CS 471 Operating Systems

Yue Cheng

George Mason University
Fall 2019

Announcement

- OS/161 Project 0 released on Blackboard
- Please complete the Google Form for OS/161 team composition

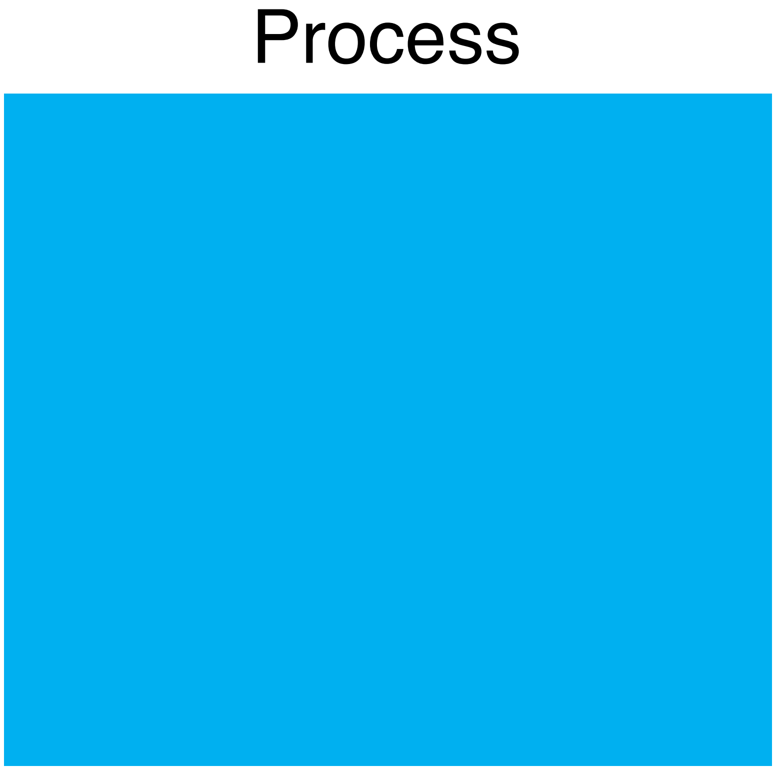
What is a Process?

What is a Process?

- Programs are code (static entity)
- **Processes** are running programs

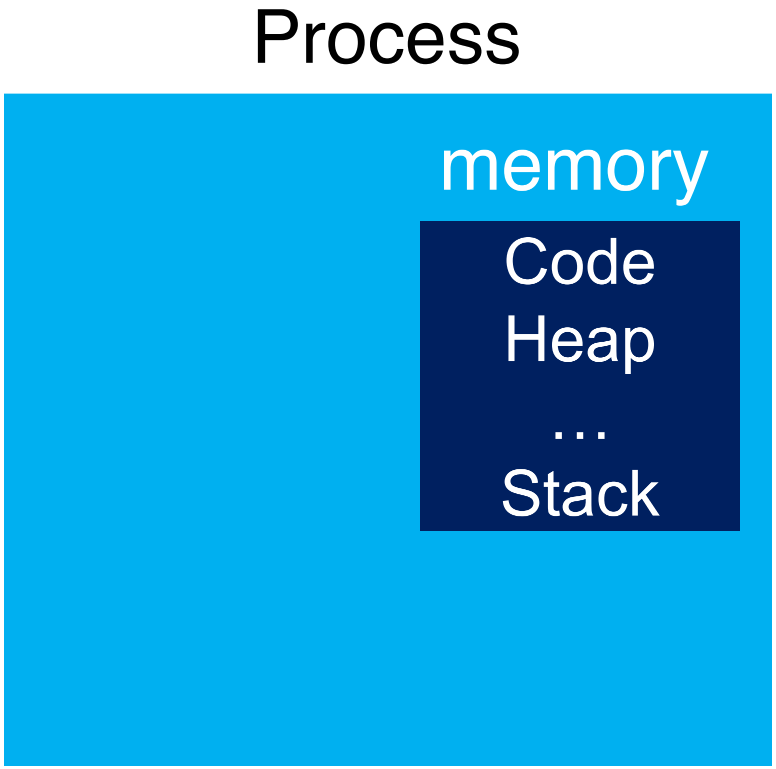
- Java analogy
  - class -> “program”
  - object -> “process”

What is in a Process?


Process

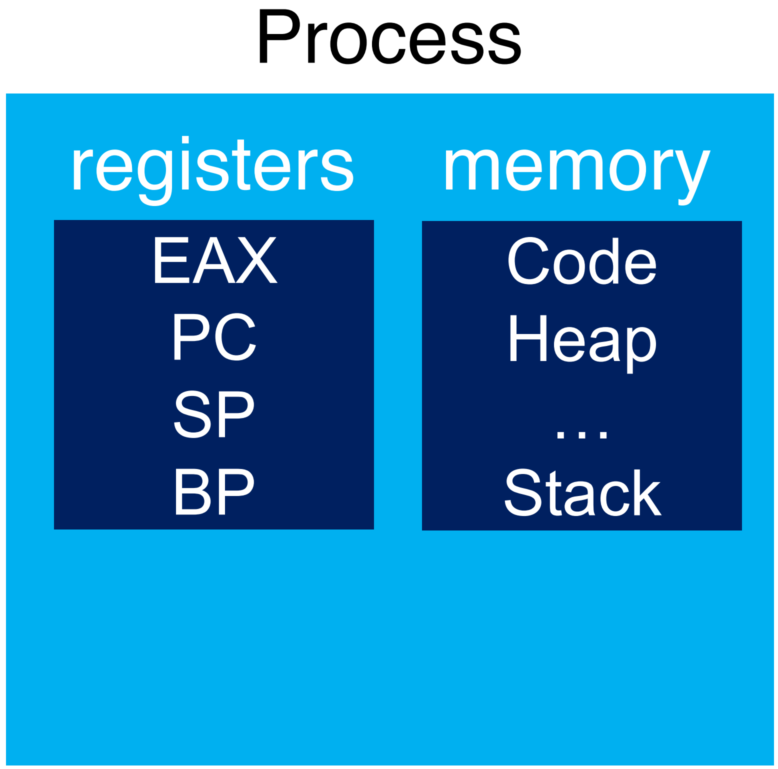
What things change as a program runs?

What is in a Process?



What things change as a program runs?

What is in a Process?



Process

registers

EAX
PC
SP
BP

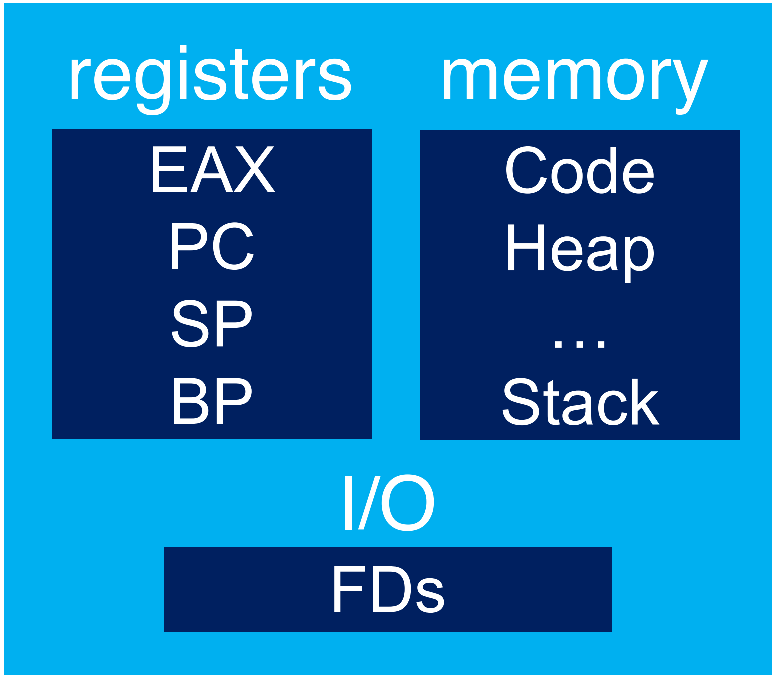
memory

Code
Heap
…
Stack

What things change as a program runs?

What is in a Process?

Process



registers

EAX
PC
SP
BP

memory

Code
Heap
…
Stack

I/O

FDs

What things change as a program runs?

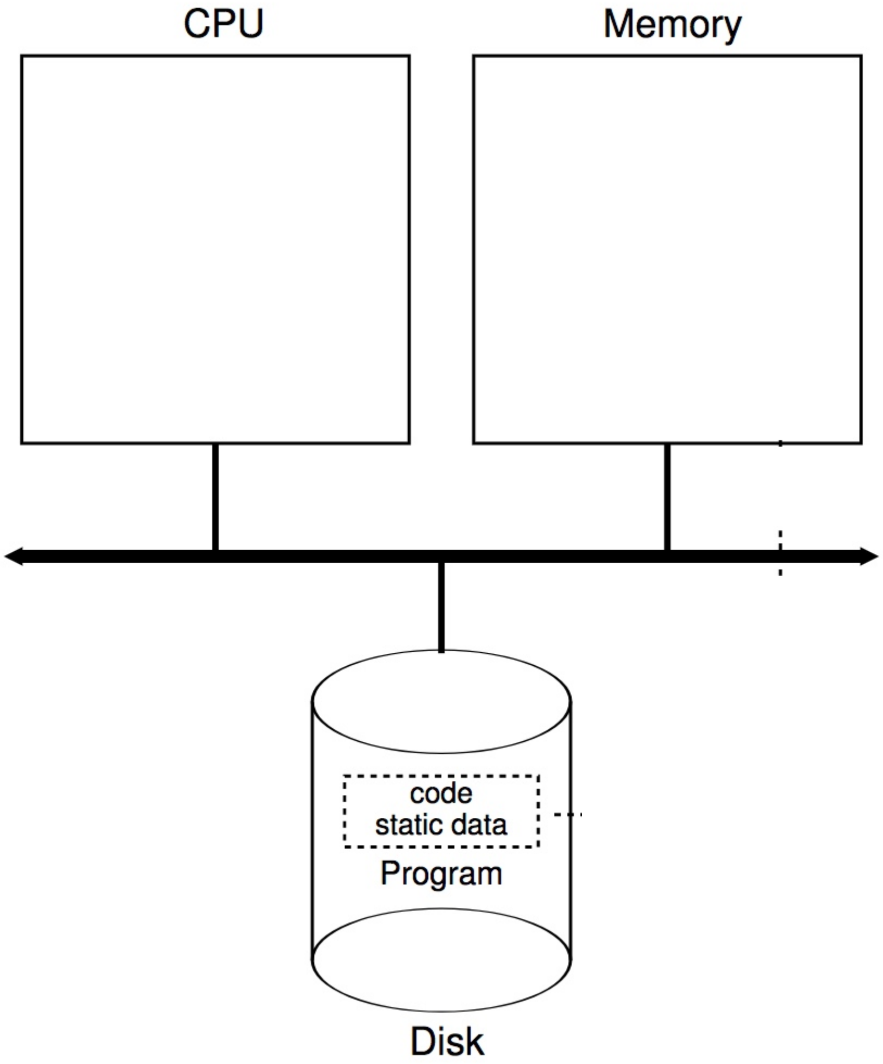
Peeking Inside

- Processes share code, but each has its own "context"
- CPU
  - Instruction pointer (Program Counter)
  - Stack pointer
- Memory
  - Set of memory addresses ("address space")
  - `cat /proc/<PID>/maps`
- Disk
  - Set of file descriptors
  - `cat /proc/<PID>/fdinfo/*`

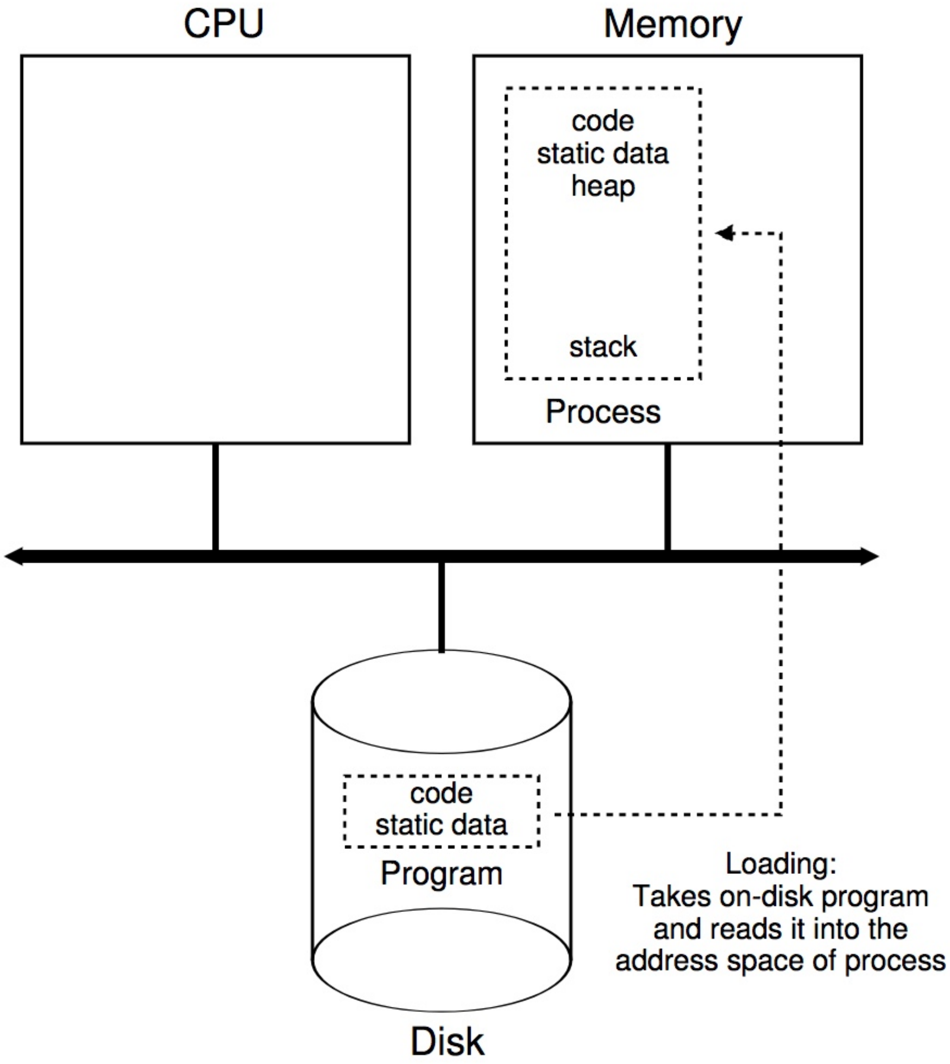
Process Creation

- Principle events that cause process creation
  - System initialization
  - Execution of a process creation system call by a running process
  - User request to create a process

Process Creation

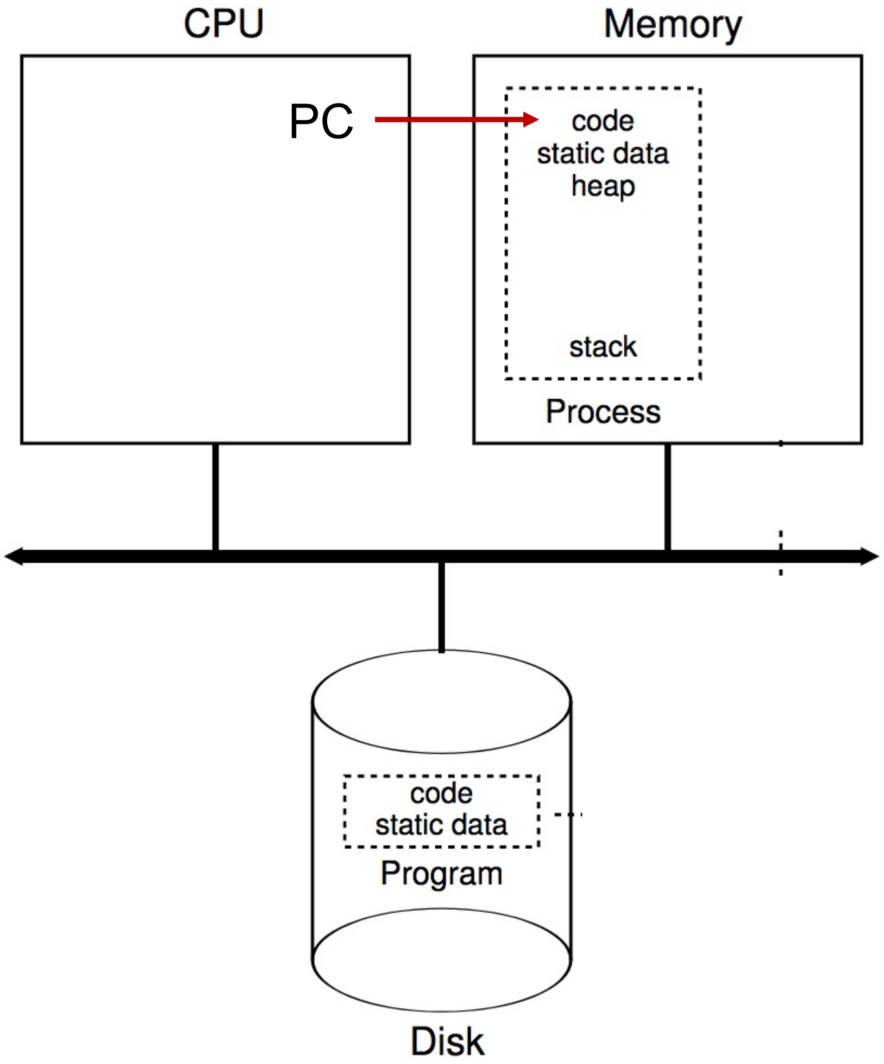


Process Creation



Loading:
Takes on-disk program
and reads it into the
address space of process

Process Creation

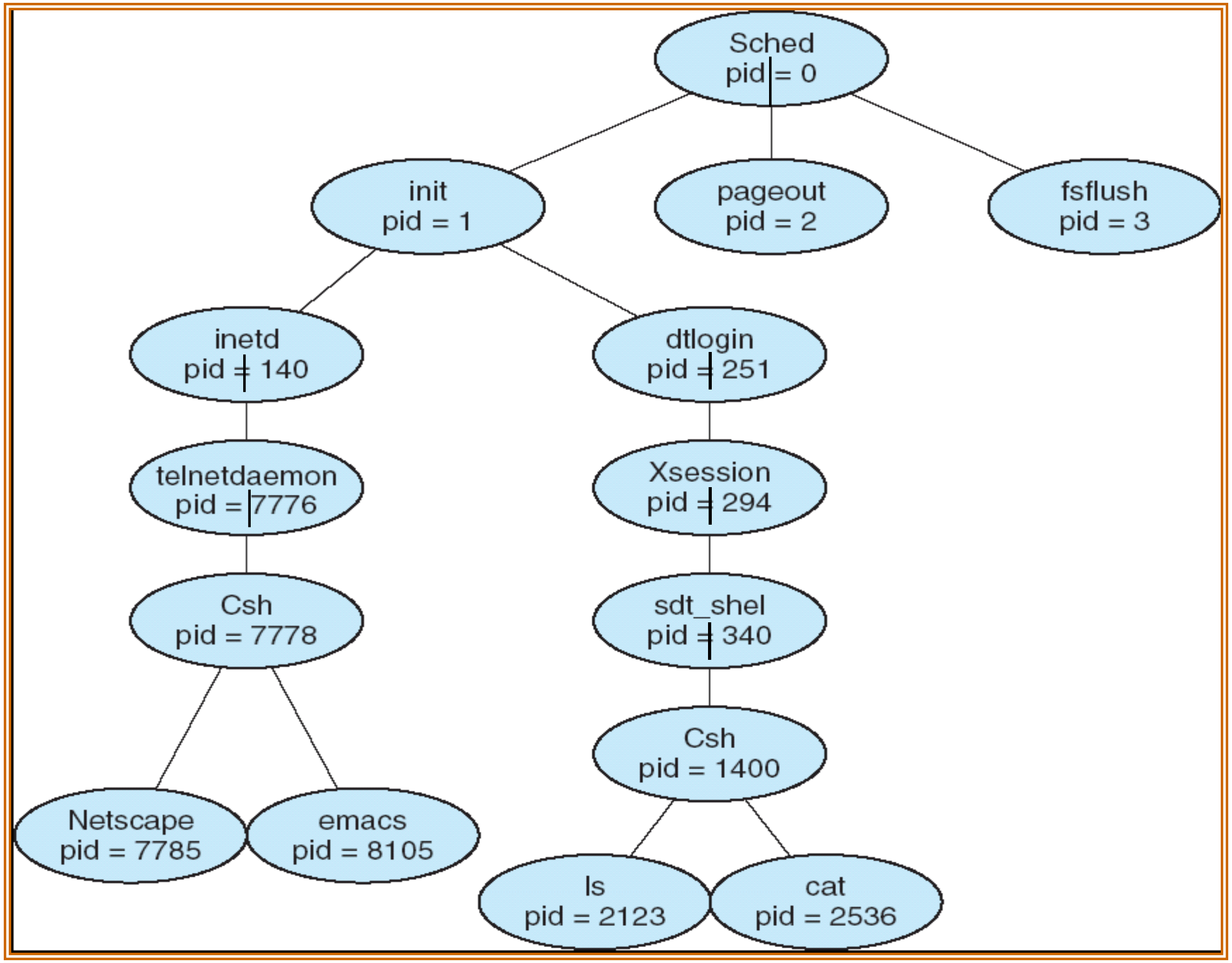


Process Creation (cont.)

- Parent process creates children processes, which, in turn create other processes, forming a tree (**hierarchy**) of processes

- **Questions**:
  - Will the parent and child execute concurrently?
  - How will the address space of the child be related to that of the parent?
  - Will the parent and child share some resources?

An Example Process Tree



How to View Process Tree in Linux?

- `% ps auxf`
  - ‘`f`’ is the option to show the process tree

- `% pstree`

Process Creation in Linux

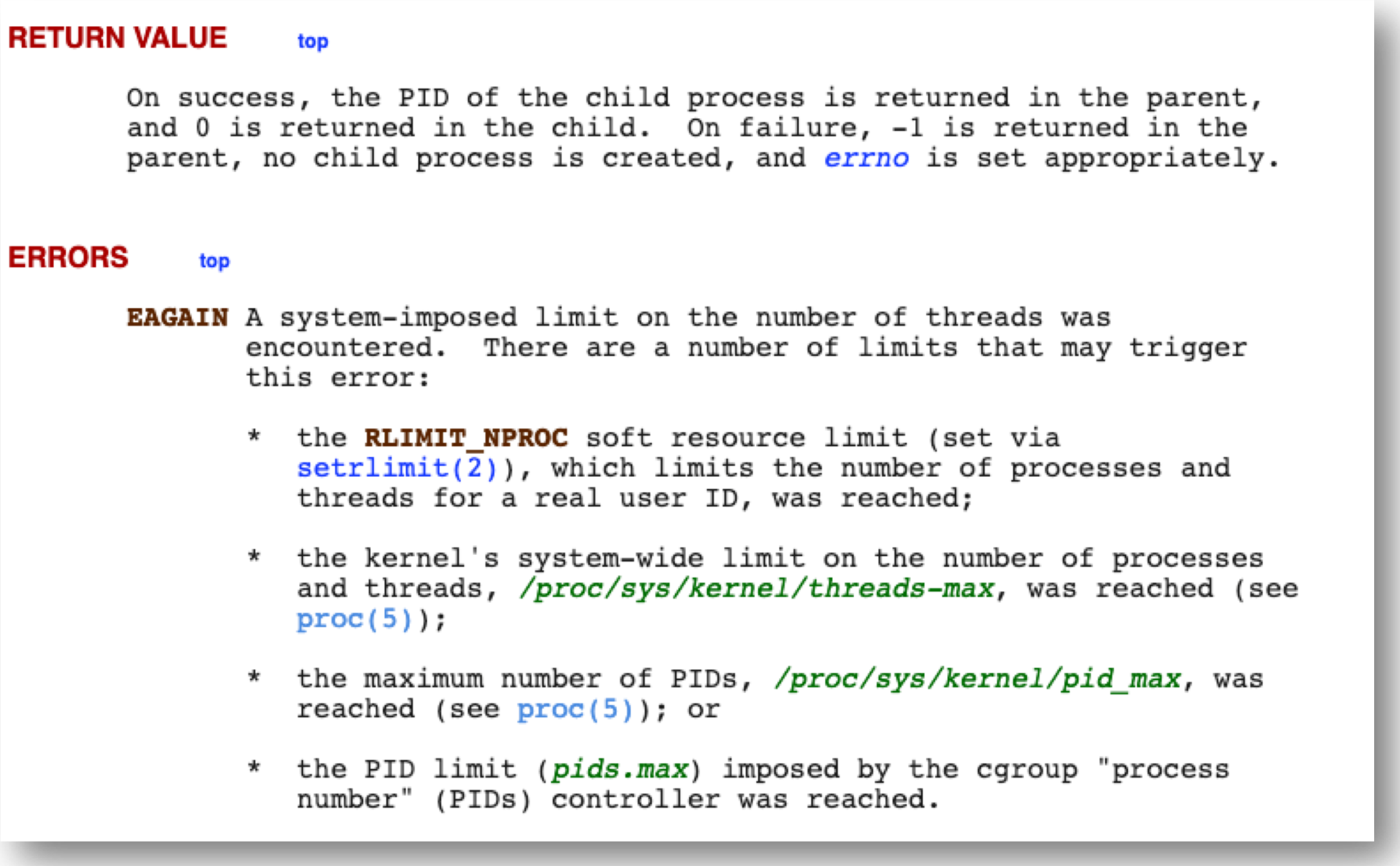
- Each process has a process identifier (pid)
- The parent executes `fork()` system call to spawn a child
- The child process has a separate copy of the parent's address space
- Both the parent and the child continue execution at the instruction following the `fork()` system call
- The return value for the `fork()` system call is
  - **zero** value for the new (**child**) process
  - **non-zero** `pid` for the **parent** process
- Typically, a process can execute a system call like `execl()` to load a binary file into memory

This is really the pid of the child process

Simply the return value of fork() in the context of the new child proc

man page of `fork()`

http://man7.org/linux/man-pages/man2/fork.2.html



RETURN VALUE top

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and *errno* is set appropriately.

ERRORS top

EAGAIN A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

* the **RLIMIT_NPROC** soft resource limit (set via setrlimit(2)), which limits the number of processes and threads for a real user ID, was reached;

* the kernel's system-wide limit on the number of processes and threads, */proc/sys/kernel/threads-max*, was reached (see proc(5));

* the maximum number of PIDs, */proc/sys/kernel/pid_max*, was reached (see proc(5)); or

* the PID limit (*pids.max*) imposed by the cgroup "process number" (PIDs) controller was reached.

Example Program with `fork()`

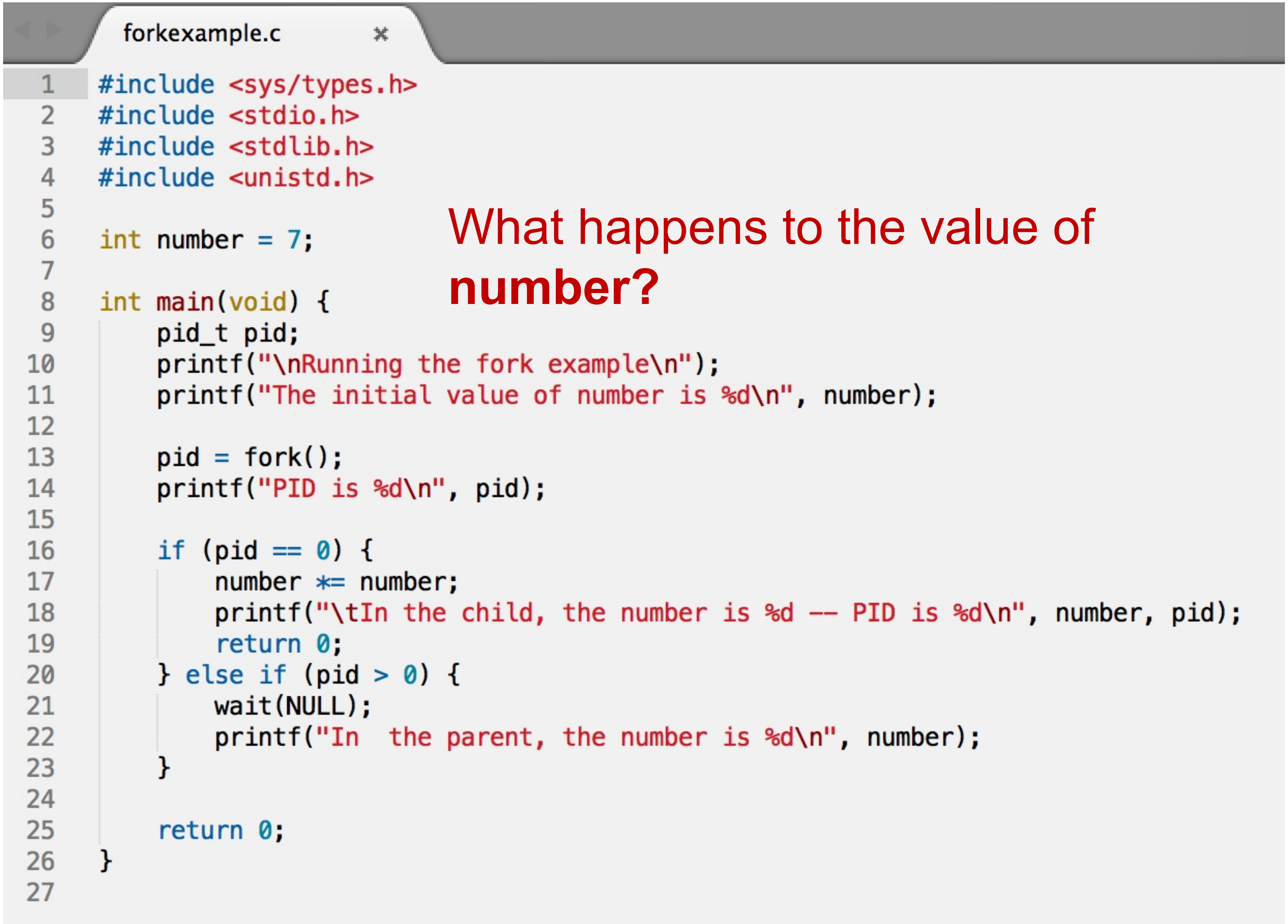
```
void main () {
     int pid;

     pid = fork();
     if  (pid < 0) {/* error_msg */}
     else if (pid == 0) {  /* child process */
               execl(“/bin/ls”, “ls”, NULL); /* execute ls */
     } else {                          /* parent process */
               /* parent will wait for the child to complete */
               wait(NULL);
               exit(0);
     }
     return;
}
```

A Very Simple Shell using `fork()`

```
while (1) {
        type_prompt();
        read_command(cmd);
        pid = fork();
         if  (pid < 0) {/* error_msg */}
         else if (pid == 0) { /* child process */
             execute_command(cmd);
         } else {              /* parent process */
              wait(NULL);
         }
}
```

More example: fork 1



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

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the fork example\n");
    printf("The initial value of number is %d\n", number);

    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        number *= number;
        printf("\tIn the child, the number is %d -- PID is %d\n", number, pid);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, the number is %d\n", number);
    }

    return 0;
}
```

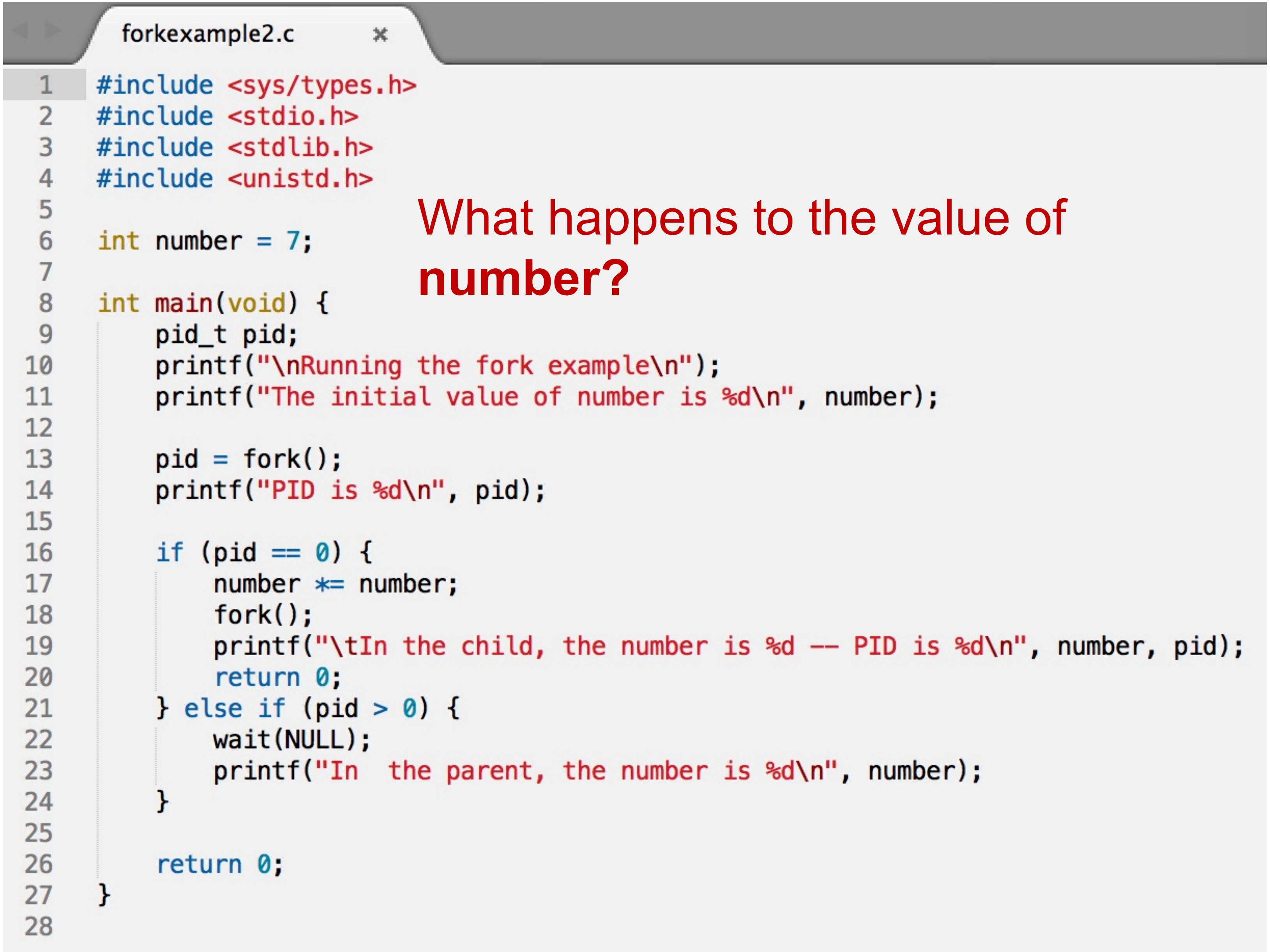
What happens to the value of **number?**

Results

```
./forkexample1


Running the fork example
The initial value of number is 7
 PID is 2137
 PID is 0
        In the child, the number is 49 -- PID is 0
In the parent, the number is 7
```

Further more example: fork 2



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

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the fork example\n");
    printf("The initial value of number is %d\n", number);

    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        number *= number;
        fork();
        printf("\tIn the child, the number is %d -- PID is %d\n", number, pid);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, the number is %d\n", number);
    }

    return 0;
}

```

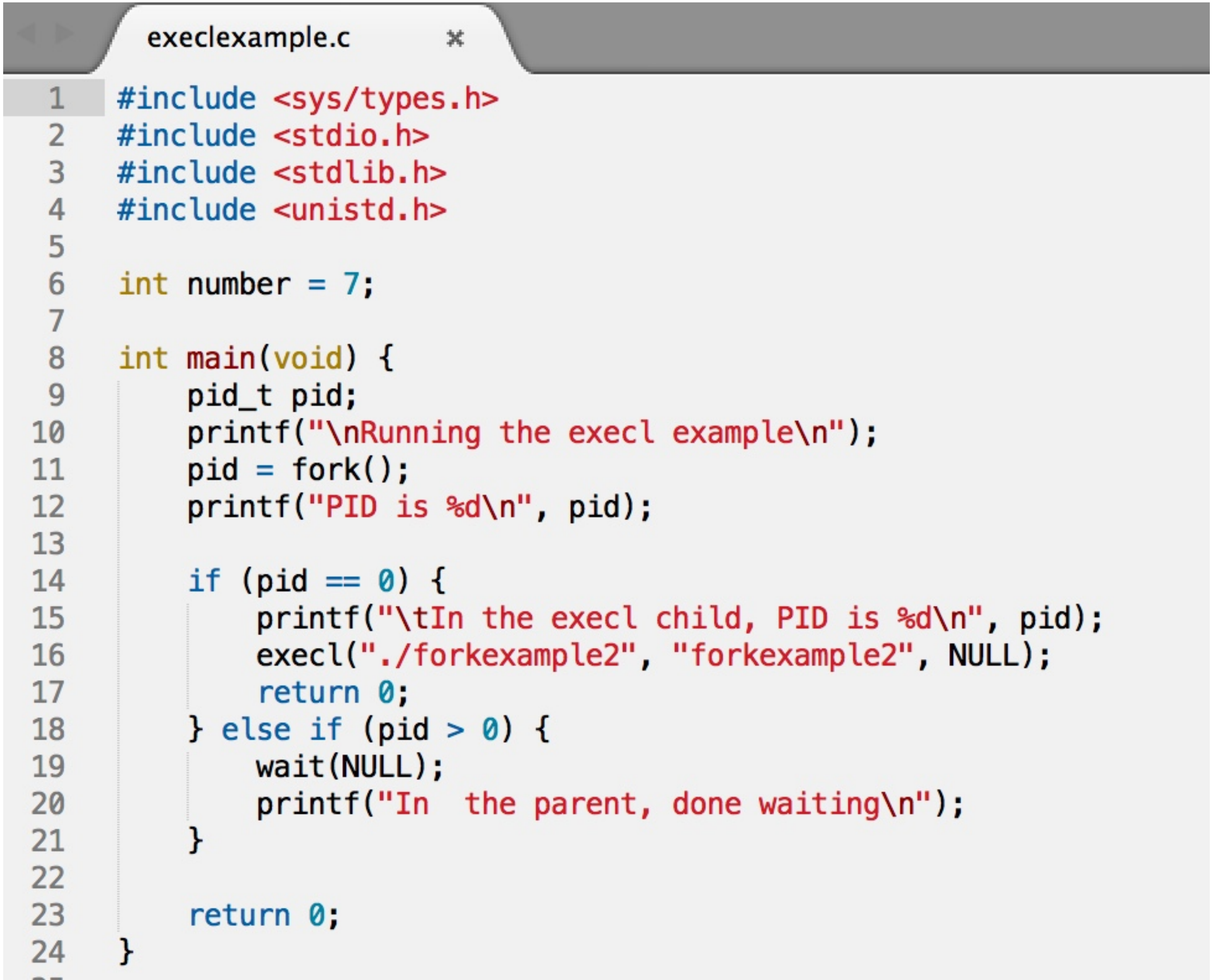
What happens to the value of **number?**

Results

```
./forkexample2

Running the fork example
The initial value of number is 7
 PID is 2164
 PID is 0
         In the child, the number is 49 -- PID is 0
         In the child, the number is 49 -- PID is 0
In the parent, the number is 7
```

execl vs. fork



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

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the execl example\n");
    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        printf("\tIn the execl child, PID is %d\n", pid);
        execl("./forkexample2", "forkexample2", NULL);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, done waiting\n");
    }

    return 0;
}
```

Results

```
./execlexample
Running the execl example
 PID is 2179
 PID is 0

                In the execl child,   PID is 0

Running the fork example
The initial value of number is 7
 PID is 2180
 PID is 0

                In the child, the number is 49 -- PID is 0
                In the child, the number is 49 -- PID is 0
In the parent, the number is 7
In the parent, done waiting
```

forkexample2