'm the parent. My child has pid %d\n", pid);

    return 0;
}
```

Child code

Parent code

Fork() and Exec()

```c
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child

        char *name[2];
        name[0] = "./hello";
        name[1] = NULL;
        execve(name[0], name, NULL);

        _exit(0);
    }
    sleep(1);
    printf("I'm the parent. My child has pid %d\n", pid);

    return 0;
}
```

output

```
[01/25/23]seed@VM:~$ ./forkexec
Hello from the C program!
I'm the parent. My child has pid 33578
```

Demo?

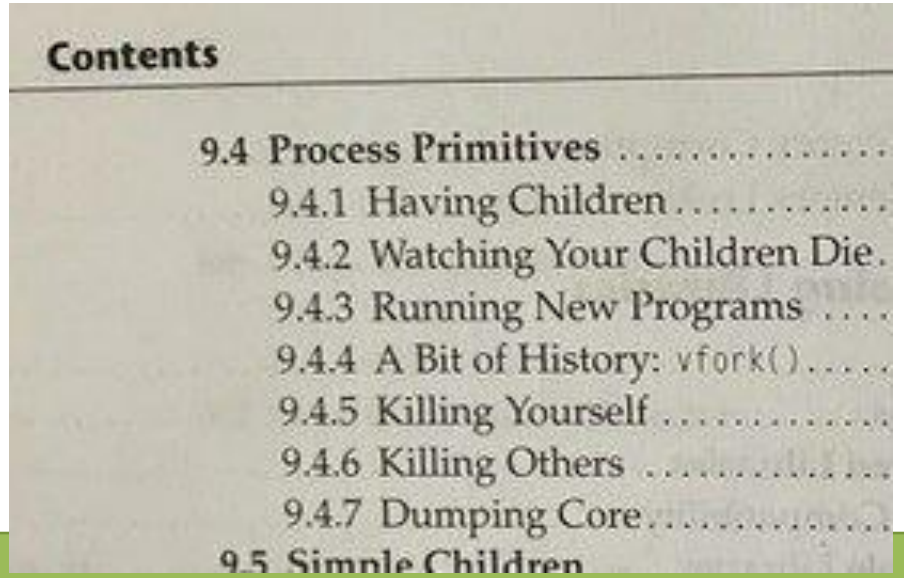
forkandexec.c

Tl;dr

The programs we run get turned into a **process**

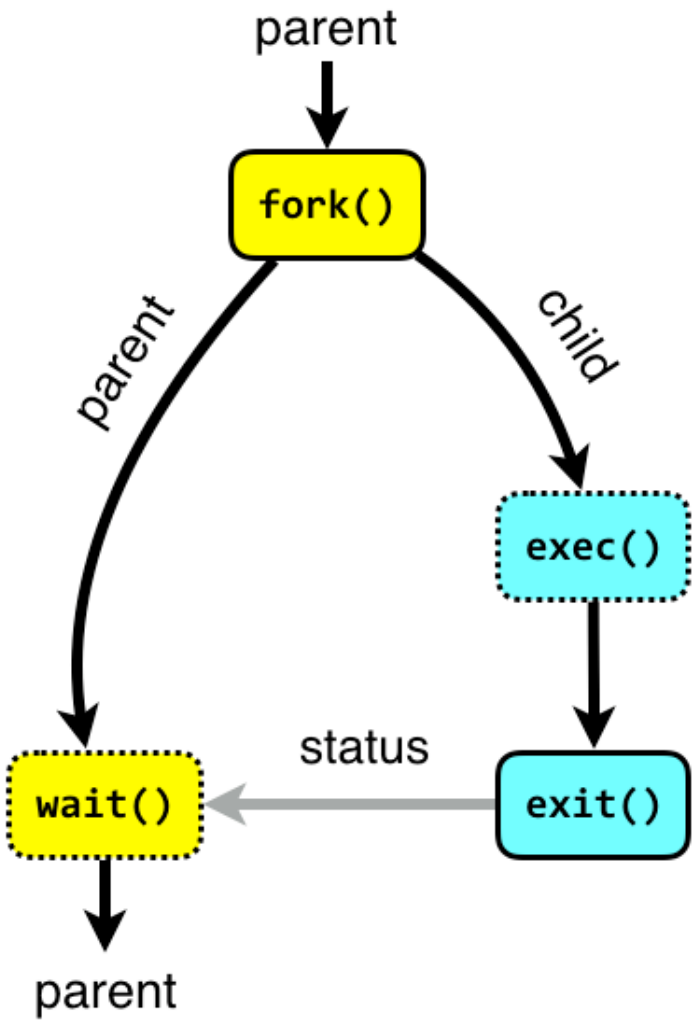
fork() is used to create a new process

- The parent process is typically the shell/terminal, and waits for the child process to finish
- The child process runs **exec()** to run our program



Contents

you can kill children with the `kill()` function or `kill` command



```c
#include <sys/types.h>
#include <unistd.h>

int main()
{
    while(1) {
        fork();
    }
    return 0;
}
```

Any ideas what might happen?

```c
#include <sys/types.h>
#include <unistd.h>

int main()
{
    while(1) {
        fork();
    }
    return 0;
}
```

"Oh, these forks() aren't homemade. They were made in factory. A **fork() bomb** factory. This is a **fork() bomb**"

A **process** is an instance of a <u>running</u> program on a computer

All processes have the following data while they are running:

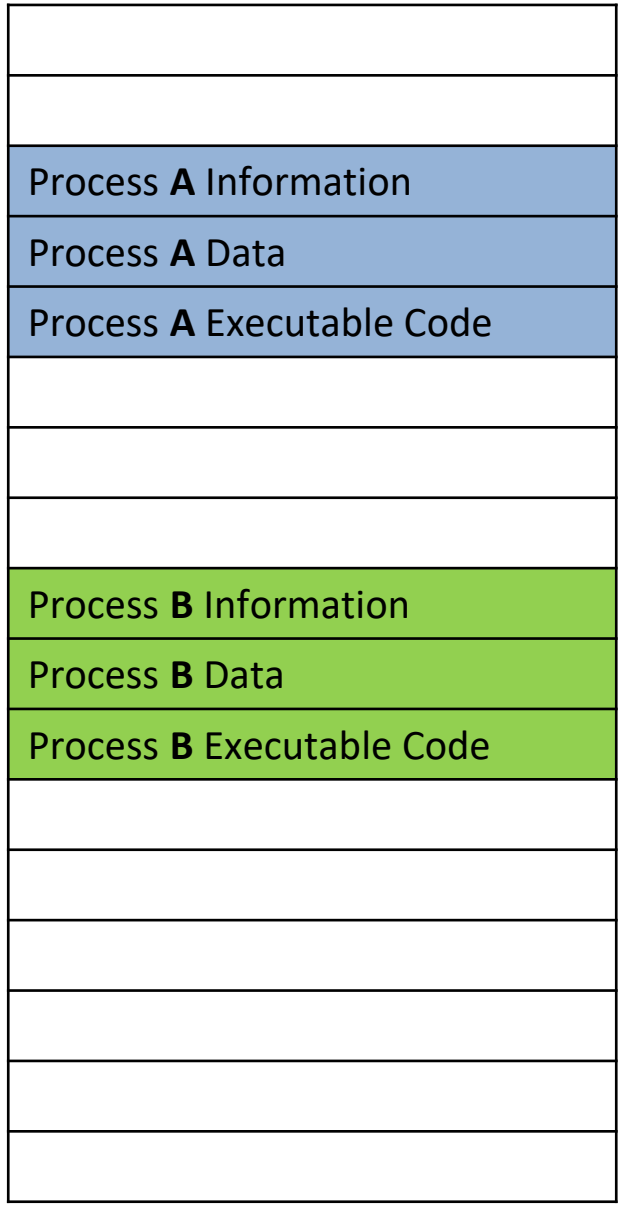
1. Executable Code

2. Associated Data

3. Execution Context/Bookkeeping information

   (info that the OS needs to handle the process)

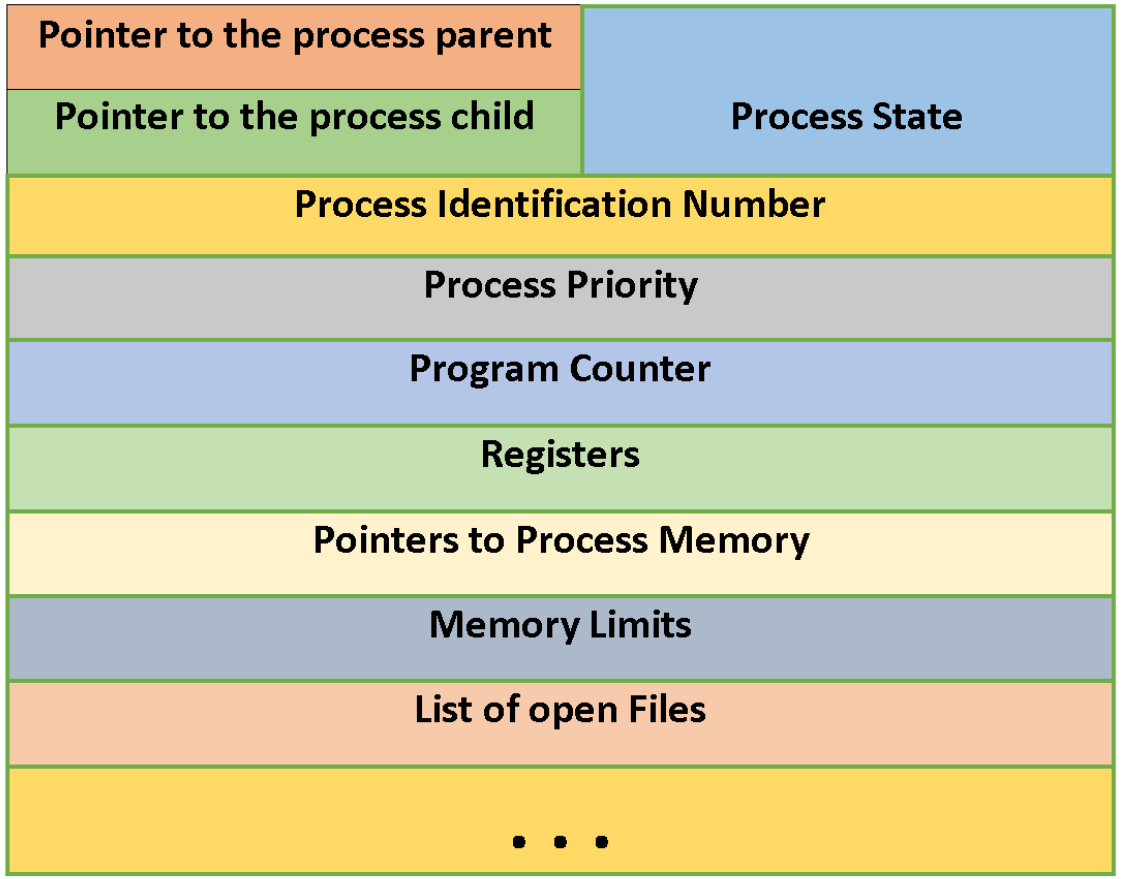
Main Memory



3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**
  - → Simply just a data structure that holds information
  - → The name of this varies by OS

Example PCB:



Created by Notes Jam

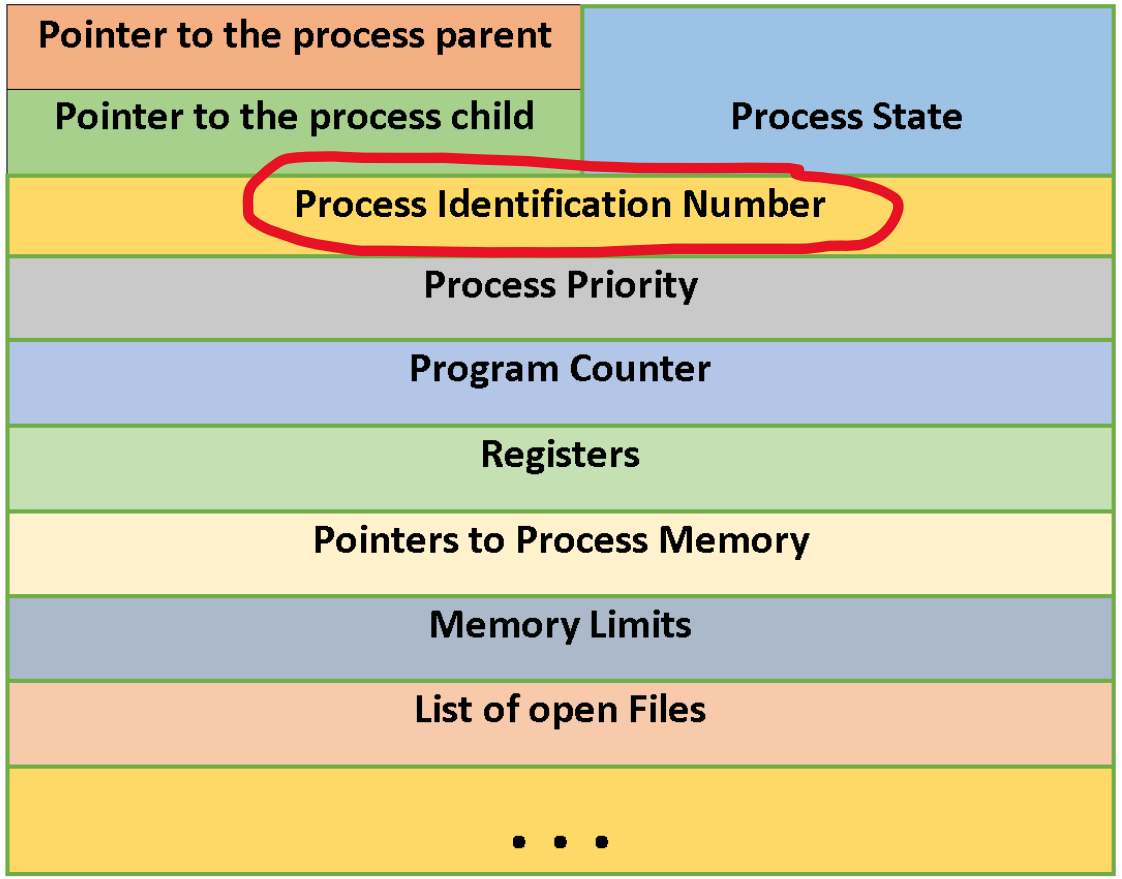
3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**
  - → Simply just a data structure that holds information
  - → The name of this varies by OS

Every process has a unique process ID (PID)

Process Name	User	% CPU	ID	Memory	Disk read tot: D
at-spi2-registryd	seed	0	1870	196.0 KiB	120.0 KiB
at-spi-bus-launcher	seed	0	1779	292.0 KiB	28.0 KiB
bash	seed	0	16245	1.6 MiB	3.1 MiB
bash	seed	0	20664	1.8 MiB	72.7 MiB
dbus-daemon	seed	0	1560	1.5 MiB	420.0 KiB

Example PCB:



We can use the PID to search for process, kill process, fork new process, etc

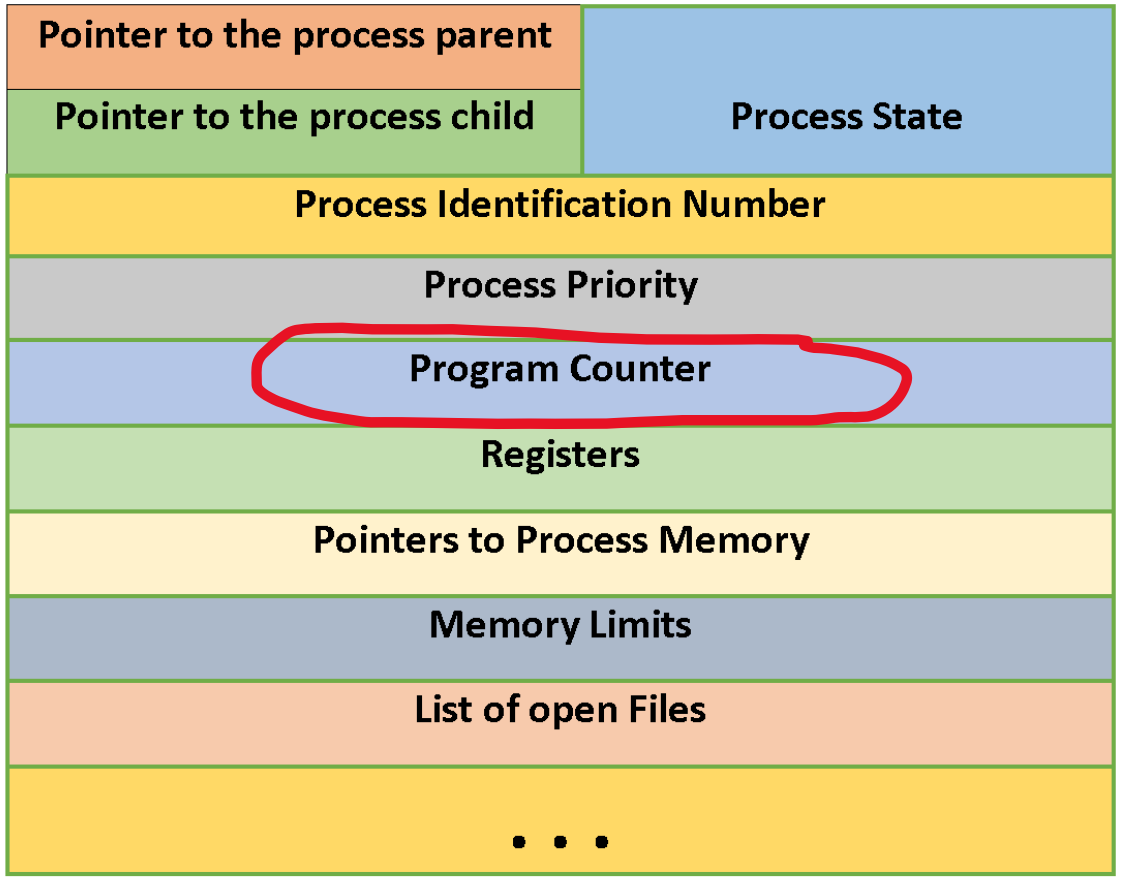
Created by Notes Jam

3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**
  - → Simply just a data structure that holds information
  - → The name of this varies by OS

Each process has a program counter (PC), which tells the CPU the next instruction to run in the process

Example PCB:



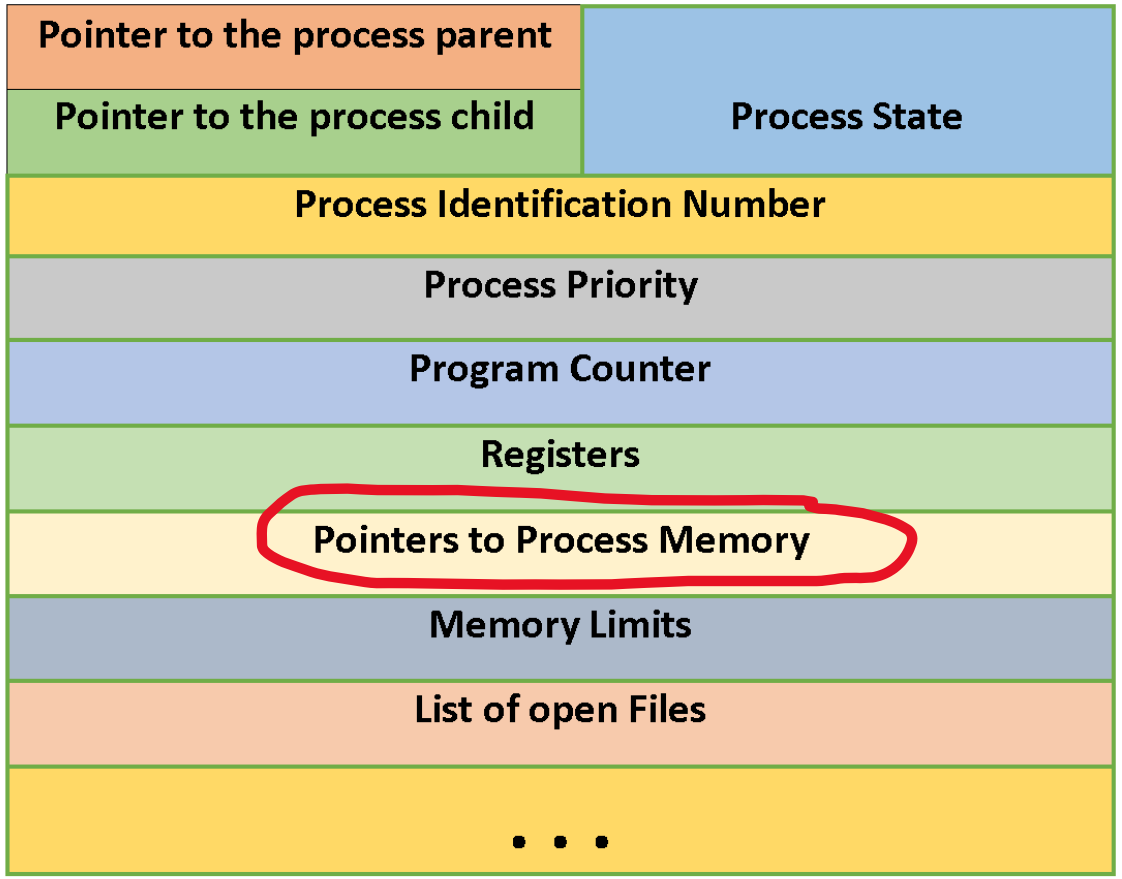
Created by Notes Jam

3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**
  - → Simply just a data structure that holds information
  - → The name of this varies by OS

PCB also maintains locations for the process Data and Code

Example PCB:

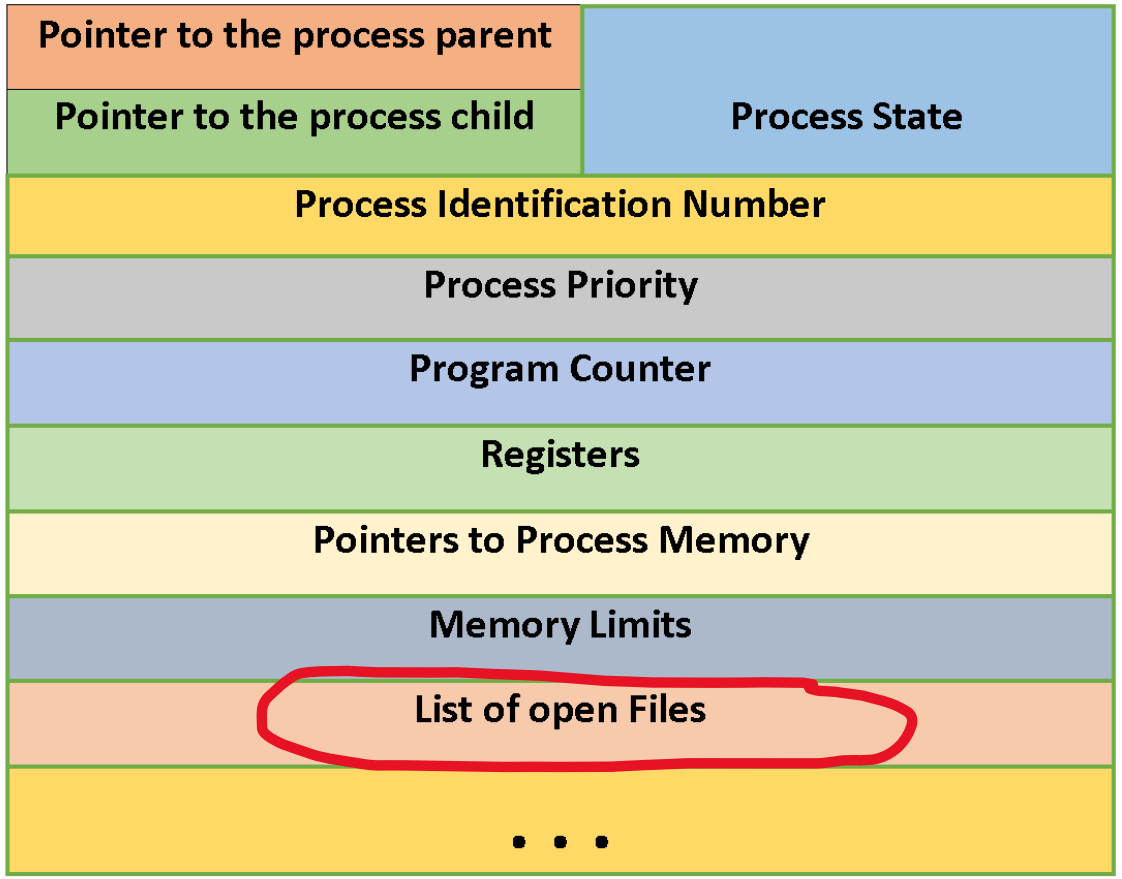


Created by Notes Jam

3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**
  - → Simply just a data structure that holds information
  - → The name of this varies by OS

Example PCB:



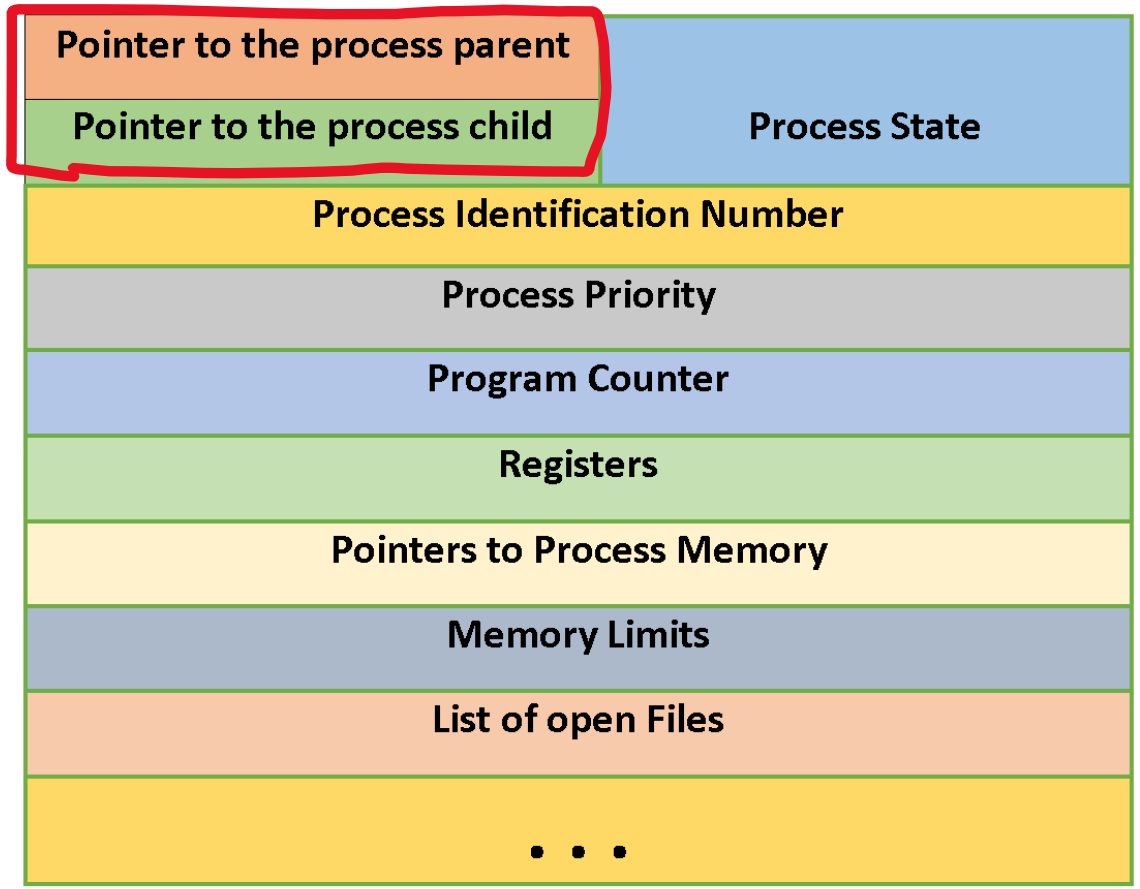
Created by Notes Jam

3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**
  - → Simply just a data structure that holds information
  - → The name of this varies by OS

PCB keeps track of who their parent is, and any child process (good parenting)

Example PCB:



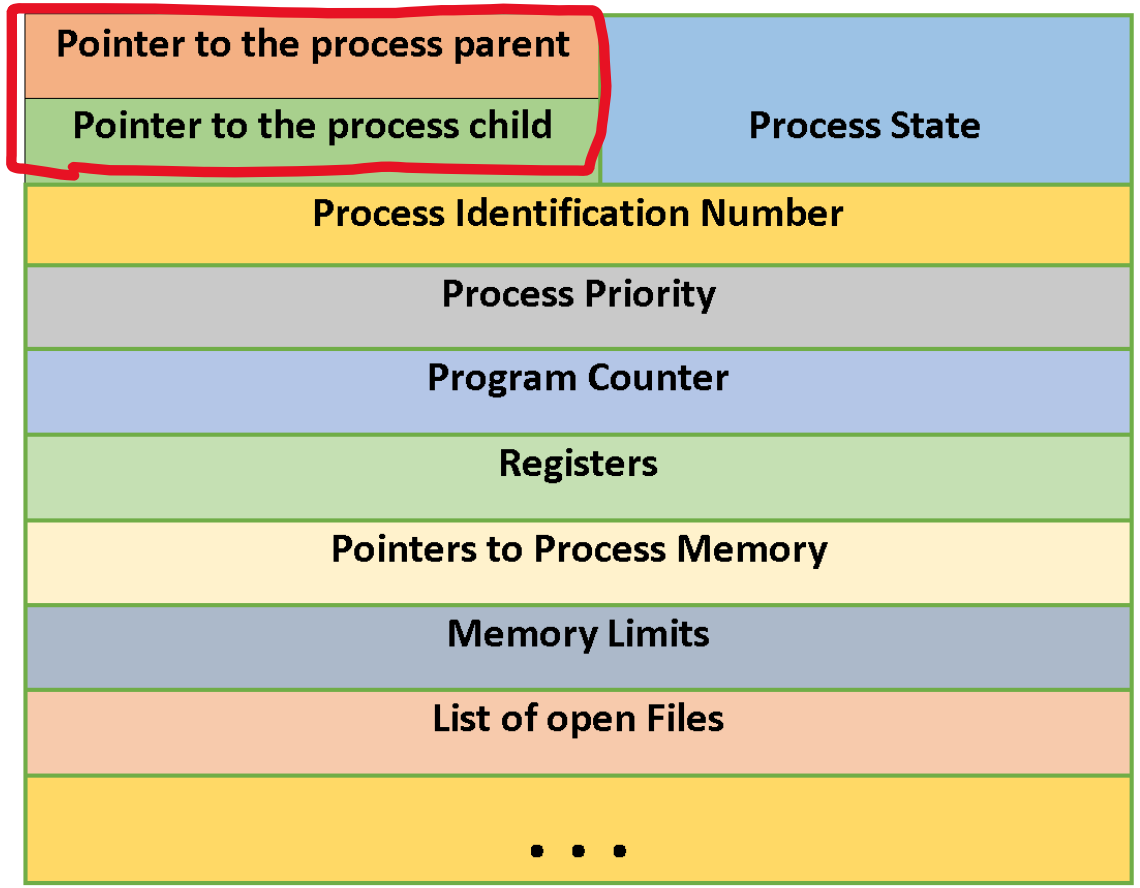
Created by Notes Jam

3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**
  - → Simply just a data structure that holds information
  - → The name of this varies by OS

PCB keeps track of who their parent is, and any child process (good parenting)

Example PCB:



Created by Notes Jam

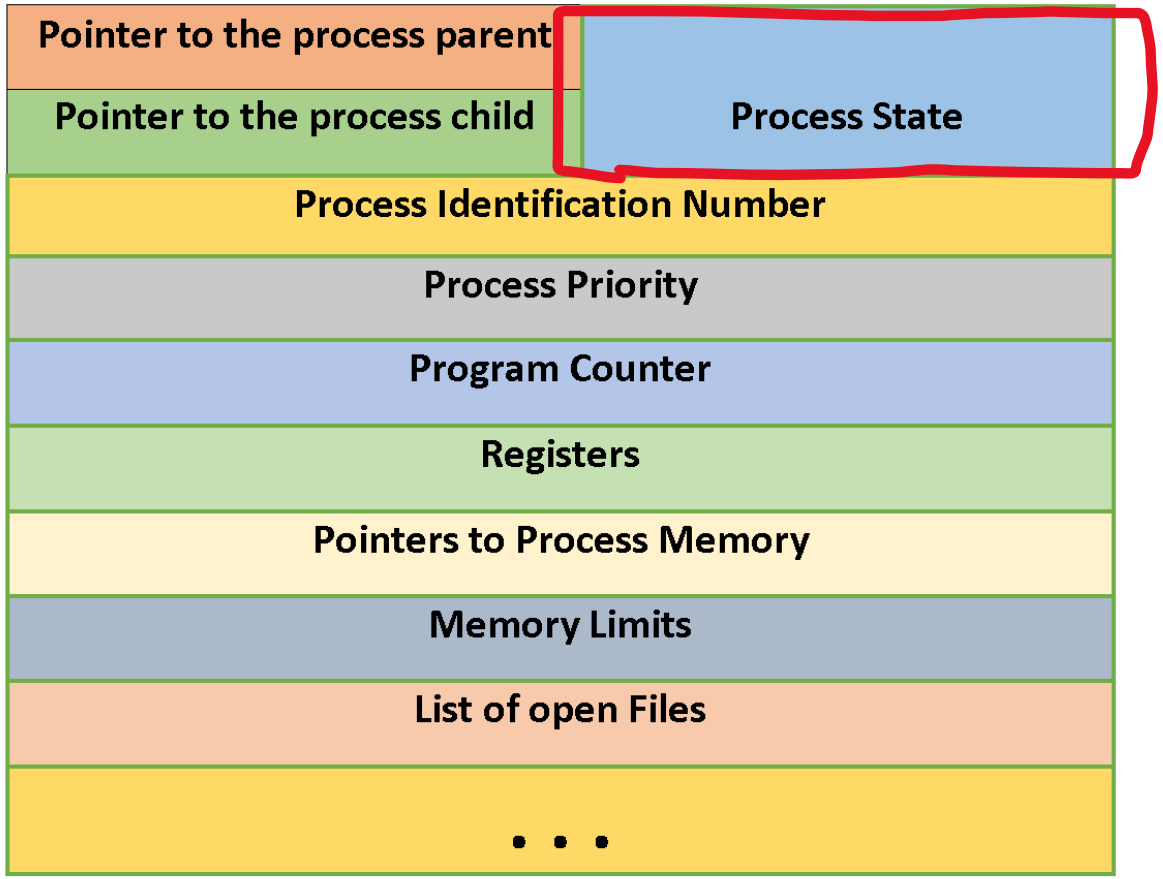
3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**
  - → Simply just a data structure that holds information
  - → The name of this varies by OS

A process goes through many **states**

- **Active (running)**
- **Blocked**
- **Waiting**
- **Suspended**

Example PCB:



Created by Notes Jam

A **process** is an instance of a <u>running</u> program on a computer

All processes have the following data while they are running:

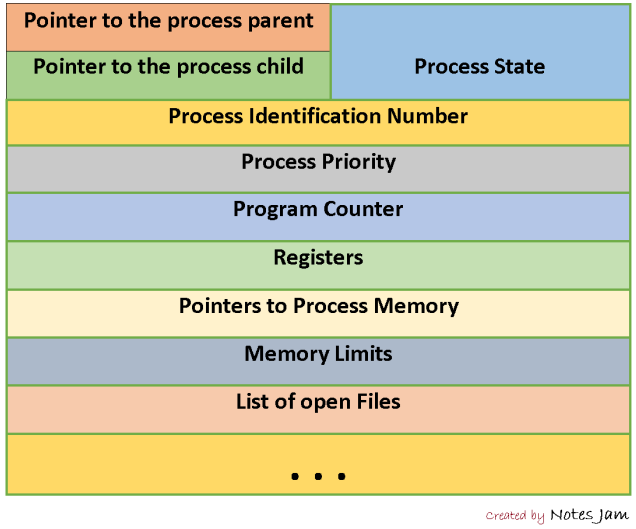
1. Executable Code

2. Associated Data

We will talk about what goes here shortly

3. Execution Context/Bookkeeping information

(info that the OS needs to handle the process)

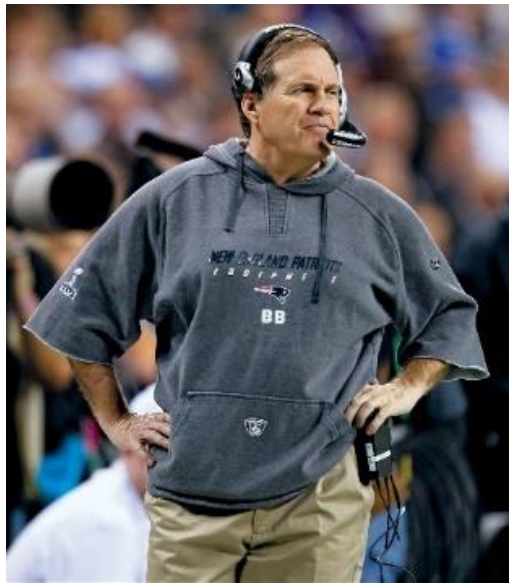


The jobs of an Operating System

1. Process Manager

"The Coach"

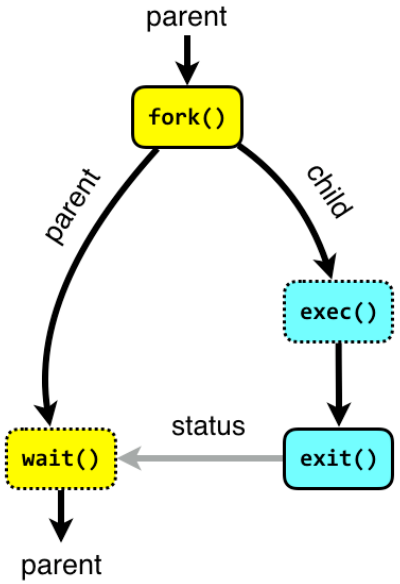
The OS manages many active processes all at once, and they must create processes, manage current process, and control which processes do what



`./hello_world` → Fork() and exec() → Program is now running as a **process**

A **process** is an instance of a <u>running</u> program on a computer

All processes have the following data while they are running:

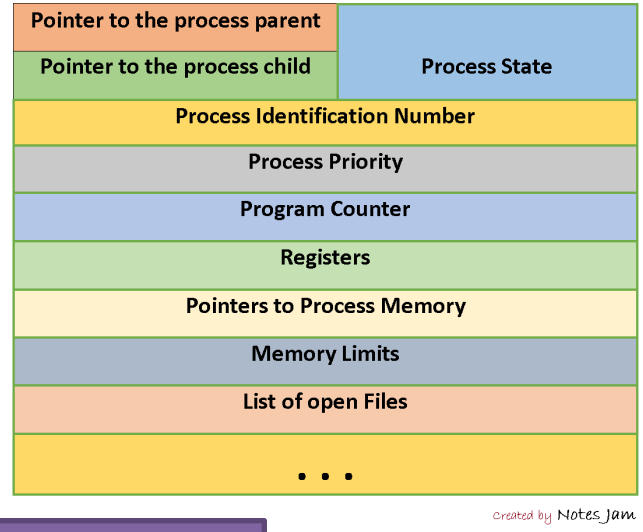


1. Executable Code

2. Associated Data

3. Execution Context/Bookkeeping information

(info that the OS needs to handle the process)



`./hello_world` → Fork() and exec() → Program is now running as a **process**

Demo time!

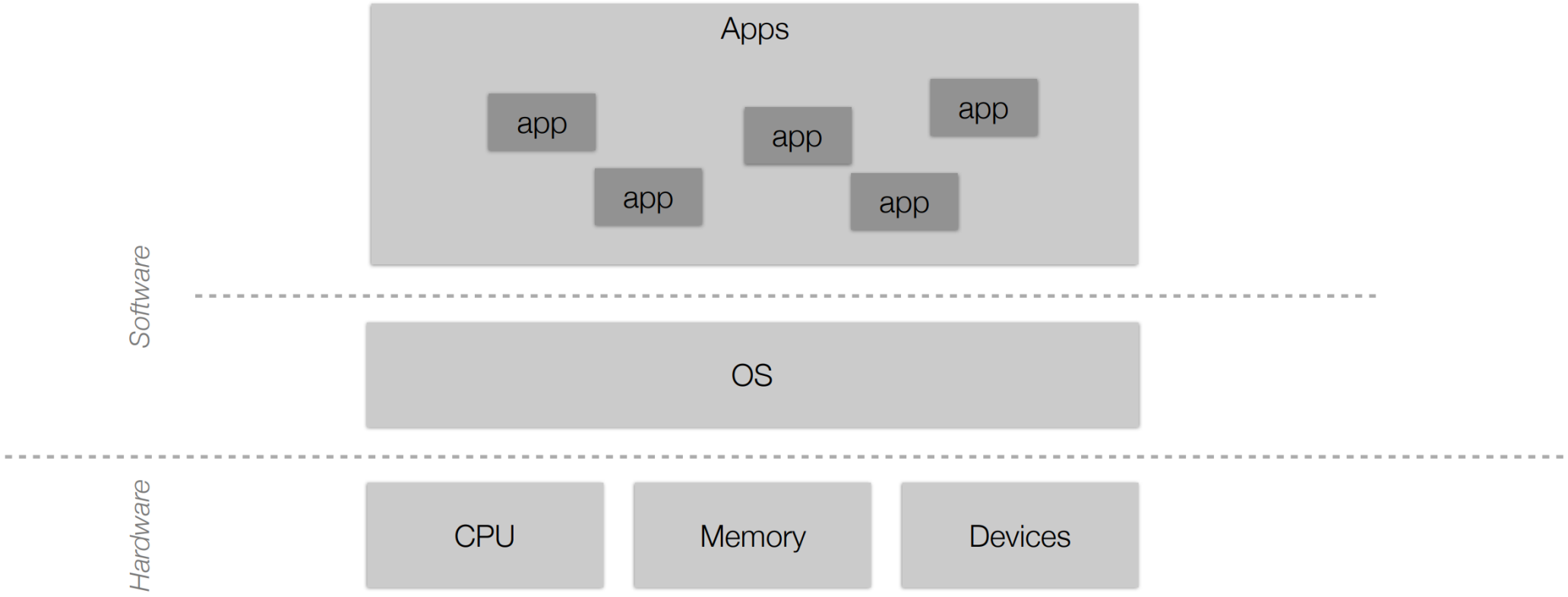
```c
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child
        printf("Hi, I'm the child. \n");
    }

    sleep(1);
    // we could wait() here
    printf("I'm the parent.);

    return 0;
}
```

```c
int main(void) {
    int pid;

    pid = fork();
    if (0 == pid) {
        // I'm the child

        char *name[2];
        name[0] = "./hello";
        name[1] = NULL;
        execve(name[0], name, NULL);

        _exit(0);
    }
    sleep(1);
    printf("I'm the parent. My child

    return 0;
}
```

The jobs of an Operating System

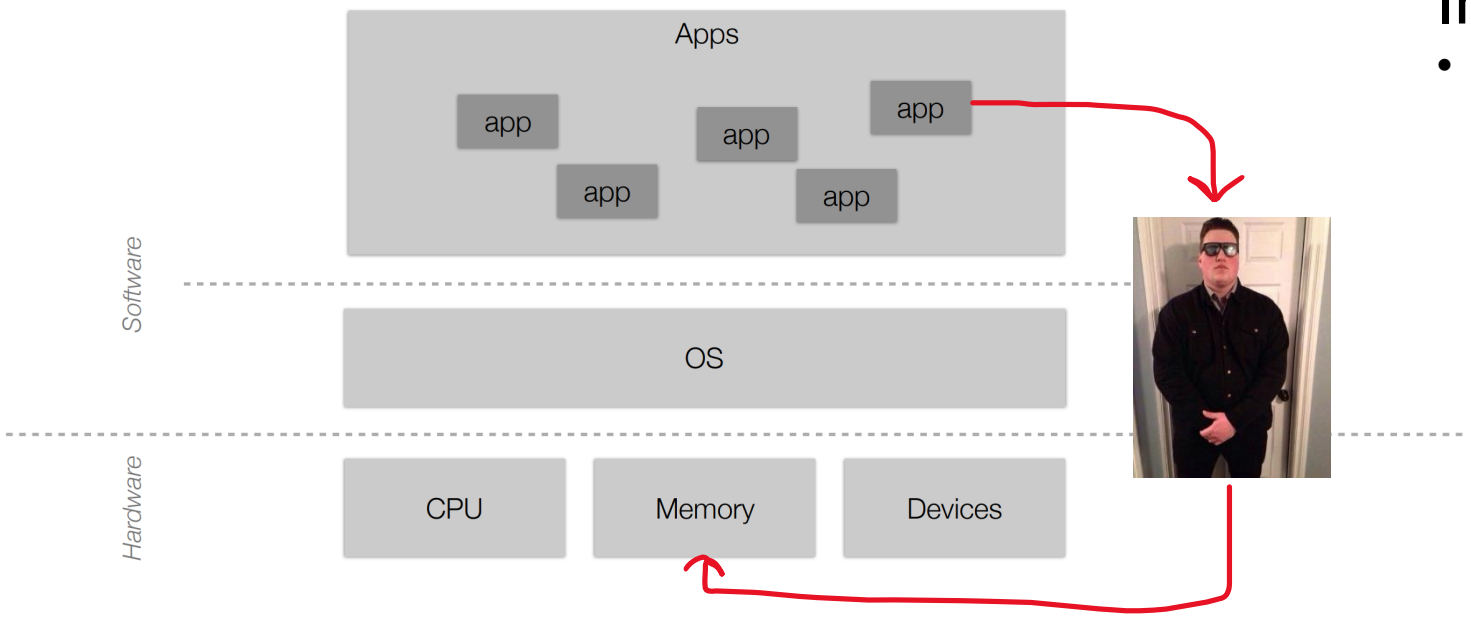
1. **Process Manager**
   "The Coach"
2. **Interface Manager**
   "The Bouncer"
3. **Memory Manager**
   "The Farmer"
4. **Traffic Manager**
   "The Judge"
5. **Illusion Manager**
   "The Illusionist"

Operating Systems Review



Operating Systems Review

Responsibilities of the OS?

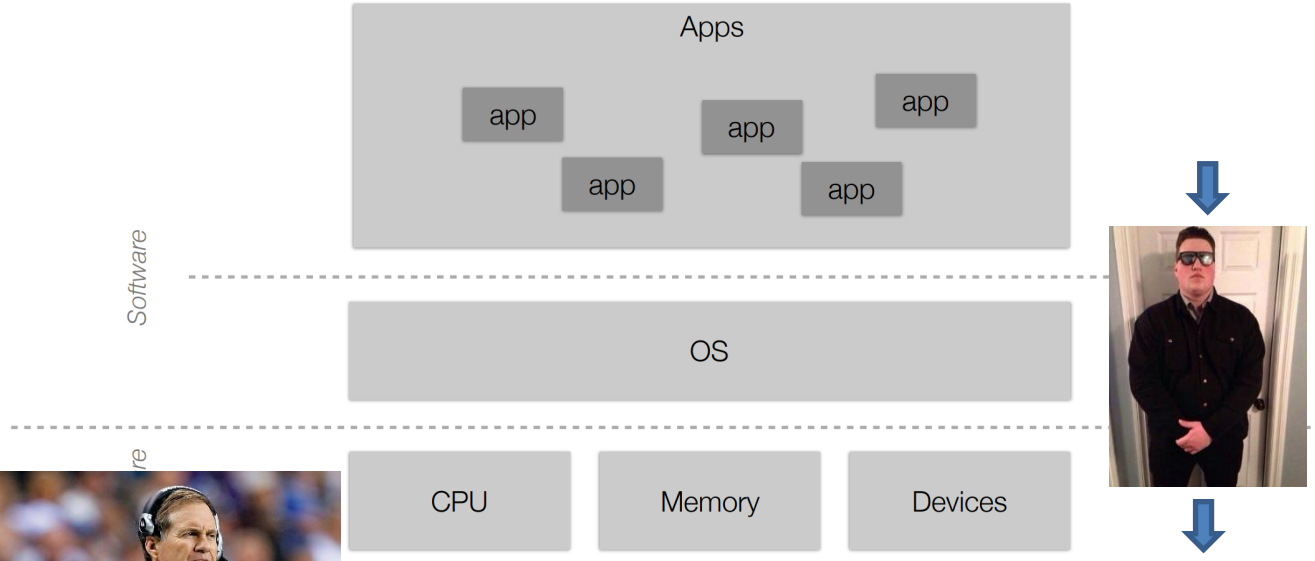


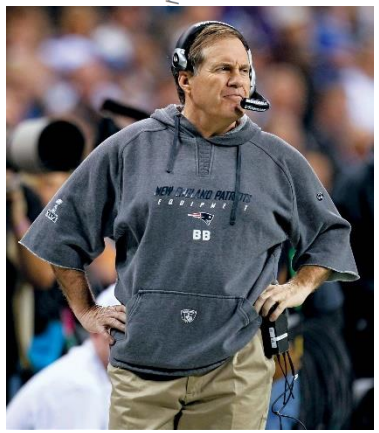
Interface Manager

- Manages communication between apps and hardware

Operating Systems Review

Responsibilities of the OS?





Interface Manager

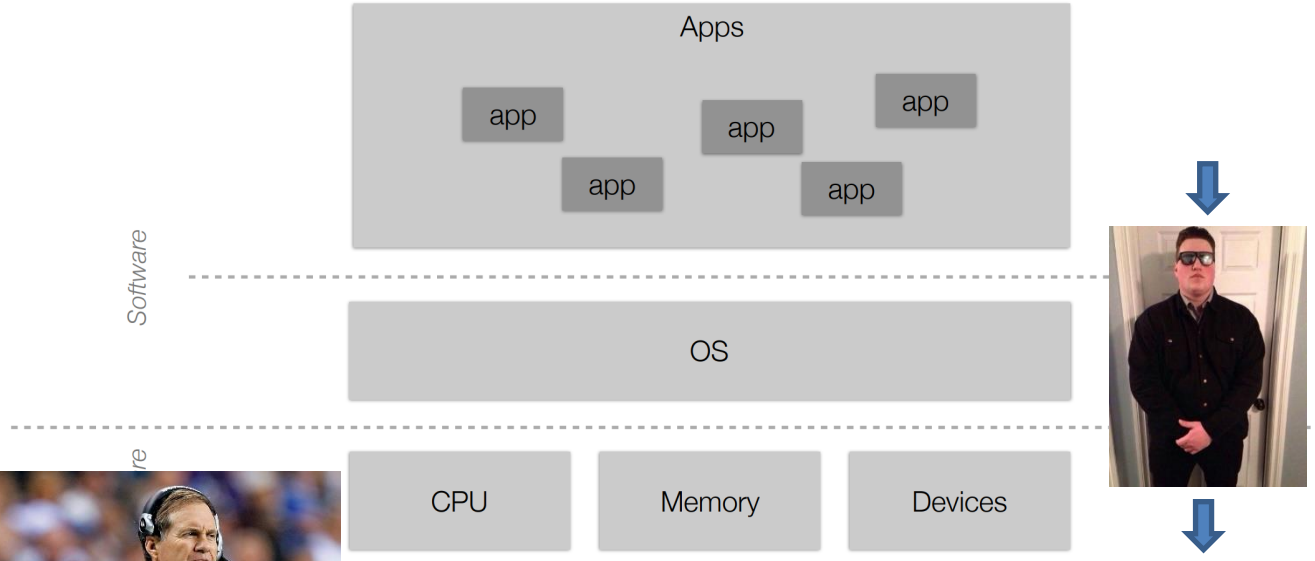
- Manages communication between apps and hardware

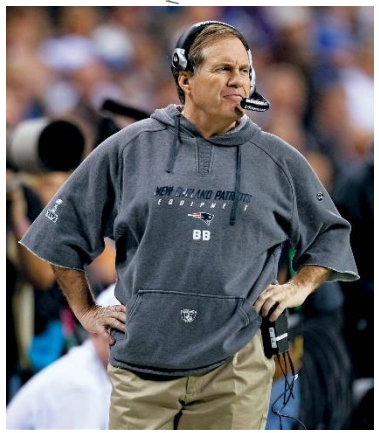
Process Manager

- Manages how processes are structured and how to handle many processes running at once

Operating Systems Review

Responsibilities of the OS?





Interface Manager

- Manages communication between apps and hardware

Process Manager

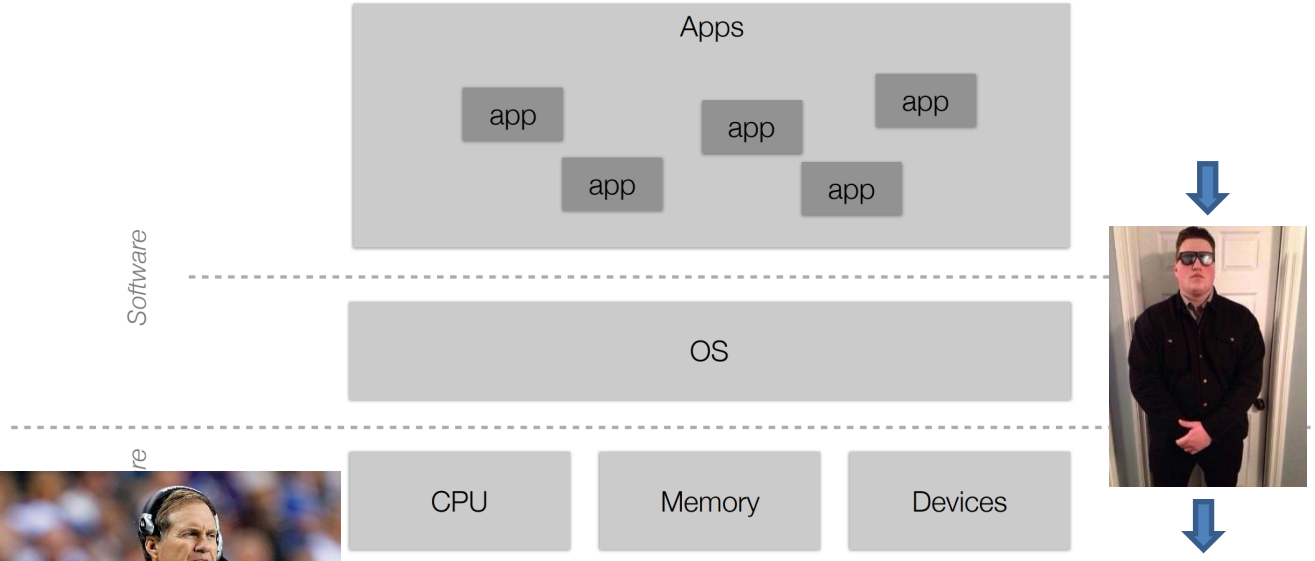
- Manages how processes are structured and how to handle many processes running at once

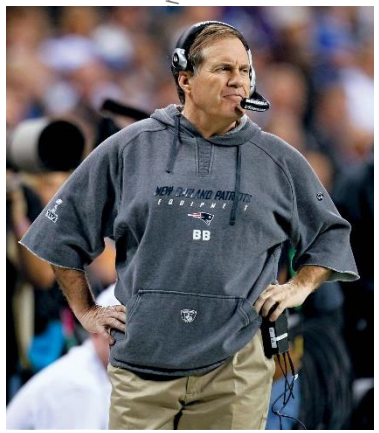
Traffic Manager

- Manages which programs should be executed by the CPU

Operating Systems Review

Responsibilities of the OS?





Interface Manager

- Manages communication between apps and hardware

Process Manager

- Manages how processes are structured and how to handle many processes running at once

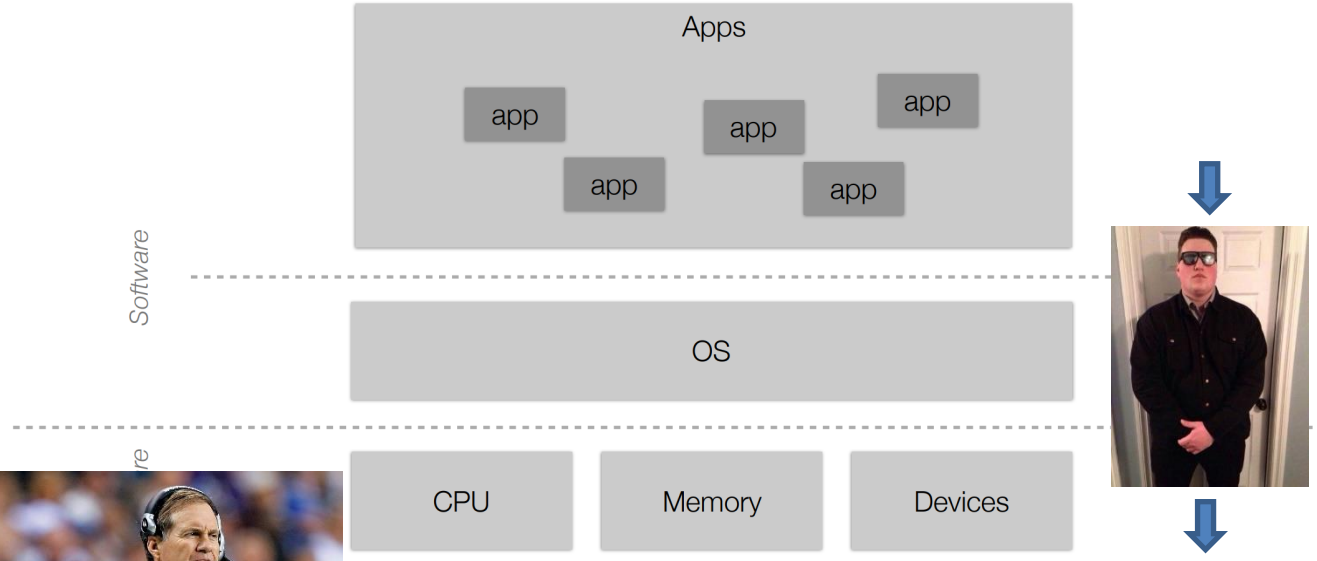
Traffic Manager

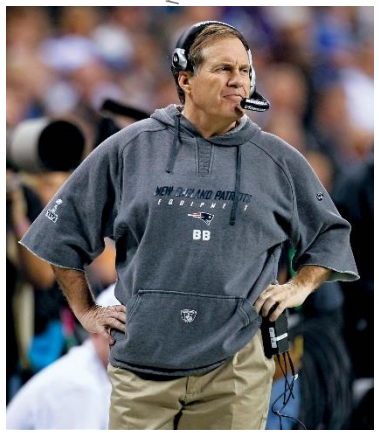
- Manages which programs should be executed by the CPU

Memory Manager

- Manages how physical memory is utilized

Operating Systems Review





Responsibilities of the OS?

Inte
- M nd
  ha

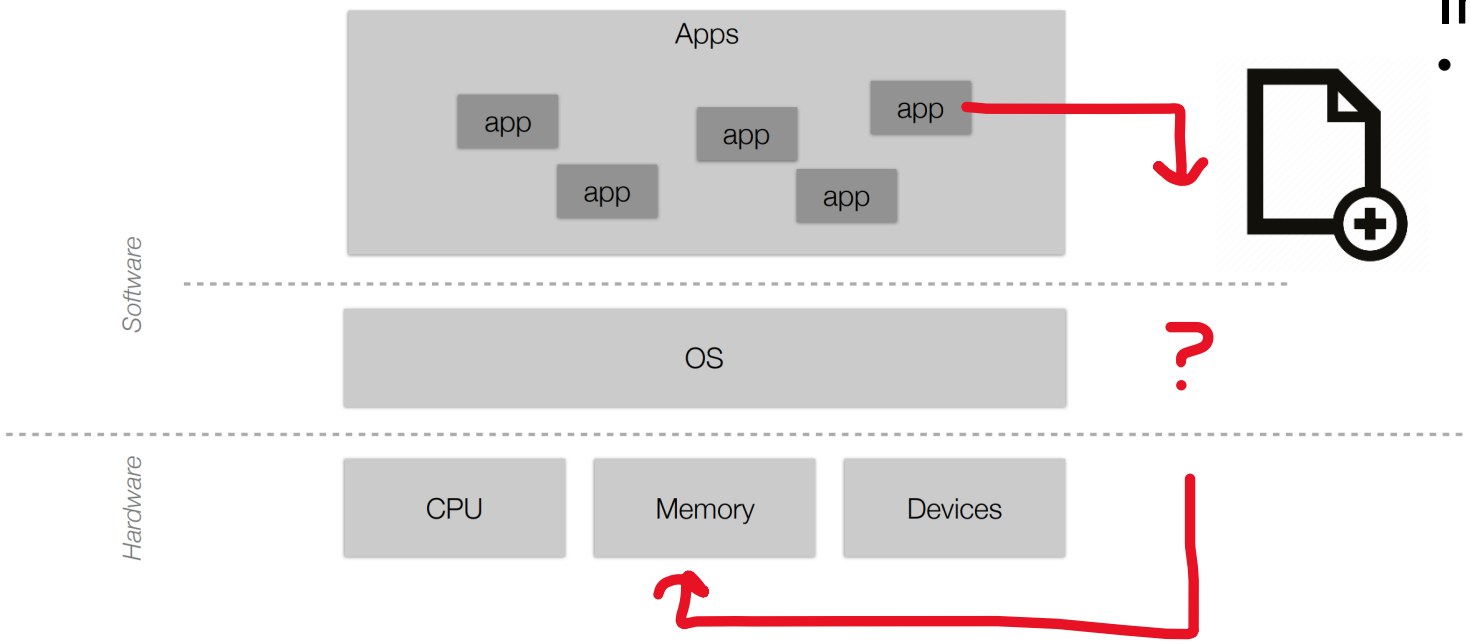
Pro
- M nd how
  to e

Traffi
- Man d by
  the

Mem
- Man

Operating Systems Review

Responsibilities of the OS?



Interface Manager

- Manages communication between apps and hardware

How does an application get access to a computer's resources?

Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

```
int main(void)
{
  printf("Hello, World!\n");

  return 0;
}
```

Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

```c
int main(void)
{
  printf("Hello, World!\n");

  return 0;
}
```

```c
int main(void)
{
  write(1, "Hello, World!\n", 14);

  return 0;
}
```

```c
int main(void)
{
  syscall(SYS_write, 1, "Hello, World!\n", 14);

  return 0;
}
```

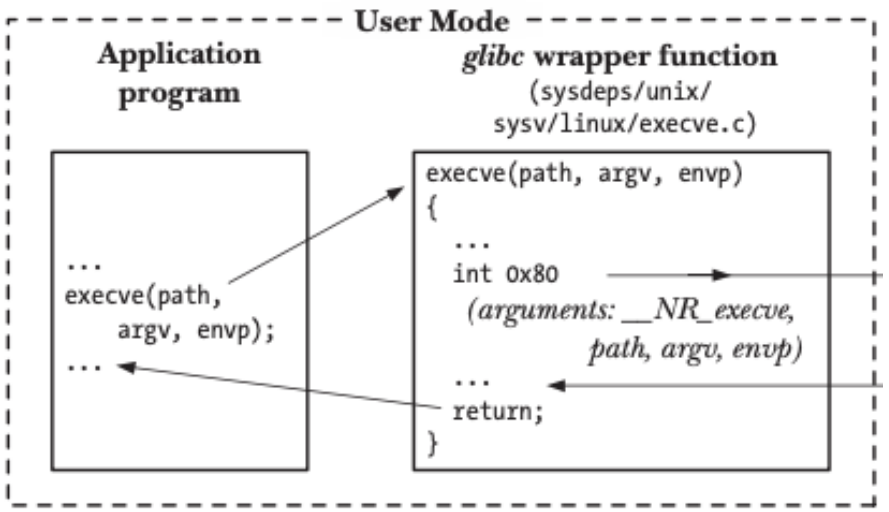
Number	Name	Description
1	exit	terminate process execution
2	fork	fork a child process
3	read	read data from a file or socket
4	write	write data to a file or socket
5	open	open a file or socket
6	close	close a file or socket
37	kill	send a kill signal
90	old_mmap	map memory
91	munmap	unmap memory
301	socket	create a socket
303	connect	connect a socket

Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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

int main()
{

        char *name[2];
        name[0] = "/bin/bc";
        name[1] = NULL;
        execve(name[0],name, NULL);
        return 0;
}
```

syscall



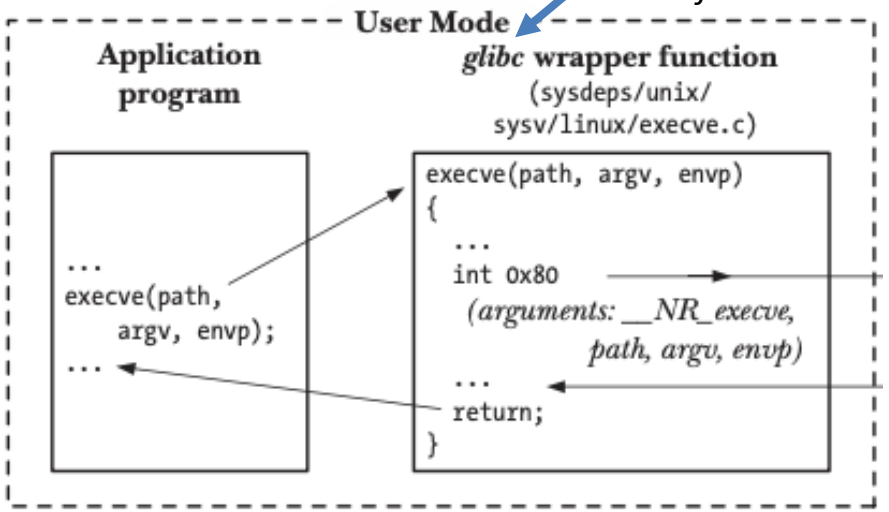
Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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

int main()
{

        char *name[2];
        name[0] = "/bin/bc";
        name[1] = NULL;
        execve(name[0],name, NULL);
        return 0;
}
```

syscall

Libraries handle the system calls for us

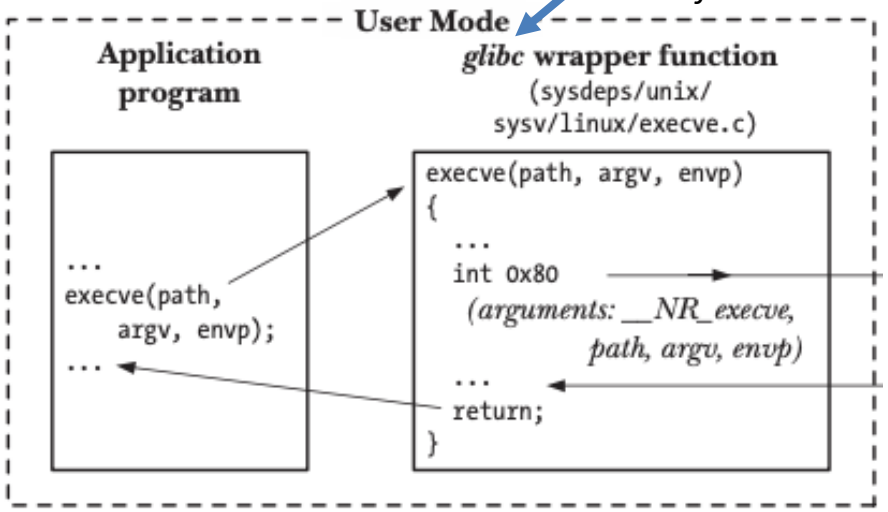


Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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

int main()
{

        char *name[2];
        name[0] = "/bin/bc";
        name[1] = NULL;
        execve(name[0],name, NULL);
        return 0;
} syscall
```

Libraries handle the system calls for us



EAX

EBX

ECX

EDX

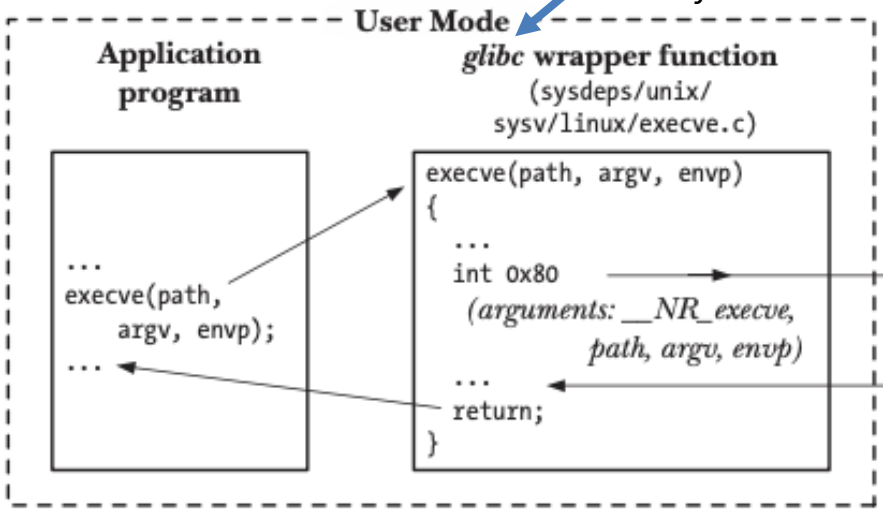
The operating system have hundreds of different syscalls, and different syscalls have different parameters, we need a way to distinguish them

Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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

int main()
{

        char *name[2];
        name[0] = "/bin/bc";
        name[1] = NULL;
        execve(name[0],name, NULL);
        return 0;
} syscall
```

Libraries handle the system calls for us



EAX	System Call Number	
EBX	Address of "`/bin/bc`"	
ECX	0 or 1	Environment variables
EDX	INT 0x80	send trap to kernel and invoke the syscall

The operating system have hundreds of different syscalls, and different syscalls have different parameters, we need a way to distinguish them

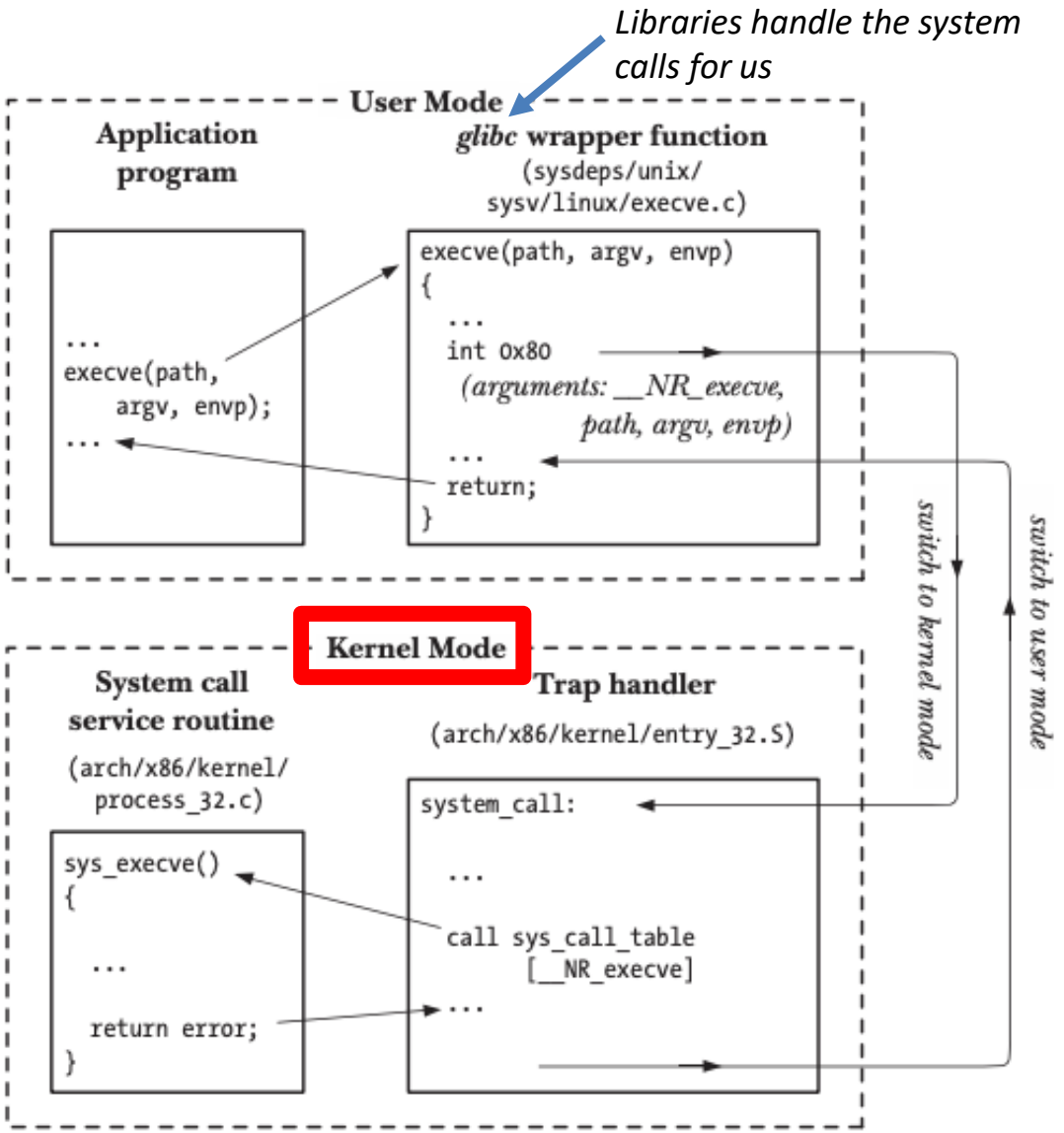
The OS will look at the values at certain registers!

Syscalls

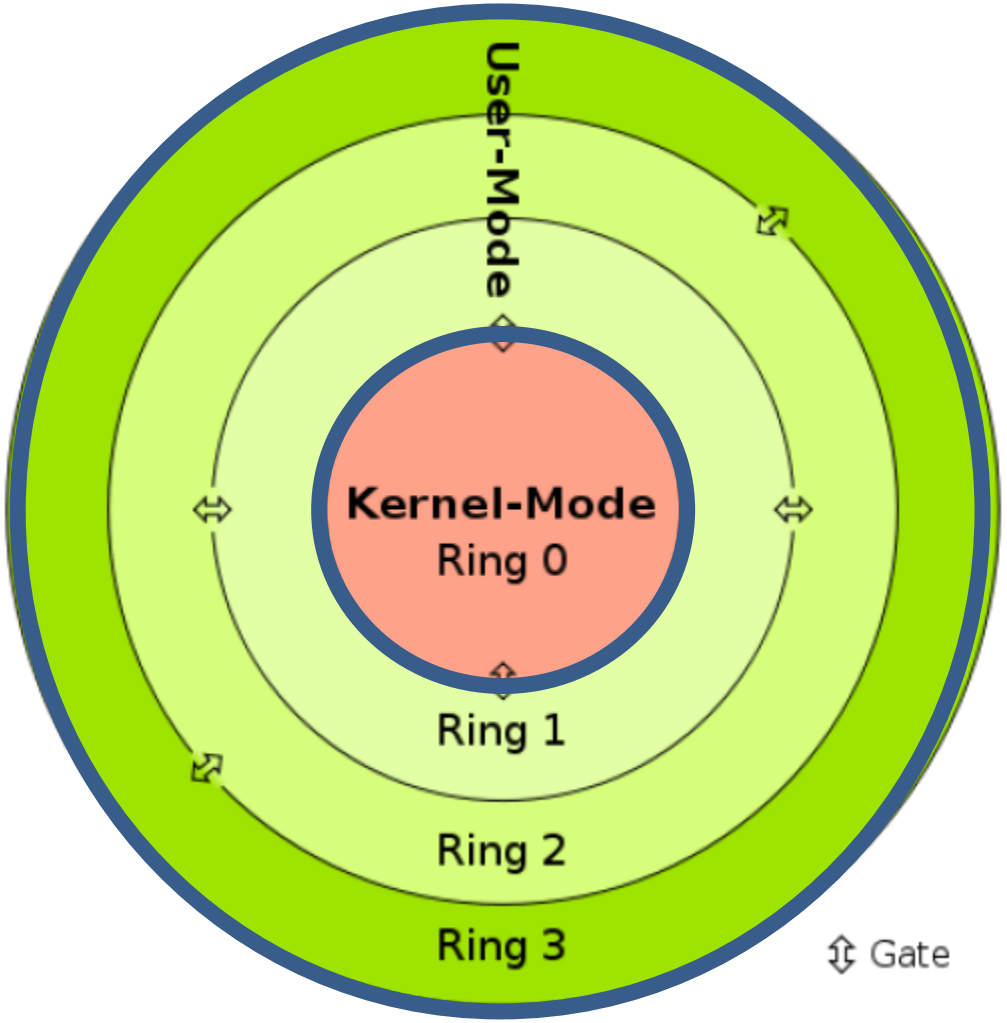
Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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

int main()
{

        char *name[2];
        name[0] = "/bin/bc";
        name[1] = NULL;
        execve(name[0],name, NULL);
        return 0;
} syscall
```

EAX	System Call Number	
EBX	Address of "`/bin/bc`"	
ECX	0 or 1	Environment variables
EDX	INT 0x80	send trap to kernel and invoke the syscall



Syscalls

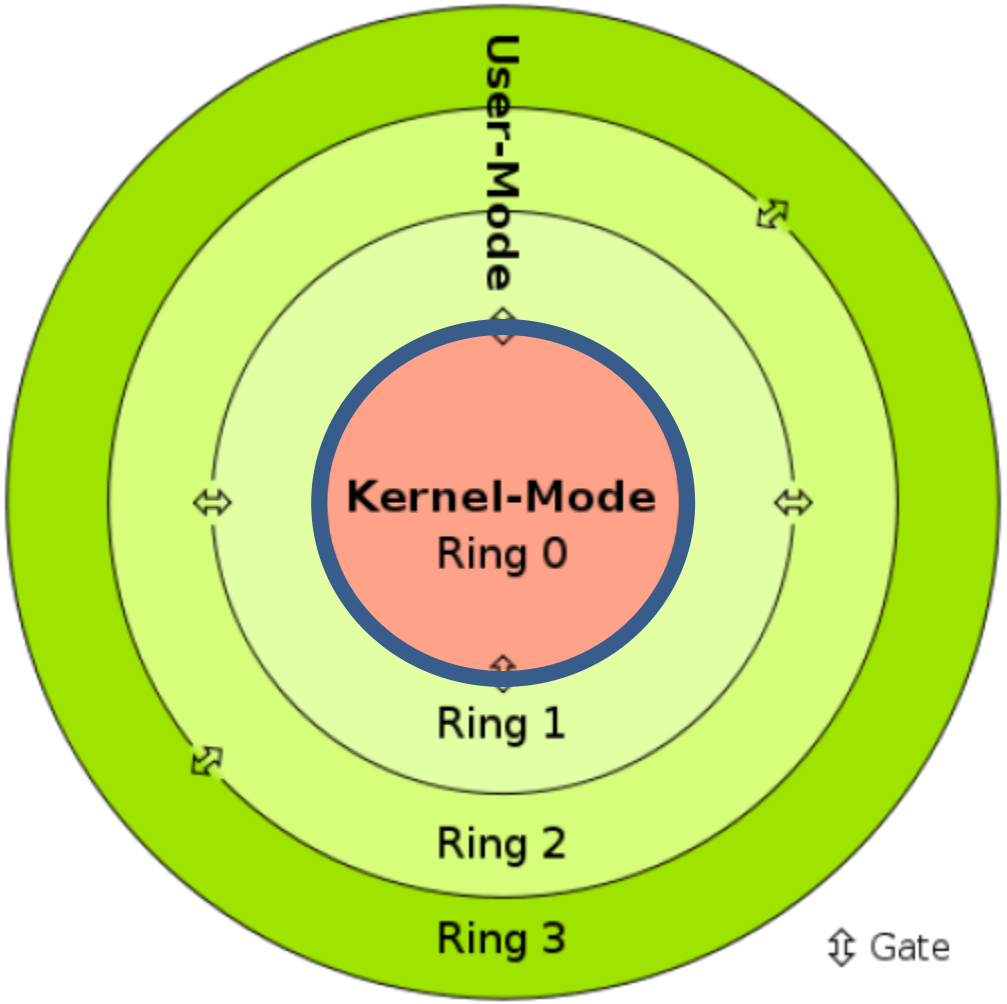


All applications run in user mode.

The code has no ability to directly access hardware

Code running in user mode must use API/syscalls to access hardware and memory

Syscalls



All applications run in user mode.

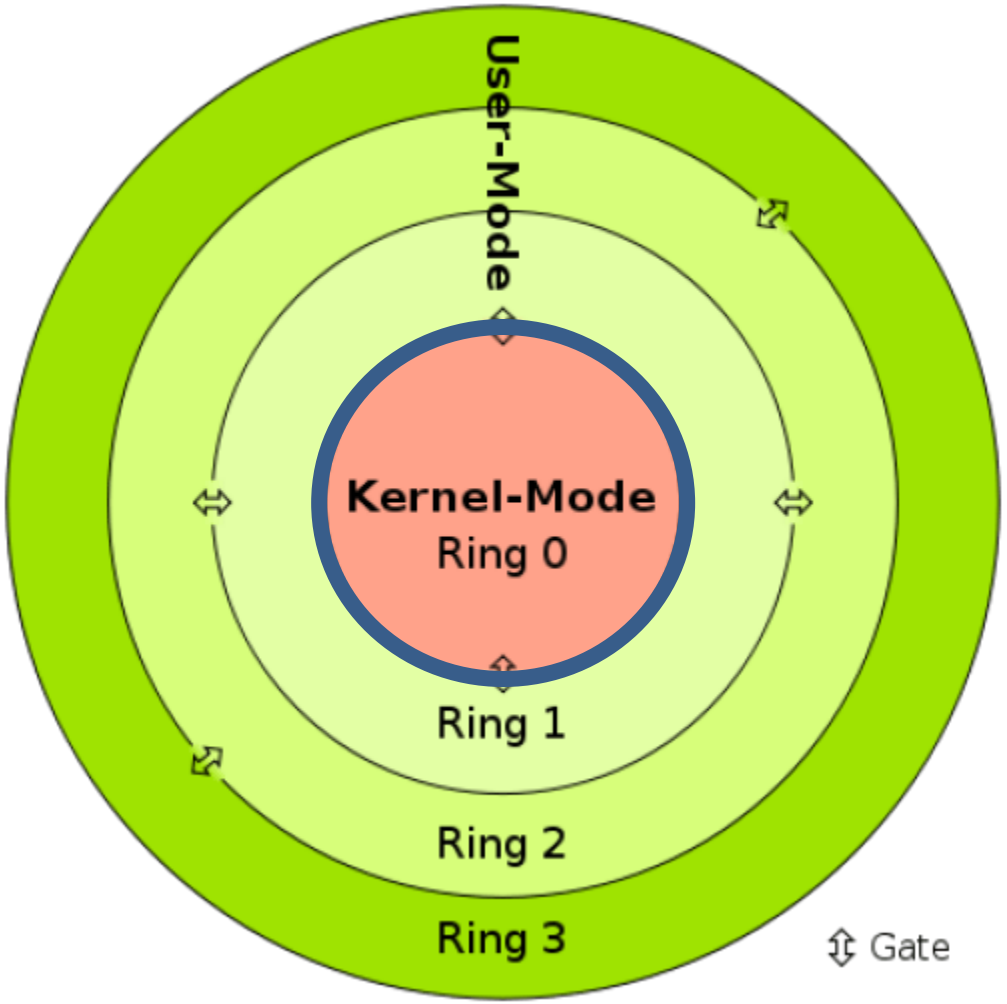
The code has no ability to directly access hardware

Code running in user mode must use API/syscalls to access hardware and memory

Code running in kernel-mode has complete, unrestricted access to computer resources

Reserved for the lowest-level trusted functions of the operating system

Syscalls



The collective functionality and services of the OS that manages the computer and its resources is called the **kernel**

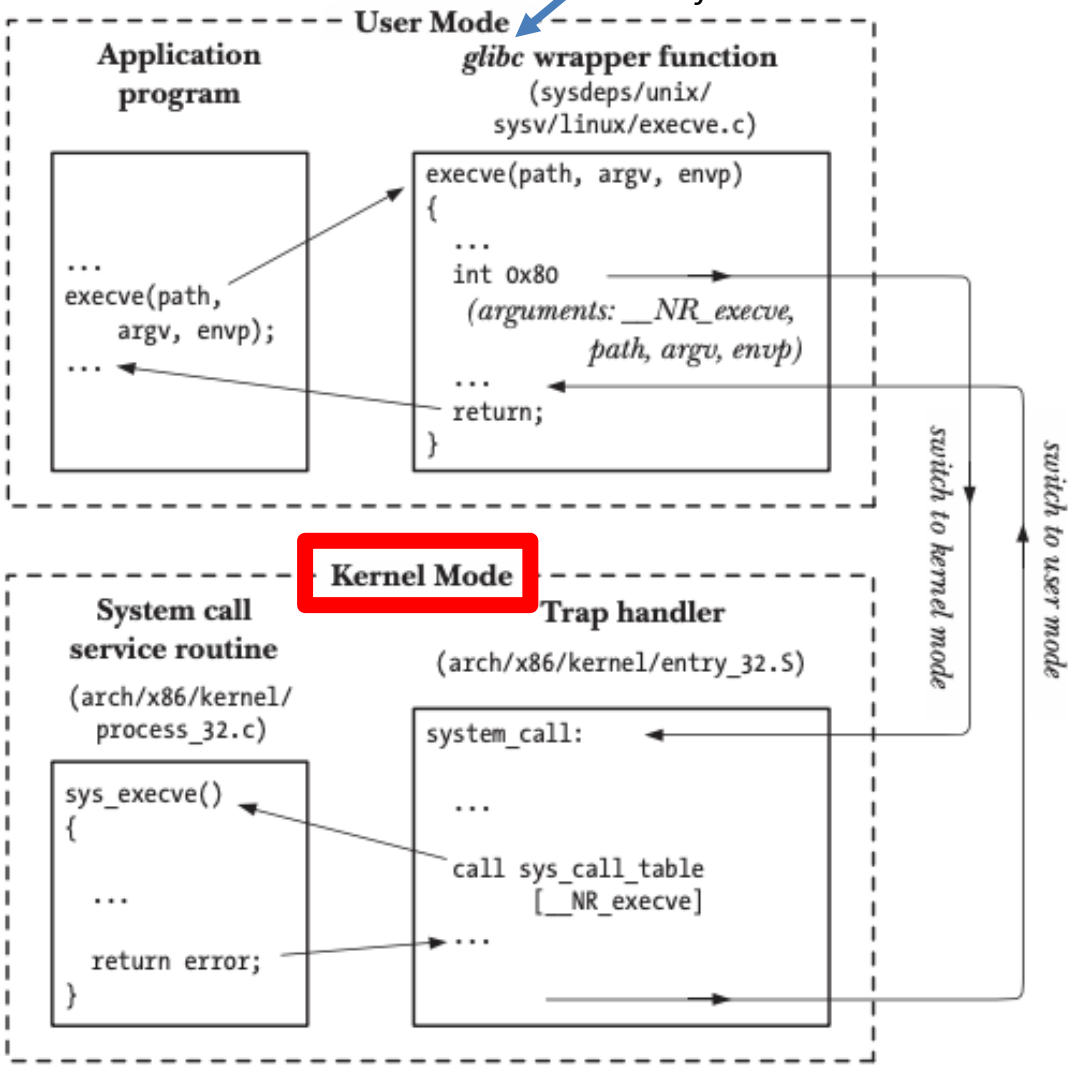
Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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

int main()
{

        char *name[2];
        name[0] = "/bin/bc";
        name[1] = NULL;
        execve(name[0],name, NULL);
        return 0;
} syscall
```

EAX	System Call Number	
EBX	Address of "`/bin/bc`"	
ECX	0 or 1	Environment variables
EDX	INT 0x80	send trap to kernel and invoke the syscall

Libraries handle the system calls for us



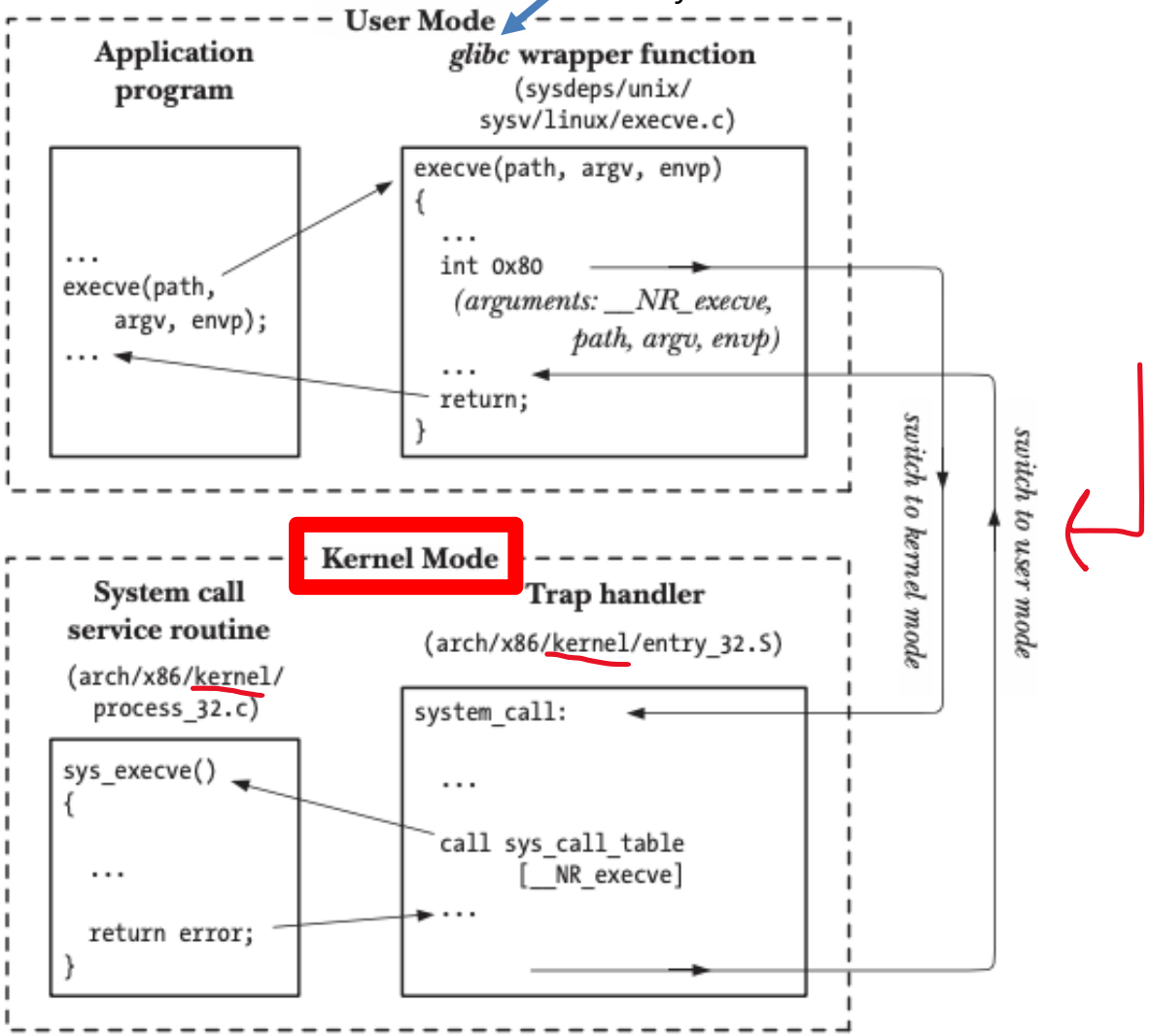
Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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

int main()
{

        char *name[2];
        name[0] = "/bin/bc";
        name[1] = NULL;
        execve(name[0],name, NULL);
        return 0;
} syscall
```

EAX	System Call Number	
EBX	Address of "`/bin/bc`"	
ECX	0 or 1	Environment variables
EDX	INT 0x80	send trap to kernel and invoke the syscall

Libraries handle the system calls for us



Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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	-	-	-	-	-

EDX

INT 0x80

send trap to kernel and invoke the syscall

Syscalls

Applications evoke operating system defined functions, or **system calls** (**syscalls**), to access computing resources

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									
5									
6									
7									
8									
9									
10	unlink	man/ cs/	0x0a	const char *pathname	-	-	-	-	-
11	execve	man/ cs/	0x0b	const char *const *filename	const char *const *argv	const char *const *envp	-	-	-
12	chdir	man/ cs/	0x0c	const char *filename	-	-	-	-	-

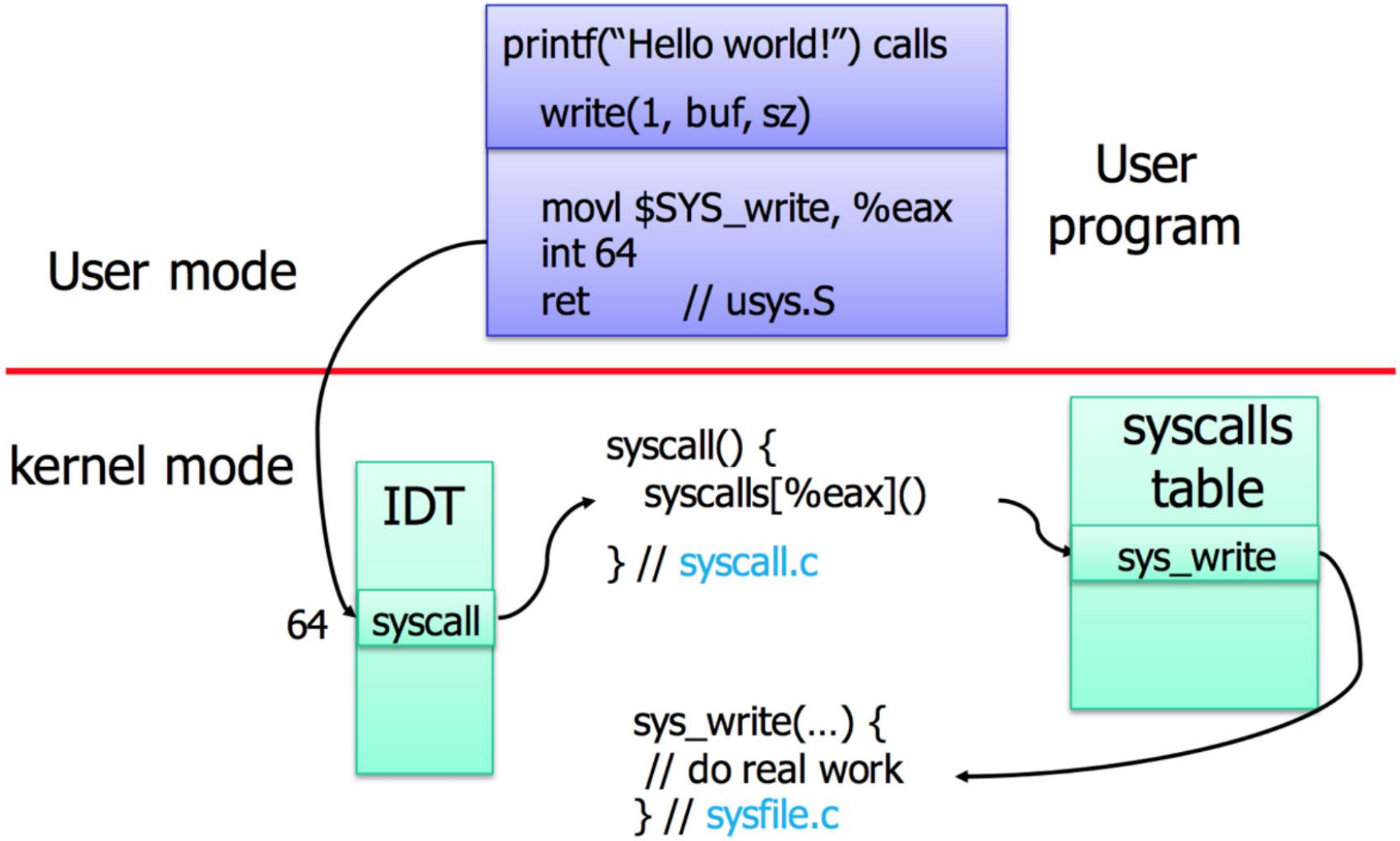
https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

EDX

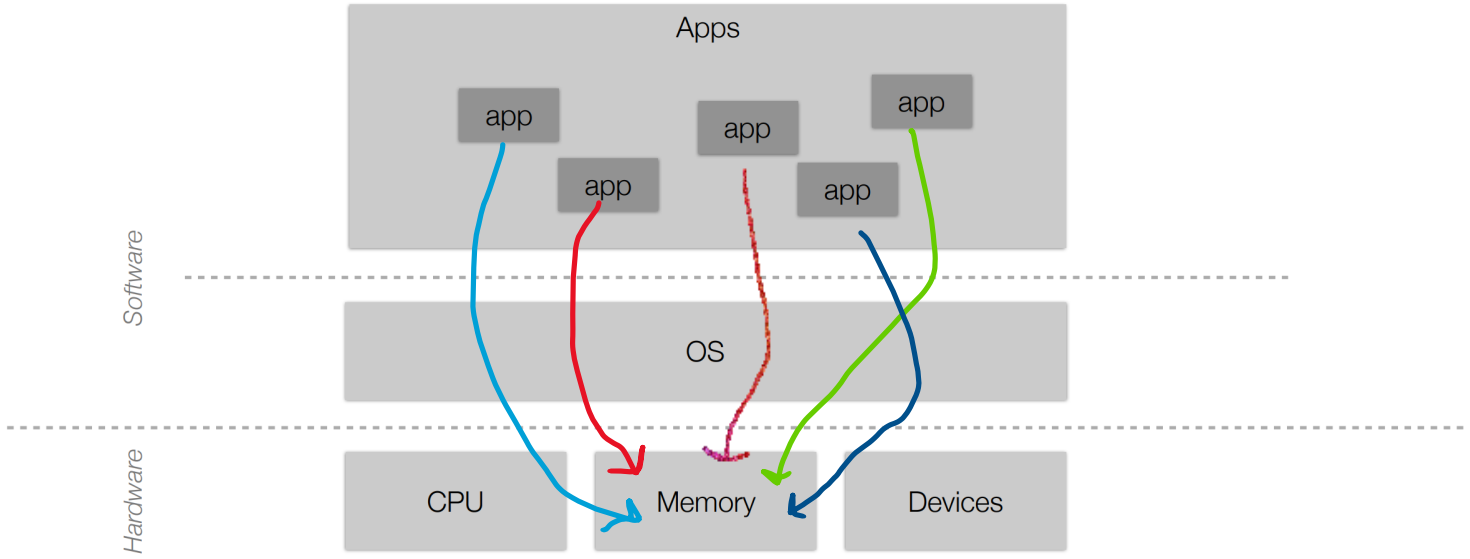
INT 0x80

send trap to kernel and invoke the syscall

Syscalls


printf("Hello world!") calls
write(1, buf, sz)
movl $SYS_write, %eax
int 64
ret // usys.S
User program
User mode
kernel mode
IDT
64
syscall
syscall() {
syscalls[%eax]()
} // syscall.c
syscalls table
sys_write
sys_write(...) {
// do real work
} // sysfile.c

Applications Layout in Memory

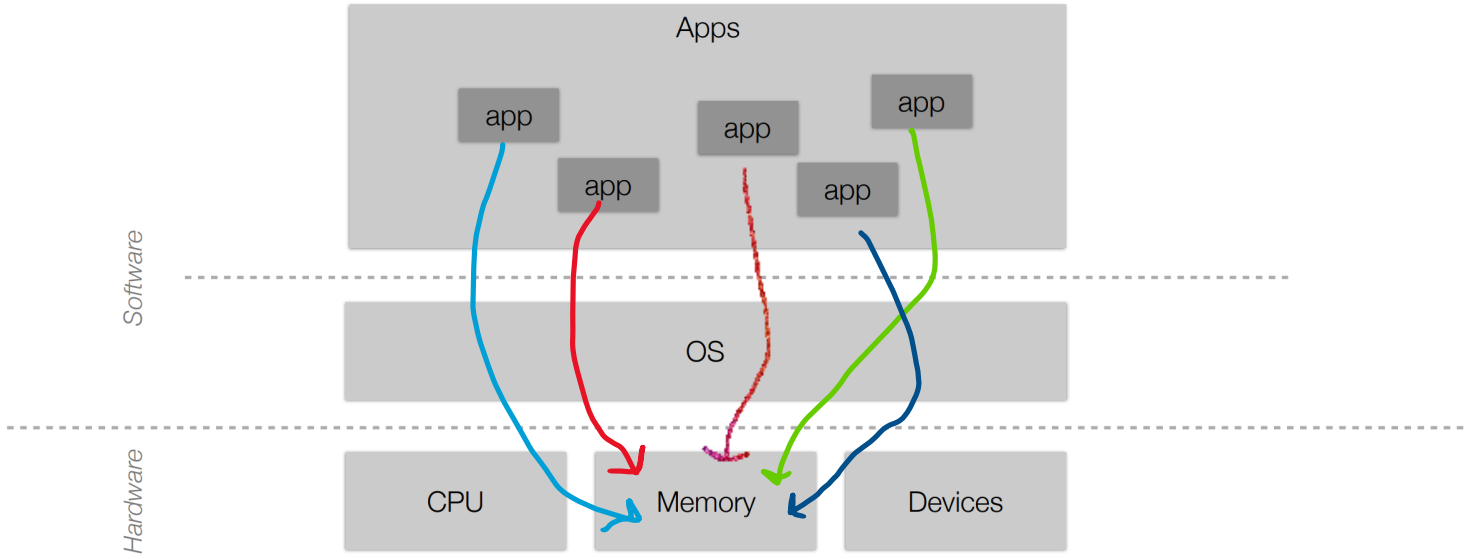


Process Manager

- Manages how processes are structured and how to handle many processes running at once

How does a **program** get loaded into memory?

Applications Layout in Memory



Process Manager

- Manages how processes are structured and how to handle many processes running at once

How does a **program** get loaded into memory?

An active program running on a computer is called a **process**

Applications Layout in Memory

Process Manager

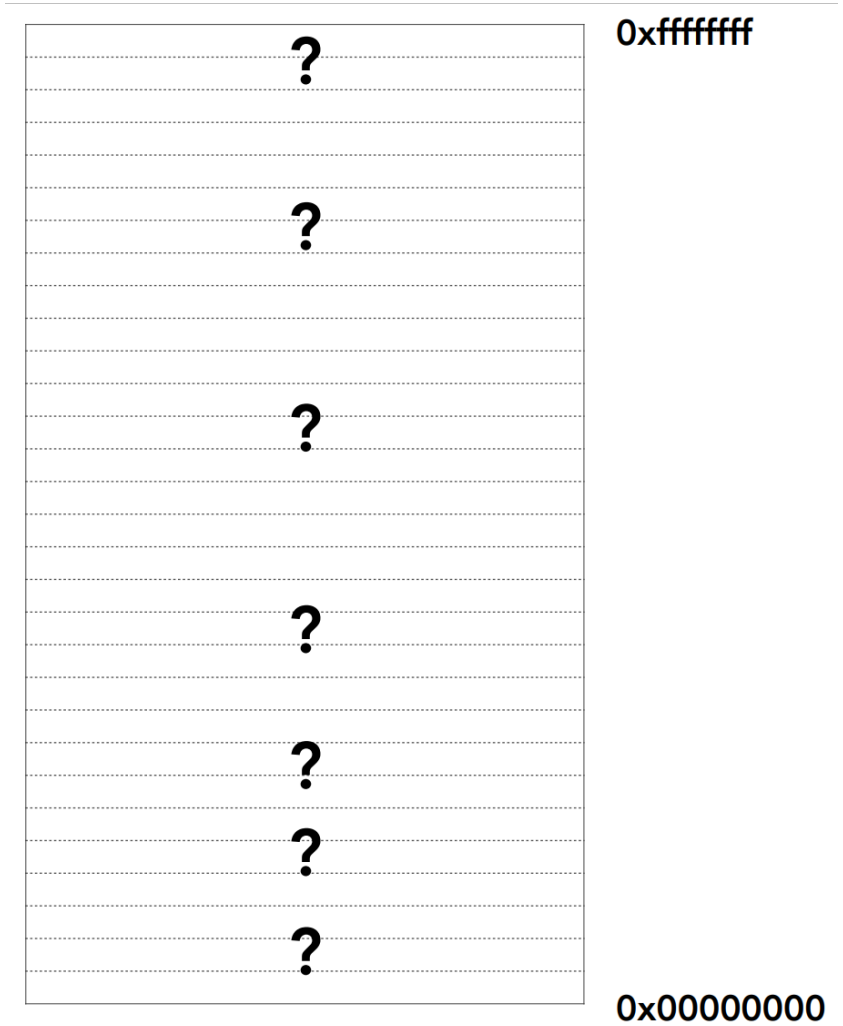
- Manages how processes are structured and how to handle many processes running at once

What does this look like?

1. **Executable Code**

2. **Associated Data**

3. Execution Context/Bookkeeping information

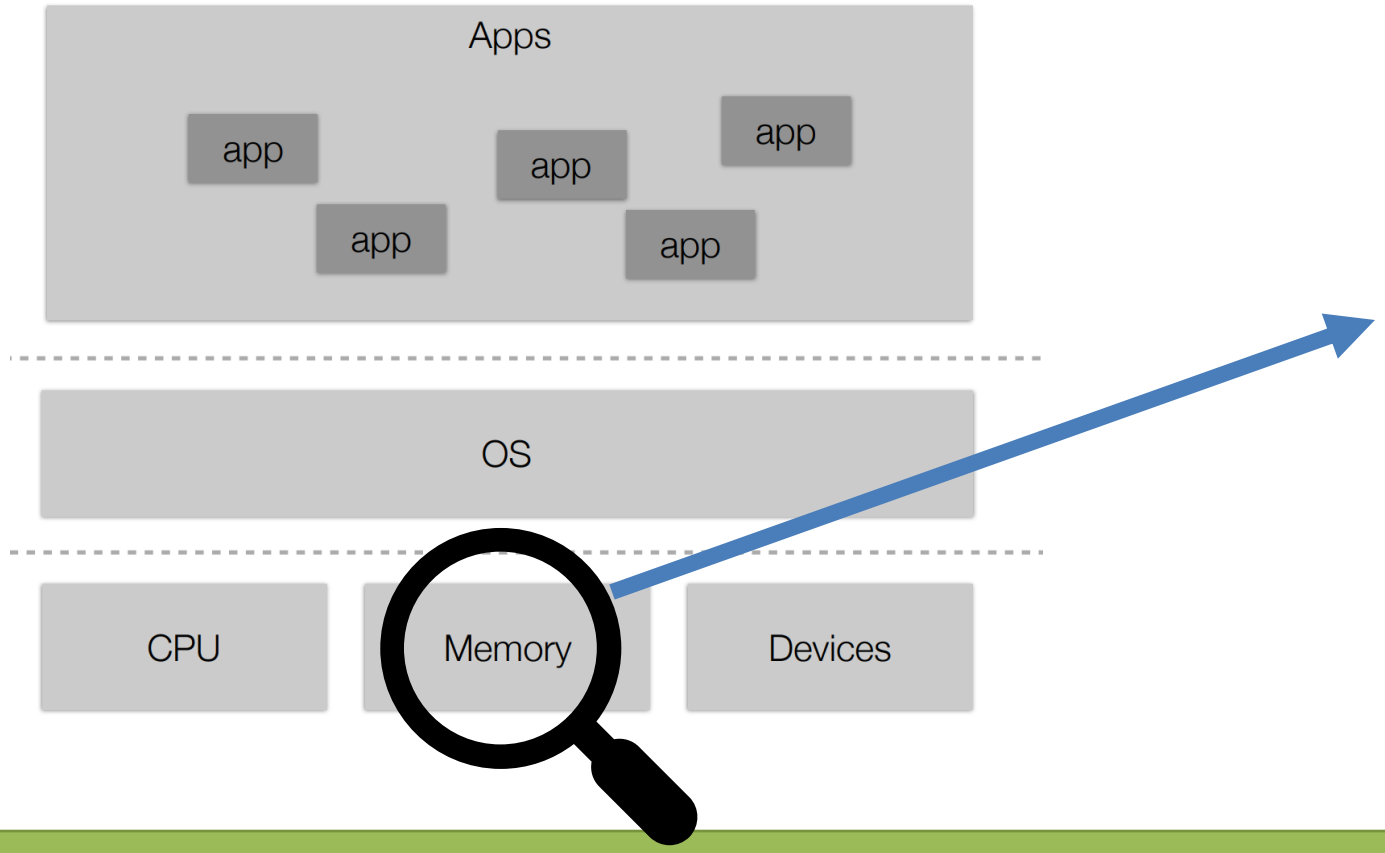


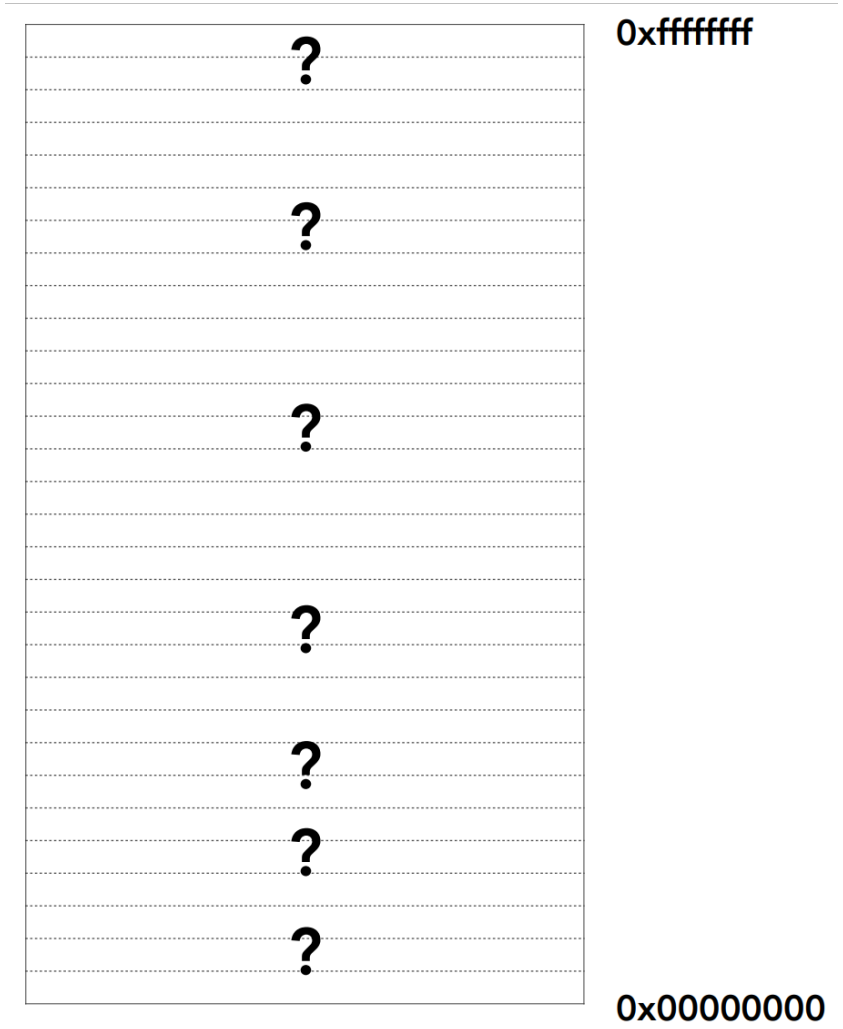
Applications Layout in Memory

Process Manager

- Manages how processes are structured and how to handle many processes running at once

What does a program look like in memory?

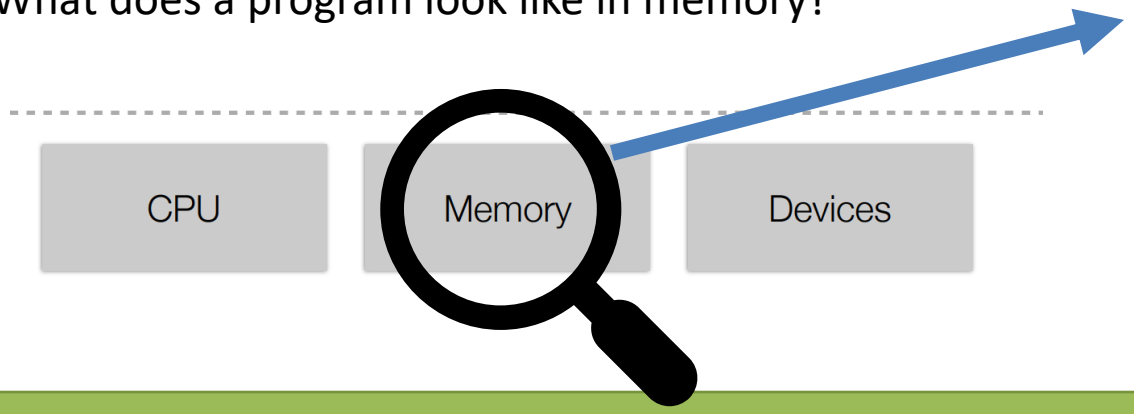




Applications Layout in Memory

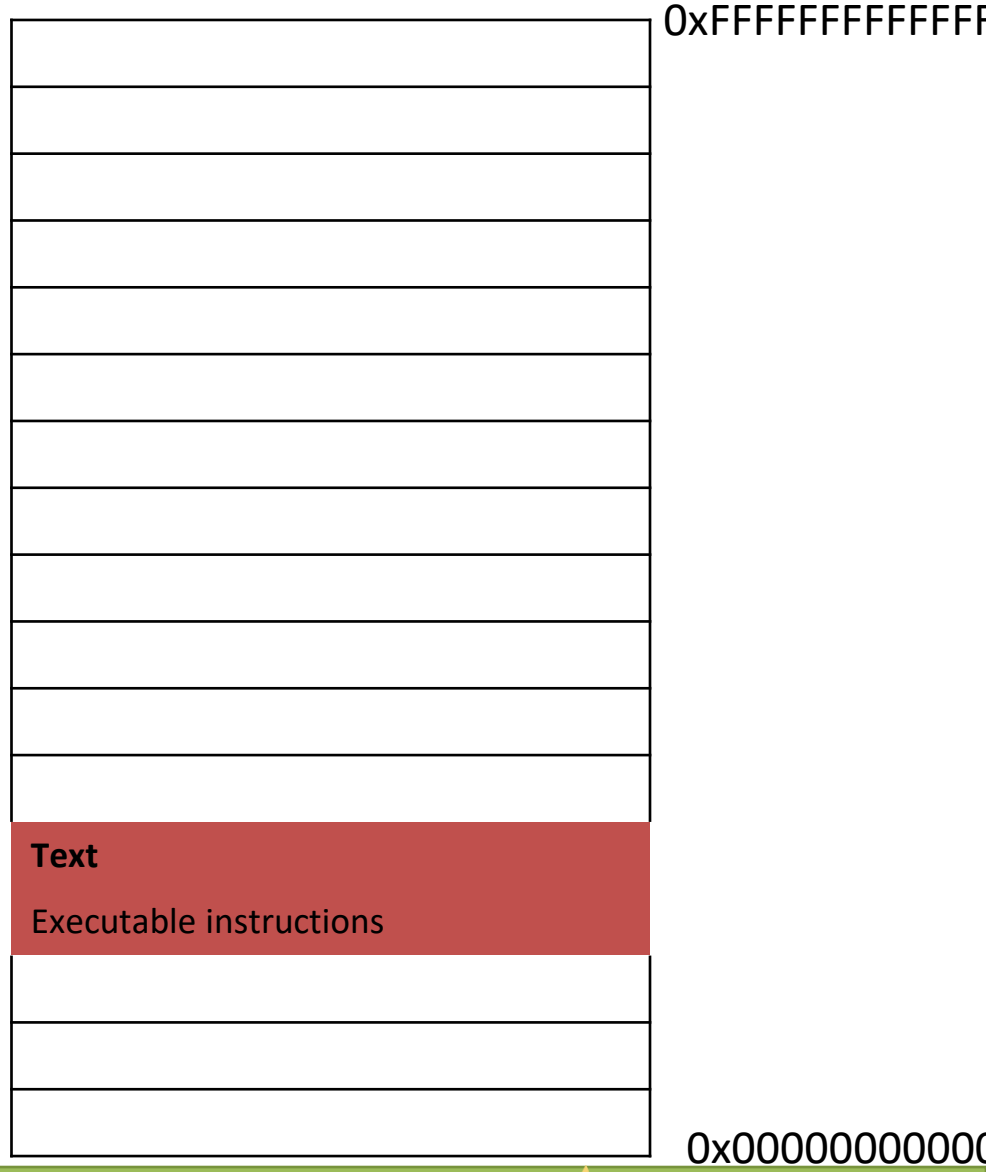
Text Segment- binary executable instructions for the process

What does a program look like in memory?



Process Manager

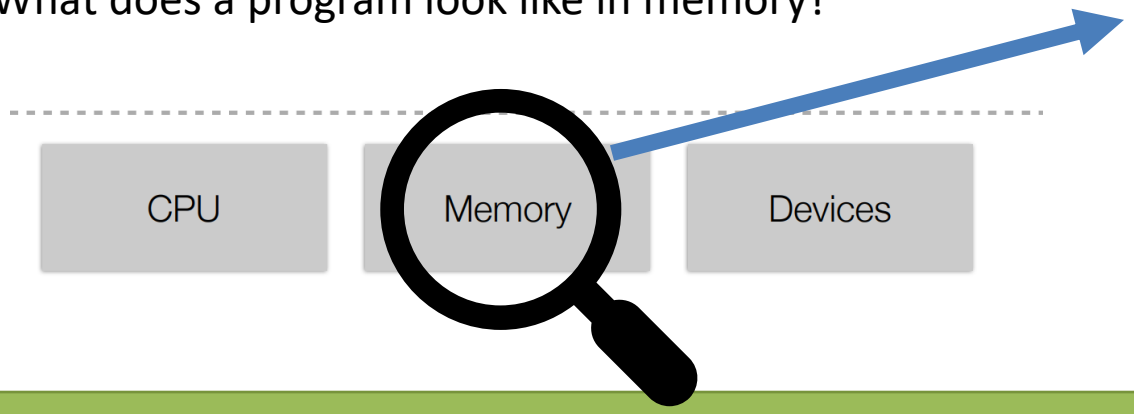
- Manages how processes are structured and how to handle many processes running at once



Applications Layout in Memory

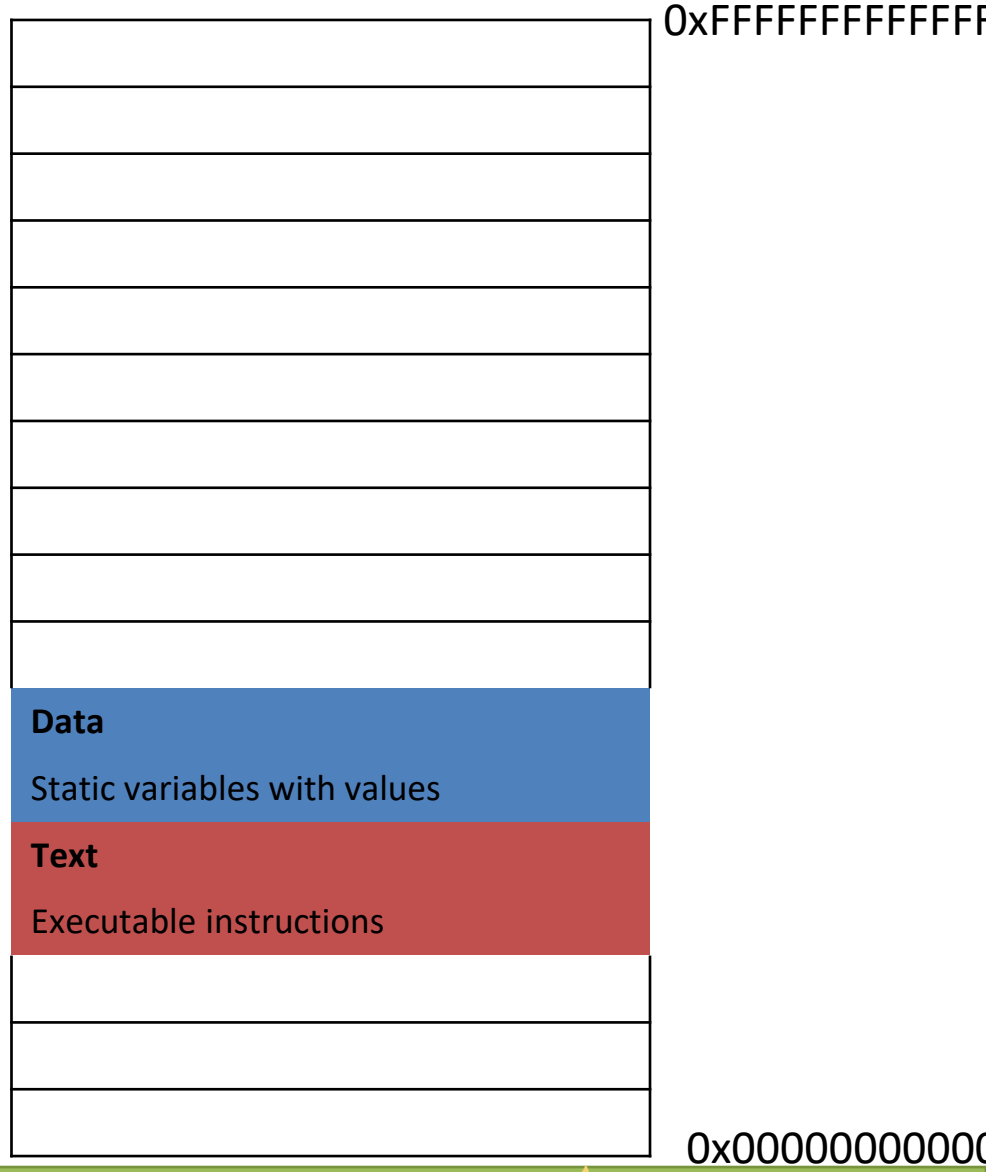
Data Segment- Static variables initialized by the programmer

What does a program look like in memory?



Process Manager

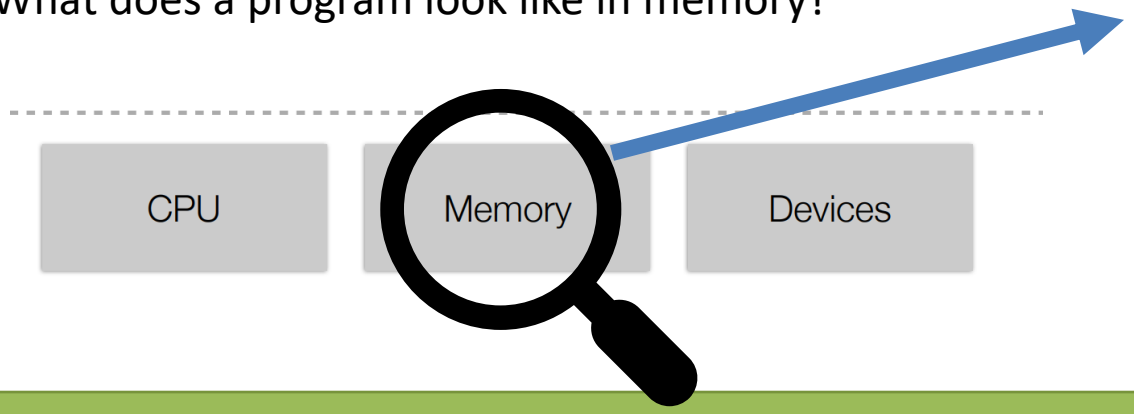
- Manages how processes are structured and how to handle many processes running at once



Applications Layout in Memory

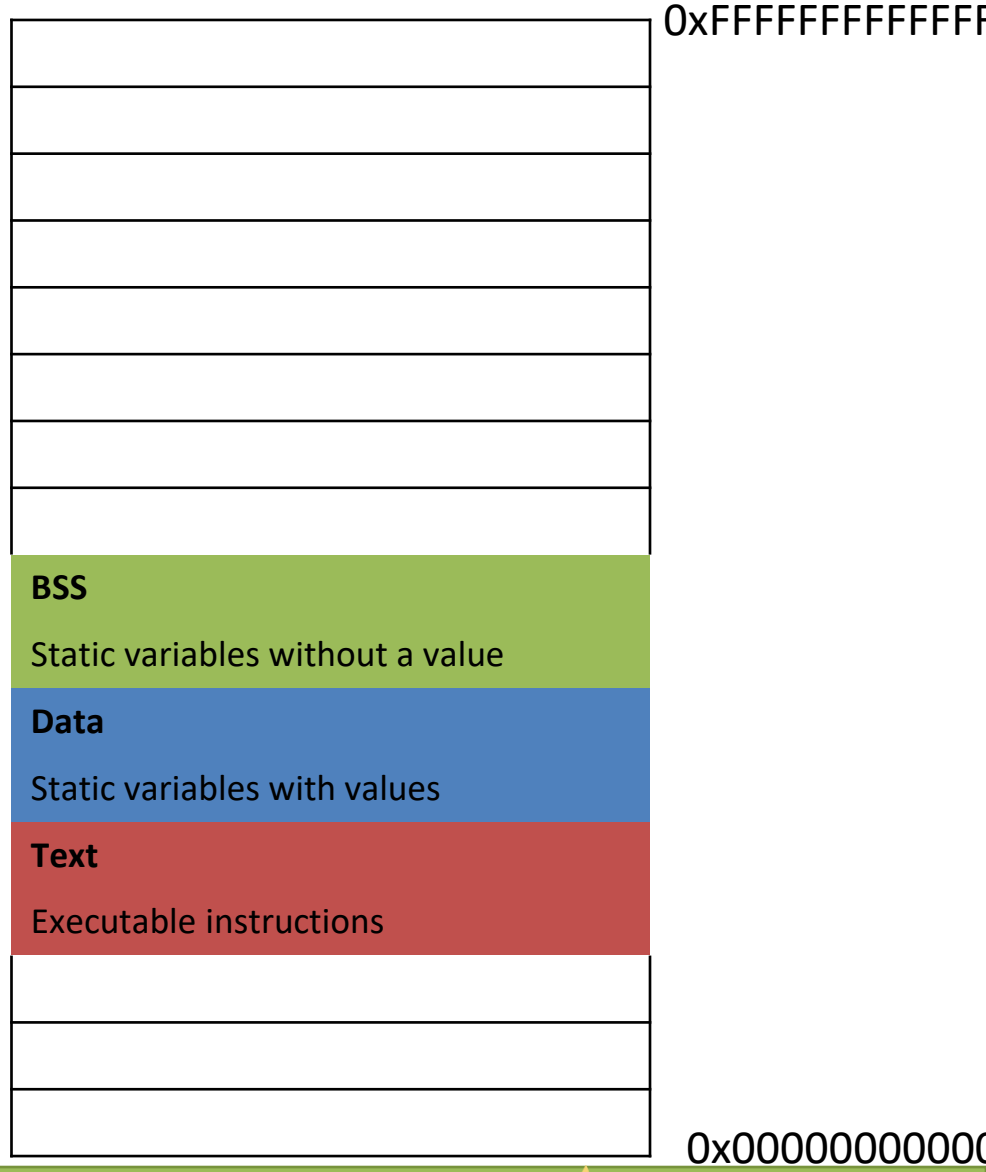
BSS Segment- contains statically allocated variables that are declared, but have not been assigned a value yet

What does a program look like in memory?



Process Manager

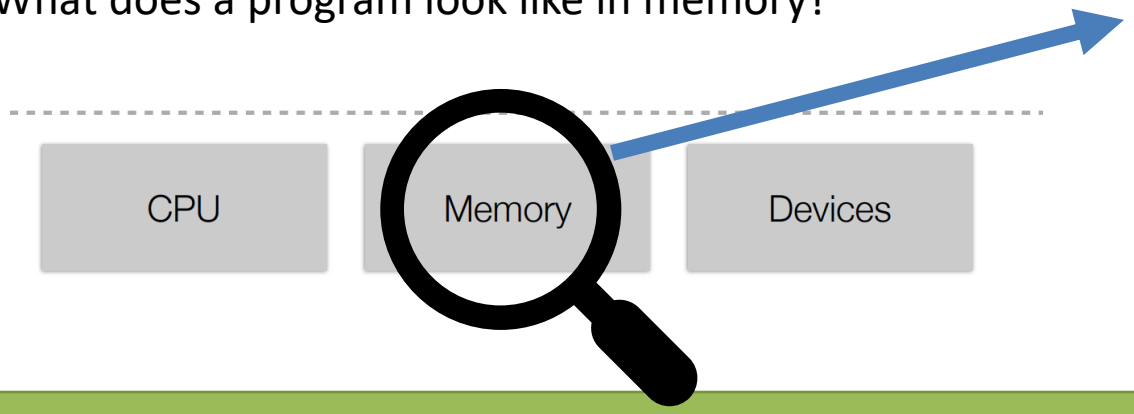
- Manages how processes are structured and how to handle many processes running at once



Applications Layout in Memory

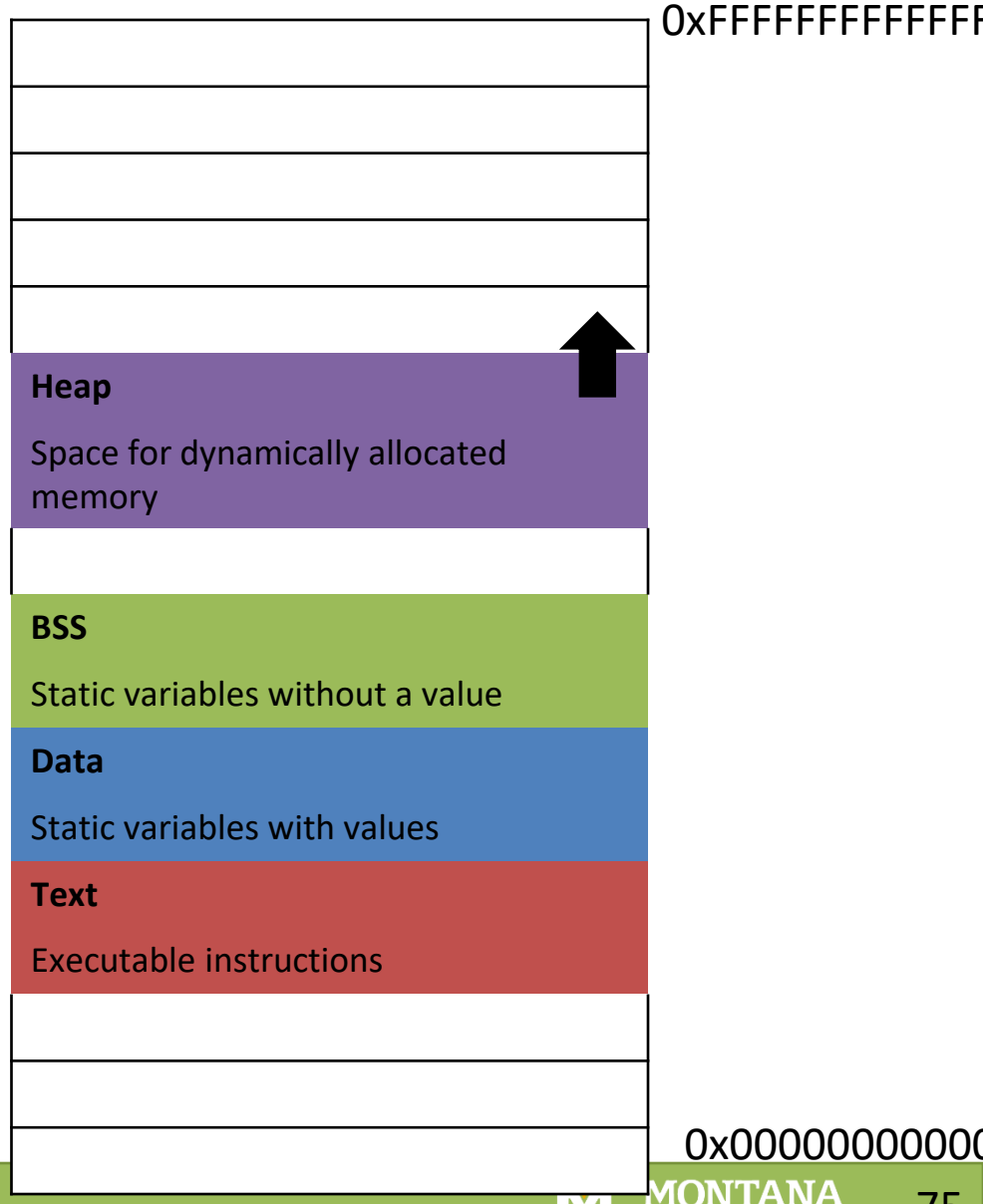
Heap- memory set aside for dynamic allocation (e.g. malloc). Grows “up” as more memory is allocated

What does a program look like in memory?



Process Manager

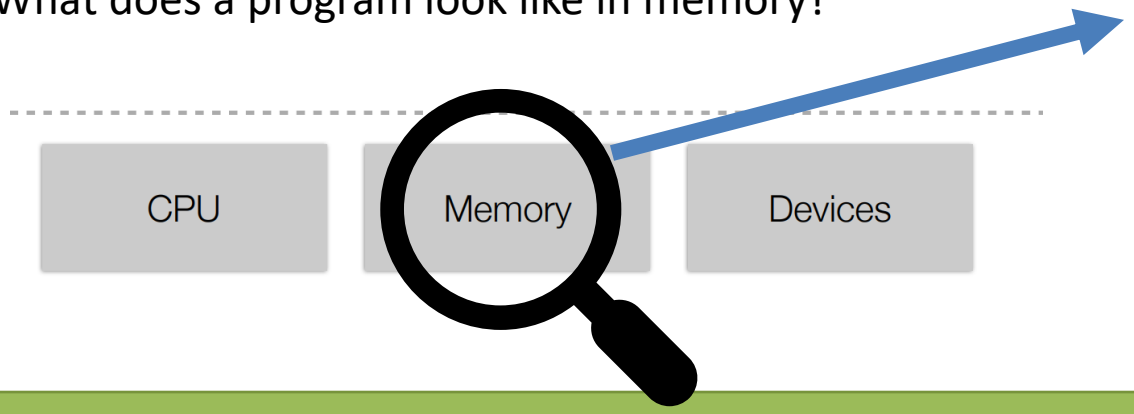
- Manages how processes are structured and how to handle many processes running at once



Applications Layout in Memory

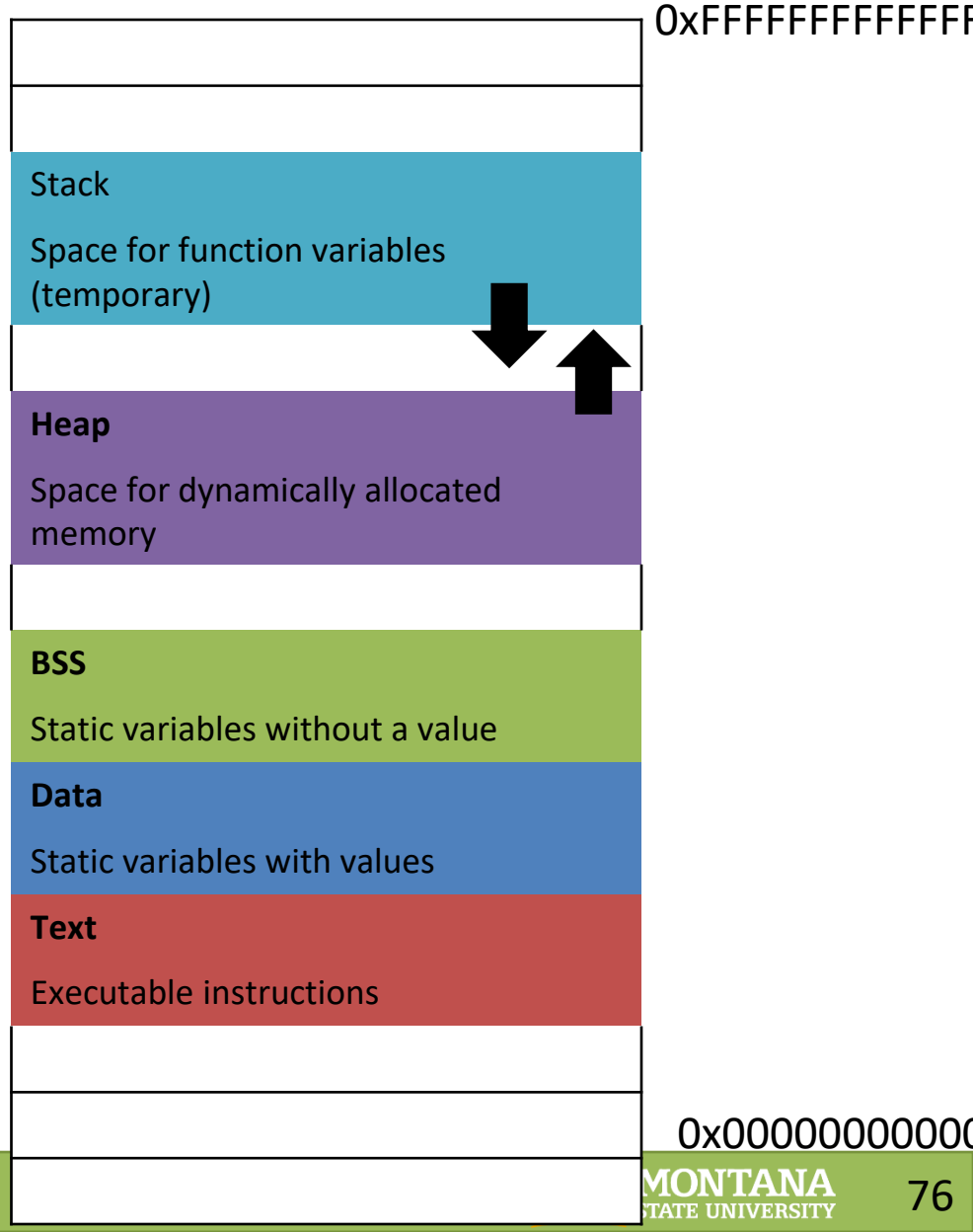
Stack – memory for storing function variables. Grows "down" as additional functions are called

What does a program look like in memory?



Process Manager

- Manages how processes are structured and how to handle many processes running at once



Applications Layout in Memory

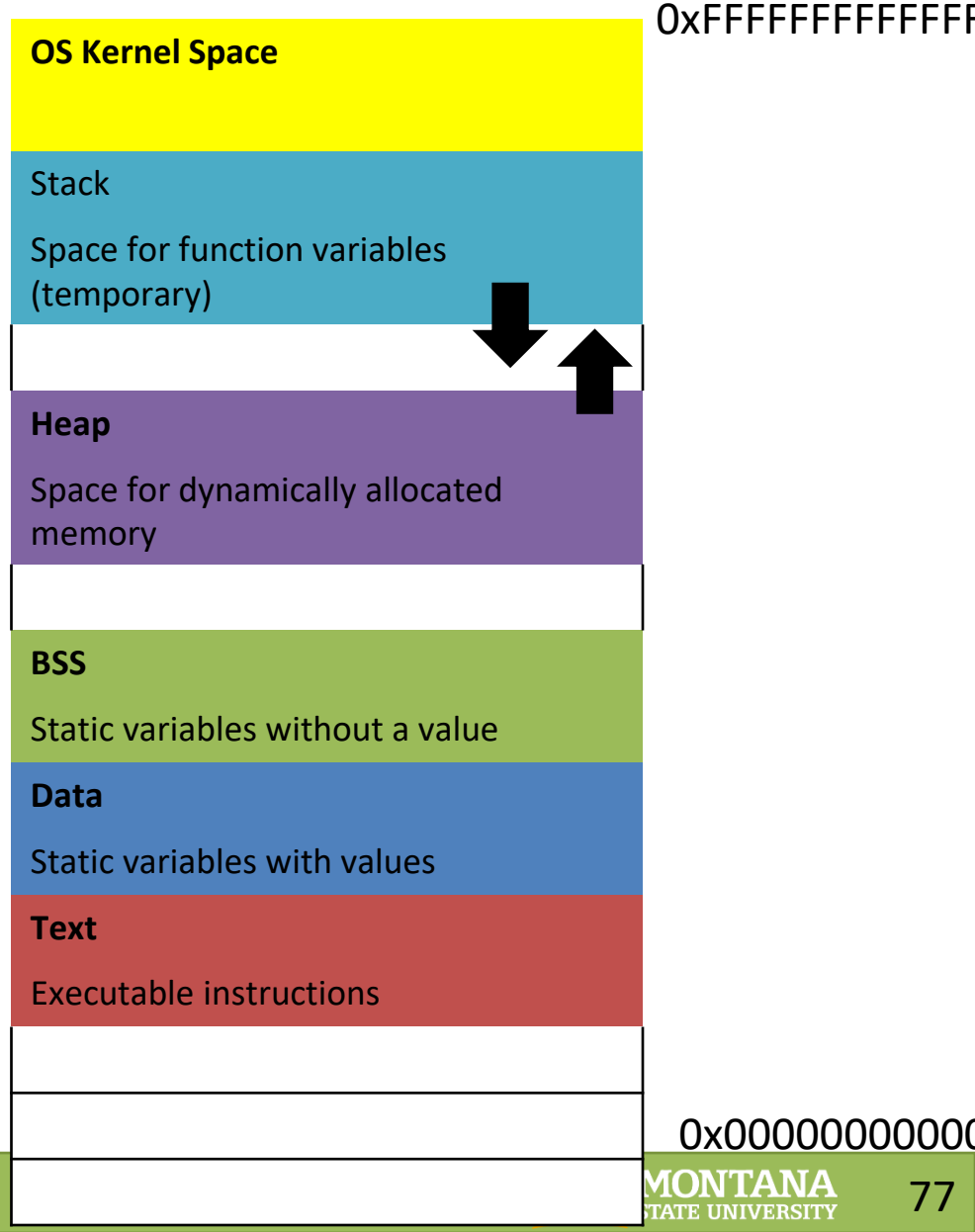
Process Manager

- Manages how processes are structured and how to handle many processes running at once

1. Executable Code

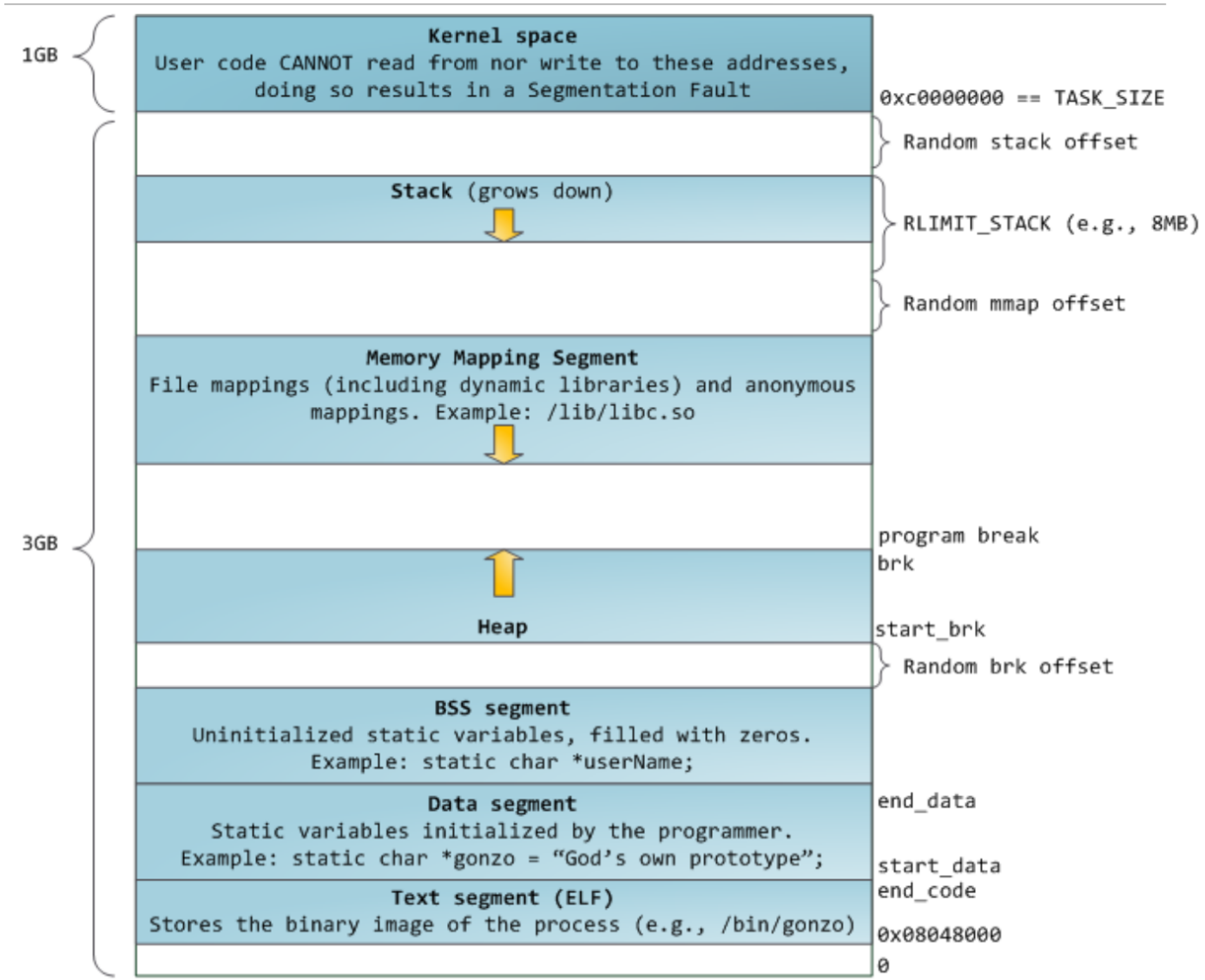
2. Associated Data

3. Execution Context/Bookkeeping information



Applications Layout in Memory

Demo?



Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
                                          2492 1544 113       1544        88        0              0             0              0               0    0       0      0           0 KB
                                          ==== ==== === ========== ======== ======== ============== ============= ============== =============== ==== ======= ====== ===========
ffffffffff600000 --xp 00000000  00:00       0    4    0   0          0         0        0              0             0              0               0    0       0      0           0 [vsyscall]
    7ffca3ddc000 r-xp 00000000  00:00       0    4    4   0          4         0        0              0             0              0               0    0       0      0           0 [vdso]
    7ffca3dd9000 r--p 00000000  00:00       0   12    0   0          0         0        0              0             0              0               0    0       0      0           0 [vvar]
    7ffca3dac000 rw-p 00000000  00:00       0  132   16  16         16        16        0              0             0              0               0    0       0      0           0 [stack]
    7f9ab45c4000 rw-p 00000000  00:00       0    4    4   4          4         4        0              0             0              0               0    0       0      0           0
    7f9ab45c3000 rw-p 0002d000  08:05 3541124    4    4   4          4         4        0              0             0              0               0    0       0      0           0 ld-2.31.so
    7f9ab45c2000 r--p 0002c000  08:05 3541124    4    4   4          4         4        0              0             0              0               0    0       0      0           0 ld-2.31.so
    7f9ab45b9000 r--p 00024000  08:05 3541124   32   32   0         32         0        0              0             0              0               0    0       0      0           0 ld-2.31.so
    7f9ab4596000 r-xp 00001000  08:05 3541124  140  140   1        140         0        0              0             0              0               0    0       0      0           0 ld-2.31.so
    7f9ab4595000 r--p 00000000  08:05 3541124    4    4   0          4         0        0              0             0              0               0    0       0      0           0 ld-2.31.so
    7f9ab457c000 rw-p 00000000  00:00       0   24   24  24         24        24        0              0             0              0               0    0       0      0           0
    7f9ab4579000 rw-p 001ea000  08:05 3541128   12   12  12         12        12        0              0             0              0               0    0       0      0           0 libc-2.31.so
    7f9ab4576000 r--p 001e7000  08:05 3541128   12   12  12         12        12        0              0             0              0               0    0       0      0           0 libc-2.31.so
    7f9ab4575000 ---p 001e7000  08:05 3541128    4    0   0          0         0        0              0             0              0               0    0       0      0           0 libc-2.31.so
    7f9ab452b000 r--p 0019d000  08:05 3541128  296  124   1        124         0        0              0             0              0               0    0       0      0           0 libc-2.31.so
    7f9ab43b3000 r-xp 00025000  08:05 3541128 1504 1000  10       1000         0        0              0             0              0               0    0       0      0           0 libc-2.31.so
    7f9ab438e000 r--p 00000000  08:05 3541128  148  140   1        140         0        0              0             0              0               0    0       0      0           0 libc-2.31.so
    558e1b9d6000 rw-p 00000000  00:00       0  132    4   4          4         4        0              0             0              0               0    0       0      0           0 [heap]
    558e1a654000 rw-p 00003000  08:05 1051705    4    4   4          4         4        0              0             0              0               0    0       0      0           0 probe
    558e1a653000 r--p 00002000  08:05 1051705    4    4   4          4         4        0              0             0              0               0    0       0      0           0 probe
    558e1a652000 r--p 00002000  08:05 1051705    4    4   4          4         0        0              0             0              0               0    0       0      0           0 probe
    558e1a651000 r-xp 00001000  08:05 1051705    4    4   4          4         0        0              0             0              0               0    0       0      0           0 probe
    558e1a650000 r--p 00000000  08:05 1051705    4    4   4          4         0        0              0             0              0               0    0       0      0           0 probe
         Address Perm   Offset Device   Inode Size  Rss Pss Referenced Anonymous LazyFree ShmemPmdMapped FilePmdMapped Shared_Hugetlb Private_Hugetlb Swap SwapPss Locked THPeligible Mapping
4175:   ./probe
```

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
                                                  KB
ffffffffff600000 --xp 00000000                    [vsyscall]
    7ffca3ddc000 r-xp 00000000                    [vdso]
    7ffca3dd9000 r--p 00000000                    [vvar]
    7ffca3dac000 rw-p 00000000                    [stack]
    7f9ab45c4000 rw-p 00000000
    7f9ab45c3000 rw-p 0002d000                    ld-2.31.so
    7f9ab45c2000 r--p 0002c000                    ld-2.31.so
    7f9ab45b9000 r--p 00024000                    ld-2.31.so
    7f9ab4596000 r-xp 00001000                    ld-2.31.so
    7f9ab4595000 r--p 00000000                    ld-2.31.so
    7f9ab457c000 rw-p 00000000
    7f9ab4579000 rw-p 001ea000                    libc-2.31.so
    7f9ab4576000 r--p 001e7000                    libc-2.31.so
    7f9ab4575000 ---p 001e7000                    libc-2.31.so
    7f9ab452b000 r--p 0019d000                    libc-2.31.so
    7f9ab43b3000 r-xp 00025000                    libc-2.31.so
    7f9ab438e000 r--p 00000000                    libc-2.31.so
    558e1b9d6000 rw-p 00000000                    [heap]
    558e1a654000 rw-p 00003000                    probe
    558e1a653000 r--p 00002000                    probe
    558e1a652000 r--p 00002000                    probe
    558e1a651000 r-xp 00001000                    probe
    558e1a650000 r--p 00000000                    probe
         Address Perm   Offset                    Mapping
4175:   ./probe
```

"probe" is the name of our executable

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   KB
    7ffca3ddc000 r-xp 00000000   [vsyscall]
    7ffca3dd9000 r--p 00000000   [vdso]
    7ffca3dac000 rw-p 00000000   [vvar]
    7f9ab45c4000 rw-p 00000000   [stack]
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000   ld-2.31.so
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   libc-2.31.so
    558e1a653000 r--p 00002000   [heap]
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r-- 00000000    probe
         Address Perm   Offset   probe
4175:   ./probe                  Mapping
```

BSS

Static variables without a value

Data

Static variables with values

"probe" is the name of out executable

This section is executable "x"

Text

Executable instructions

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   KB
    7ffca3ddc000 r-xp 00000000   [vsyscall]
    7ffca3dd9000 r--p 00000000   [vdso]
    7ffca3dac000 rw-p 00000000   [vvar]
    7f9ab45c4000 rw-p 00000000   [stack]
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000   ld-2.31.so
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   libc-2.31.so
    558e1a653000 r--p 00002000   [heap]
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r-- 00000000    probe
         Address Perm   Offset   probe
4175:   ./probe                  Mapping
```

Heap

Space for dynamically allocated memory

BSS

Static variables without a value

Data

Static variables with values

Beginning of heap

"probe" is the name of out executable

This section is executable "x"

Text

Executable instructions

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   KB
    7ffca3ddc000 r-xp 00000000   [vsyscall]
    7ffca3dd9000 r--p 00000000   [vdso]
    7ffca3dac000 rw-p 00000000   [vvar]
    7f9ab45c4000 rw-p 00000000   [stack]
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000   ld-2.31.so
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   [heap]
    558e1a653000 r--p 00002000   probe
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r-- 00000000    probe
         Address Perm   Offset   probe
4175:   ./probe                  Mapping
```

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

Space for dynamically allocated memory

Beginning of heap

BSS

Static variables without a value

Data

Static variables with values

"probe" is the name of out executable

This section is executable "x"

Text

Executable instructions

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   KB
    7ffca3ddc000 r-xp 00000000   [vsyscall]
    7ffca3dd9000 r--p 00000000   [vdso]
    7ffca3dac000 rw-p 00000000   [vvar]
    7f9ab45c4000 rw-p 00000000   [stack]
    7f9ab45c3000 rw-p 0002d000   ld-2.31.so
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000
    7f9ab4579000 rw-p 001ea000   libc-2.31.so
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   [heap]
    558e1a653000 r--p 00002000   probe
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r-- 00000000    probe
         Address Perm   Offset   probe
4175:   ./probe                  Mapping
```

Stack

Space for function variables (temporary)

Beginning of stack

Memory Mapping Segment

File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

Space for dynamically allocated memory

Beginning of heap

BSS

Static variables without a value

Data

Static variables with values

"probe" is the name of out executable

This section is executable "x"

Text

Executable instructions

Applications Layout in Memory

When you allocate variables on the stack

When you allocate variables on the heap

Ouput of `pmap` (process mapping tool)

Stack

Space for function variables (temporary)

Beginning of stack

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

Space for dynamically allocated memory

Beginning of heap

BSS

Static variables without a value

Data

Static variables with values

"probe" is the name of out executable

able "x"

Text

Executable instructions

```
KB

[vsyscall]
[vdso]
[vvar]
[stack]

ld-2.31.so
ld-2.31.so
ld-2.31.so
ld-2.31.so
ld-2.31.so

libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
[heap]
probe
probe
probe
probe
probe
Mapping
```

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   [vsyscall]
    7ffca3ddc000 r-xp 00000000   [vdso]
    7ffca3dd9000 r--p 00000000   [vvar]
    7ffca3dac000 rw-p 00000000   [stack]
    7f9ab45c4000 rw-p 00000000   ld-2.31.so
    7f9ab45c3000 rw-p 0002d000   ld-2.31.so
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000
    7f9ab4579000 rw-p 001ea000   libc-2.31.so
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   [heap]
    558e1a653000 r--p 00002000   probe
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r-- 00000000    probe
         Address Perm   Offset   Mapping
4175:   ./probe
```

OS Kernel Space

Stack

Space for function variables (temporary)

Beginning of stack

Memory Mapping Segment

File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

Space for dynamically allocated memory

Beginning of heap

BSS

Static variables without a value

Data

Static variables with values

"probe" is the name of out executable

This section is executable "x"

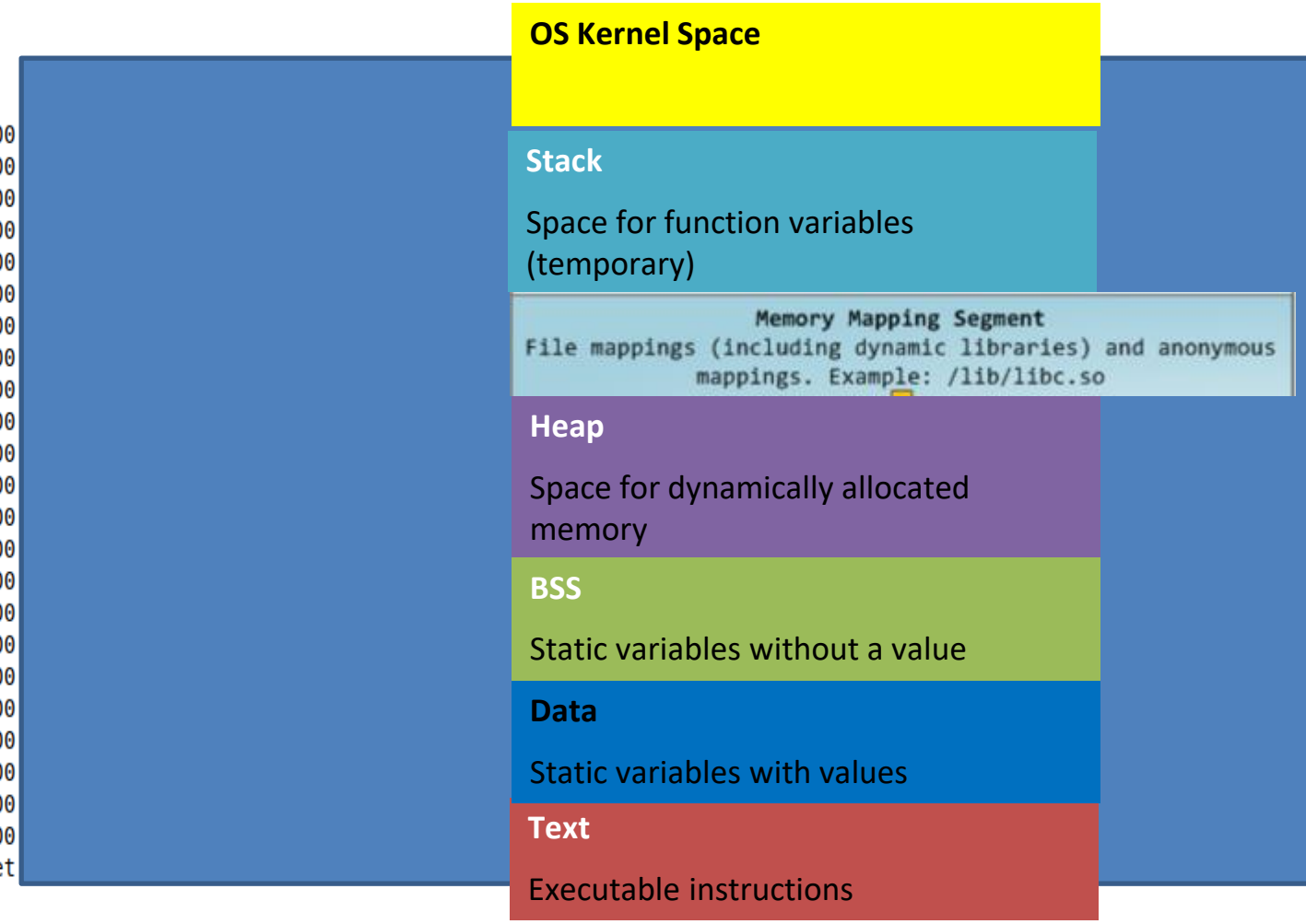
Text

Executable instructions

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   [vsyscall]
    7ffca3ddc000 r-xp 00000000   [vdso]
    7ffca3dd9000 r--p 00000000   [vvar]
    7ffca3dac000 rw-p 00000000   [stack]
    7f9ab45c4000 rw-p 00000000
    7f9ab45c3000 rw-p 0002d000   ld-2.31.so
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000
    7f9ab4579000 rw-p 001ea000   libc-2.31.so
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   [heap]
    558e1a654000 rw-p 00003000   probe
    558e1a653000 r--p 00002000   probe
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r--p 00000000   probe
         Address Perm   Offset   Mapping
4175:   ./probe
```



Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   KB
    7ffca3ddc000 r-xp 00000000   [vsyscall]
    7ffca3dd9000 r--p 00000000   [vdso]
    7ffca3dac000 rw-p 00000000   [vvar]
    7f9ab45c4000 rw-p 00000000   [stack]
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000   ld-2.31.so
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   [heap]
    558e1a653000 r--p 00002000   probe
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r--p 00000000   probe
         Address Perm   Offset   Mapping
4175:   ./probe
```

```
-> the address of main   = 0x558e1a651249
-> the address of printf = 0x7f9ab43f2e10
-> the address of getenv = 0x7f9ab43d7020
-> a stack address       = 0x7ffca3dcb3b0
-> a global address      = 0x558e1a6540c4
-> the argv address      = 0x7ffca3dcb4f8
-> argv[0]               = 0x7ffca3dcc45f
   value is [./probe]
-> the environ address   = 0x7ffca3dcb508
-> the envp address      = 0x7ffca3dcb508
-> getenv("PWD")         = 0x7ffca3dcc5ff
   value is [/home/seed/os-review]
-> a heap address        = 0x558e1b9d66b0
```

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   KB
    7ffca3ddc000 r-xp 00000000   [vsyscall]
    7ffca3dd9000 r--p 00000000   [vdso]
    7ffca3dac000 rw-p 00000000   [vvar]
    7f9ab45c4000 rw-p 00000000   [stack]
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000   ld-2.31.so
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   [heap]
    558e1a653000 r--p 00002000   probe
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r--p 00000000   probe
         Address Perm   Offset   Mapping
4175:   ./probe
```

Where is "main" located in memory?

```
-> the address of main    = 0x558e1a651249
-> the address of printf  = 0x7f9ab43f2e10
-> the address of getenv  = 0x7f9ab43d7020
-> a stack address        = 0x7ffca3dcb3b0
-> a global address       = 0x558e1a6540c4
-> the argv address       = 0x7ffca3dcb4f8
-> argv[0]                = 0x7ffca3dcc45f
   value is [./probe]
-> the environ address    = 0x7ffca3dcb508
-> the envp address       = 0x7ffca3dcb508
-> getenv("PWD")          = 0x7ffca3dcc5ff
   value is [/home/seed/os-review]
-> a heap address         = 0x558e1b9d66b0
```

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000
    7ffca3ddc000 r-xp 00000000
    7ffca3dd9000 r--p 00000000
    7ffca3dac000 rw-p 00000000
    7f9ab45c4000 rw-p 00000000
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000
    7f9ab45b9000 r--p 00024000
    7f9ab4596000 r-xp 00001000
    7f9ab4595000 r--p 00000000
    7f9ab457c000 rw-p 00000000
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000
    7f9ab4575000 ---p 001e7000
    7f9ab452b000 r--p 0019d000
    7f9ab43b3000 r-xp 00025000
    7f9ab438e000 r--p 00000000
    558e1b9d6000 rw-p 00000000
    558e1a654000 rw-p 00003000
    558e1a653000 r--p 00002000
    558e1a651000 r-xp 00001000
         Address Perm   Offset
4175:   ./probe
```

Where is "main" located in memory?

```
-> the address of main    = 0x558e1a651249
-> the address of printf  = 0x7f9ab43f2e10
-> the address of getenv  = 0x7f9ab43d7020
-> a stack address        = 0x7ffca3dcb3b0
-> a global address       = 0x558e1a6540c4
-> the argv address       = 0x7ffca3dcb4f8
-> argv[0]                = 0x7ffca3dcc45f
   value is [./probe]
-> the environ address    = 0x7ffca3dcb508
-> the envp address       = 0x7ffca3dcb508
-> getenv("PWD")          = 0x7ffca3dcc5ff
   value is [/home/seed/os-review]
-> a heap address         = 0x558e1b9d66b0
```

```
KB

[vsyscall]
[vdso]
[vvar]
[stack]

ld-2.31.so
ld-2.31.so
ld-2.31.so
ld-2.31.so
ld-2.31.so

libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
[heap]
probe
probe
probe
Mapping
```

`main` is code in our program, so it goes inside the text segment

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   KB
    7ffca3ddc000 r-xp 00000000   [vsyscall]
    7ffca3dd9000 r--p 00000000   [vdso]
    7ffca3dac000 rw-p 00000000   [vvar]
    7f9ab45c4000 rw-p 00000000   [stack]
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000   ld-2.31.so
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   [heap]
    558e1a653000 r--p 00002000   probe
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r--p 00000000   probe
         Address Perm   Offset   Mapping
4175:   ./probe
```

Where is “printf” located in memory?

```
-> the address of main    = 0x558e1a651249
-> the address of printf  = 0x7f9ab43f2e10
-> the address of getenv  = 0x7f9ab43d7020
-> a stack address        = 0x7ffca3dcb3b0
-> a global address       = 0x558e1a6540c4
-> the argv address       = 0x7ffca3dcb4f8
-> argv[0]                = 0x7ffca3dcc45f
   value is [./probe]
-> the environ address    = 0x7ffca3dcb508
-> the envp address       = 0x7ffca3dcb508
-> getenv("PWD")          = 0x7ffca3dcc5ff
   value is [/home/seed/os-review]
-> a heap address         = 0x558e1b9d66b0
```

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000   KB
    7ffca3ddc000 r-xp 00000000   [vsyscall]
    7ffca3dd9000 r--p 00000000   [vdso]
    7ffca3dac000 rw-p 00000000   [vvar]
    7f9ab45c4000 rw-p 00000000   [stack]
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000   ld-2.31.so
    7f9ab45b9000 r--p 00024000   ld-2.31.so
    7f9ab4596000 r-xp 00001000   ld-2.31.so
    7f9ab4595000 r--p 00000000   ld-2.31.so
    7f9ab457c000 rw-p 00000000   ld-2.31.so
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000   libc-2.31.so
    7f9ab4575000 ---p 001e7000   libc-2.31.so
    7f9ab452b000 r--p 0019d000   libc-2.31.so
    7f9ab43b3000 r-xp 00025000   libc-2.31.so
    7f9ab438e000 r--p 00000000   libc-2.31.so
    558e1b9d6000 rw-p 00000000   libc-2.31.so
    558e1a654000 rw-p 00003000   [heap]
    558e1a653000 r--p 00002000   probe
    558e1a652000 r--p 00002000   probe
    558e1a651000 r-xp 00001000   probe
    558e1a650000 r--p 00000000   probe
         Address Perm   Offset   Mapping
4175:   ./probe
```

Where is “printf” located in memory?

```
-> the address of main    = 0x558e1a651249
-> the address of printf  = 0x7f9ab43f2e10
-> the address of getenv  = 0x7f9ab43d7020
-> a stack address        = 0x7ffca3dcb3b0
-> a global address       = 0x558e1a6540c4
-> the argv address       = 0x7ffca3dcb4f8
-> argv[0]                = 0x7ffca3dcc45f
   value is [./probe]
-> the environ address    = 0x7ffca3dcb508
-> the envp address       = 0x7ffca3dcb508
-> getenv("PWD")          = 0x7ffca3dcc5ff
   value is [/home/seed/os-review]
-> a heap address         = 0x558e1b9d66b0
```

`printf` is executable code from a shared library (libc) so we are in the memory mapping segment!

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000
    7ffca3ddc000 r-xp 00000000
    7ffca3dd9000 r--p 00000000
    7ffca3dac000 rw-p 00000000
    7f9ab45c4000 rw-p 00000000
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000
    7f9ab45b9000 r--p 00024000
    7f9ab4596000 r-xp 00001000
    7f9ab4595000 r--p 00000000
    7f9ab457c000 rw-p 00000000
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000
    7f9ab4575000 ---p 001e7000
    7f9ab452b000 r--p 0019d000
    7f9ab43b3000 r-xp 00025000
    7f9ab438e000 r--p 00000000
    558e1b9d6000 rw-p 00000000
    558e1a654000 rw-p 00003000
    558e1a653000 r--p 00002000
    558e1a652000 r--p 00002000
    558e1a651000 r-xp 00001000
    558e1a650000 r--p 00000000
         Address Perm   Offset
4175:   ./probe
```

Where is "argv" located in memory?

```
-> the address of main   = 0x558e1a651249
-> the address of printf = 0x7f9ab43f2e10
-> the address of getenv = 0x7f9ab43d7020
-> a stack address       = 0x7ffca3dcb3b0
-> a global address      = 0x558e1a6540c4
-> the argv address      = 0x7ffca3dcb4f8
-> argv[0]               = 0x7ffca3dcc45f
   value is [./probe]
-> the environ address   = 0x7ffca3dcb508
-> the envp address      = 0x7ffca3dcb508
-> getenv("PWD")         = 0x7ffca3dcc5ff
   value is [/home/seed/os-review]
-> a heap address        = 0x558e1b9d66b0
```

```
KB

[vsyscall]
[vdso]
[vvar]
[stack]

ld-2.31.so
ld-2.31.so
ld-2.31.so
ld-2.31.so
ld-2.31.so

libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
[heap]
probe
probe
probe
probe
probe
Mapping
```

`argv` is an array that holds the command line parameters passed into this program

Applications Layout in Memory

Ouput of `pmap` (process mapping tool)

```
ffffffffff600000 --xp 00000000
    7ffca3ddc000 r-xp 00000000
    7ffca3dd9000 r--p 00000000
    7ffca3dac000 rw-p 00000000
    7f9ab45c4000 rw-p 00000000
    7f9ab45c3000 rw-p 0002d000
    7f9ab45c2000 r--p 0002c000
    7f9ab45b9000 r--p 00024000
    7f9ab4596000 r-xp 00001000
    7f9ab4595000 r--p 00000000
    7f9ab457c000 rw-p 00000000
    7f9ab4579000 rw-p 001ea000
    7f9ab4576000 r--p 001e7000
    7f9ab4575000 ---p 001e7000
    7f9ab452b000 r--p 0019d000
    7f9ab43b3000 r-xp 00025000
    7f9ab438e000 r--p 00000000
    558e1b9d6000 rw-p 00000000
    558e1a654000 rw-p 00003000
    558e1a653000 r--p 00002000
    558e1a652000 r--p 00002000
    558e1a651000 r-xp 00001000
    558e1a650000 r--p 00000000
         Address Perm   Offset
4175:   ./probe
```

Where is "argv" located in memory?

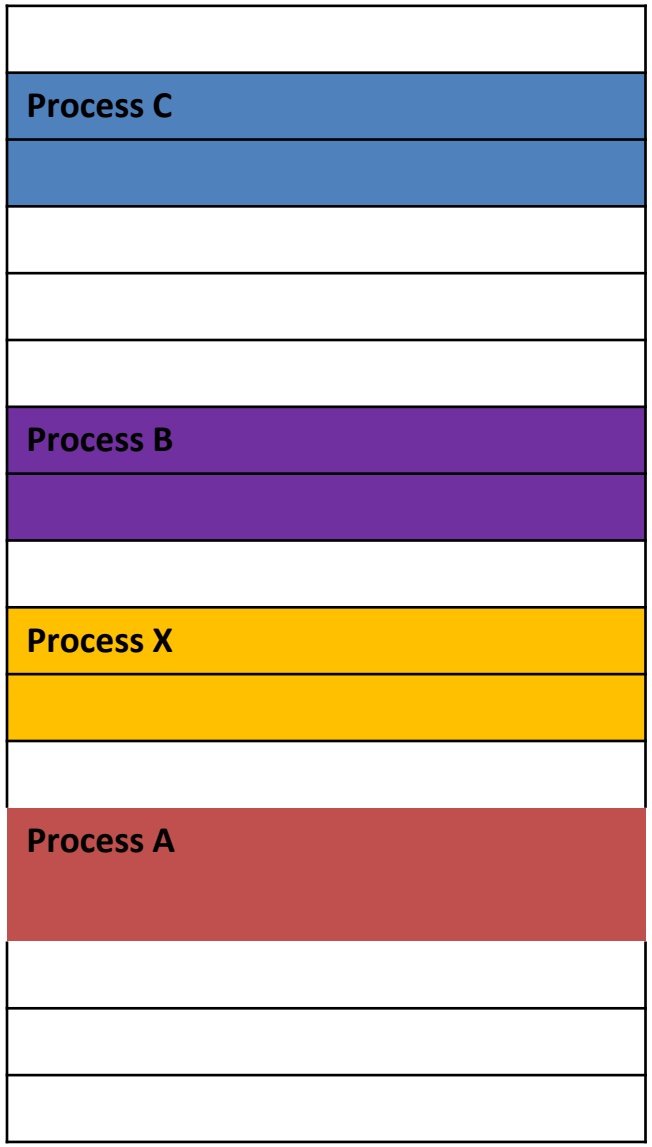
```
-> the address of main    = 0x558e1a651249
-> the address of printf  = 0x7f9ab43f2e10
-> the address of getenv  = 0x7f9ab43d7020
-> a stack address        = 0x7ffca3dcb3b0
-> a global address       = 0x558e1a6540c4
-> the argv address       = 0x7ffca3dcb4f8
-> argv[0]                = 0x7ffca3dcc45f
   value is [./probe]
-> the environ address    = 0x7ffca3dcb508
-> the envp address       = 0x7ffca3dcb508
-> getenv("PWD")          = 0x7ffca3dcc5ff
   value is [/home/seed/os-review]
-> a heap address         = 0x558e1b9d66b0
```

```
KB

[vsyscall]
[vdso]
[vvar]
[stack]

ld-2.31.so
ld-2.31.so
ld-2.31.so
ld-2.31.so
ld-2.31.so

libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
libc-2.31.so
[heap]
probe
probe
probe
probe
probe
Mapping
```

`argv` is the argument to the main function, so we are in the stack!

Applications Layout in Memory

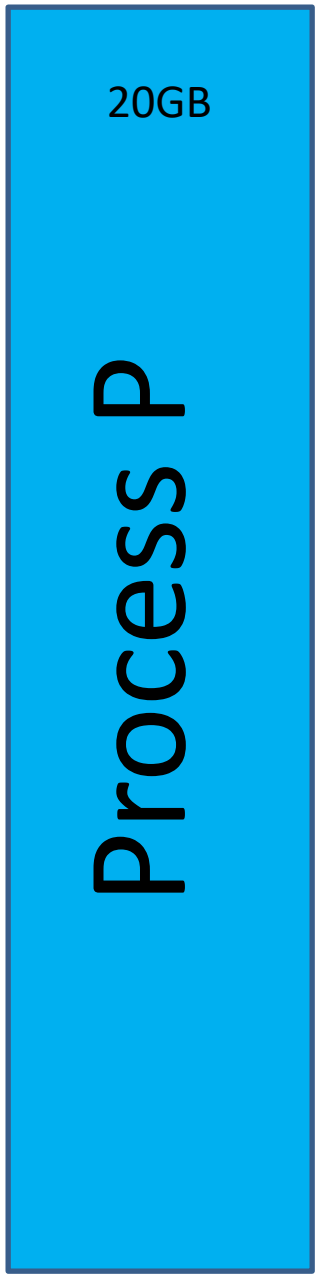
We have many programs that are actively running on our computer

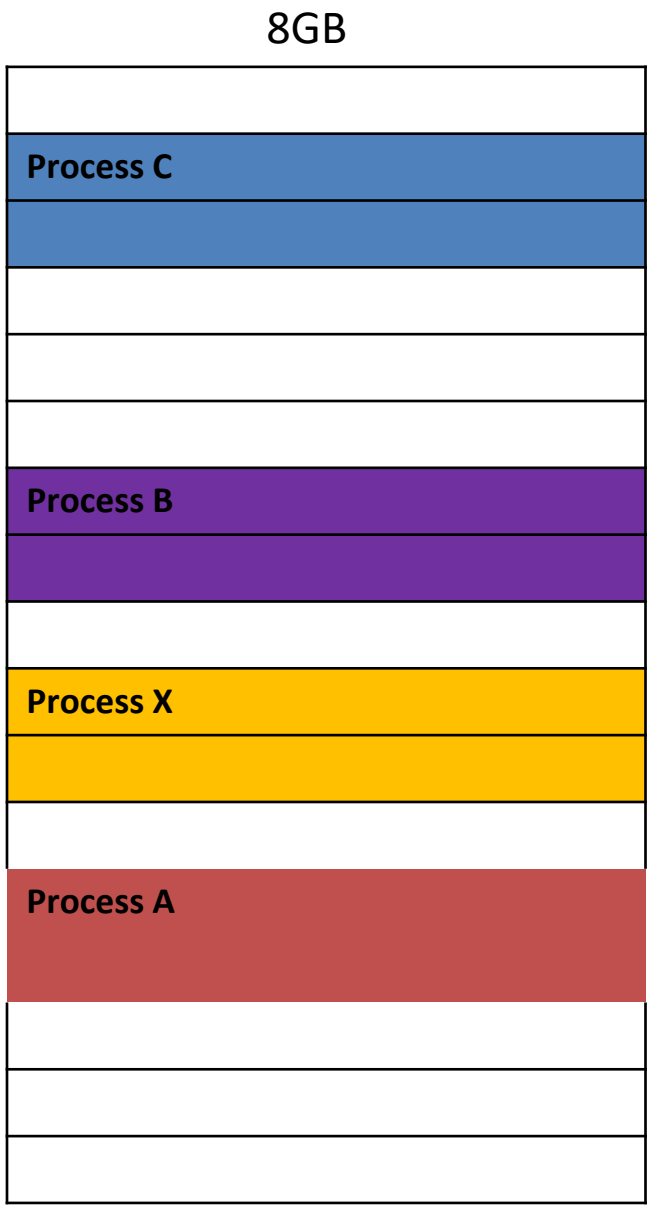


Applications Layout in Memory

We have many programs that are actively running on our computer

What if we have a program that is bigger than out entire main memory?



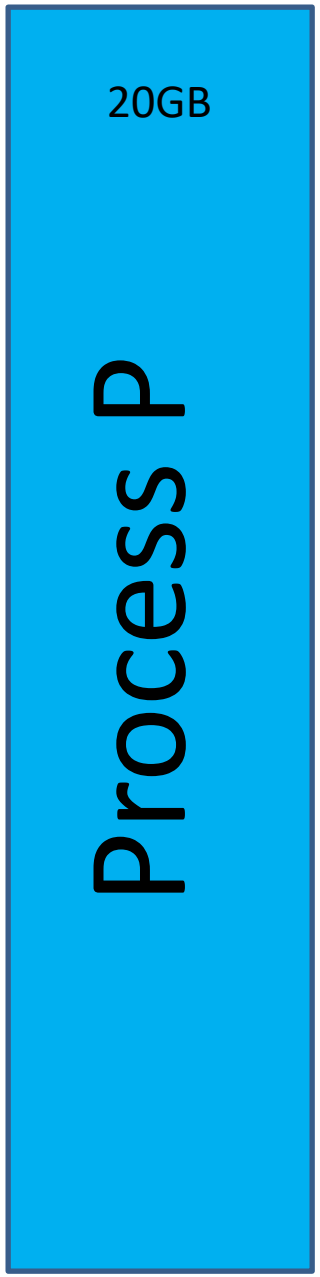


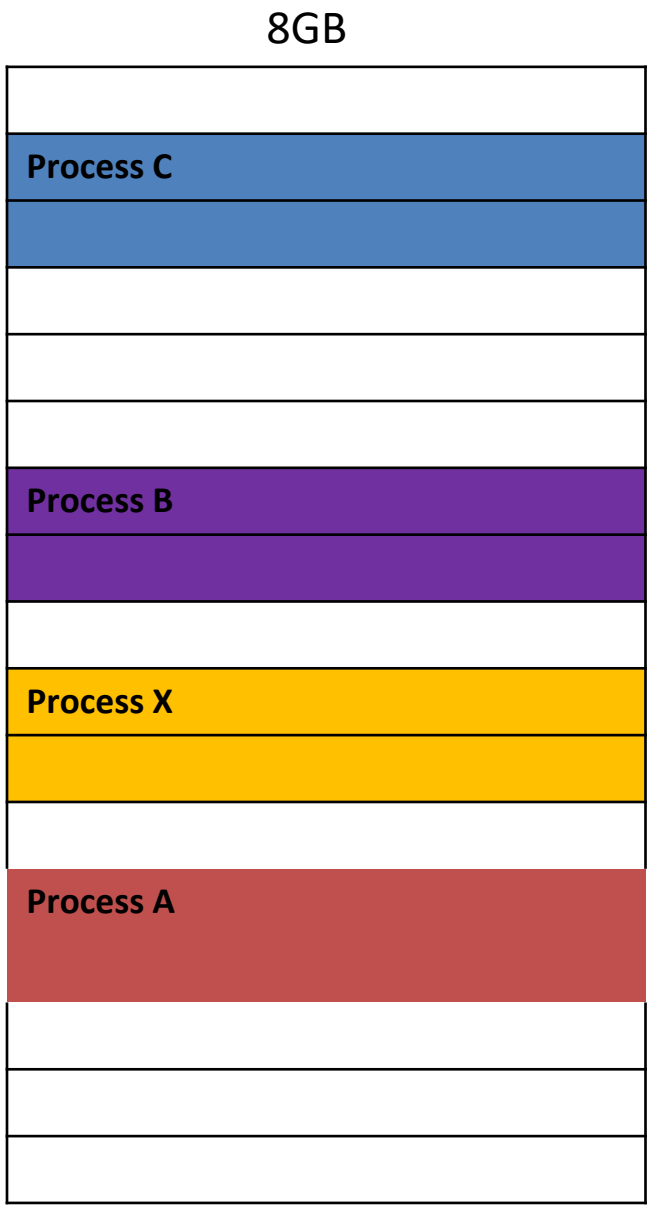
Applications Layout in Memory

We have many programs that are actively running on our computer

What if we have a program that is bigger than out entire main memory?

Does our computer crash?

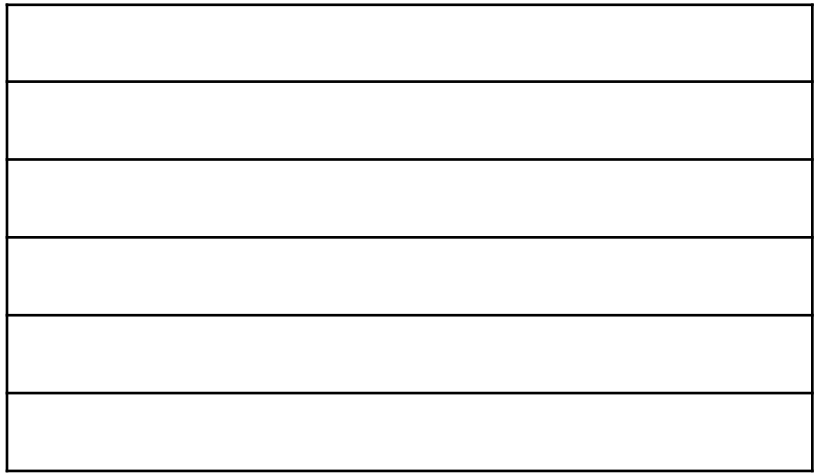




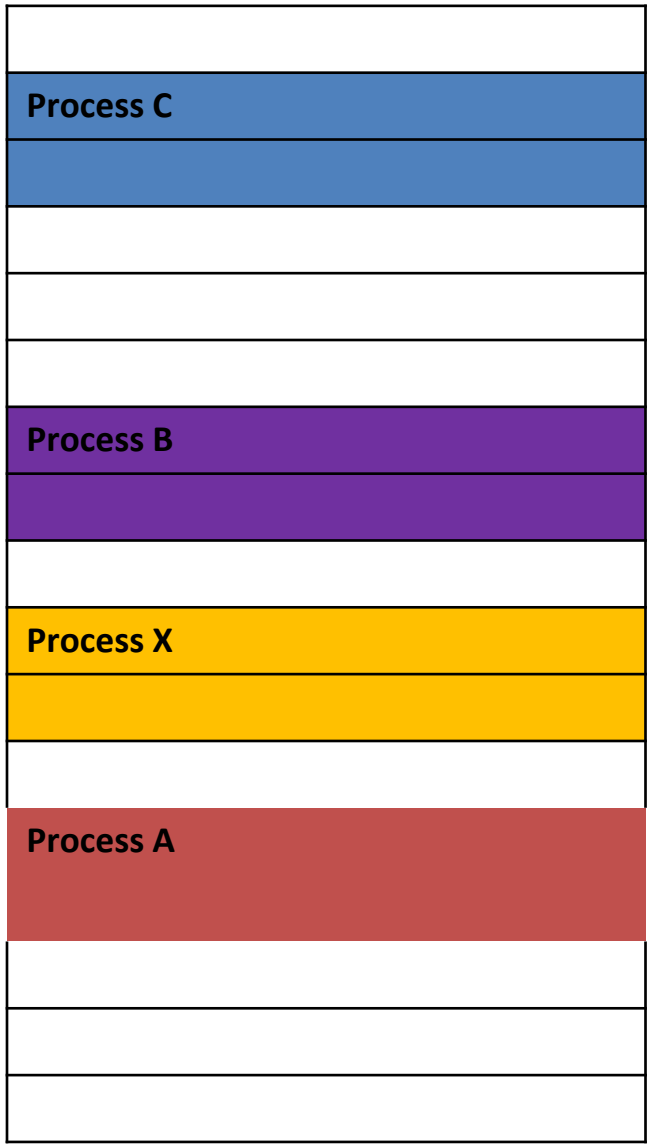
Memory management

Virtual Memory uses secondary storage to give programs the illusion that they have infinite storage

Secondary Storage



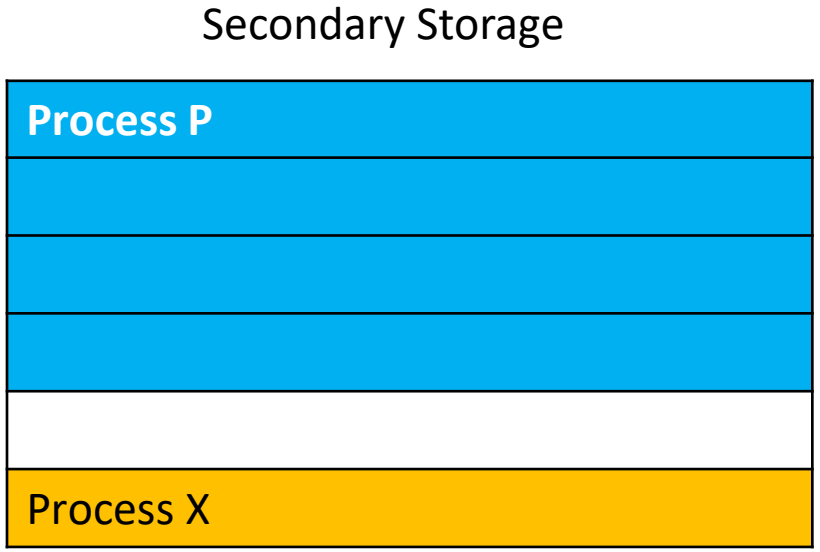
8GB

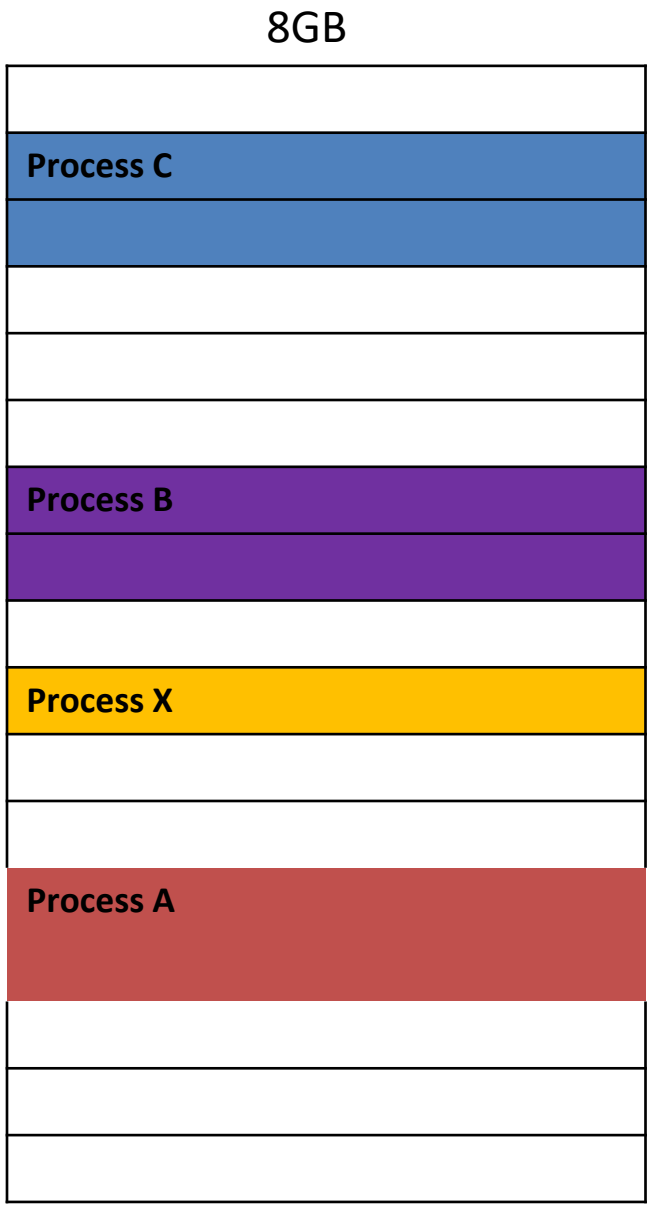


Memory management

Virtual Memory uses secondary storage to give programs the illusion that they have infinite storage

We split the process into smaller **pages**. Load pages into memory only when needed

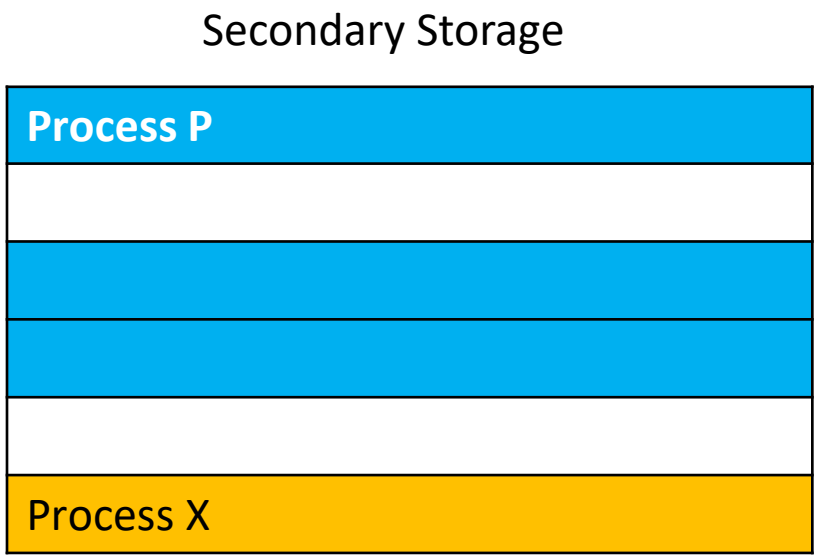


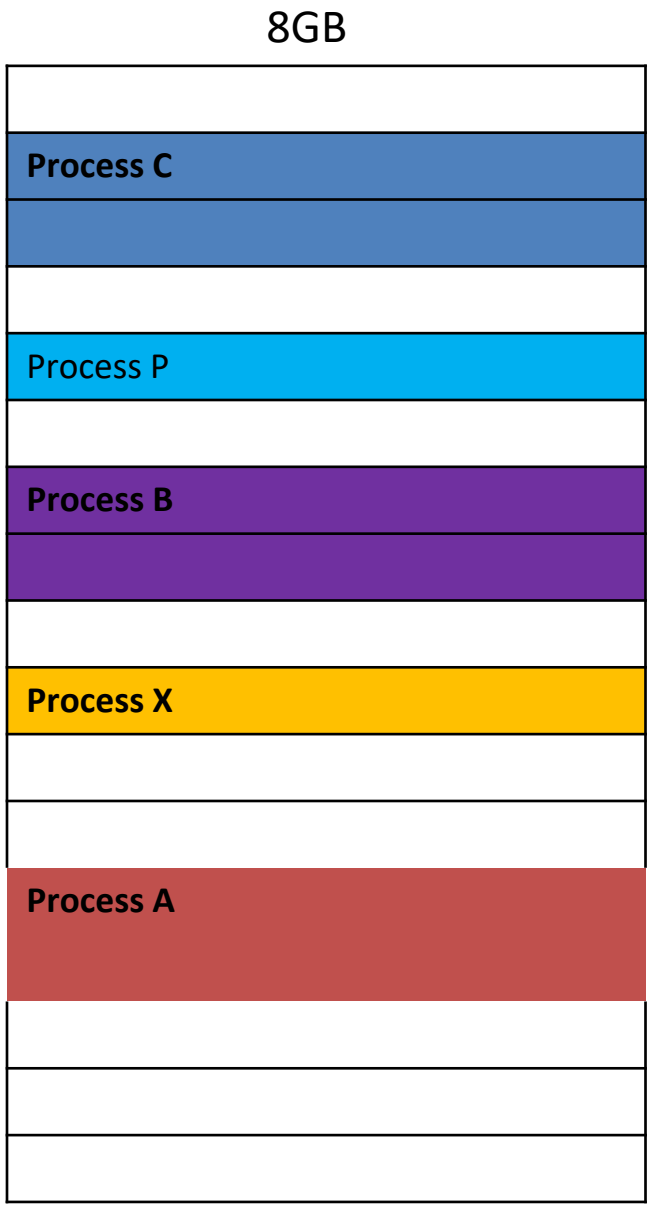


Memory management

Virtual Memory uses secondary storage to give programs the illusion that they have infinite storage

We split the process into smaller **pages**. Load pages into memory only when needed

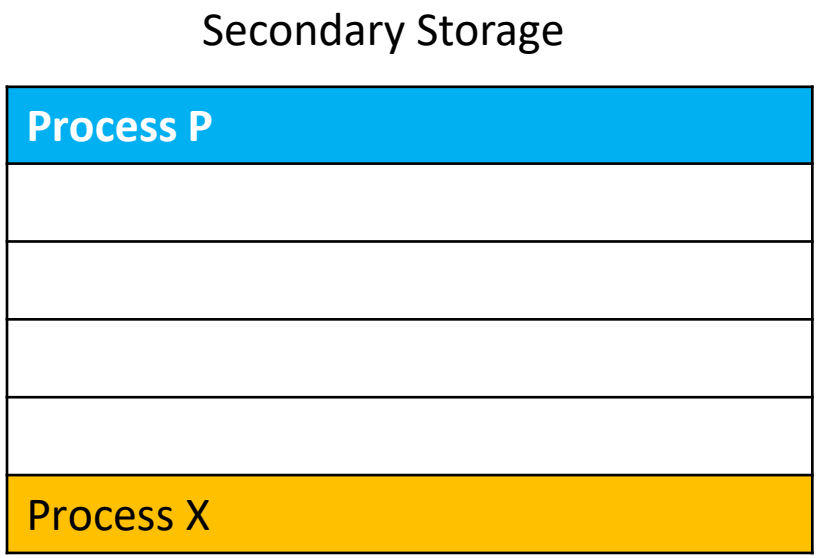


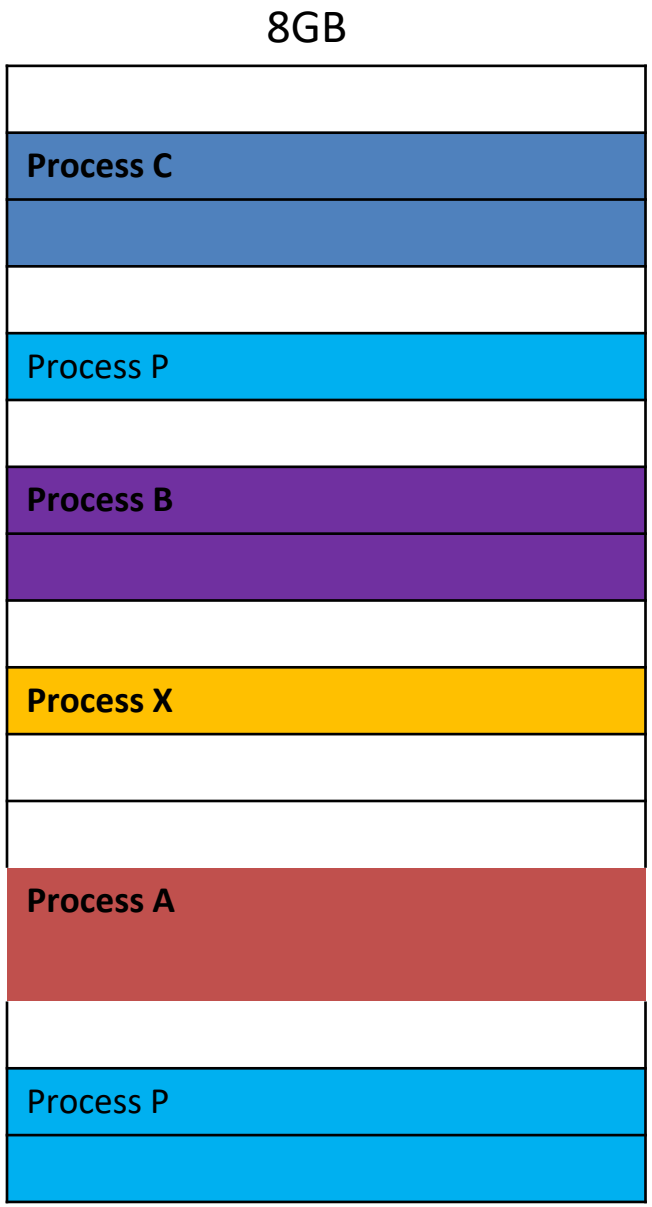


Memory management

Virtual Memory uses secondary storage to give programs the illusion that they have infinite storage

We split the process into smaller, fixed-size, **pages**. Load pages into memory only when needed

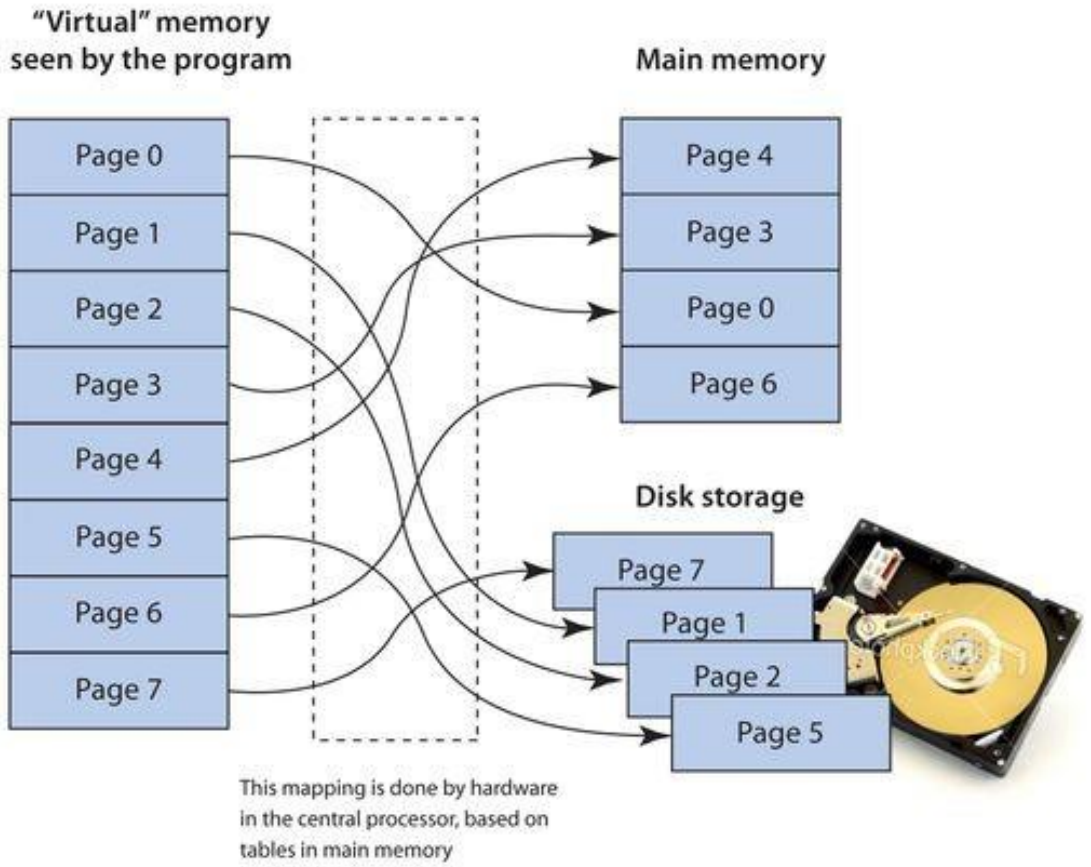




Memory management

Virtual Memory uses secondary storage to give programs the illusion that they have infinite storage

We split the process into smaller, fixed-size, **pages**. Load pages into memory only when needed

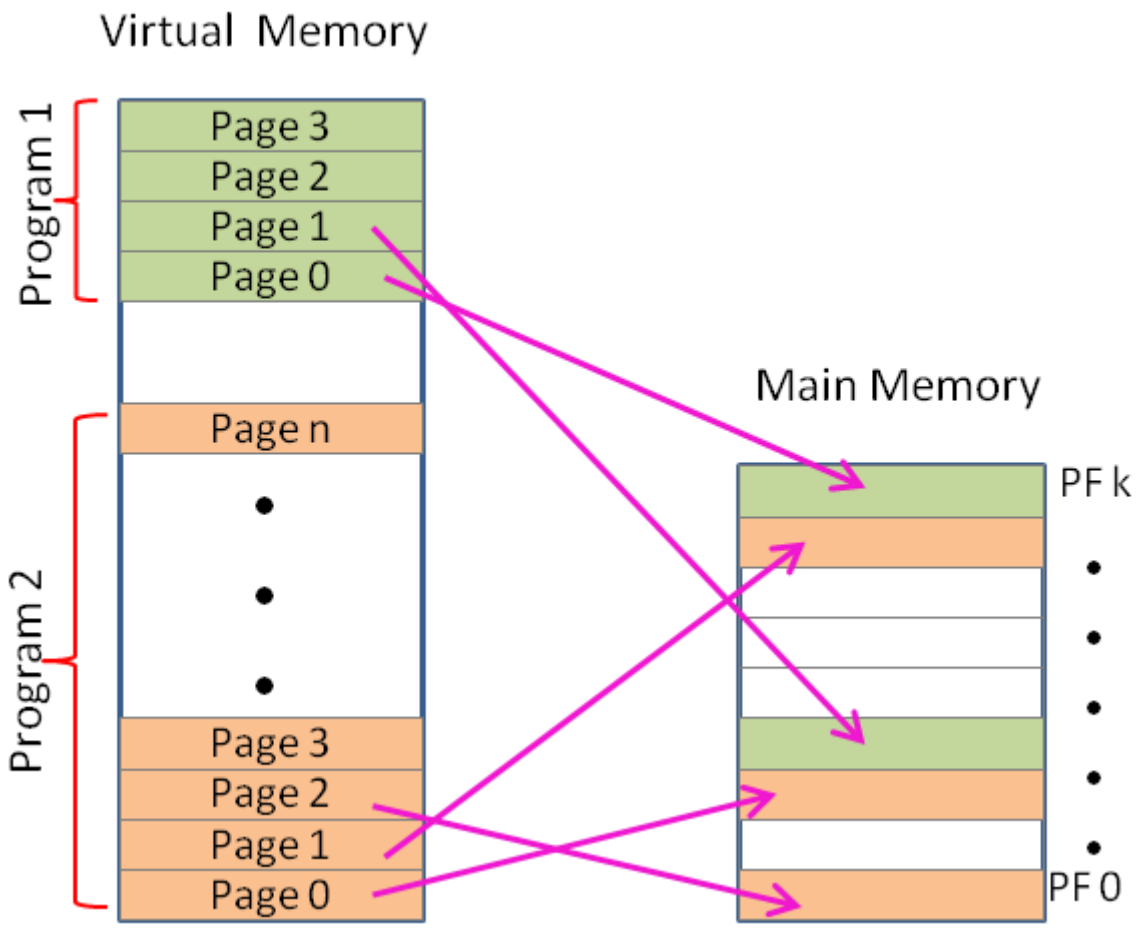


Constantly swapping stuff in and out of main memory

Memory management

Virtual Memory uses secondary storage to give programs the illusion that they have infinite storage

We split the process into smaller, fixed-size, **pages**. Load pages into memory only when needed

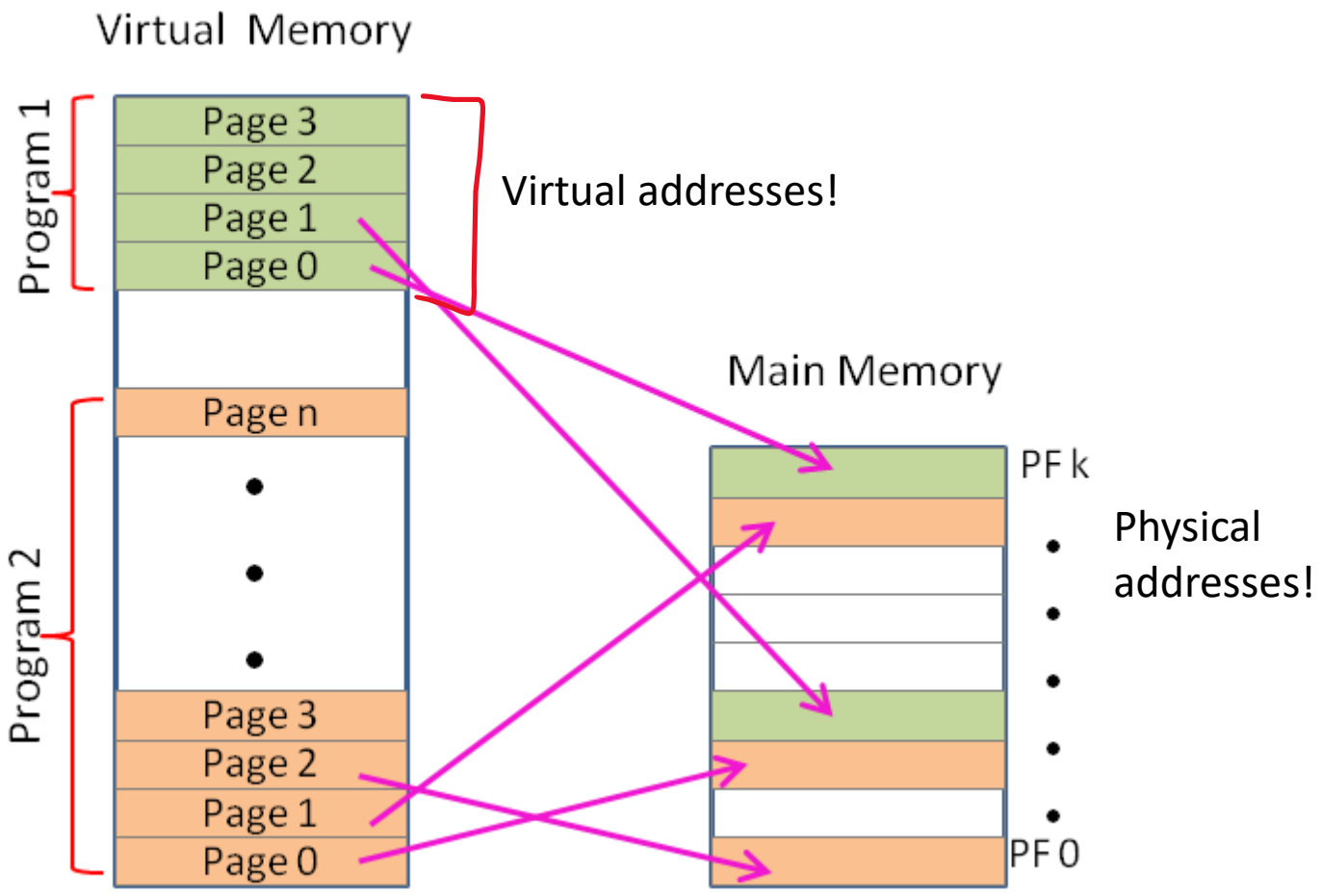


A process in memory is not contiguous

Memory management

Virtual Memory uses secondary storage to give programs the illusion that they have infinite storage

We split the process into smaller, fixed-size, **pages**. Load pages into memory only when needed



A process in memory is not contiguous

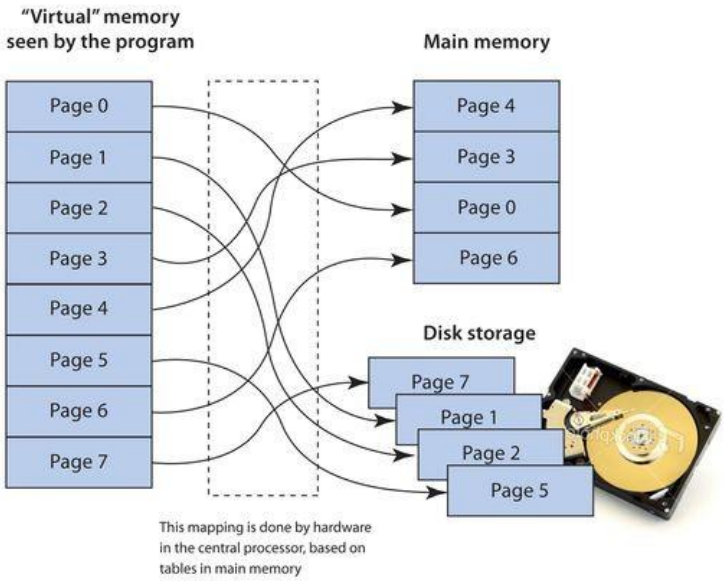
In probe.c, we are seeing virtual addresses!

Internal fragmentation vs external fragmentation

OS Review

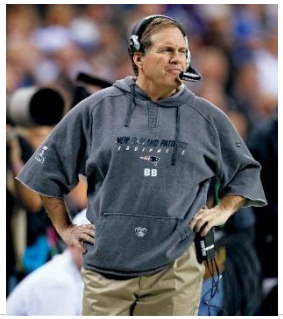
Memory Manager

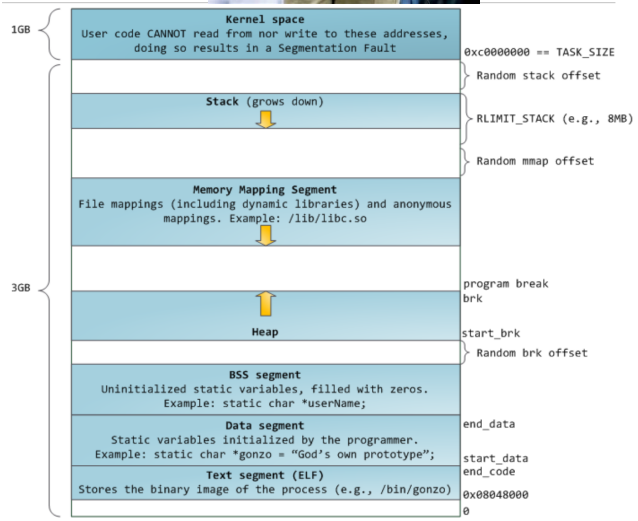
- Manages how physical memory is utilized



Process Manager

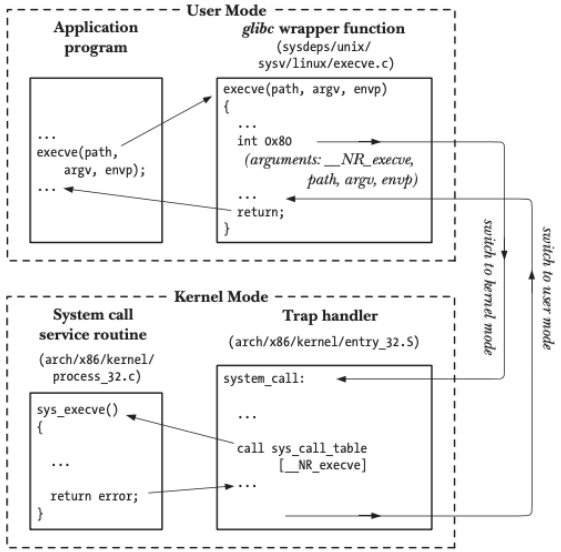
- Manages how processes are structured and how to handle many processes running at once





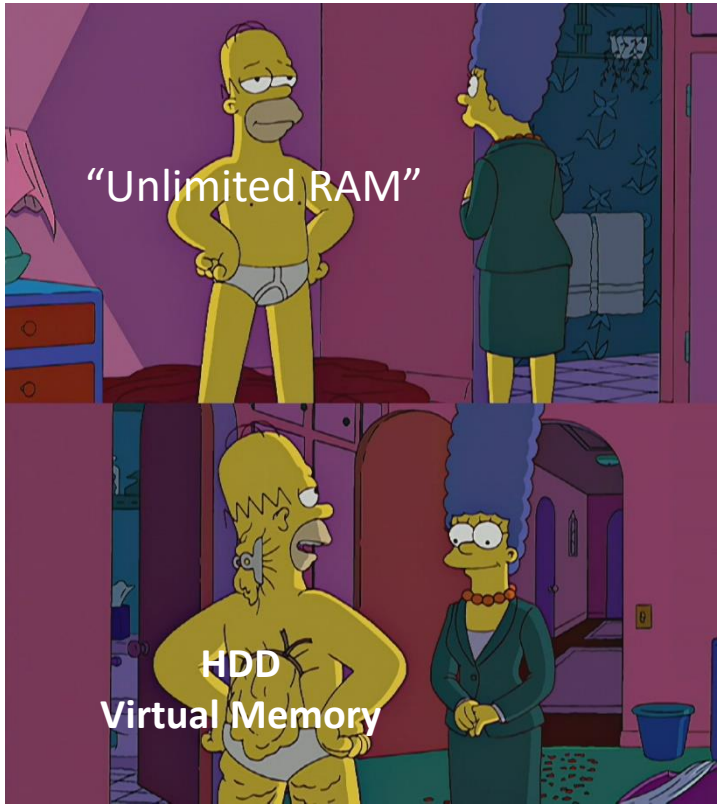
Interface Manager

- Manages communication between apps and hardware



Illusion Manager

- Gives applications the illusion that they have infinite storage and resources



Traffic Manager

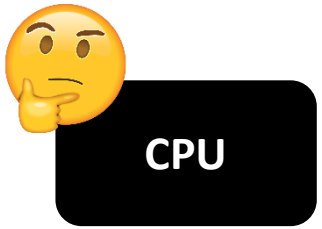
- Manages which programs should be executed by the CPU

Process A *(Ready)*

Process B *(Urgent)*

Process C *(Ready)*

Process D *(Blocked)*



The jobs of an Operating System

1. **Process Manager**
   "The Coach"
2. **Interface Manager**
   "The Bouncer"
3. **Memory Manager**
   "The Farmer"
4. **Traffic Manager**
   "The Judge"
5. **Illusion Manager**
   "The Illusionist"