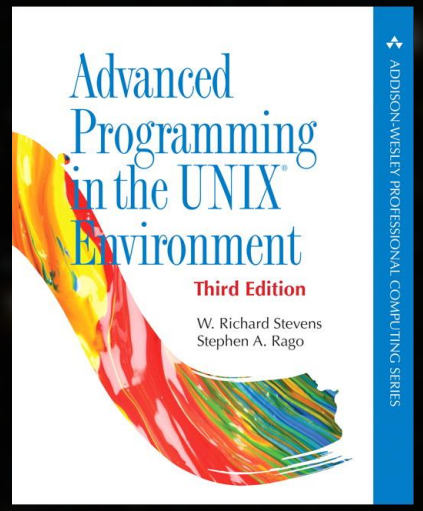
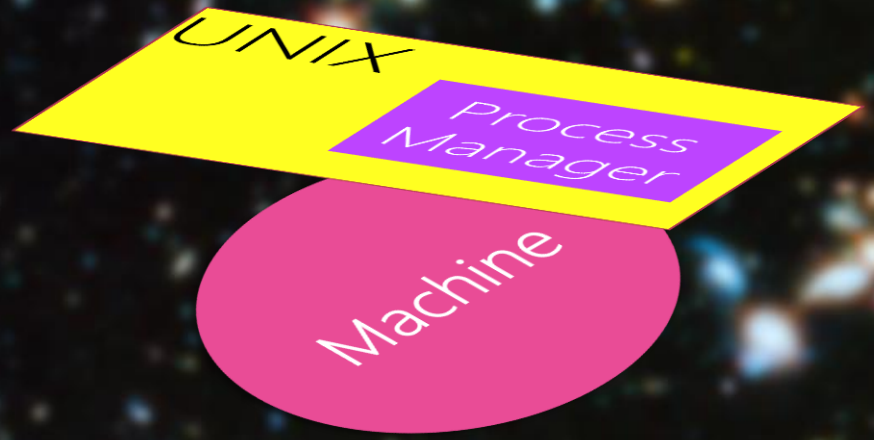
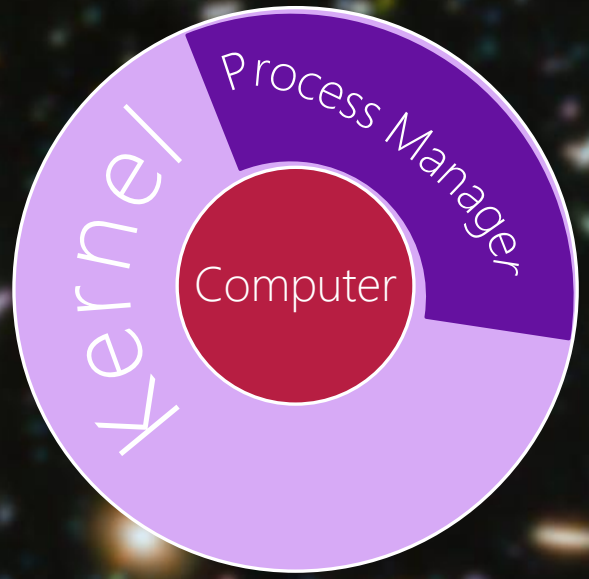
Floating Zombie, Resident Evil (2002) - Paul W. S. Anderson

Marks for Lab04/Lec04 and Lab06/Lec06 will be out soon!



Chapter 08: Process Control





Computer

Memory

Kernel: Device Manager

Kernel: Memory Manager

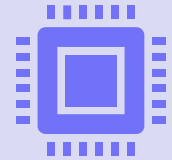
Kernel: File Manager

Kernel: Network Manager

Kernel: Process Manager

Bus

Processor



Multiprocessing

aka multiprogramming

Single Processor ~~Multiprocessor~~

Whether Busy Waiting or HALT

Waste of Processor

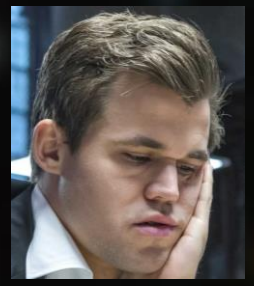
Share it with another process

Processor Sharing → Time Sharing/Slicing

Single Processor ~~Multiprocessor~~

It's not that simple, tho!

Process Context Switch



Magnus Carlsen

Sure! →

← Sure!

100 nano to 10 microseconds!

Hikaru Nakamura

Eris Li

"Normal people should see Naples before they die, but the great chess masters have to win the Wijk aan Zee tournament first of all"-Bent Larsen

Sharing 1 Chessboard

Creating a New Process

Creating a New Process

System Calls: `fork()` in `unistd.h`

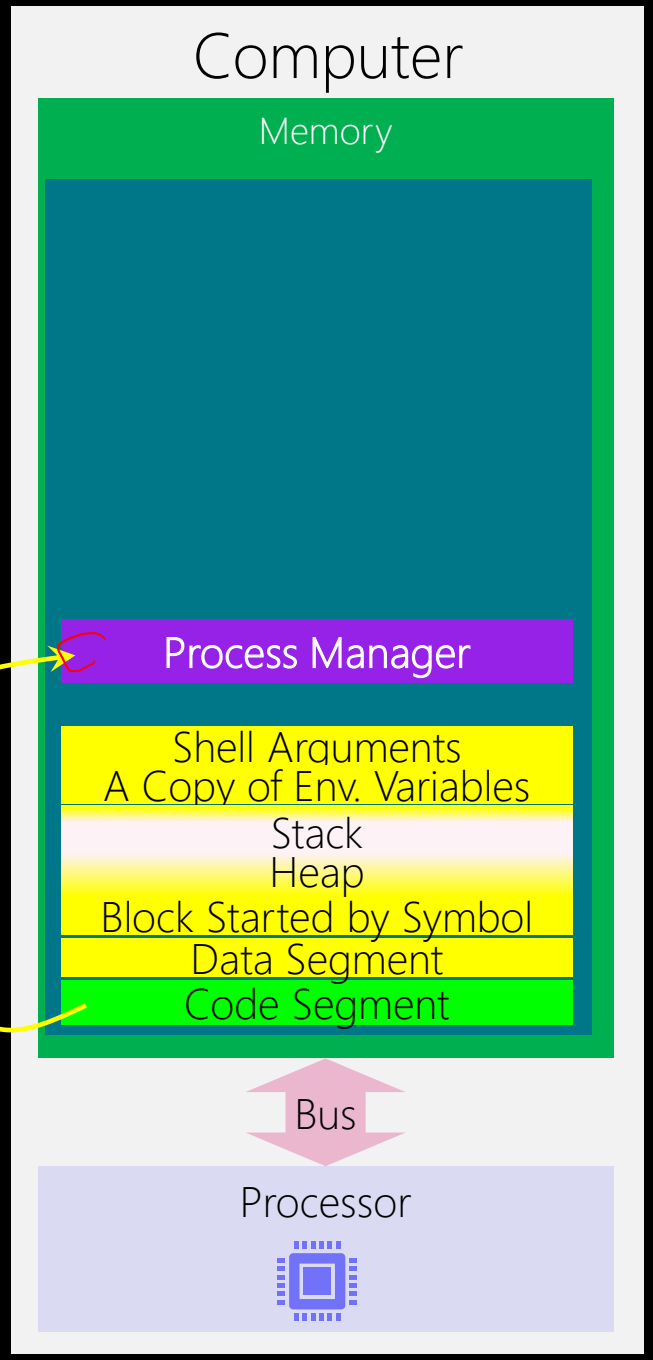
```
#include <unistd.h>
pid_t fork(void);
```

Returns: 0 in child, PID of child in parent, –1 on error

Parent vs. Child Process

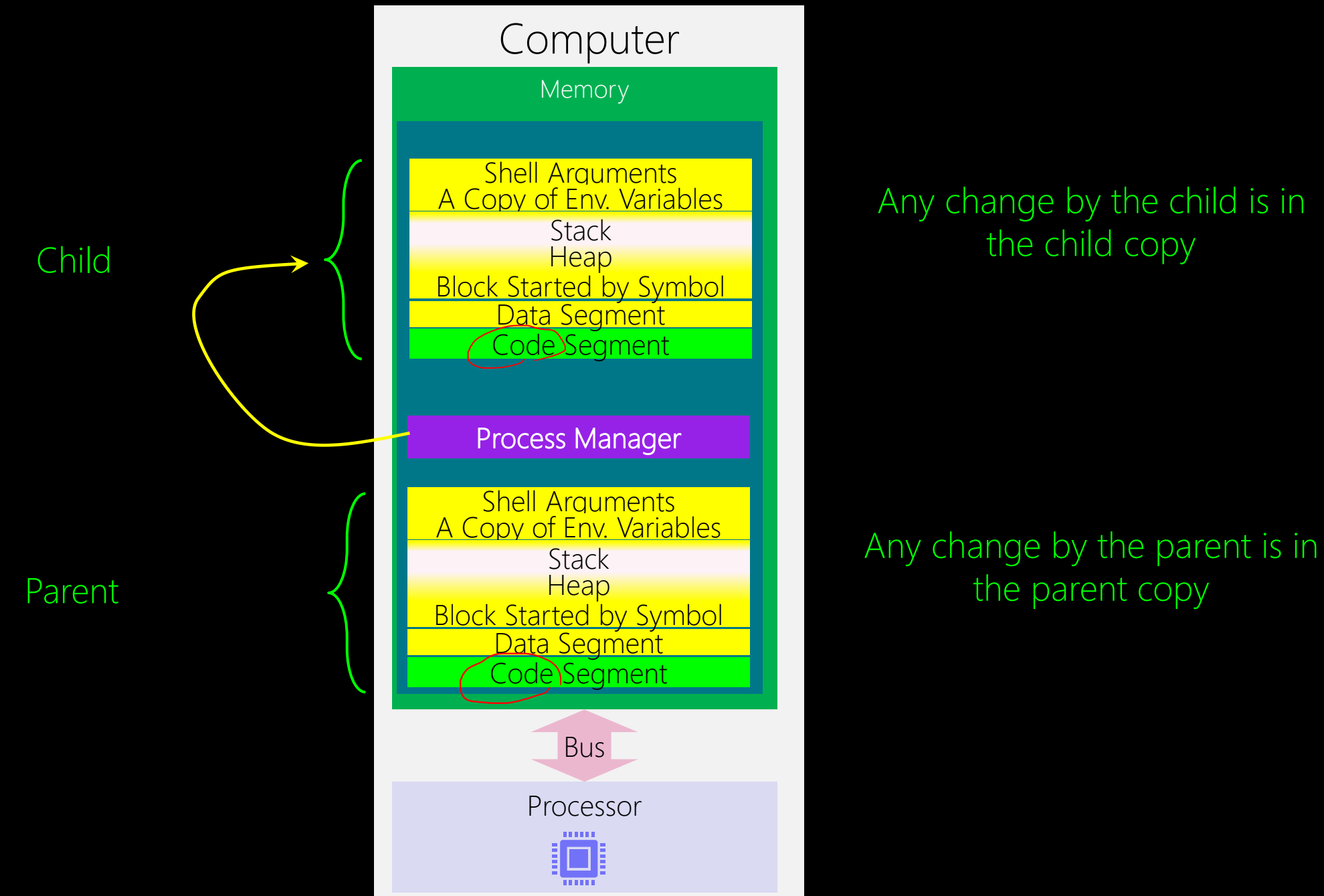
System Calls: `fork()` in `unistd.h`

Only an existing process can create a new process.
Because somebody should do the system call!



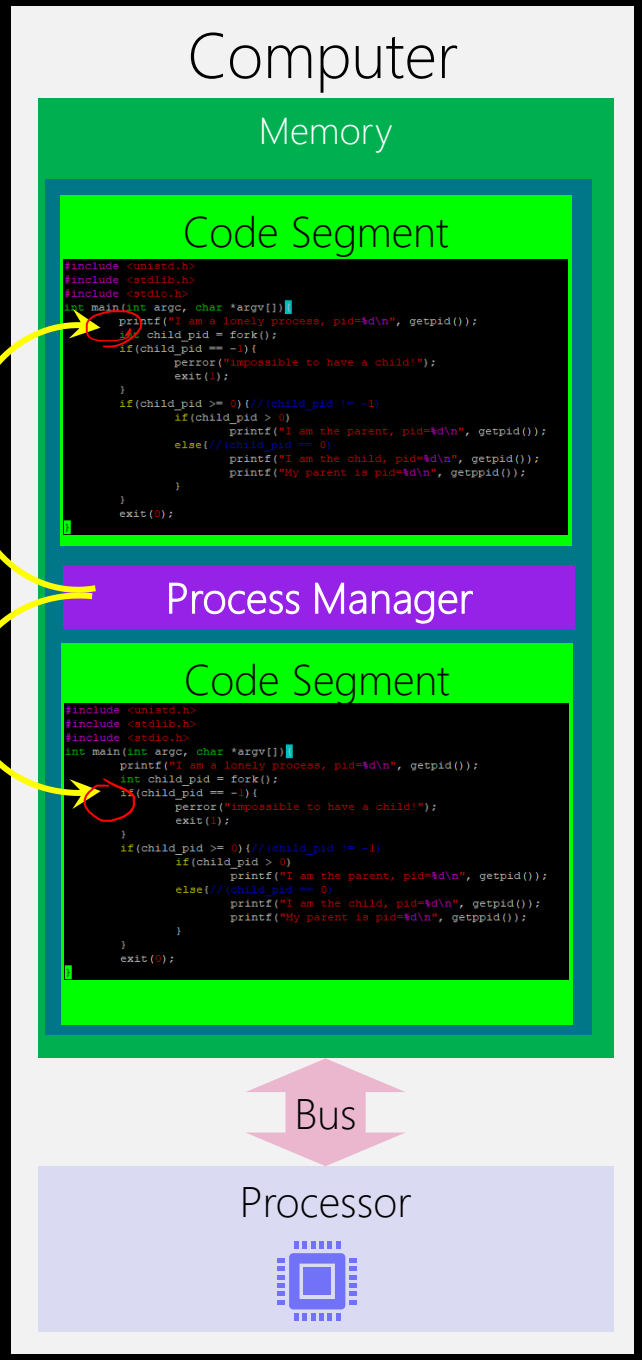

```
int child_pid = fork();
```

Exact copy at `fork()`


Computer
Memory
Shell Arguments
A Copy of Env. Variables
Stack
Heap
Block Started by Symbol
Data Segment
Code Segment
Child
Any change by the child is in the child copy
Process Manager
Shell Arguments
A Copy of Env. Variables
Stack
Heap
Block Started by Symbol
Data Segment
Code Segment
Parent
Any change by the parent is in the parent copy
Bus
Processor

Child

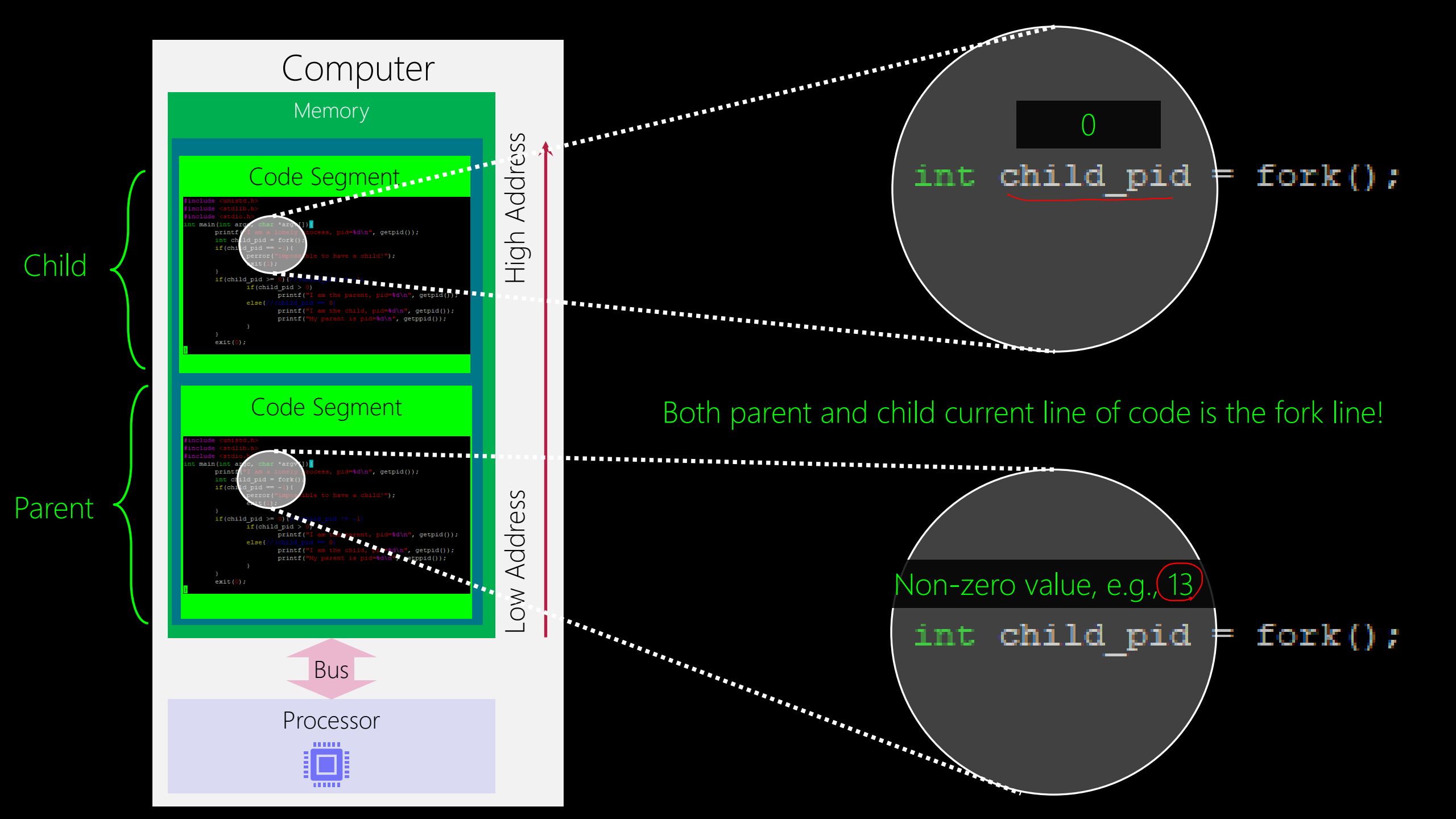
Parent



0 at `fork()` for child

This system call is amazing as it returns two values to two different processes!

child_pid at `fork()` for parent


Computer
Memory
Code Segment
Code Segment
High Address
Low Address
Child
Parent
Bus
Processor
0
int child_pid = fork();
Both parent and child current line of code is the fork line!
Non-zero value, e.g., 13
int child_pid = fork();

Parent

13

```
int child_pid = fork();
```

Child

0

```
int child_pid = fork();
```

Let's be fair

Give 1 statement to each

Parent (pid=4)

Processor

Child (pid=13)

13

```
int child_pid = fork();
if(child_pid == -1){
```

F

0

```
int child_pid = fork();
```

Parent (pid=4)

Processor

Child (pid=13)

13

```
int child_pid = fork();
if(child_pid == -1){
```

0

```
int child_pid = fork();
if(child_pid == -1){
```

F

Parent (pid=4) Processor Child (pid=13)

13

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
```

F

T

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
```

Parent (pid=4)

13

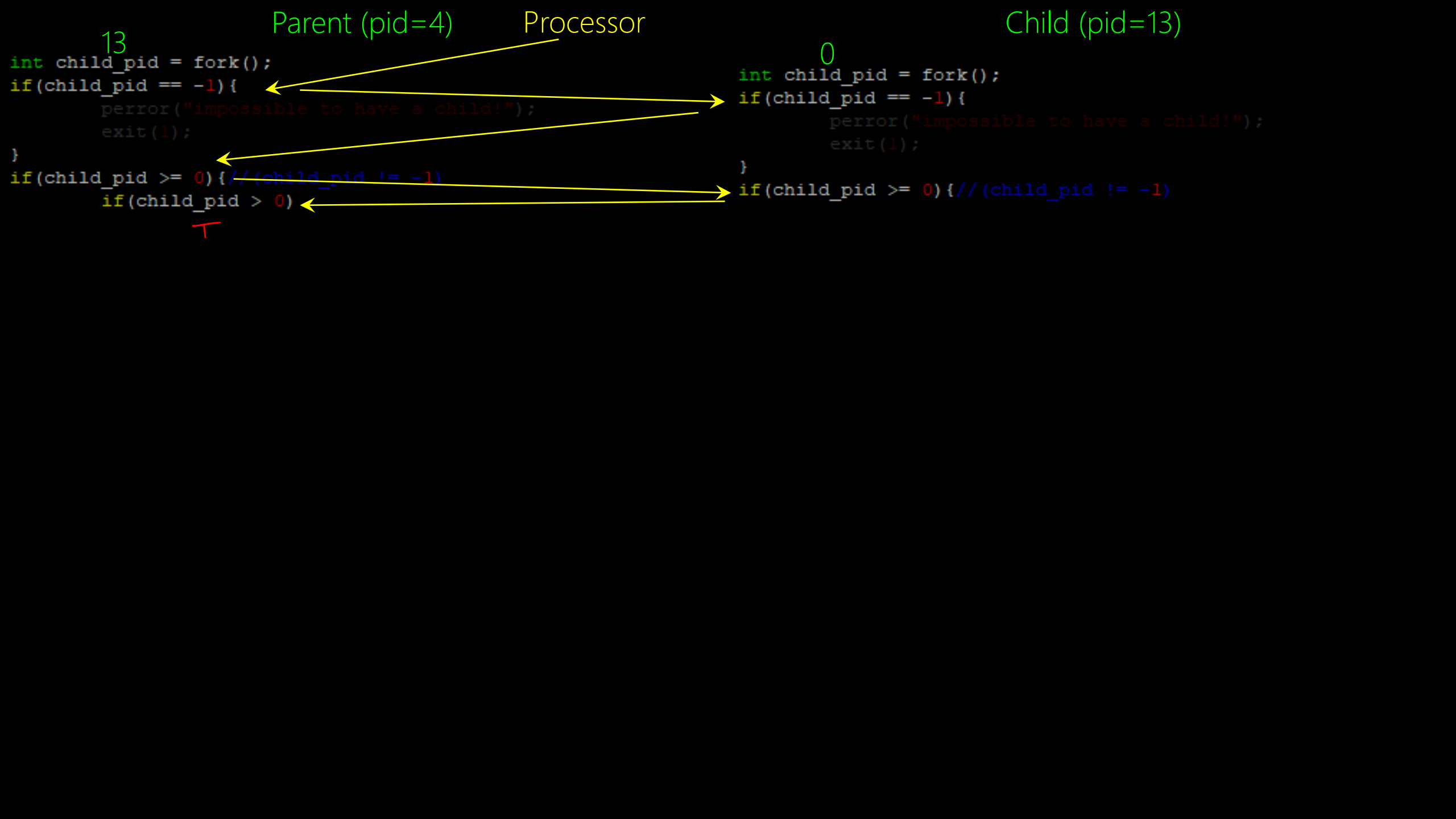
```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
```

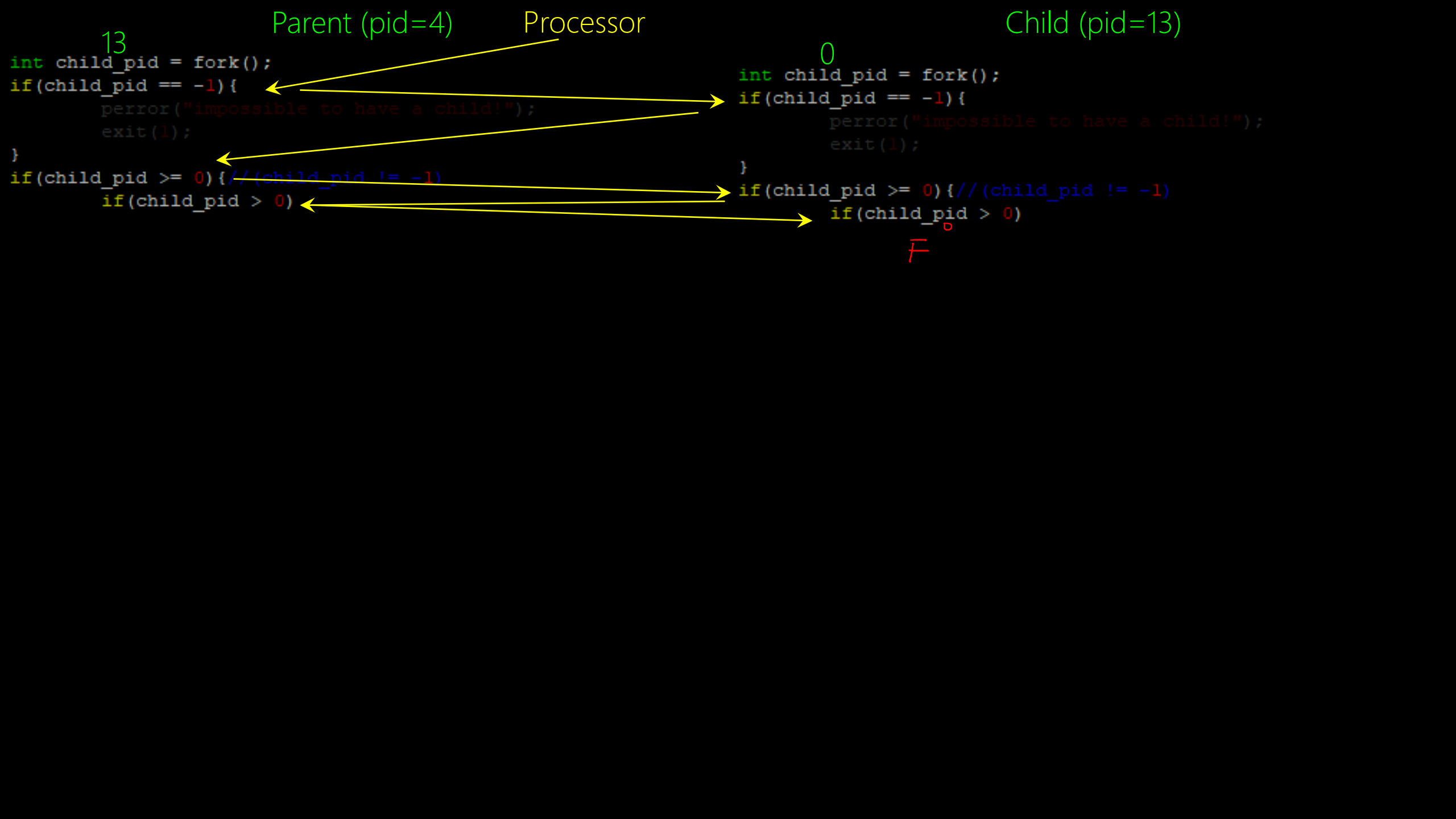
Processor

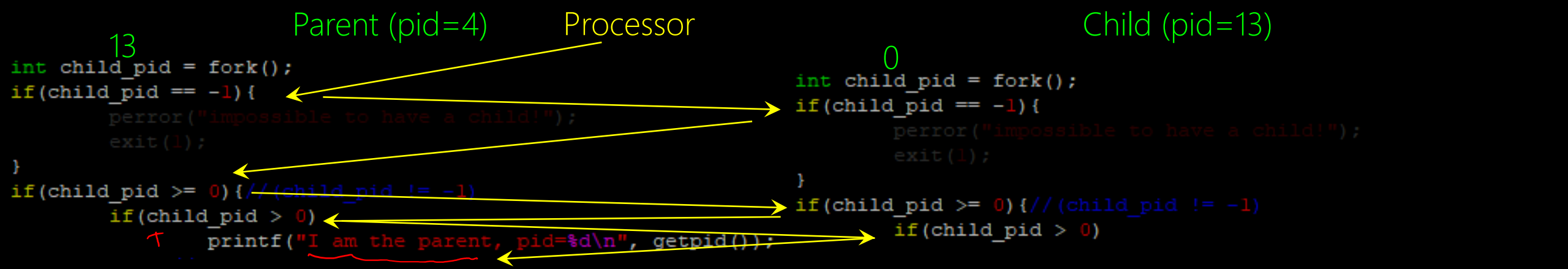
Child (pid=13)

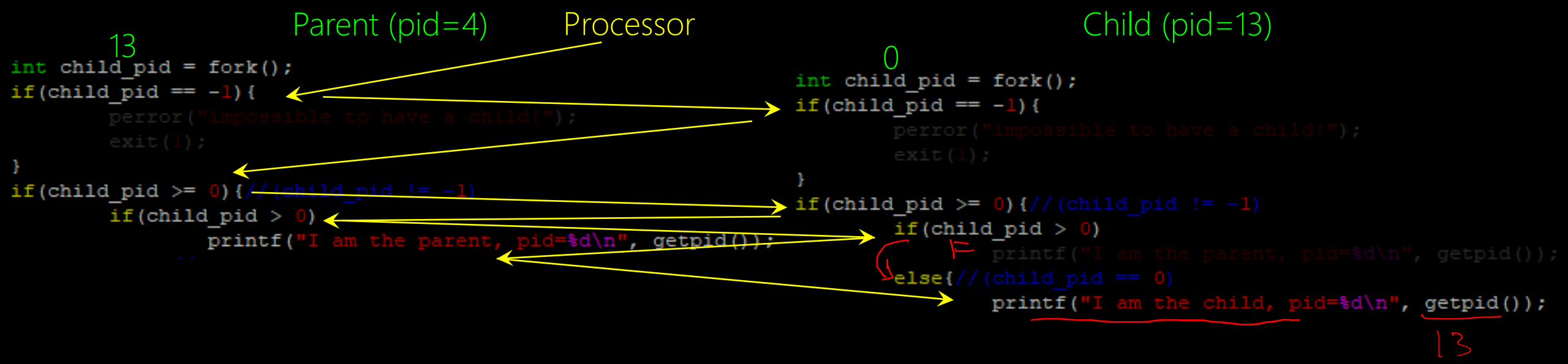
0

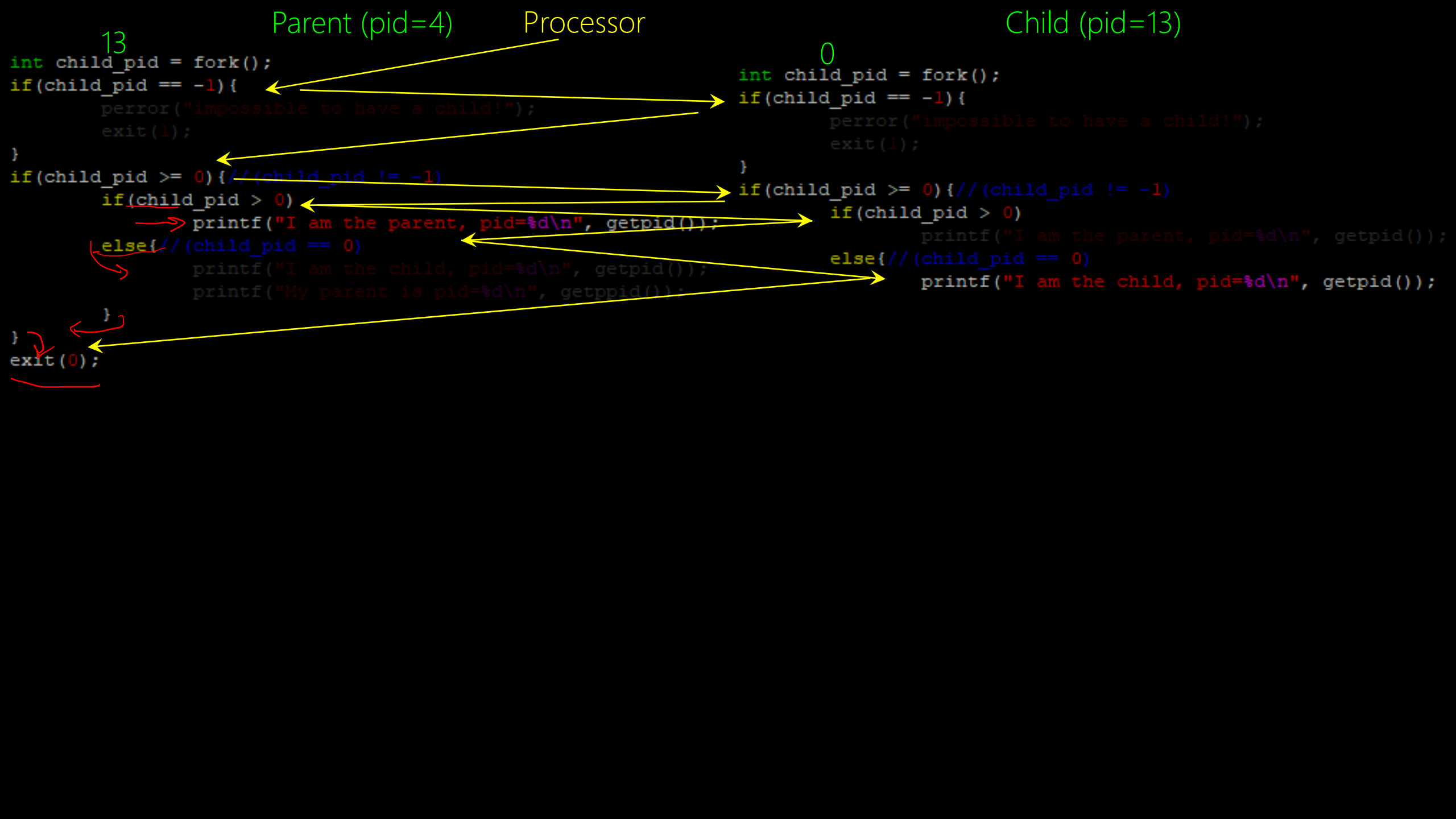
```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
```


Parent (pid=4)
Processor
Child (pid=13)
13
0
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
if(child_pid > 0)
T
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)


Parent (pid=4)
Processor
Child (pid=13)
13
0
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
if(child_pid > 0)
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
if(child_pid > 0)
F


Parent (pid=4)
Processor
Child (pid=13)
13
0
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
if(child_pid > 0)
printf("I am the parent, pid=%d\n", getpid());
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
if(child_pid > 0)

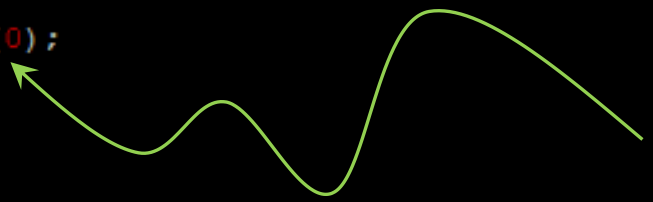

Parent (pid=4)
Processor
Child (pid=13)
13
0
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
if(child_pid > 0)
printf("I am the parent, pid=%d\n", getpid());
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
if(child_pid > 0)
printf("I am the parent, pid=%d\n", getpid());
else{//(child_pid == 0)
printf("I am the child, pid=%d\n", getpid());
13


Parent (pid=4)
Processor
Child (pid=13)
13
0
int child_pid = fork();
if(child_pid == -1){
perror("impossible to have a child!");
exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
if(child_pid > 0)
printf("I am the parent, pid=%d\n", getpid());
else{//(child_pid == 0)
printf("I am the child, pid=%d\n", getpid());
printf("My parent is pid=%d\n", getppid());
}
}
exit(0);

Parent (pid=4)

13

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
exit(0);
```



Child (pid=13)

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
```

Child (pid=13)

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
```

0

X = fork()

getppid()

Orphan

No Parent → Grandparent adopts the Child

Child' PPID → Grandparent → … → Shell → Kernel

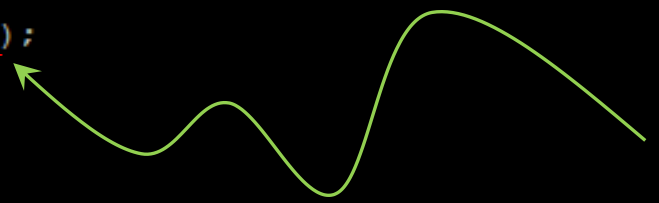
1

0

Child

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
exit(0);
```



Put the child first, please!

Parent (pid=4)

13

```
int child_pid = fork();
```

Processor

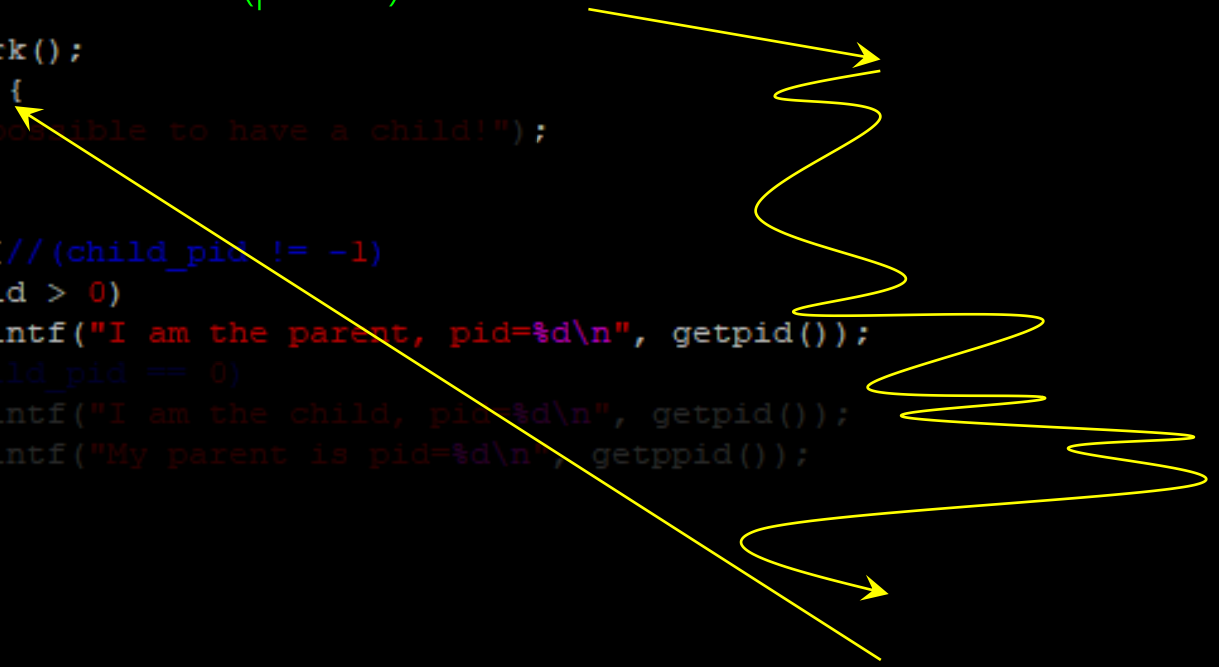
Child (pid=13)

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
exit(0);
```

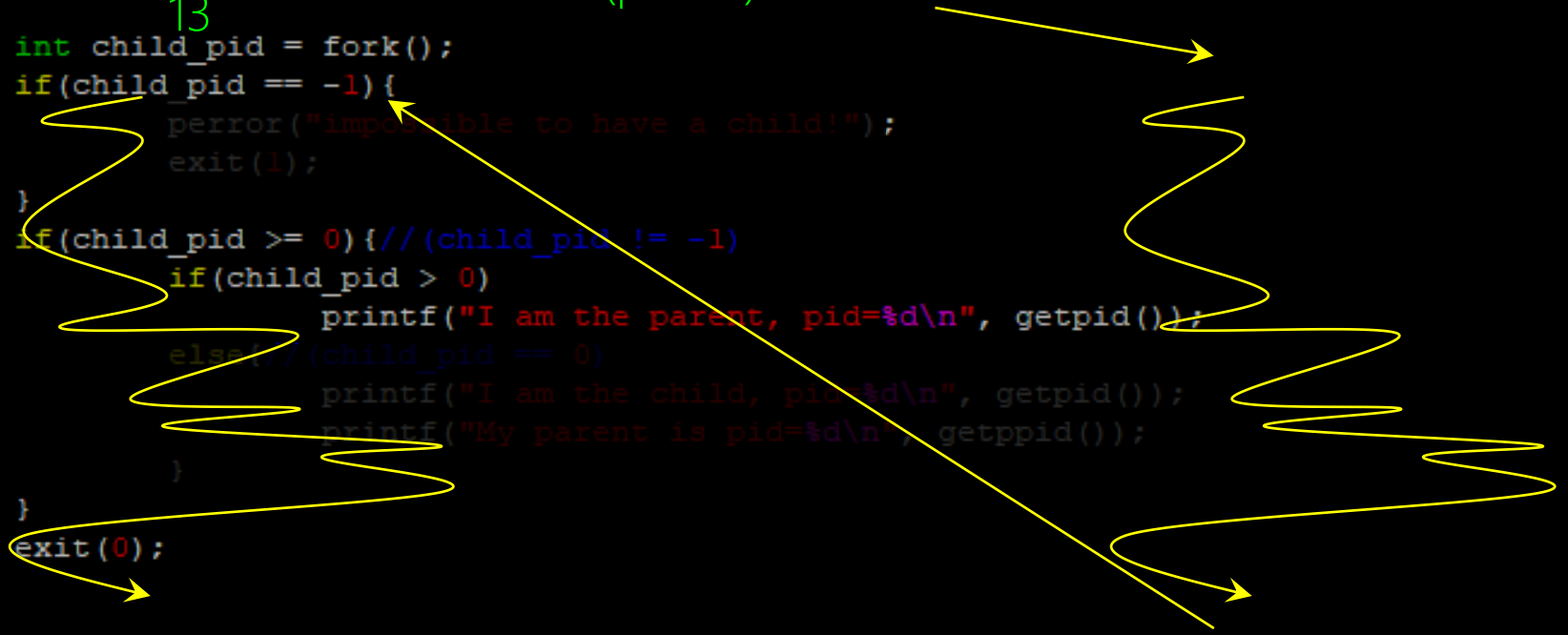
Parent (pid=4) Processor

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
exit(0);
```



Parent (pid=4) Processor

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
exit(0);
```



Resident Evil (2002) - Paul W. S. Anderson

Chronology of Family Planning in China

(I) Generative Model for New Process

A) Parent process carries the task of 1) Itself and 2) The Child
B) Parent does the `fork()` to pass the child's task to the child

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());


        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent's pid=%d\n", getppid());
                //Assign child's tasks here


        }
}

 exit(0);
```

Parent's Tasks

Child's Tasks

Processor Share | Processor Slice
Time Share | Time Slice

As programmer, we don't know. We should not assume anything.

E.g., parent gets 20 slices/minutes, child gets 1 slice/1 minute

E.g., parent gets 1 slice/minute, child gets 100 slices/10 minutes

Parent

13

```
int child_pid = fork();
```

Child

0

```
int child_pid = fork();
```

If parent gets higher priority …

Parent

13

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
exit(0);
```

Child

0

```
int child_pid = fork();
```

Child

0

```
int child_pid = fork();
```

Child

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
exit(0);
```

Orphan Child

Best Pattern/Practice

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent pid=%d\n", getppid());
                //Assign child's tasks here
                exit(0);
        }
}
//Assign parent tasks here
exit(0);
```

Child's Tasks

Parent's Tasks

Still there is a chance for having orphaned child. Why?

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent's pid=%d\n", getppid());
                //Assign child's tasks here
                exit(0);
        }
}
//Assign parent tasks here
exit(0);
```

Child's Tasks

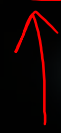
Parent's Tasks

Wait for the child

Resident Evil (2002) - Paul W. S. Anderson

Wait for Child Process be over

System Calls: `wait()` in `sys/wait.h`



Like HLT (HALT) to processor, kernel can also halt a process:

- Not give any processor time/slices
- It is called blocking for processes instead of halting.

Wait for Child Process be over

System Calls: `wait()` in `sys/wait.h`

```
#include <sys/wait.h>
pid_t wait(int *statloc);
```

Return Child's PID if OK, or –1 on error

Parent

13

```
int child_pid = fork();
```

Child

0

```
int child_pid = fork();
```

Parent

Child

13

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
 //Assign parent tasks here
  int child_exit = 0;
  wait(&child_exit);
```

0

```
int child_pid = fork();
```

Parent blocks

No processor share or time slice will be given

Parent

13

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
```

```
//Assign parent tasks here
 int child_exit = 0;
 wait(&child_exit);
```

Child

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
                exit(0);
        }
}
```

Child to Kernel: I am done!

Kernel to Parent: Wake up! Send a SIGCHLD + Child's Exit Status

Parent

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}

 //Assign parent tasks here
  int child_exit = 0;
  wait(&child_exit);
 exit(0);
```

Parent become unblocked.

Receive Some Share of Processor

Parent

13

```
int child_pid = fork();
```

Child

0

```
int child_pid = fork();
```

What if child gets higher priority?

Parent

13

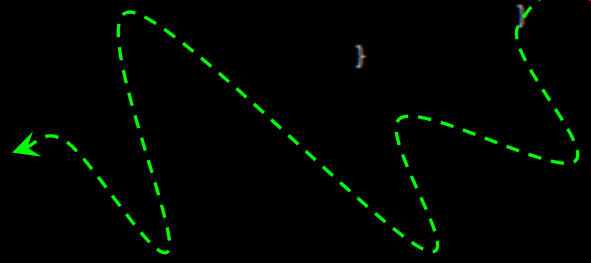
```
int child_pid = fork();
```

wait()

Child

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
                exit(0);
        }
}
```



Child to Kernel: I am done!

Kernel to ???: I have no sleeping parent to wakeup!

Parent

13

```
int child_pid = fork();
```

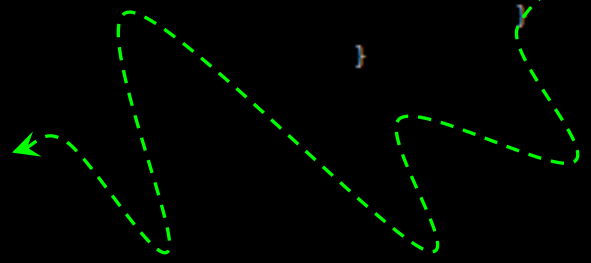
exit(1)

wait()

Child

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
                exit(0);
}
```



Zombie

Child finishes before parent's `wait()`

Child is waiting for the parent to receive the exit status.

Parent MUST have a `wait()`, otherwise Zombie never exits!

Parent

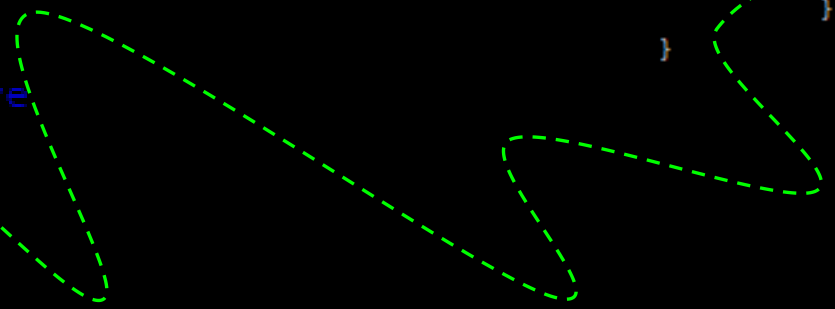
13

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}

 //Assign parent tasks here
 int child_exit = 0;
 wait(&child_exit);
```

Child

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }       exit(0);
}
```



No blocking!

Wait for nothing. The child was already done.

Returns immediately with exit status of the child

Parent

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}
```

```
//Assign parent tasks here
int child_exit = 0;
wait(&child_exit);
exit(0);
```

Laser Room, Resident Evil (2002) - Paul W. S. Anderson

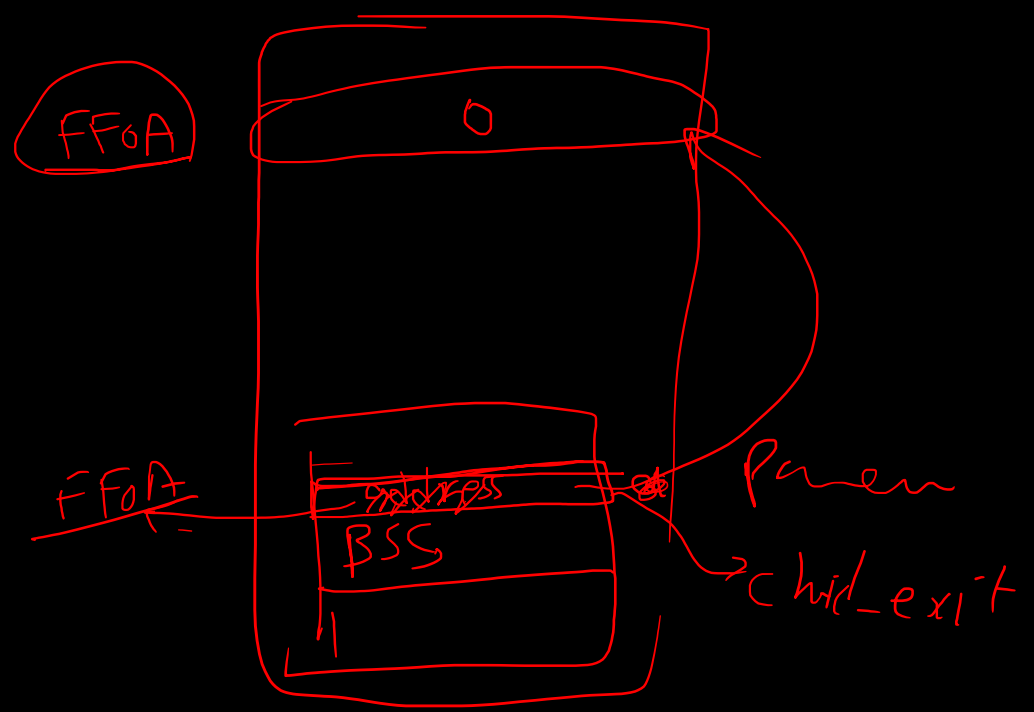
Example I

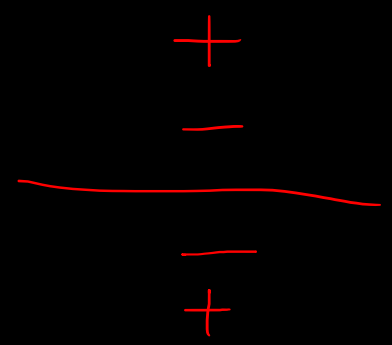
We want to ADD and SUB two numbers

A) The parent does the ADD

B) The child does the SUB

```
hfani@bravo:~$ vi fork_model_I.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
int main(int argc, char *argv[]){
        int a = 0;
        int b = 0;
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        printf("I am a lonely process, pid=%d\n", getpid());
        int child_pid = fork();
        if(child_pid == -1){
                perror("impossible to have a child!\n");
                exit(1);
        }
        if(child_pid >= 0){//(child_pid != -1)
                if(child_pid > 0)
                        printf("I am the parent, pid=%d\n", getpid());
                else{//(child_pid == 0)
                        printf("I am the child, pid=%d\n", getpid());
                        //Assign child's tasks here
                        printf("child: %d - %d = %d\n", a, b, a - b);
                        exit(0);
                }
        }
        //Assign parent tasks here
        printf("parent: %d + %d = %d\n", a, b, a + b);
        int *child_exit;
        wait(child_exit);
        exit(0);
}
```





```
hfani@alpha: ./fork_model_I 10 30
I am a lonely process, pid=483542
I am the parent, pid=483542
parent: 10 + 30 = 40
I am the child, pid=483543
child: 10 + 30 = -20
```

Parent gets priority

```
hfani@alpha: ./fork_model_I 10 30
I am a lonely process, pid=483542
I am the child, pid=483543
child: 10 + 30 = -20
I am the parent, pid=483542
parent: 10 + 30 = 40
```

Child gets priority

(11) Generative Model for New Process II

A) Parent process carries the task of 1) Itself and 2) A path to the task of child
B) Parent does the `fork()` to create the child and pass the path to the tasks
C) Child does the `exec()` to fetch the tasks (Lab07)

```c
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)

                exit(0);
        }
}
//Assign parent tasks here

 exit(0);
```

Path to Child's Tasks: exec (path)

Parent's Tasks

Wait for the child

Example II

We want to ADD and GCD of two numbers

A) The parent does the ADD

B) The child does the GCD (Great Common Devisor)

Example II

Child: Not fair! You do a simple task, but I do the task that I have no idea about.
Parent: Don't worry. Hossein already have the program. Simply `exec()` it.

```
hfani@bravo:~$ vi gcd.c
```

```c
//from https://www.javatpoint.com/gcd-of-two-numbers-in-c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int n1 = atoi(argv[1]);
    int n2 = atoi(argv[2]);
    int i, gcd;
    // use for loop
    for( i = 1; i <= n1 && i <= n2; ++i)
    {
        if (n1 % i ==0 && n2 % i == 0)
            gcd = i; /* if n1 and n2 is completely divisible by i, the divisible number will be the G
    }
    // print the GCD of two numbers
    printf ("PID: %d => gcd of %d and %d is %d\n", getpid(), n1, n2, gcd);
    return 0;
}
```

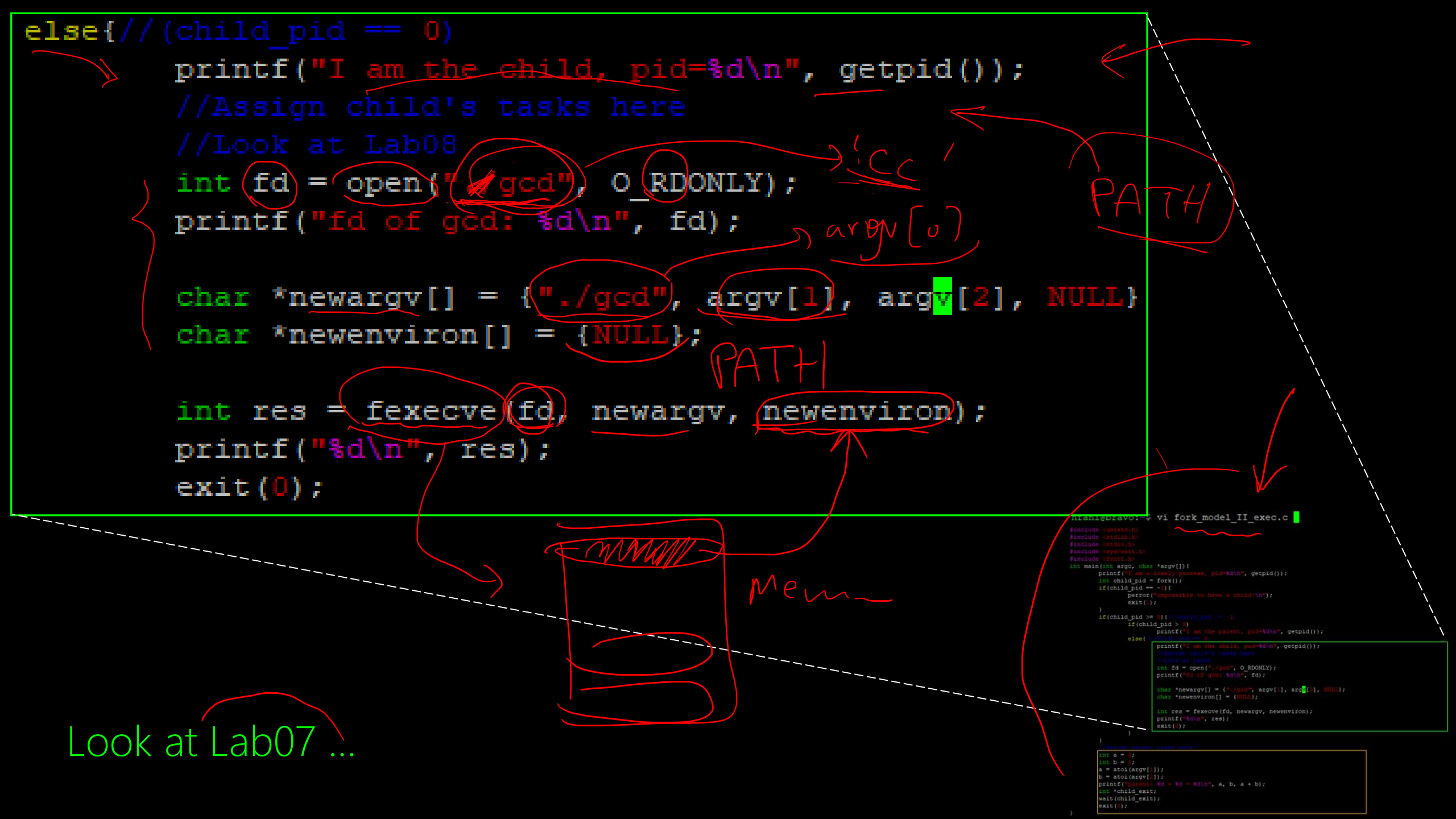
Even Hossein may not know how to calculate GCD!

```
hfani@bravo:~$ ./gcd 105 200
PID: 3359754 => gcd of 105 and 200 is 5
```

```
hfani@bravo:~$ vi fork_model_II_exec.c
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <fcntl.h>
int main(int argc, char *argv[]){
        printf("I am a lonely process, pid=%d\n", getpid());
        int child_pid = fork();
        if(child_pid == -1){
                perror("impossible to have a child!\n");
                exit(1);
        }
        if(child_pid >= 0){//(child_pid != -1)
                if(child_pid > 0)
                        printf("I am the parent, pid=%d\n", getpid());
                else{//(child_pid == 0)
                        printf("I am the child, pid=%d\n", getpid());
                        //Assign child's tasks here
                        //Look at Lab08
                        int fd = open("./gcd", O_RDONLY);
                        printf("fd of gcd: %d\n", fd);

                        char *newargv[] = {"./gcd", argv[1], argv[2], NULL};
                        char *newenviron[] = {NULL};

                        int res = fexecve(fd, newargv, newenviron);
                        printf("%d\n", res);
                        exit(0);
                }
        }
        //Assign parent tasks here
        int a = 0;
        int b = 0;
        a = atoi(argv[1]);
        b = atoi(argv[2]);
        printf("parent: %d + %d = %d\n", a, b, a + b);
        int child_exit = 0;
        wait(&child_exit);
        exit(0);
}
```

```c
else{//(child_pid == 0)
        printf("I am the child, pid=%d\n", getpid());
        //Assign child's tasks here
        //Look at Lab08
        int fd = open("./gcd", O_RDONLY);
        printf("fd of gcd: %d\n", fd);

        char *newargv[] = {"./gcd", argv[1], argv[2], NULL};
        char *newenviron[] = {NULL};

        int res = fexecve(fd, newargv, newenviron);
        printf("%d\n", res);
        exit(0);
```

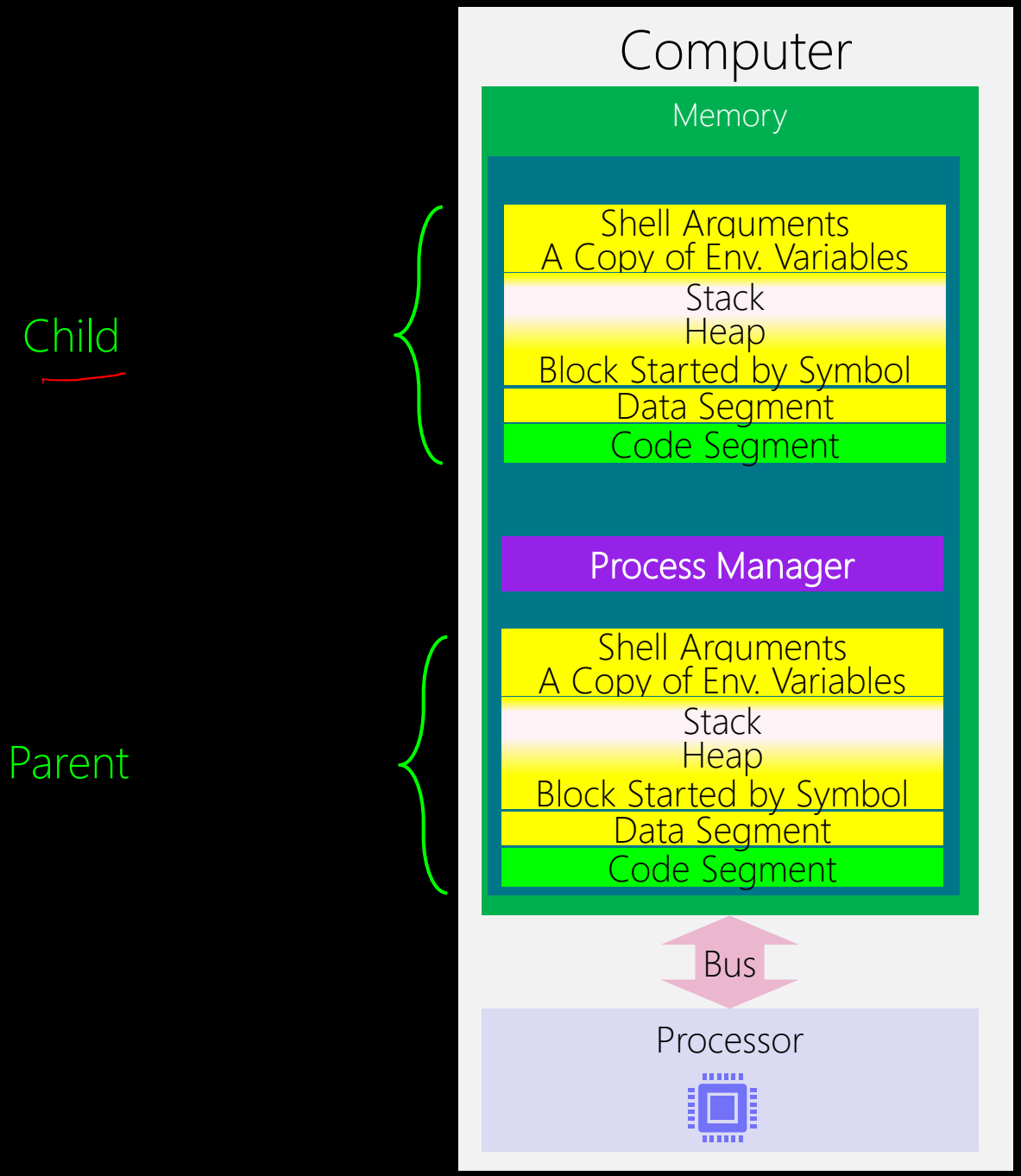


```
hfani@bravo:~$ ./fork_model_II_exec 105 200
I am a lonely process, pid=3370346
I am the parent, pid=3370346
parent: 105 + 200 = 305
I am the child, pid=3370347
fd of gcd: 3
PID: 3370347 => gcd of 105 and 200 is 5
```

```
hfani@bravo:~$ ./fork_model_II_exec 105 200
I am a lonely process, pid=3370346
I am the parent, pid=3370346
parent: 105 + 200 = 305
I am the child, pid=3370347
fd of gcd: 3
PID: 3370347 => gcd of 105 and 200 is 5
```

The child embed the program file inside itself

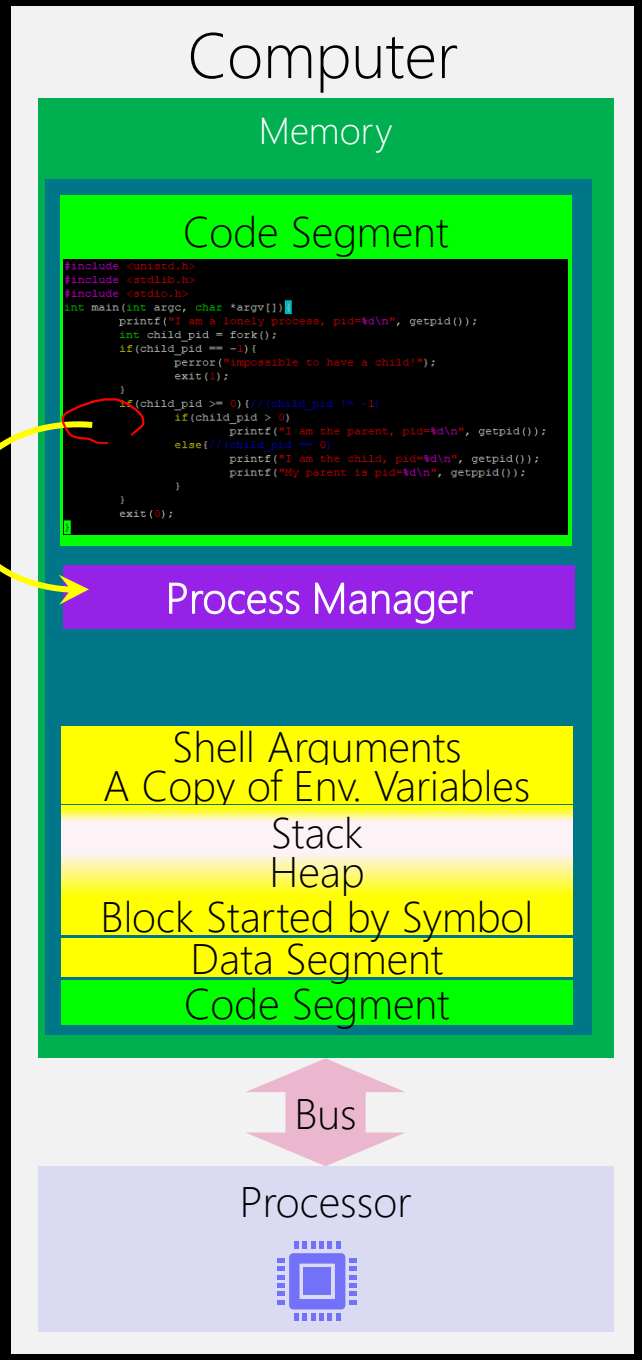
Not as a separate process, but as itself!


Computer
Memory
Shell Arguments
A Copy of Env. Variables
Stack
Heap
Block Started by Symbol
Data Segment
Code Segment
Process Manager
Shell Arguments
A Copy of Env. Variables
Stack
Heap
Block Started by Symbol
Data Segment
Code Segment
Bus
Processor
Child
Parent

Child

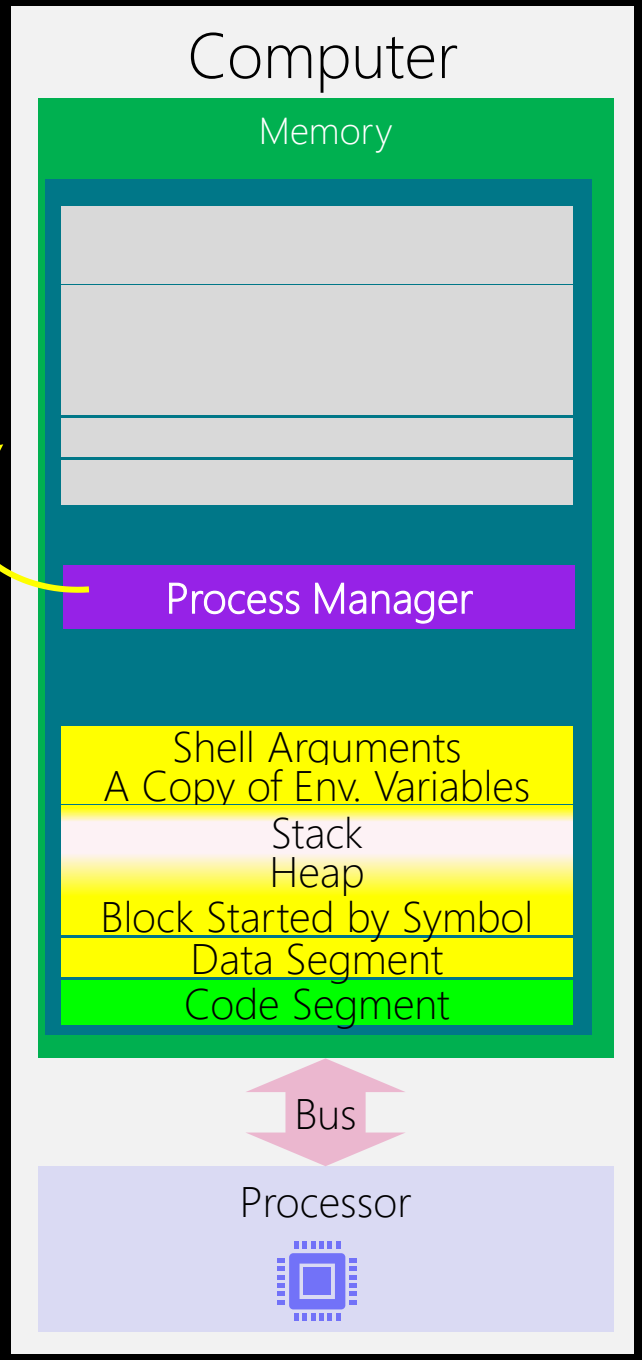
Parent

Child does `exec(path)`



Child

Parent

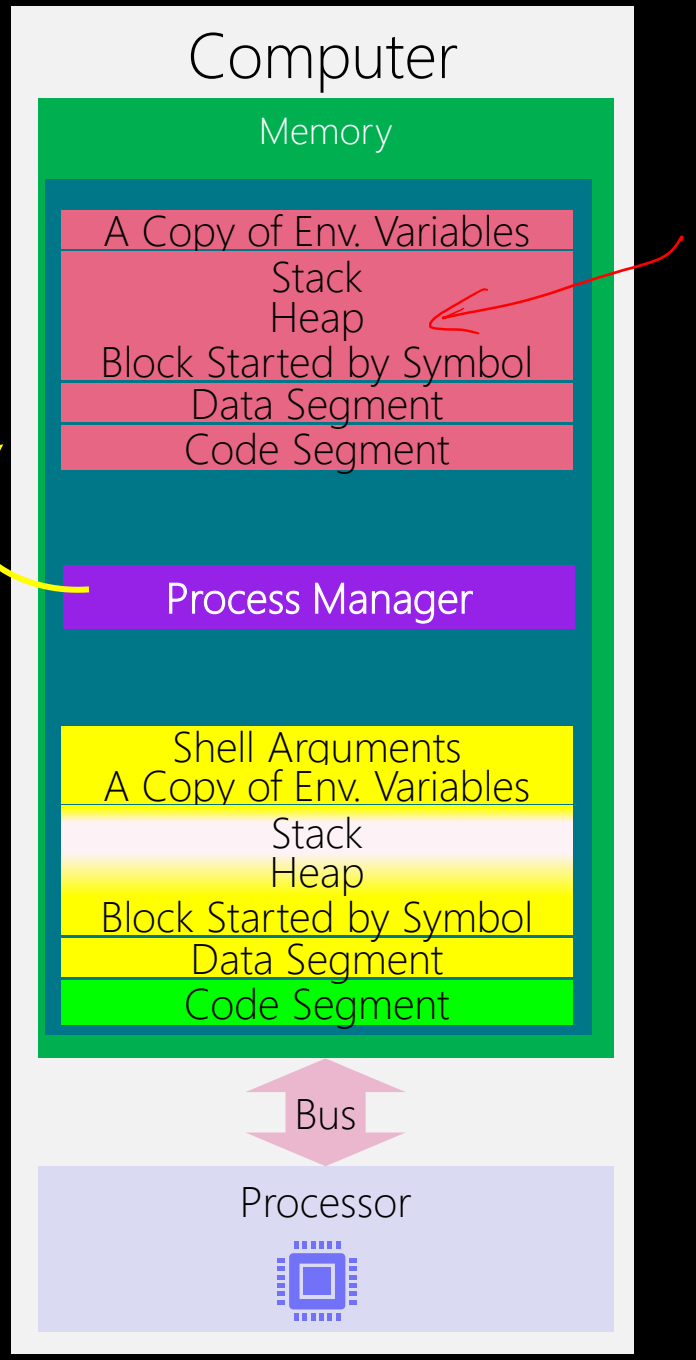


Child does `exec(path)`

Kernel empty the child space

Child

Parent



Child does `exec(path)`

Kernel empty the child space

Kernel fills it with the program file in path.

No new process!

No new PID!

```
else{//(child_pid == 0)
        printf("I am the child, pid=%d\n", getpid());
        //Assign child's tasks here
        //Look at Lab08
        int fd = open("./gcd", O_RDONLY);
        printf("fd of gcd: %d\n", fd);

        char *newargv[] = {"./gcd", argv[1], argv[2], NULL};
        char *newenviron[] = {NULL};

        int res = fexecve(fd, newargv, newenviron);
        printf("%d\n", res);
        exit(0);
```

Do not exist in memory!!
They're replaced w/ gcd's codes.

Chestburster Scene, Alien (1979) - Ridley Scott

Process Life Cycle

Process States