CSCI 476: Computer Security

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

Reese Pearsall

Spring 2023

Announcements

Gerard is here

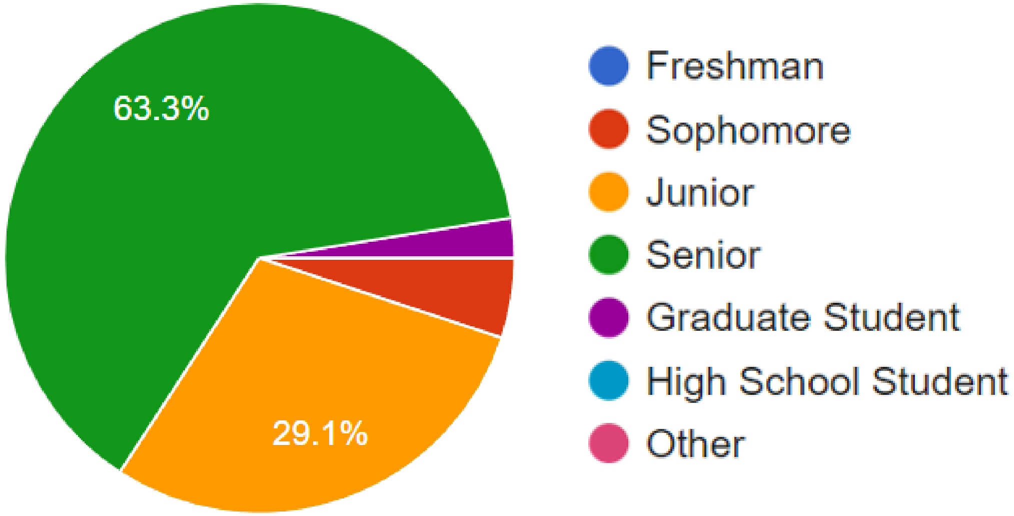
Lab 0 due on Sunday 1/29 @ 11:59 PM

No in-person lecture next Wednesday (2/1)

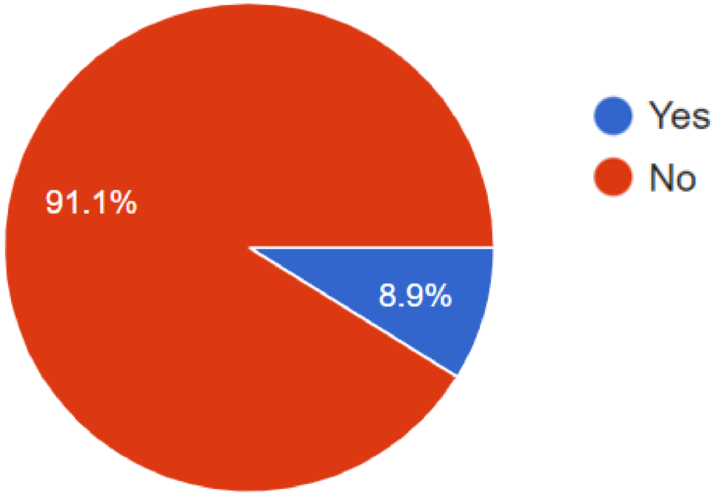
- I’ll post an asynchronous lecture video to the course web page

Course Questionnaire Results

Class?

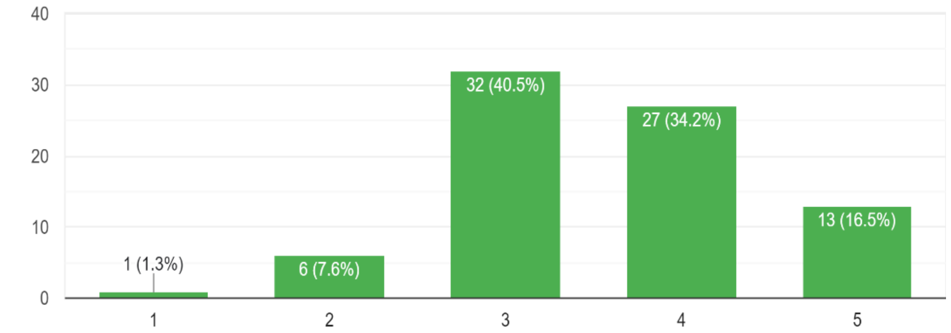


Have you taken Operating Systems (CSCI 460)



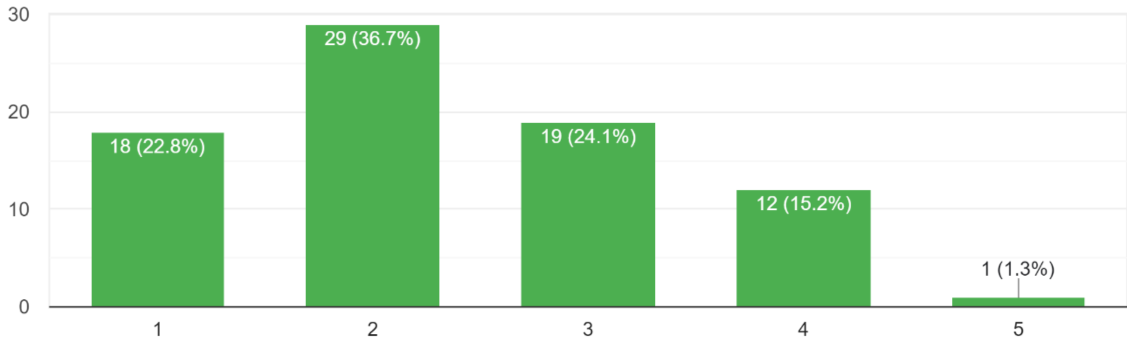
How comfortable are you C?

79 responses



How comfortable are you with reading assembly code?

79 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”

Course Questionnaire Results

“I am a big procrastinator”

“Im interested in learning about penetration testing”

“This class seems relevant to my career path”

Course Questionnaire Results

“I am a big procrastinator”

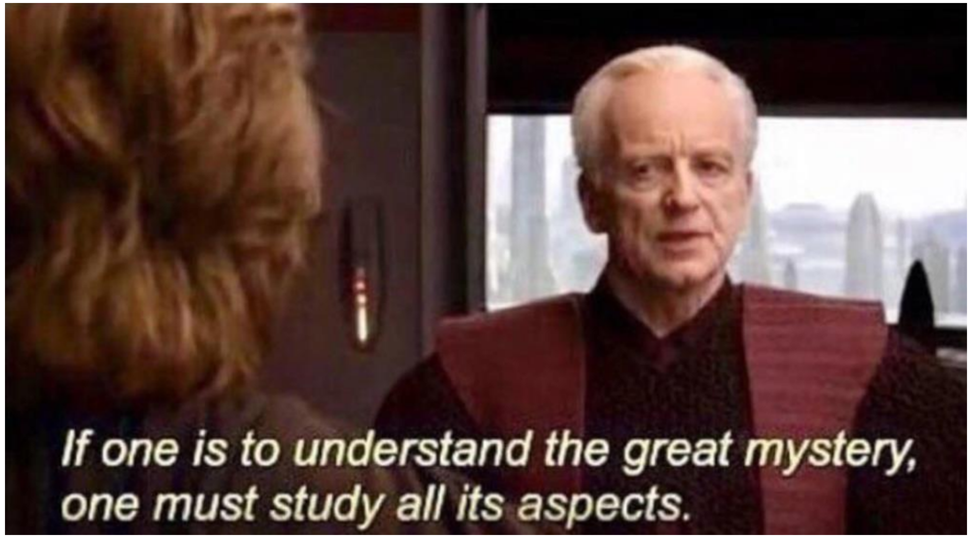
“Im interested in learning about penetration testing”

“This class seems relevant to my career path”

“The best cereal is *just milk*”

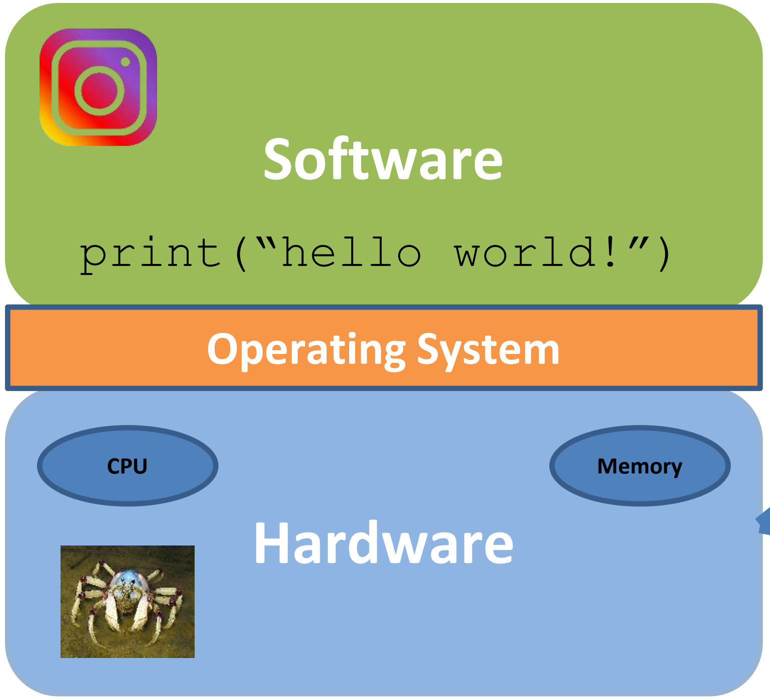
To understand the technical aspects of security, we must have a good understanding of how ~~computers~~ work

operating systems



If one is to understand the great mystery, one must study all its aspects.

The Operating System



Software

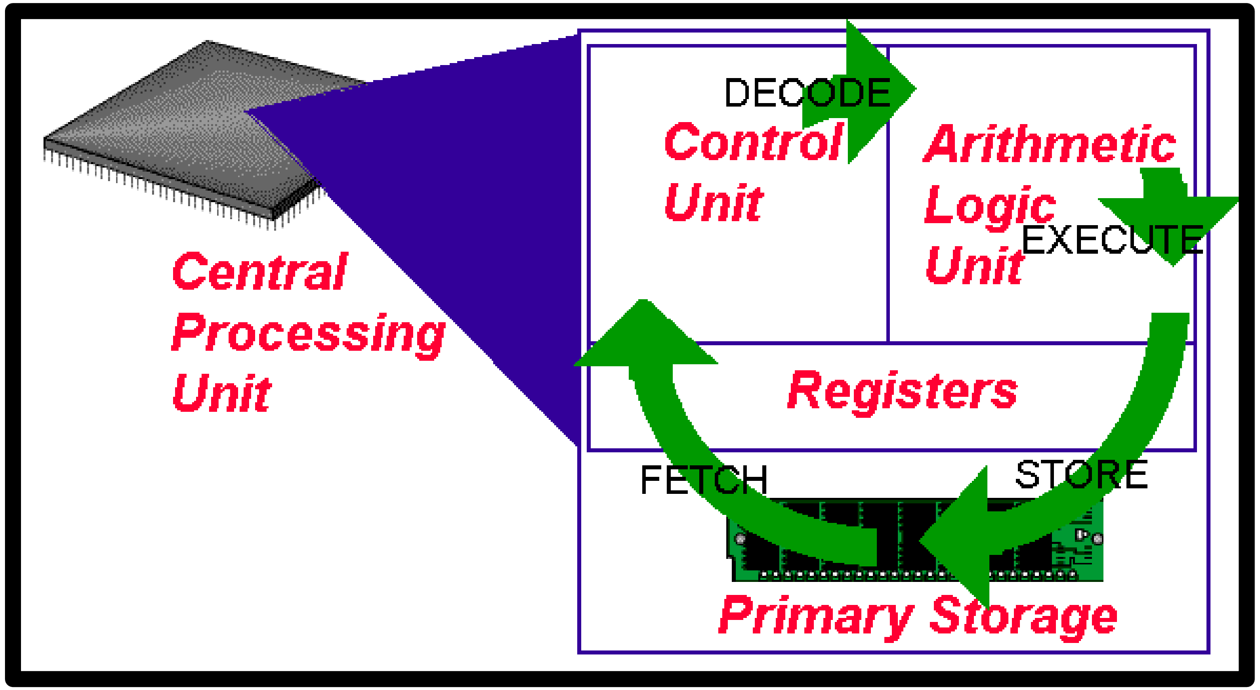
print(“hello world!”)

Operating System

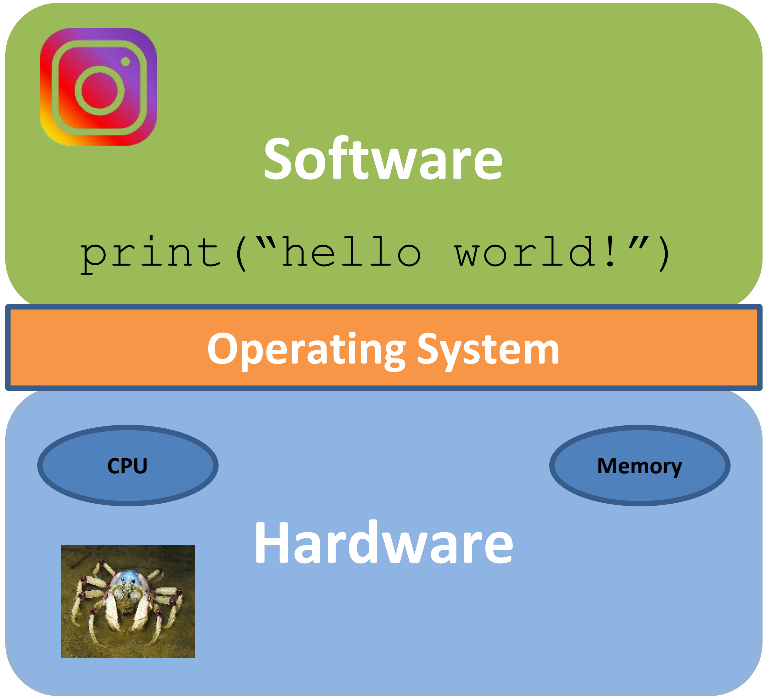
CPU

Memory

Hardware



The Operating System



Software

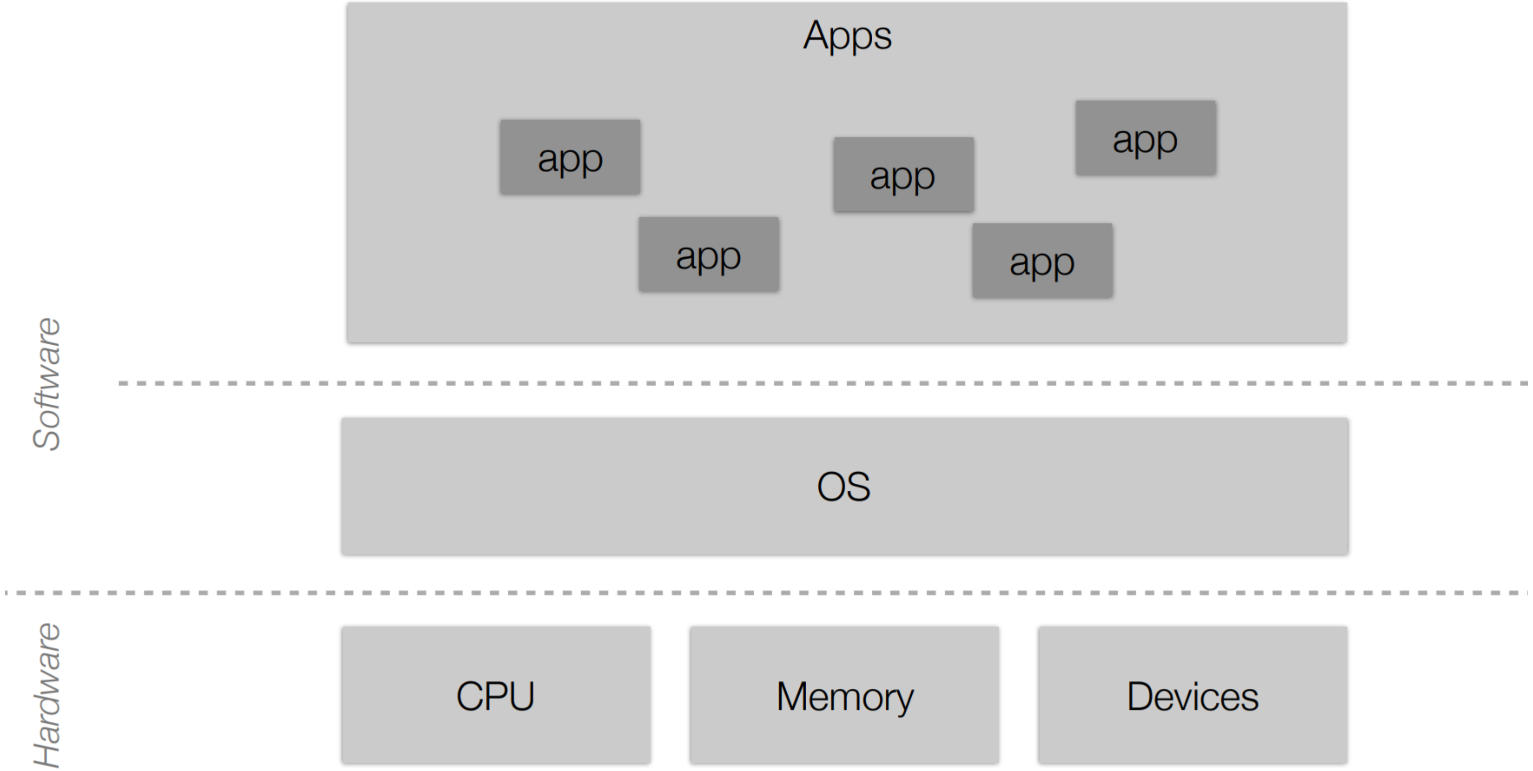
```
print(“hello world!”)
```

Operating System

CPU

Memory

Hardware



Apps

app

app

app

app

app

Software

OS

Hardware

CPU

Memory

Devices

The jobs of an Operating System

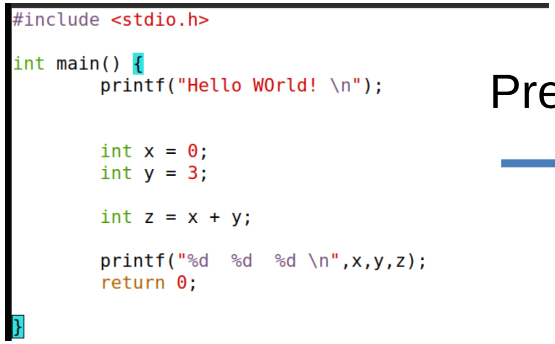
1. **Process Manager**
   “The Coach”
2. **Interface Manager**
   “The Bouncer”
3. **Memory Manager**
   “The Farmer”
4. **Traffic Manager**
   “The Judge”
5. **Illusion Manager**
   “The Illusionist”

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”

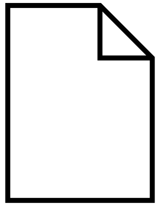
This will be the focus of today’s lecture

Source code to binary



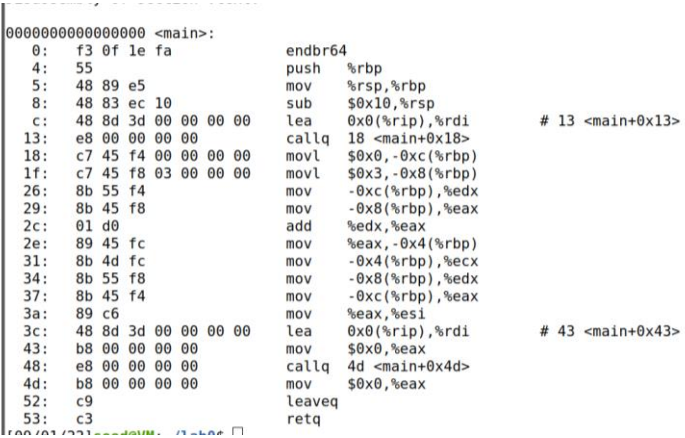
```
#include <stdio.h>

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

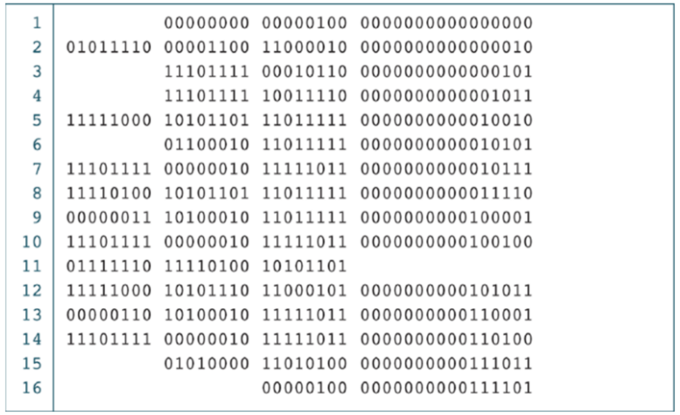
Compiler



```
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
```

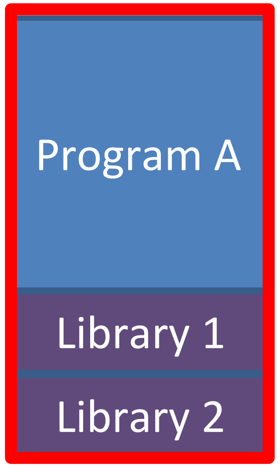
- Converted to assembly code
- .s file

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

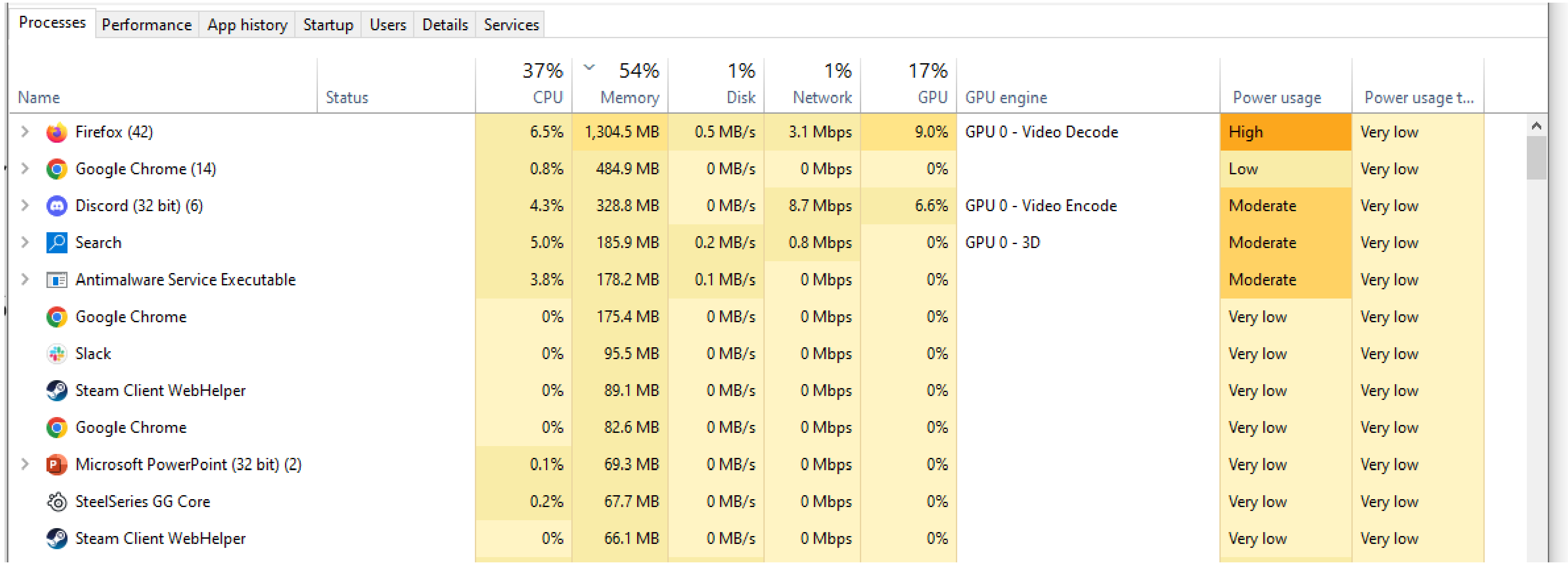
./hello_world

**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 running program on a computer



		37%	54%	1%	1%	17%			
Name	Status	CPU	Memory	Disk	Network	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 running program on a computer

All processes have the following data while they are running:

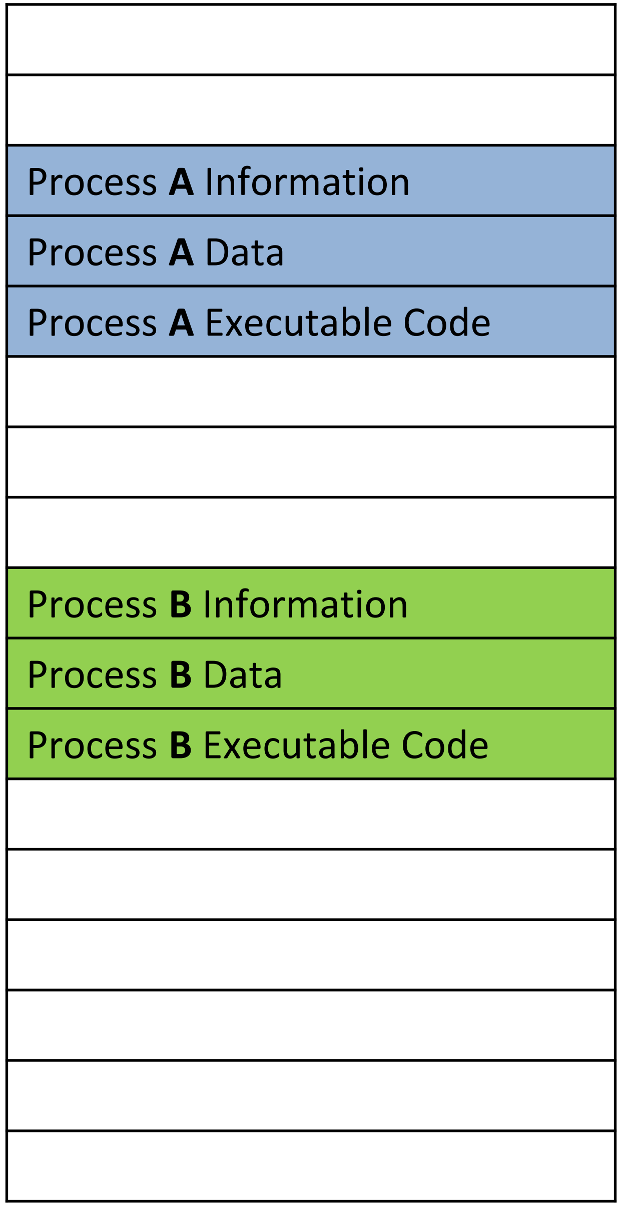
1. Executable Code

2. Associated Data

3. Execution Context/Bookkeeping information

   (info that the OS needs to handle the process)

Main Memory



Process **A** Information

Process **A** Data

Process **A** Executable Code

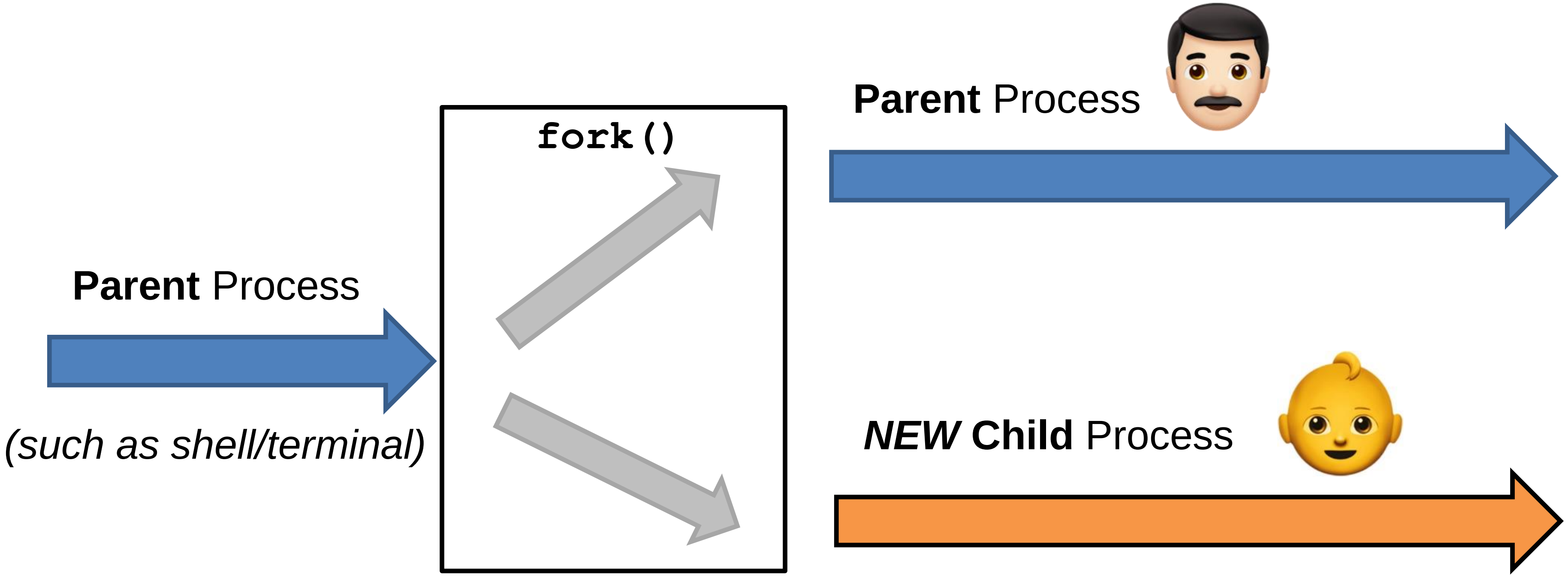
Process **B** Information

Process **B** Data

Process **B** Executable Code

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

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



Parent Process

(*such as shell/terminal)*

`fork()`

Parent Process

***NEW* Child** Process

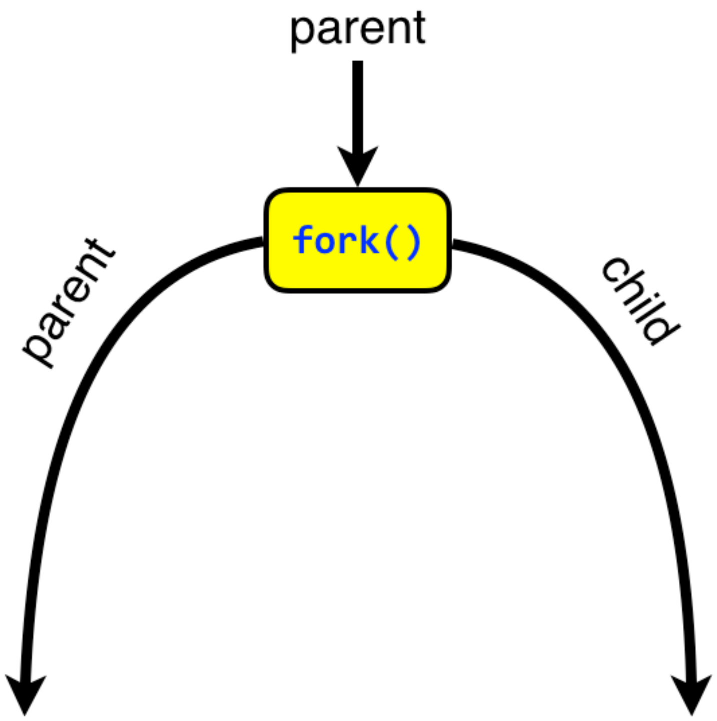
`fork()` duplicates a process so that instead of one process, you get two!

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!

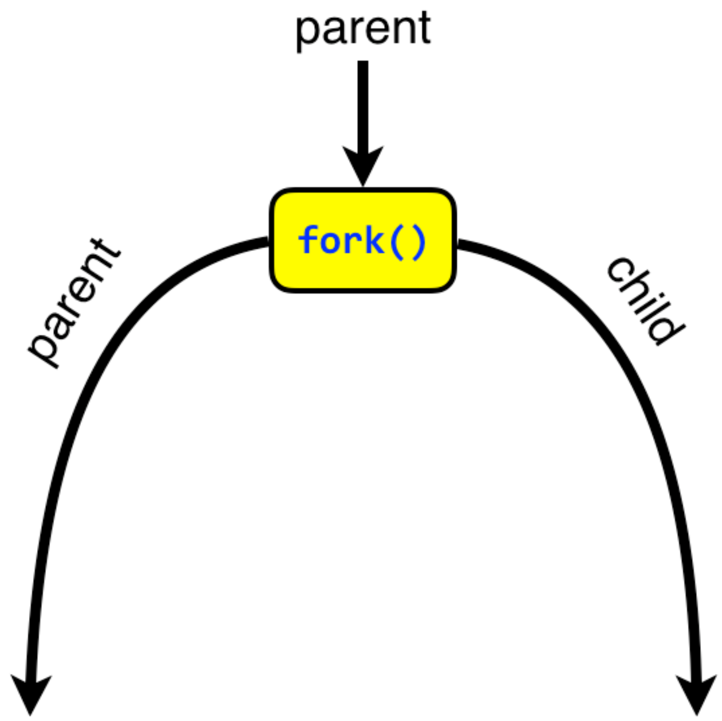
How can we tell the parent and child apart?

```
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!

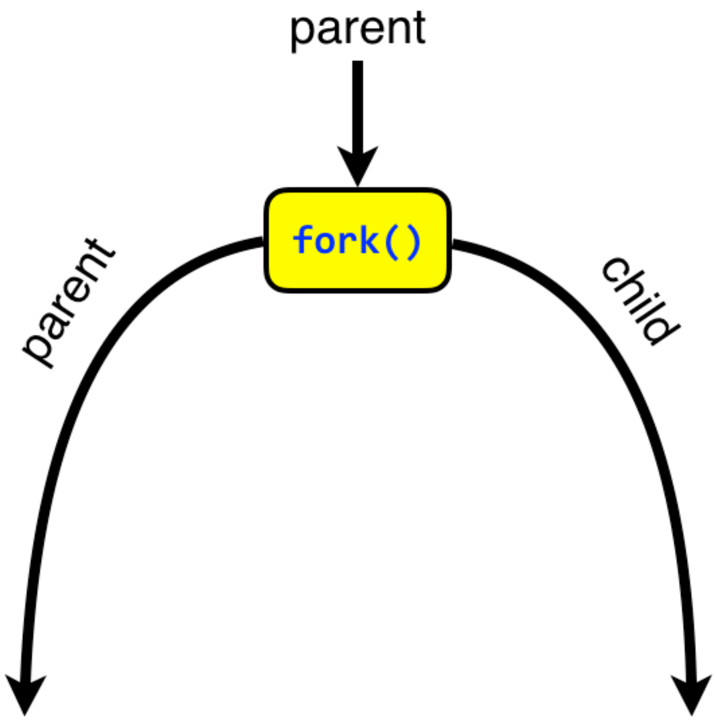
How can we tell the parent and child apart?

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

```
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;
}
```

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!

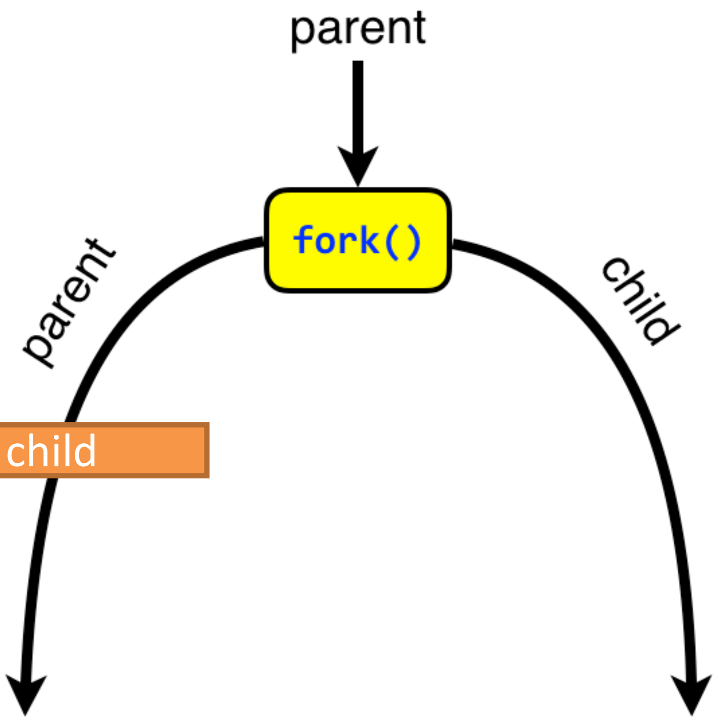
How can we tell the parent and child apart?

```
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()`**!

child

parent



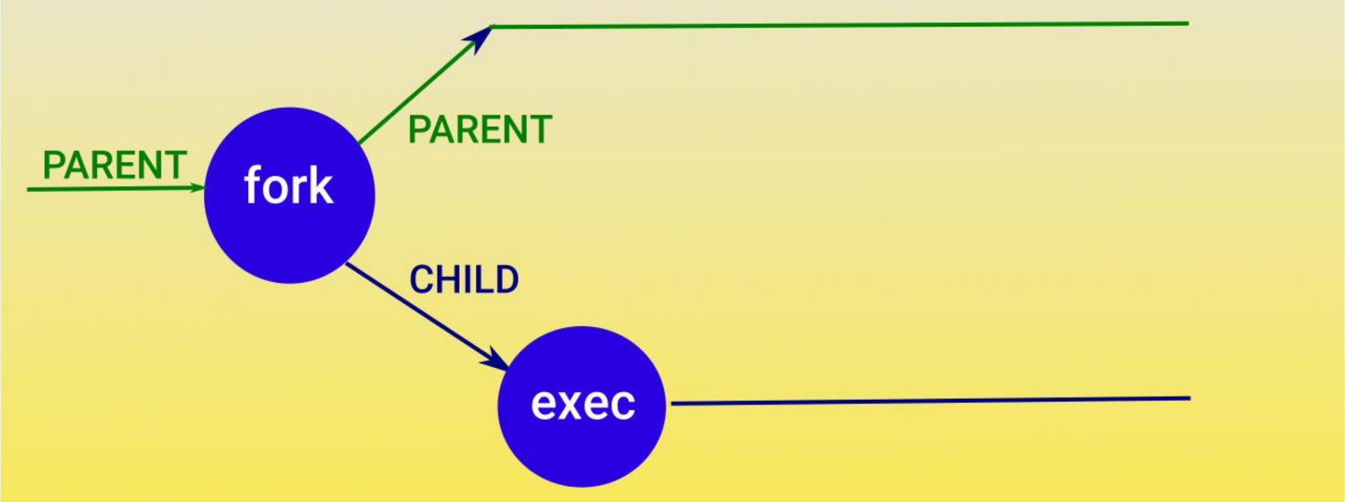
3. **`fork()`** always returns 0 for the child process, so the child process will execute the code in the if statement

Demo?

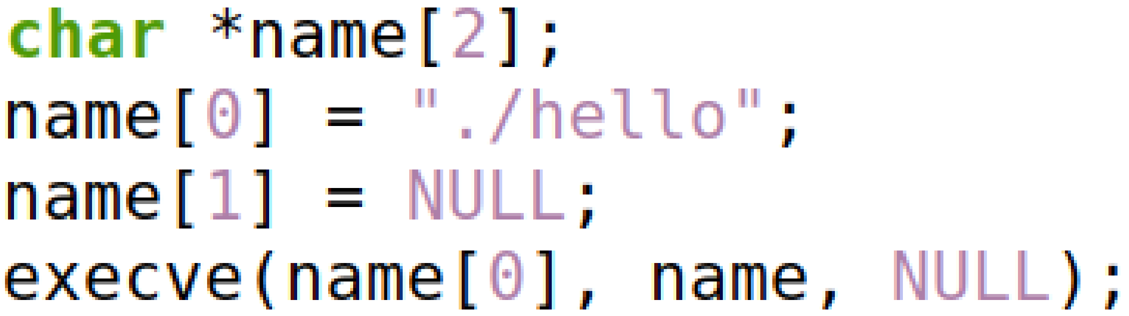
fork1.c

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



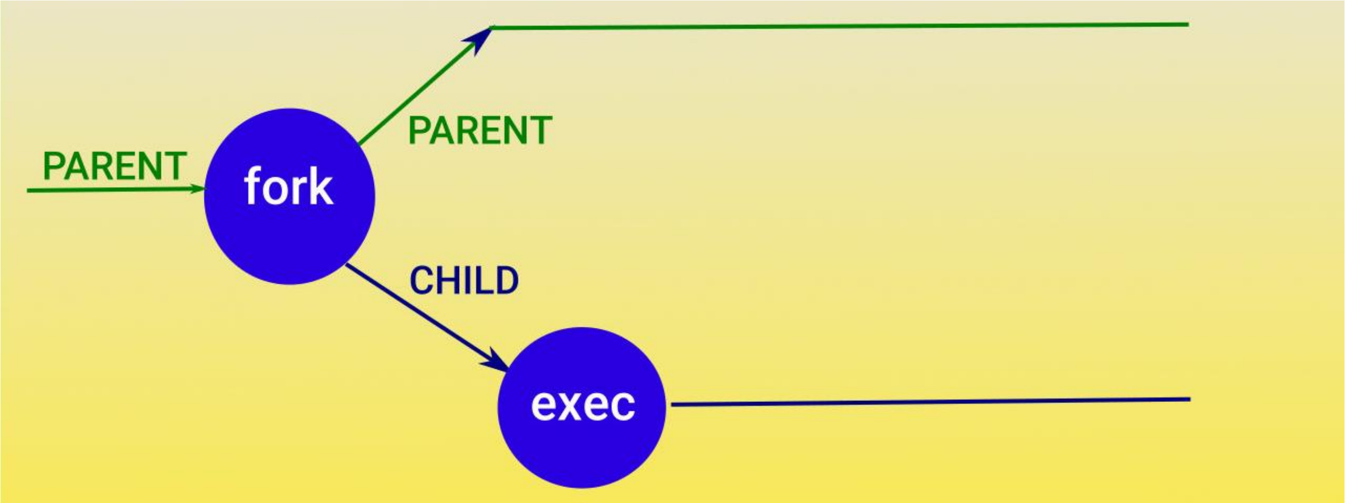
There are many different forms of the **`exec()`** function call



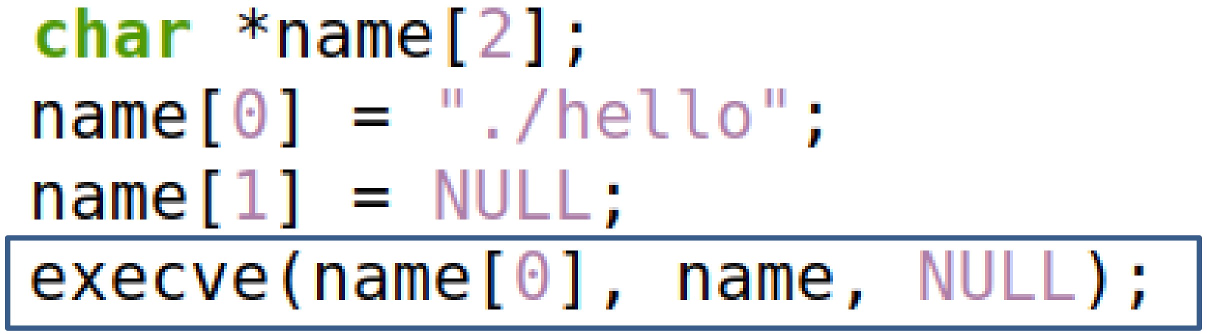
```
char *name[2];
name[0] = "./hello";
name[1] = NULL;
execve(name[0], name, NULL);
```

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



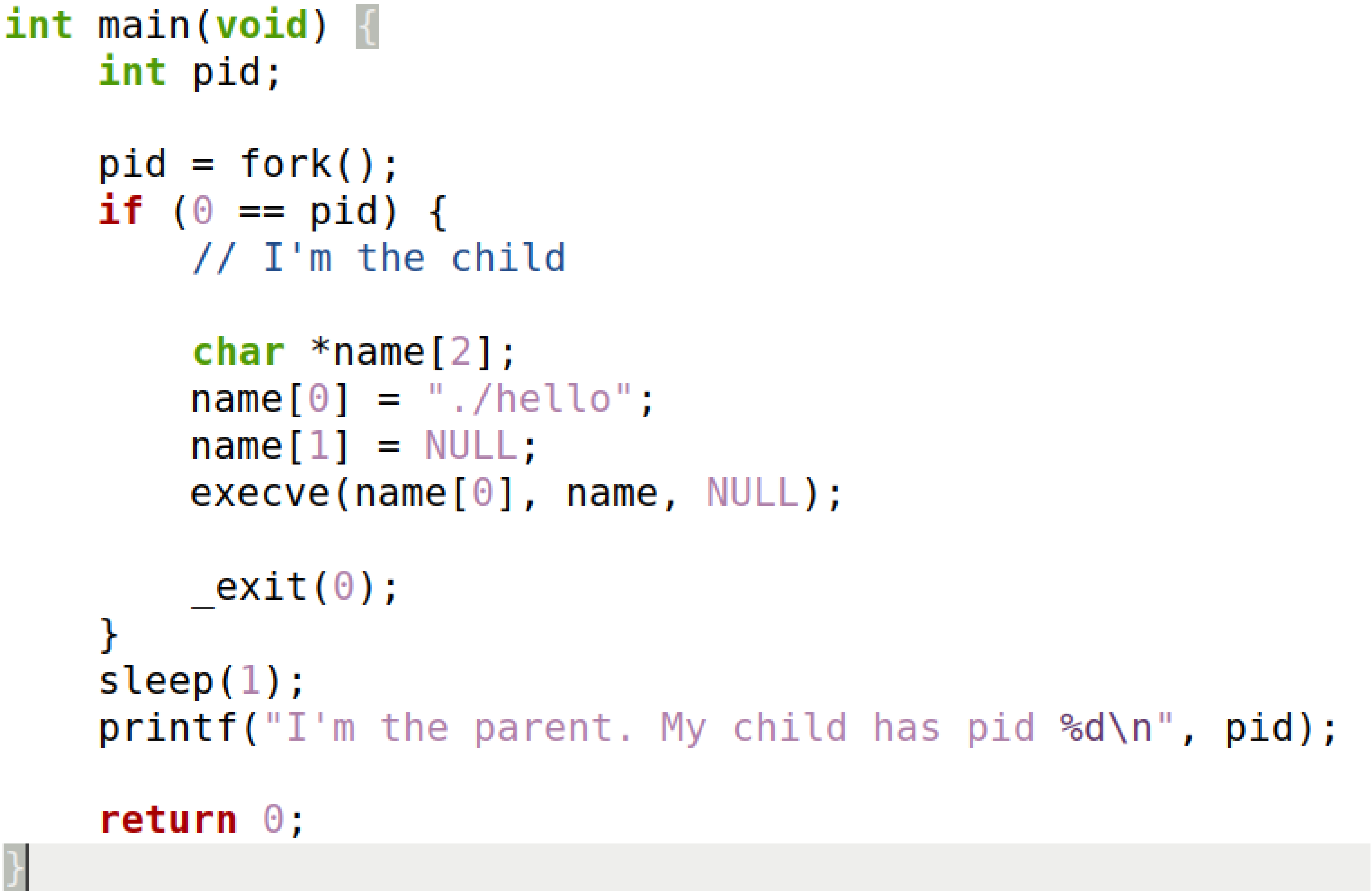
There are many different forms of the **`exec()`** function call



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

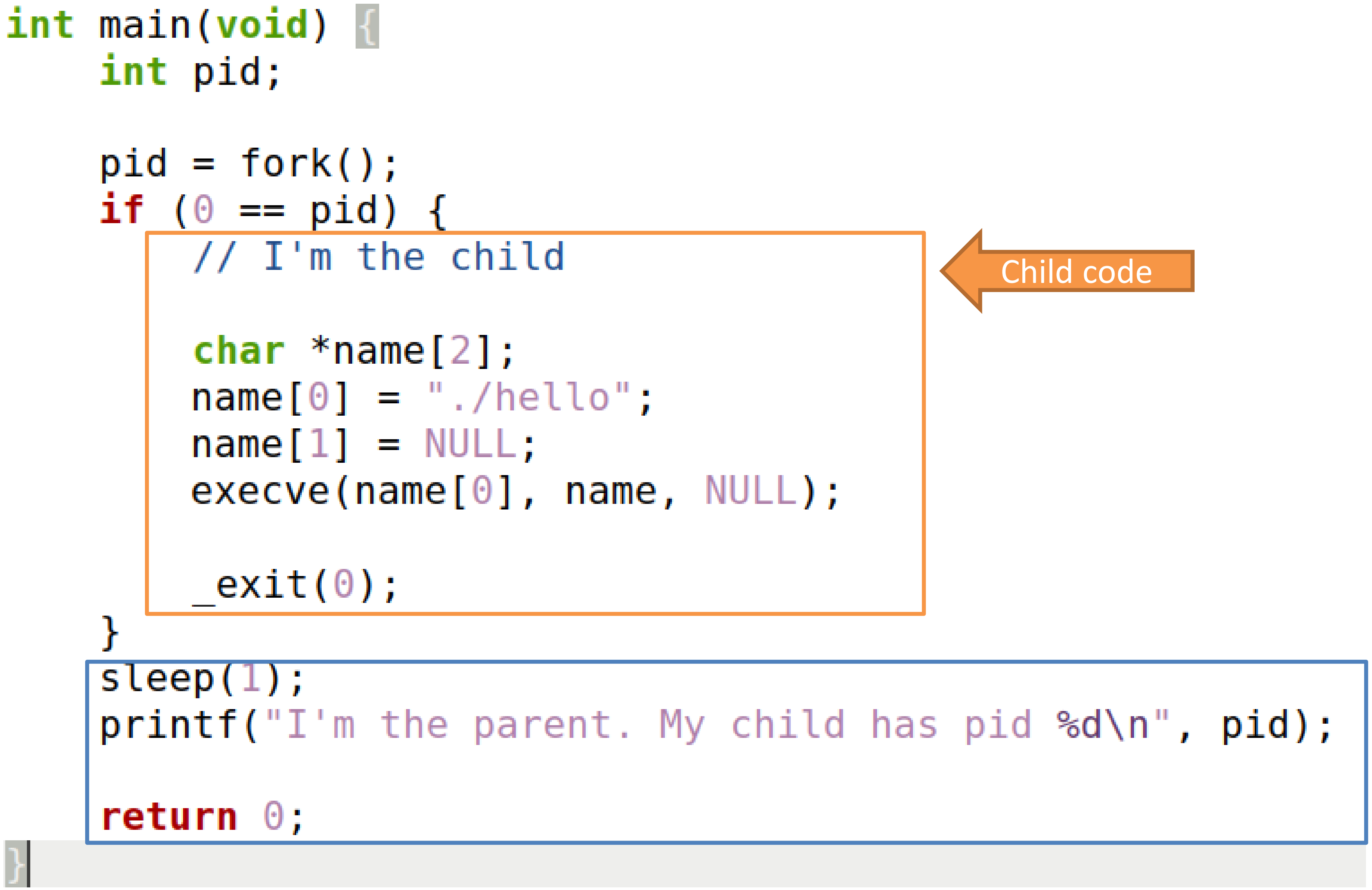
This will invoke a program called `hello`

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()

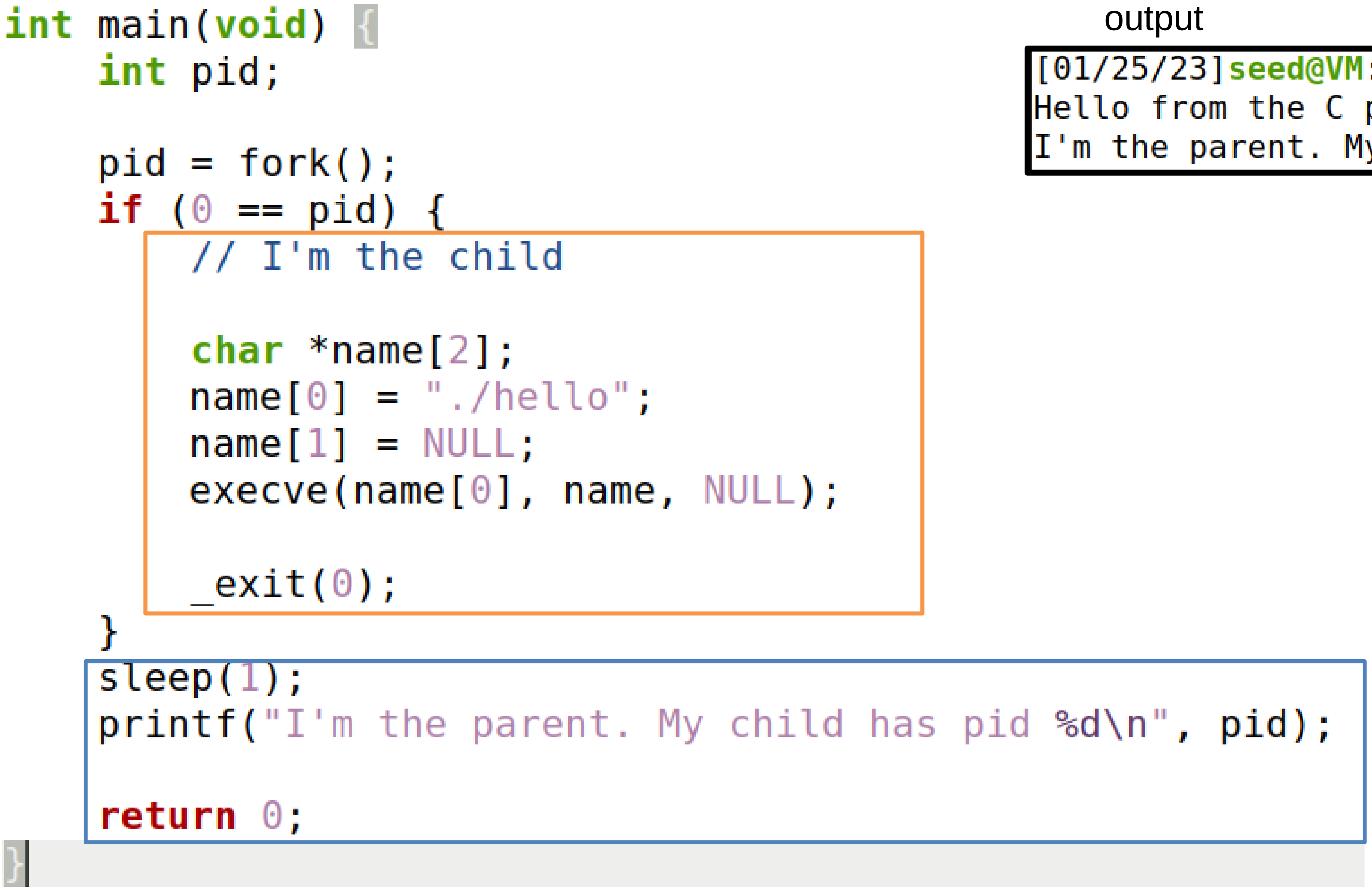


```
int main(void) {
  int pid;

  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

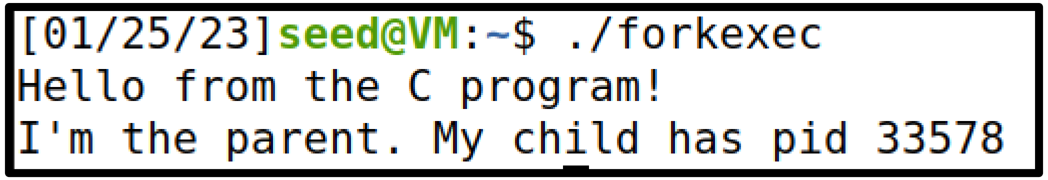
Parent code

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?

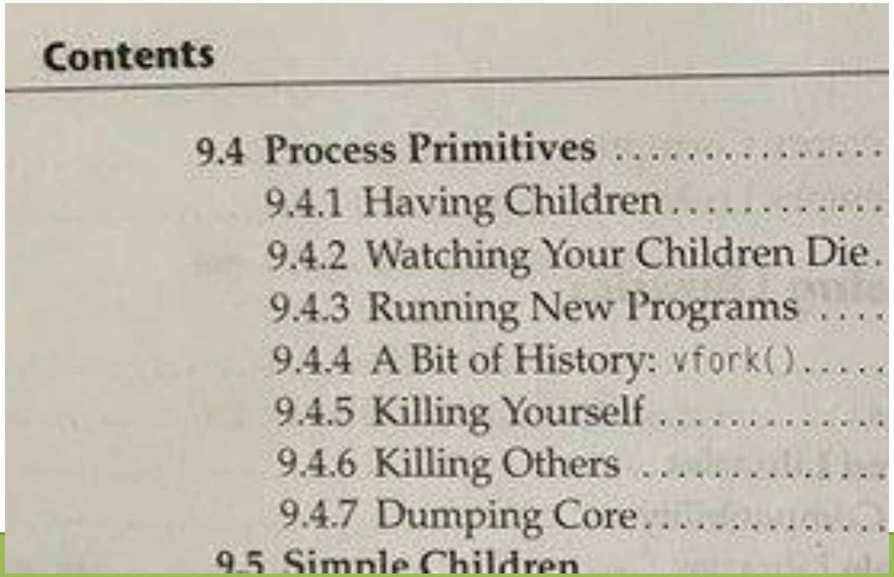
forkandexec.c

Tl;dr

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

fork() is used to create a new process

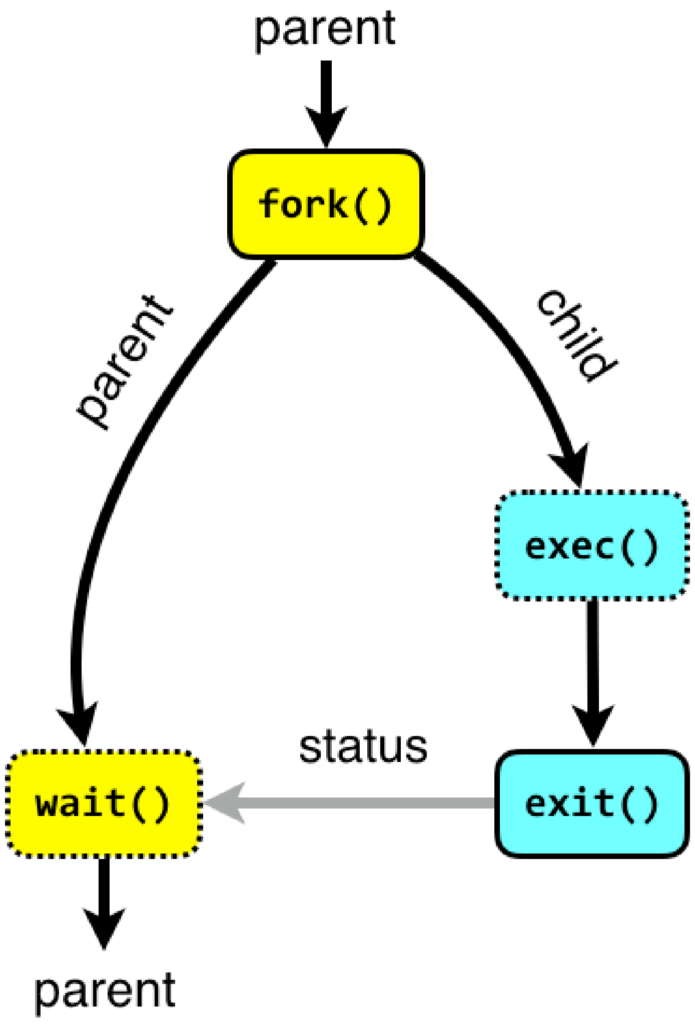
- The parent process is typically the shell/terminal, and waits for the child process to finish
- The child process runs **exec()** to run our program

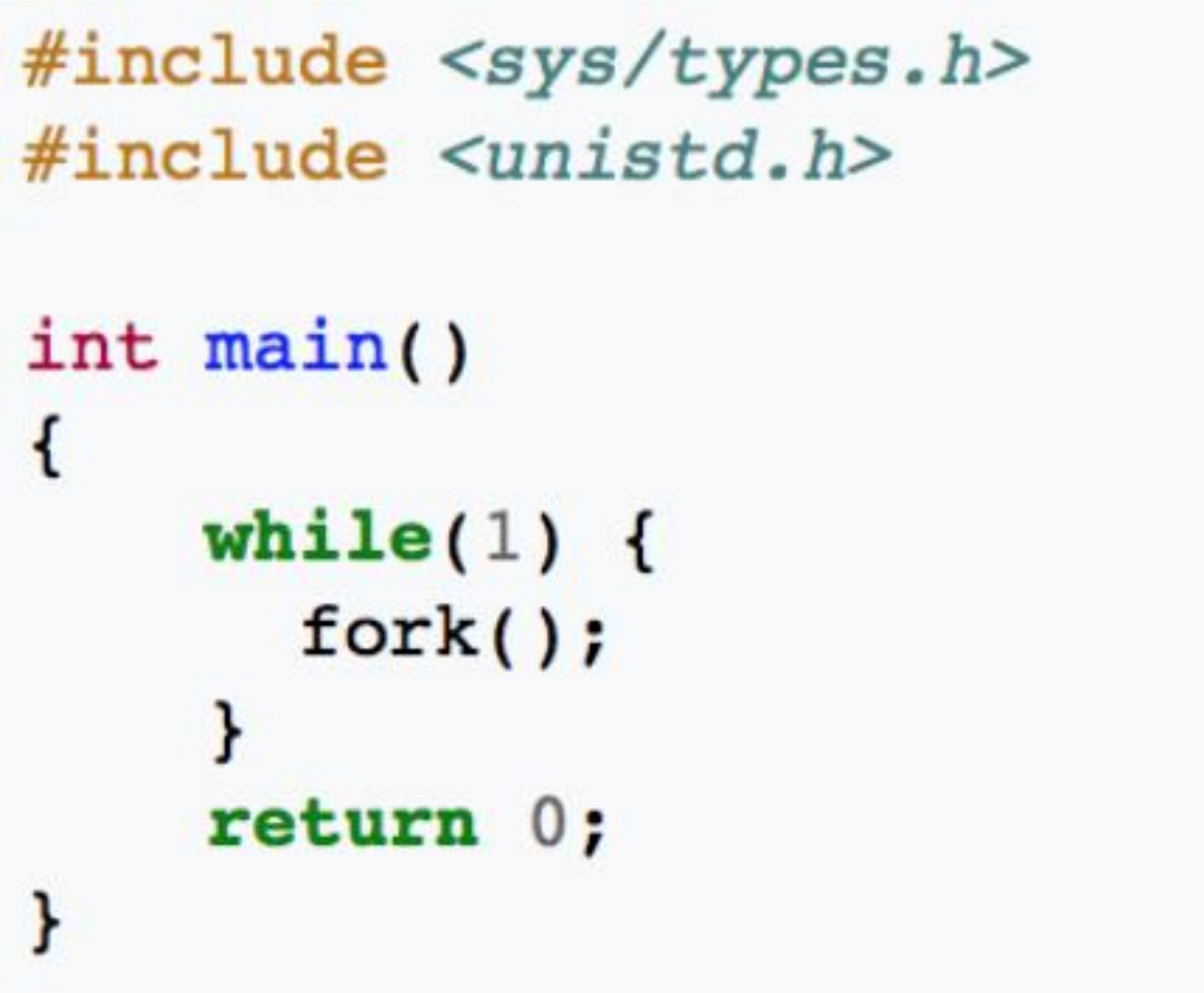


Contents

9.4 Process Primitives . . . .
9.4.1 Having Children . . . .
9.4.2 Watching Your Children Die.
9.4.3 Running New Programs . . . .
9.4.4 A Bit of History: vfork() . . . . .
9.4.5 Killing Yourself . . . .
9.4.6 Killing Others . . . .
9.4.7 Dumping Core . . . .
9.5 Simple Children

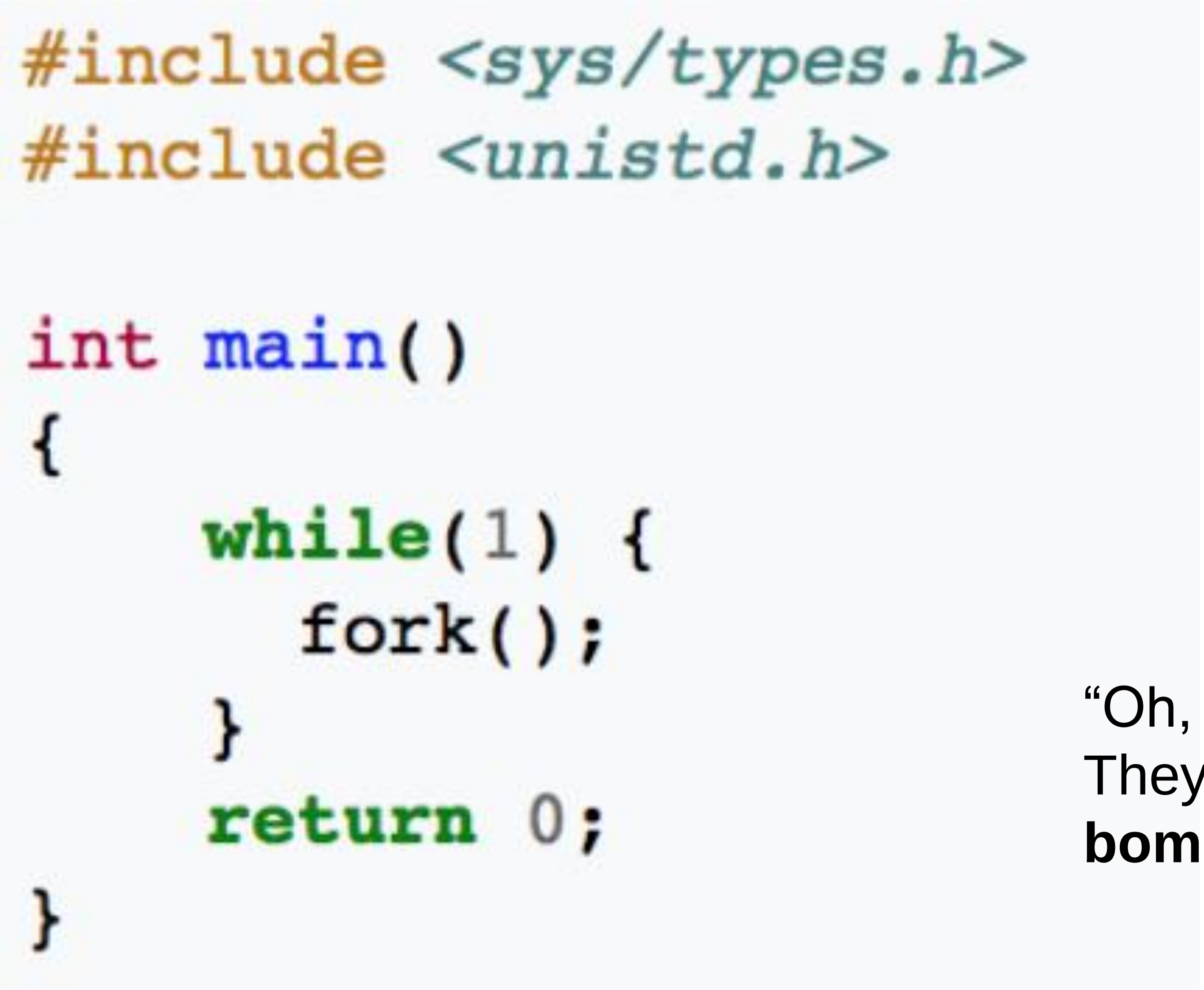
you can kill children with the `kill()` function or `kill` command





```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 running program on a computer

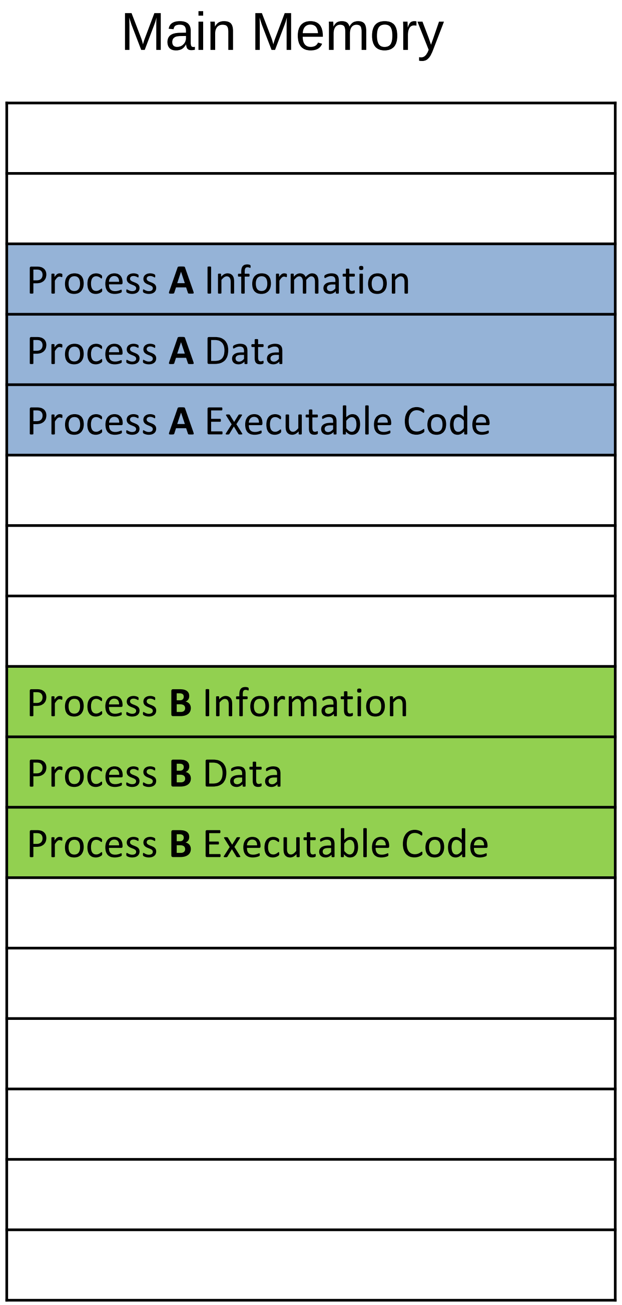
All processes have the following data while they are running:

1. Executable Code

2. Associated Data

3. Execution Context/Bookkeeping information

   (info that the OS needs to handle the process)



Main Memory

Process **A** Information

Process **A** Data

Process **A** Executable Code

Process **B** Information

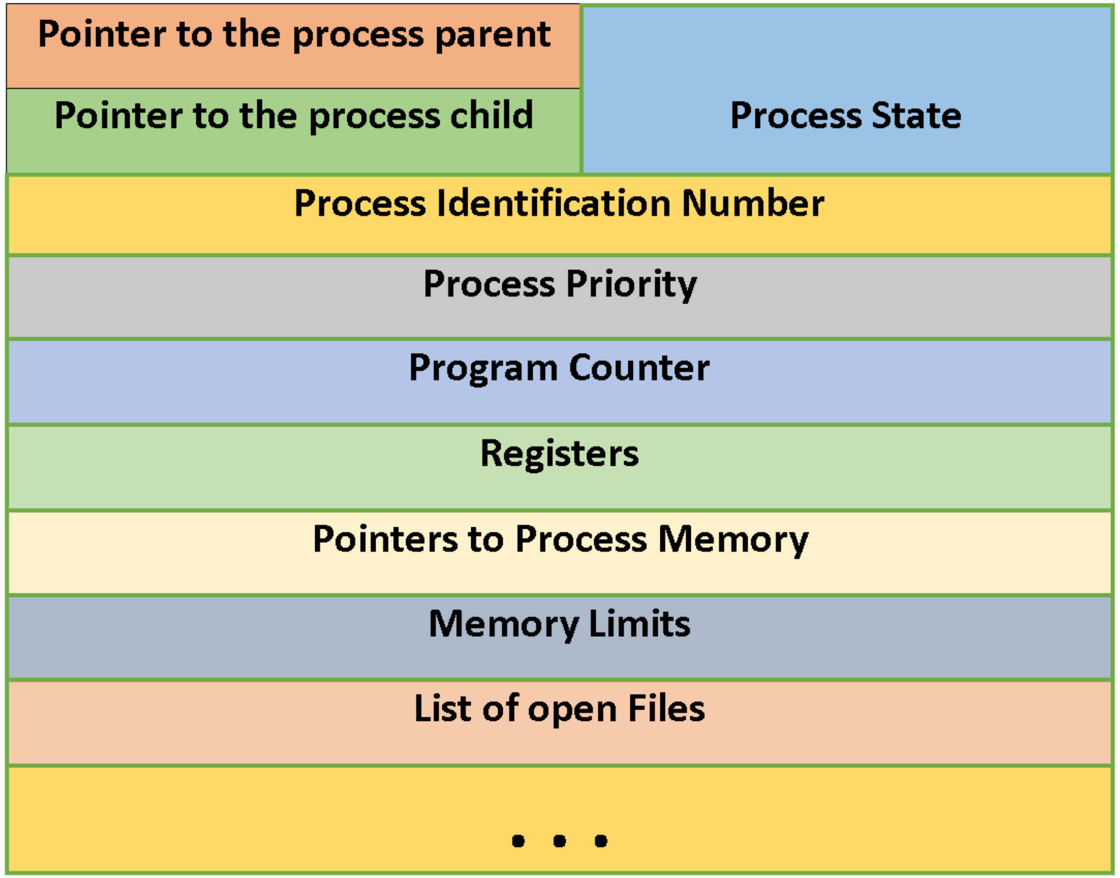
Process **B** Data

Process **B** Executable Code

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:



Pointer to the process parent

Pointer to the process child

Process State

Process Identification Number

Process Priority

Program Counter

Registers

Pointers to Process Memory

Memory Limits

List of open Files

. . .

Created by Notes Jam

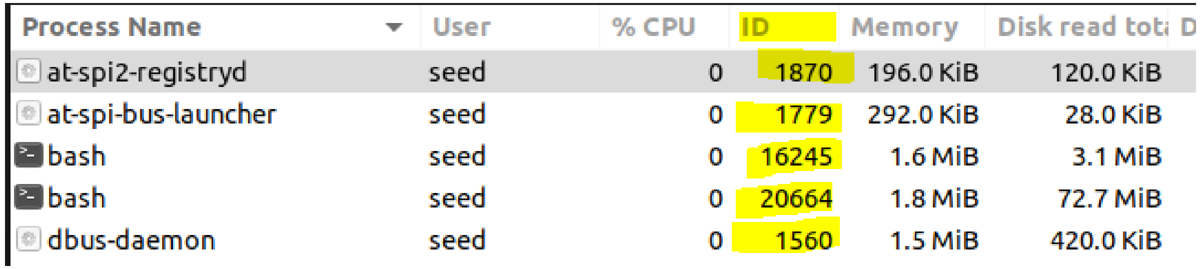
3. Execution Context/Bookkeeping information

- Each process has a **Process Control Block (PCB)**

  → Simply just a data structure that holds information

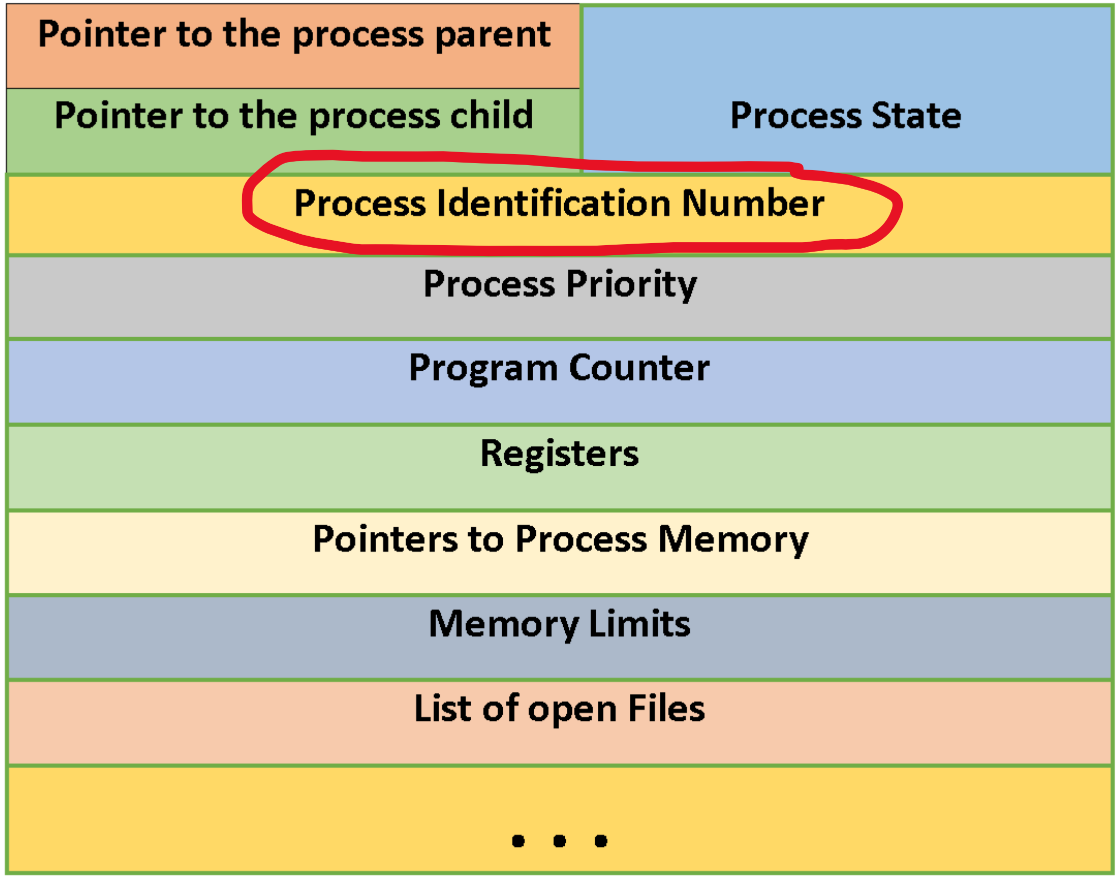
  → The name of this varies by OS

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:



Pointer to the process parent

Pointer to the process child

Process State

Process Identification Number

Process Priority

Program Counter

Registers

Pointers to Process Memory

Memory Limits

List of open Files

. . .

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

Created by Notes Jam

3. Execution Context/Bookkeeping information

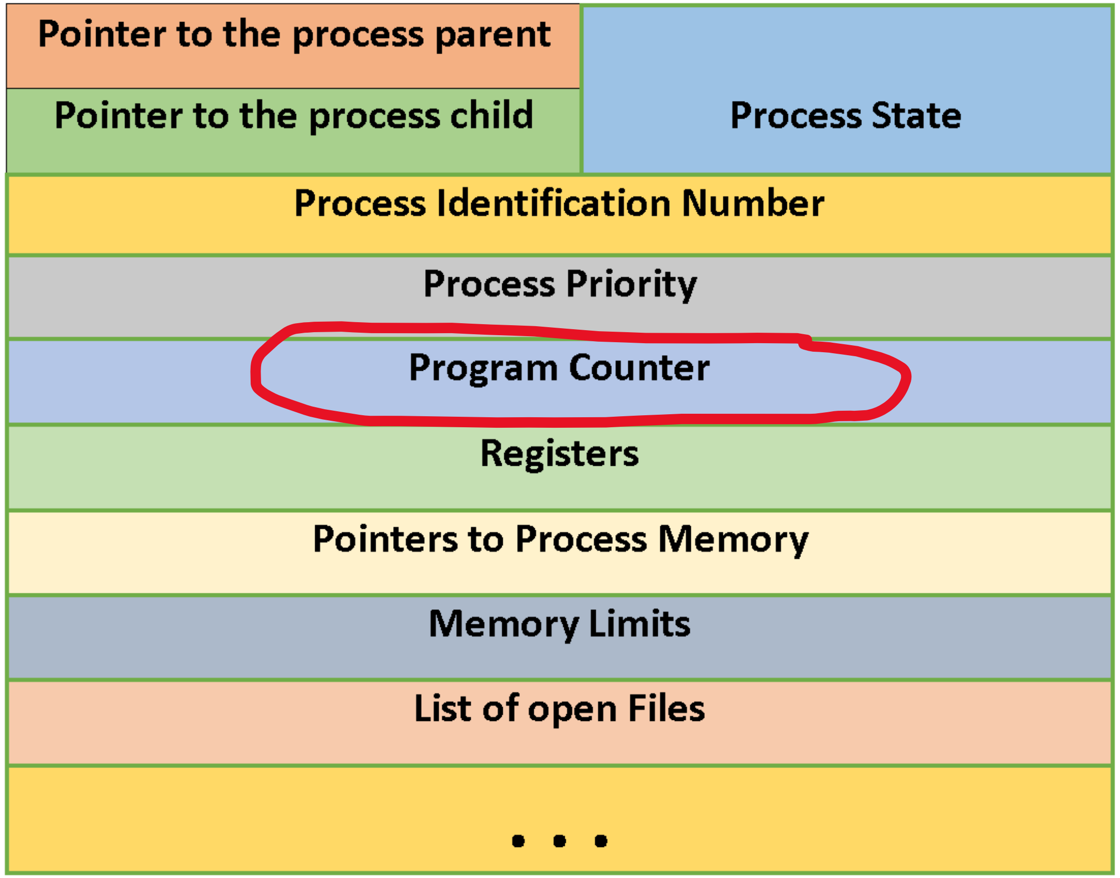
- Each process has a **Process Control Block (PCB)**

  → 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:



Pointer to the process parent

Pointer to the process child

Process State

Process Identification Number

Process Priority

Program Counter

Registers

Pointers to Process Memory

Memory Limits

List of open Files

• • •

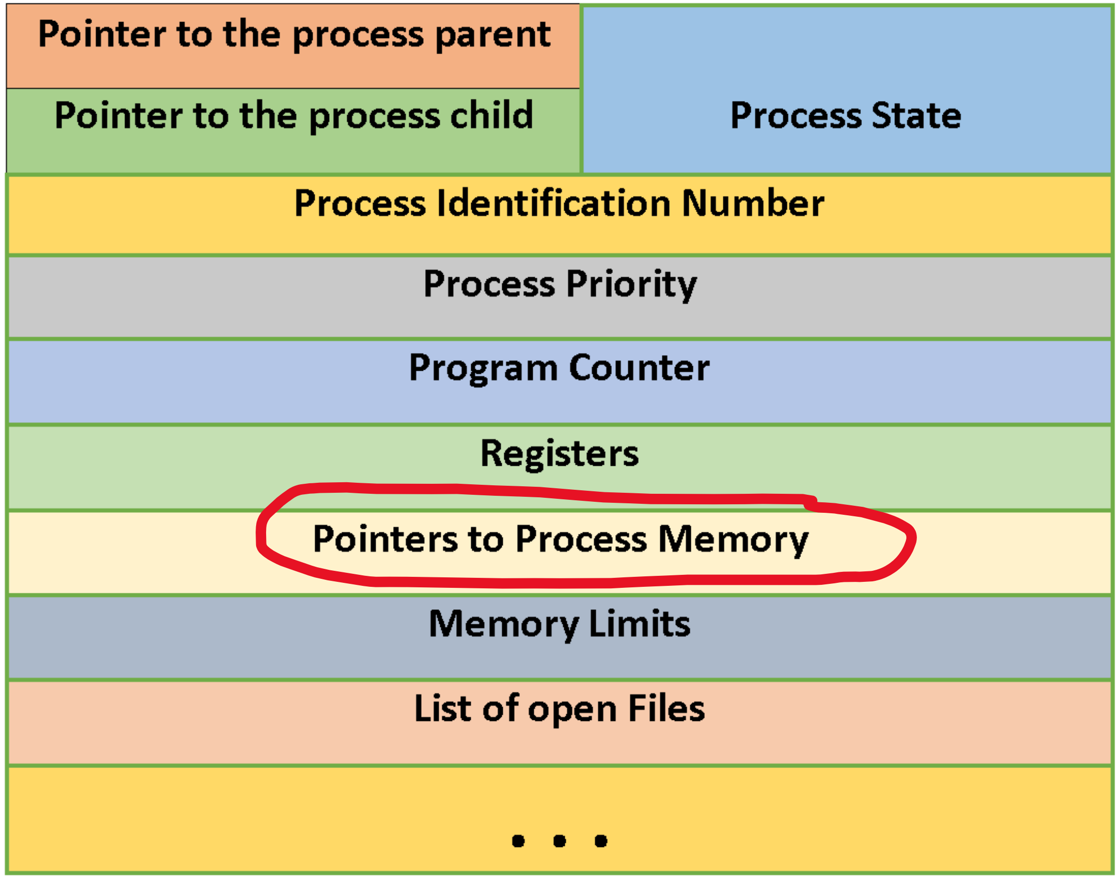
Created by Notes Jam

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:



Pointer to the process parent

Pointer to the process child

Process State

Process Identification Number

Process Priority

Program Counter

Registers

Pointers to Process Memory

Memory Limits

List of open Files

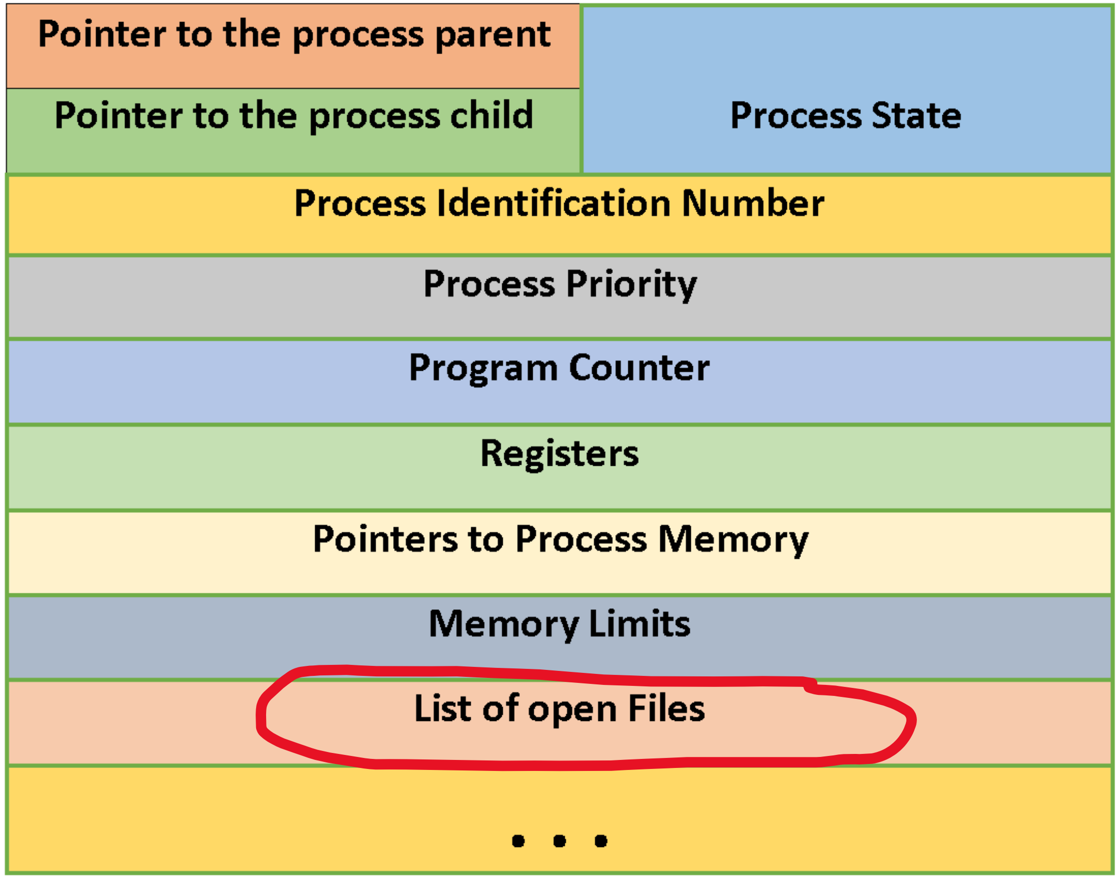
. . .

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:



Pointer to the process parent

Pointer to the process child

Process State

Process Identification Number

Process Priority

Program Counter

Registers

Pointers to Process Memory

Memory Limits

List of open Files

. . .

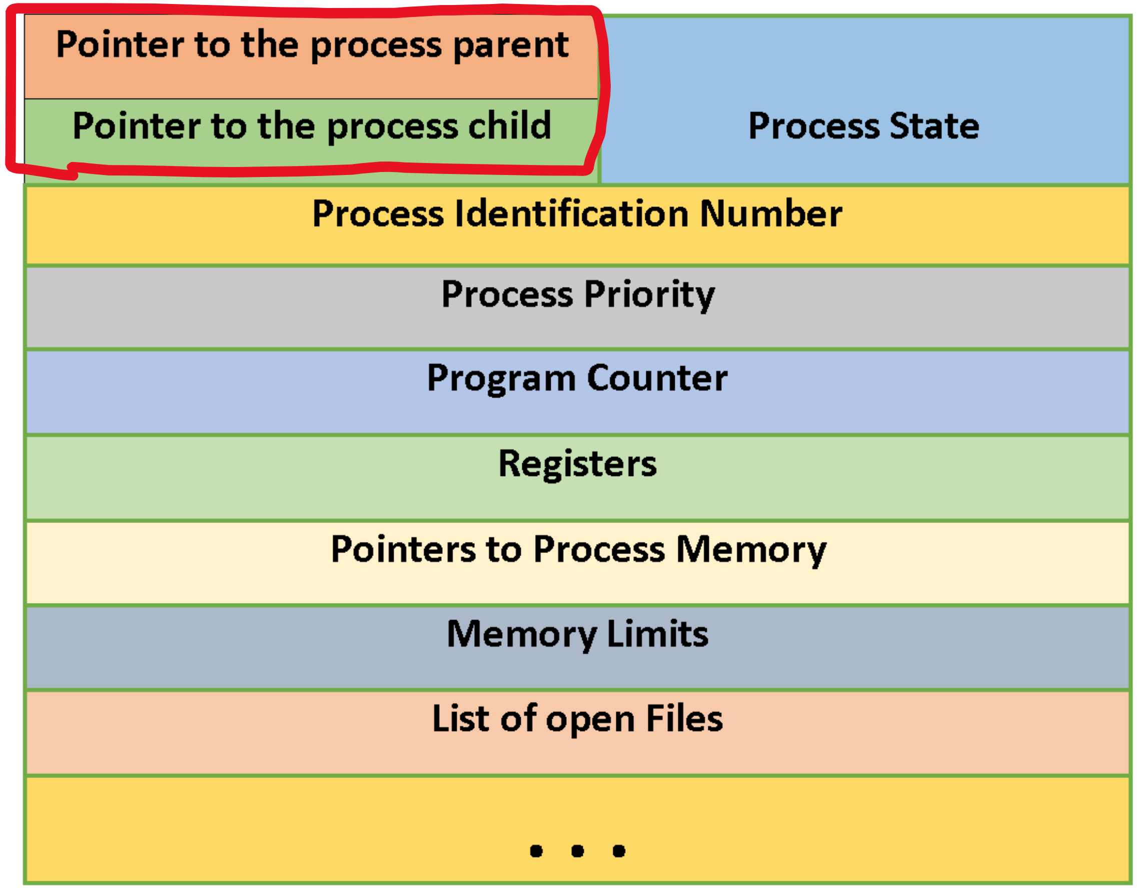
Created by Notes Jam

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:



Pointer to the process parent

Pointer to the process child

Process State

Process Identification Number

Process Priority

Program Counter

Registers

Pointers to Process Memory

Memory Limits

List of open Files

. . .

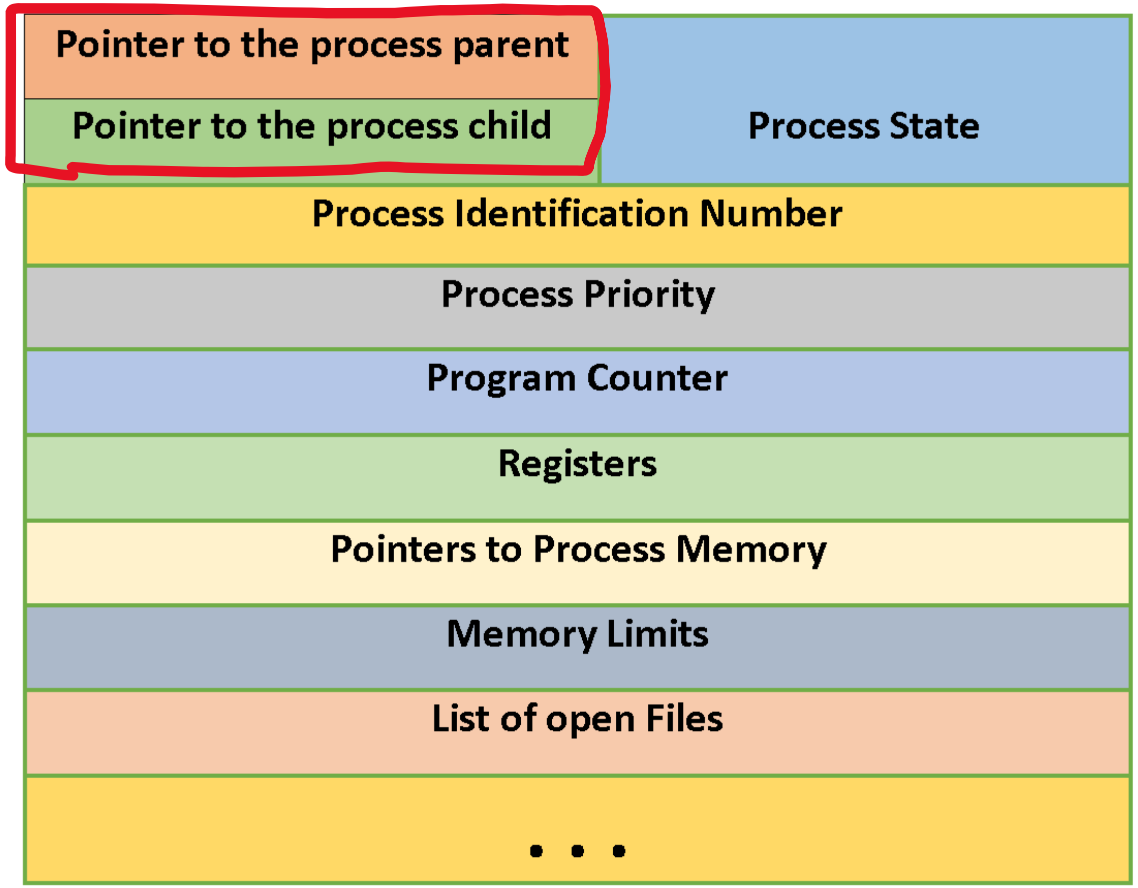
Created by Notes Jam

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:



Pointer to the process parent

Pointer to the process child

Process State

Process Identification Number

Process Priority

Program Counter

Registers

Pointers to Process Memory

Memory Limits

List of open Files

. . .

Created by Notes Jam

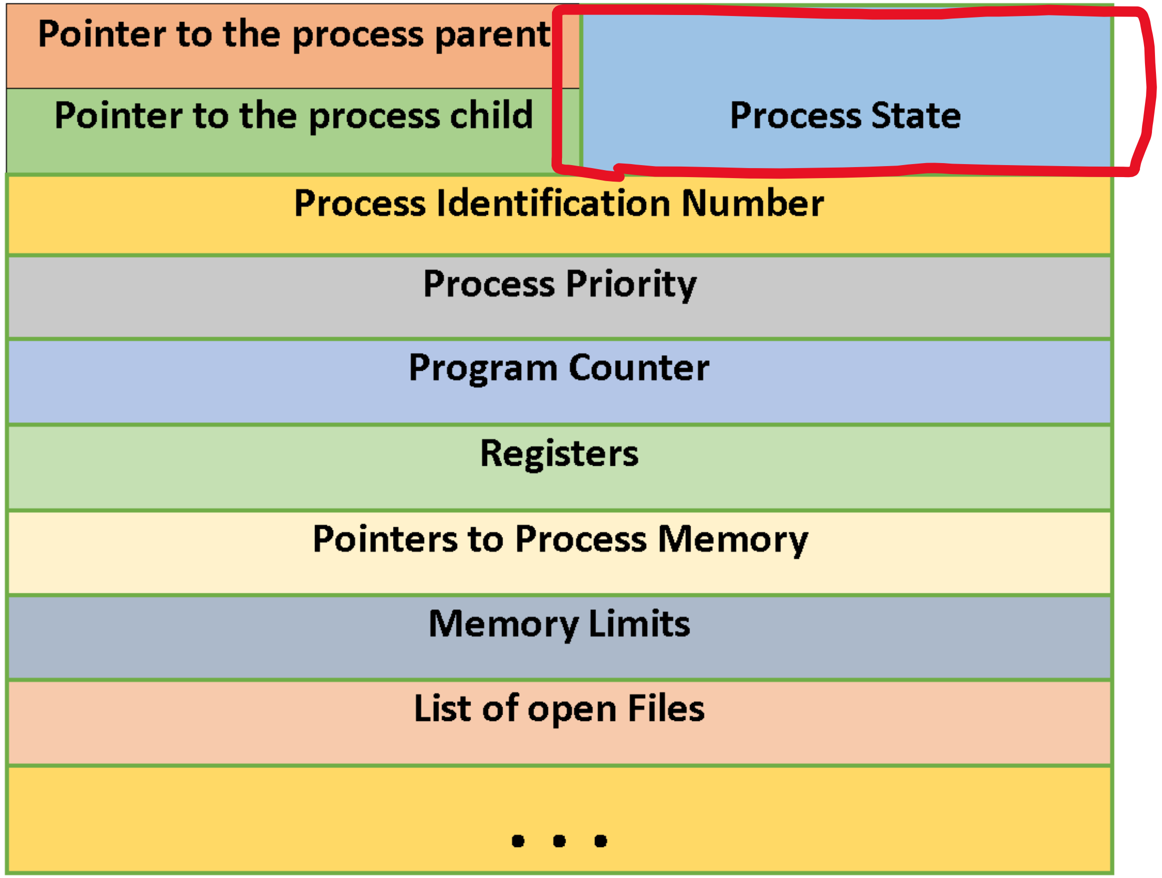
3. Execution Context/Bookkeeping information

- 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:



Pointer to the process parent

Pointer to the process child

Process State

Process Identification Number

Process Priority

Program Counter

Registers

Pointers to Process Memory

Memory Limits

List of open Files

. . .

A **process** is an instance of a running program on a computer

All processes have the following data while they are running:

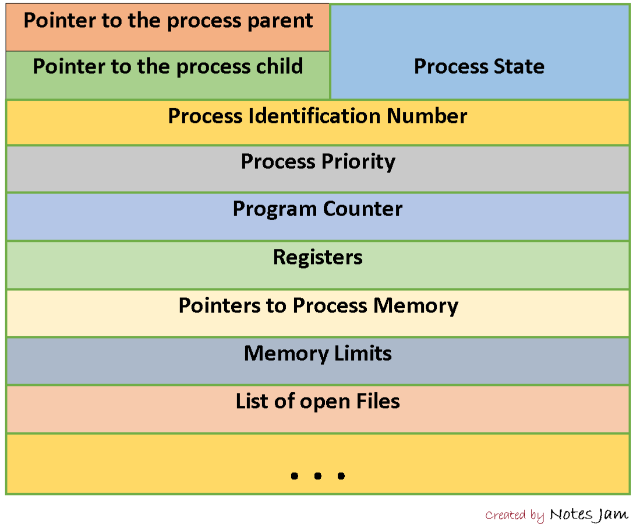
1. Executable Code

2. Associated Data

We will talk about what goes here on Friday

3. Execution Context/Bookkeeping information

(info that the OS needs to handle the process)



Pointer to the process parent	Process State
Pointer to the process child	
Process Identification Number	
Process Priority	
Program Counter	
Registers	
Pointers to Process Memory	
Memory Limits	
List of open Files	
. . .	

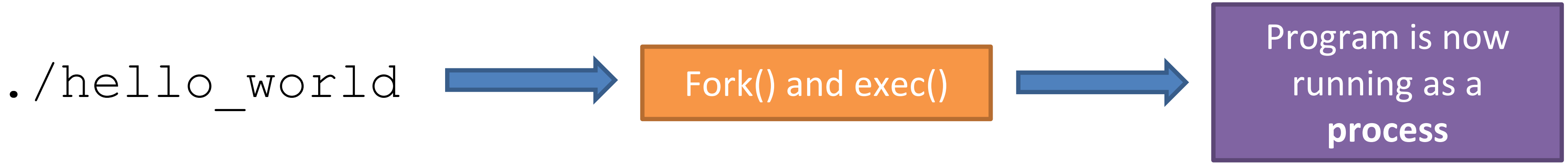
Created by Notes Jam

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**

The jobs of an Operating System

Next time…

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”