Notes on fork() function

Yogesh Virkar

Department of Computer Science,

University of Colorado, Boulder.

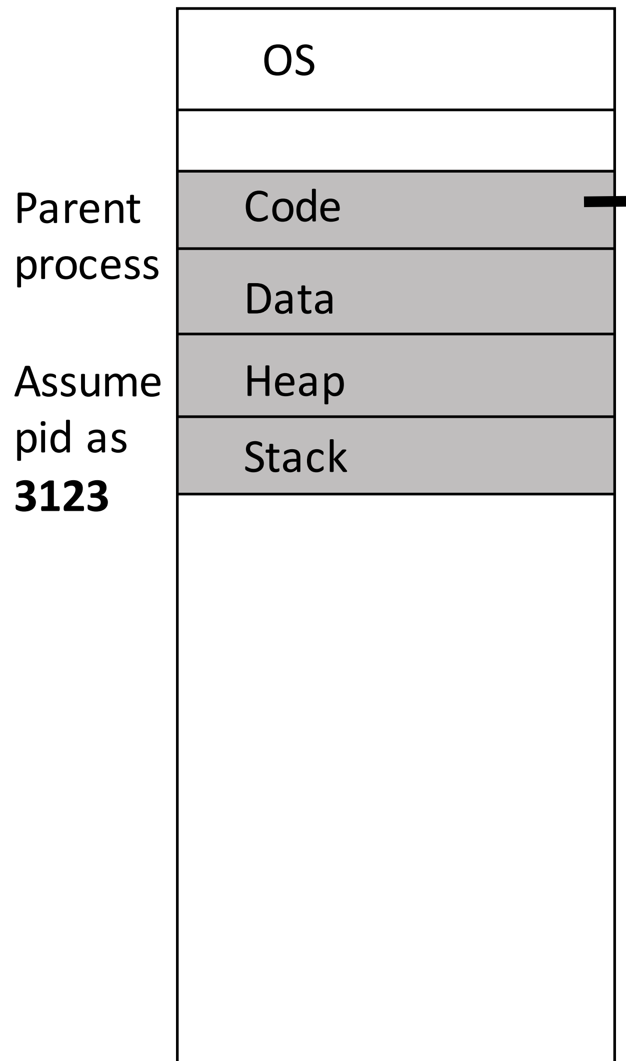
Forking

- A process is a running instance of a program.
- We can multiple instances of the same program.
- Each process has a unique id called as the process id.
- The fork() function creates a new process, i.e., new instance of the same program. The newly created process is called the child process.
  - fork() is **called once by the parent**
  - fork() **returns twice:**

  1. **returns 0 to child process**
  2. **returns child's process id to the parent process**

Forking (2)
(based on Prof. Han's lecture slides Chpt 8.1)

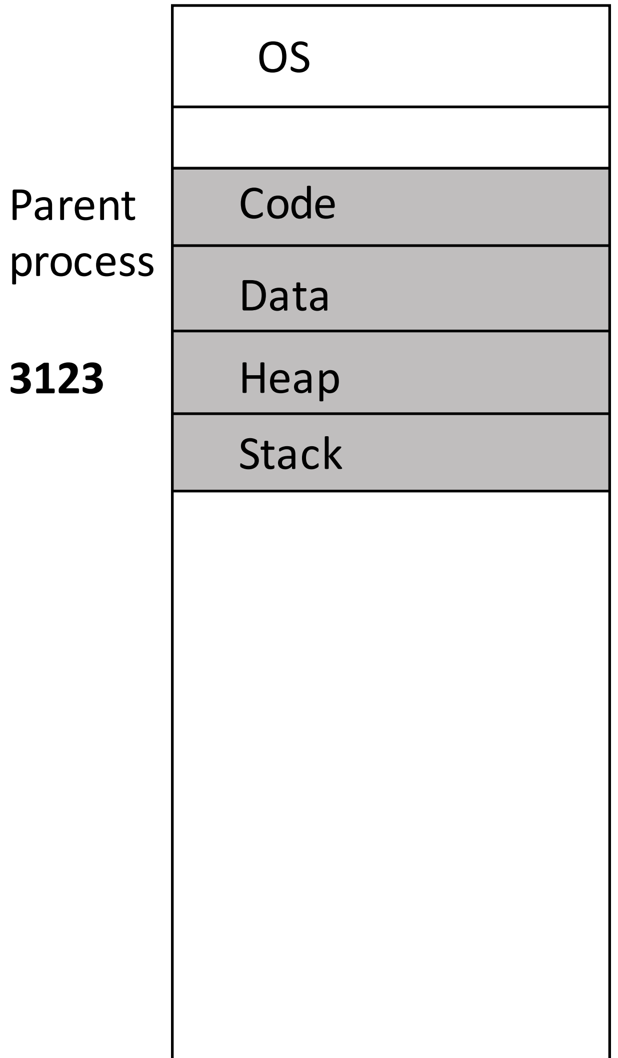
Memory before fork()



```
int retval = fork();

if (returnvalue == 0) {
    printf("I am the child.");
}
else {
    printf("I am the parent.");
}
```

Forking (3)
(based on Prof. Han's lecture slides Chpt 8.1)

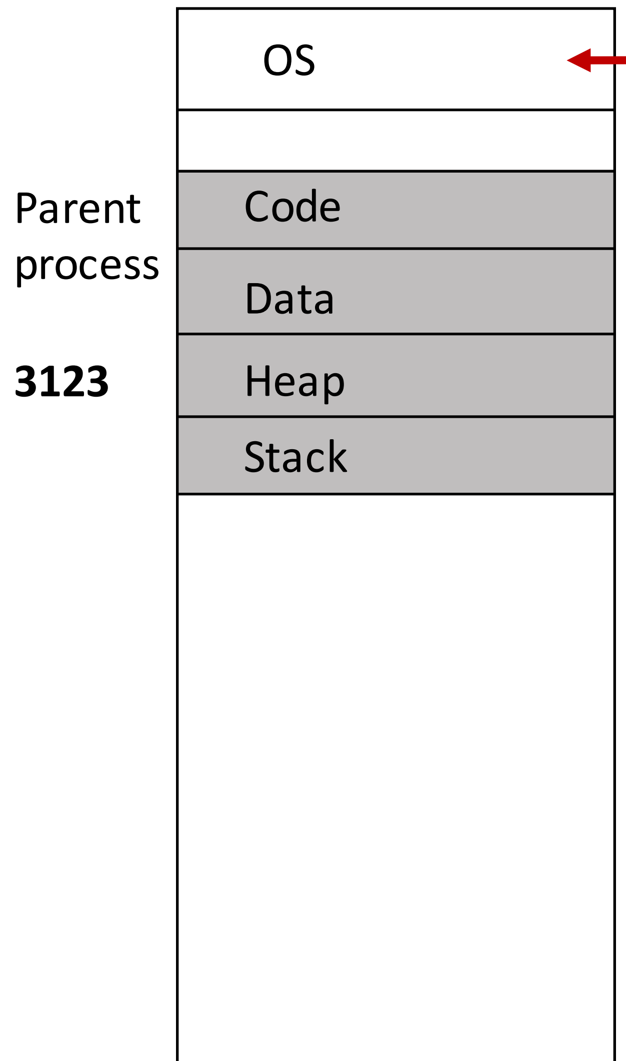
Memory before fork()



```
→ int retval = fork();

  if (retval == 0) {
     printf("I am the child.");
  }
  else {
     printf("I am the parent.");
  }
```

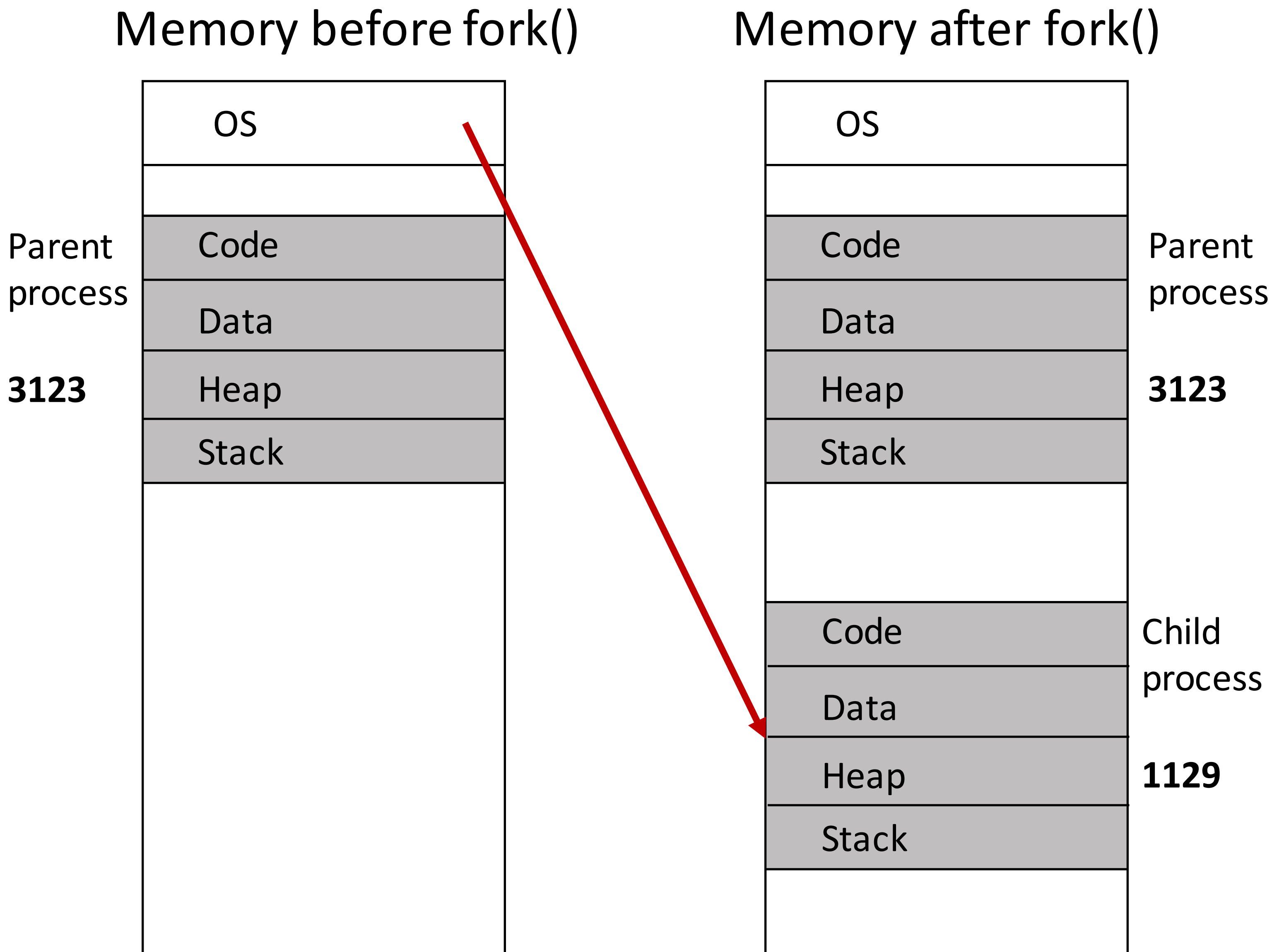
Forking (4)
(based on Prof. Han’s lecture slides Chpt 8.1)

Memory before fork()



```
int retval = fork();

if (retval == 0) {
    printf(“I am the child.”);
}
else {
    printf(“I am the parent.”);
}
```

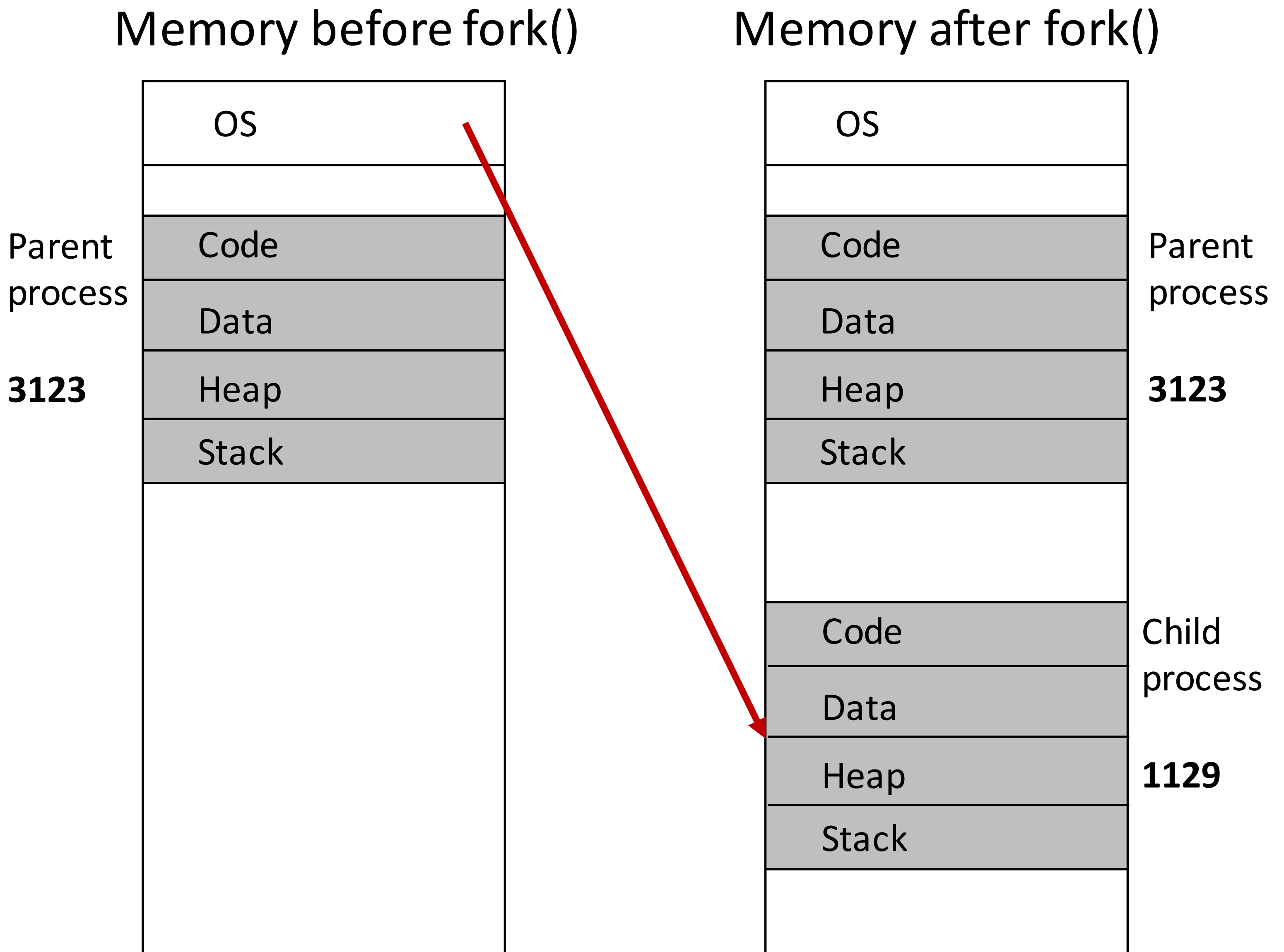
Forking (5)
(based on Prof. Han's lecture slides Chpt 8.1)



```
int retval = fork();

if (retval == 0) {
    printf("I am the child.");
}
else {
    printf("I am the parent.");
}
```

fork() talks to the operating system to create a new process (child process). Let us assume it gets a process id 1129.

Forking (6)
(based on Prof. Han's lecture slides Chpt 8.1)

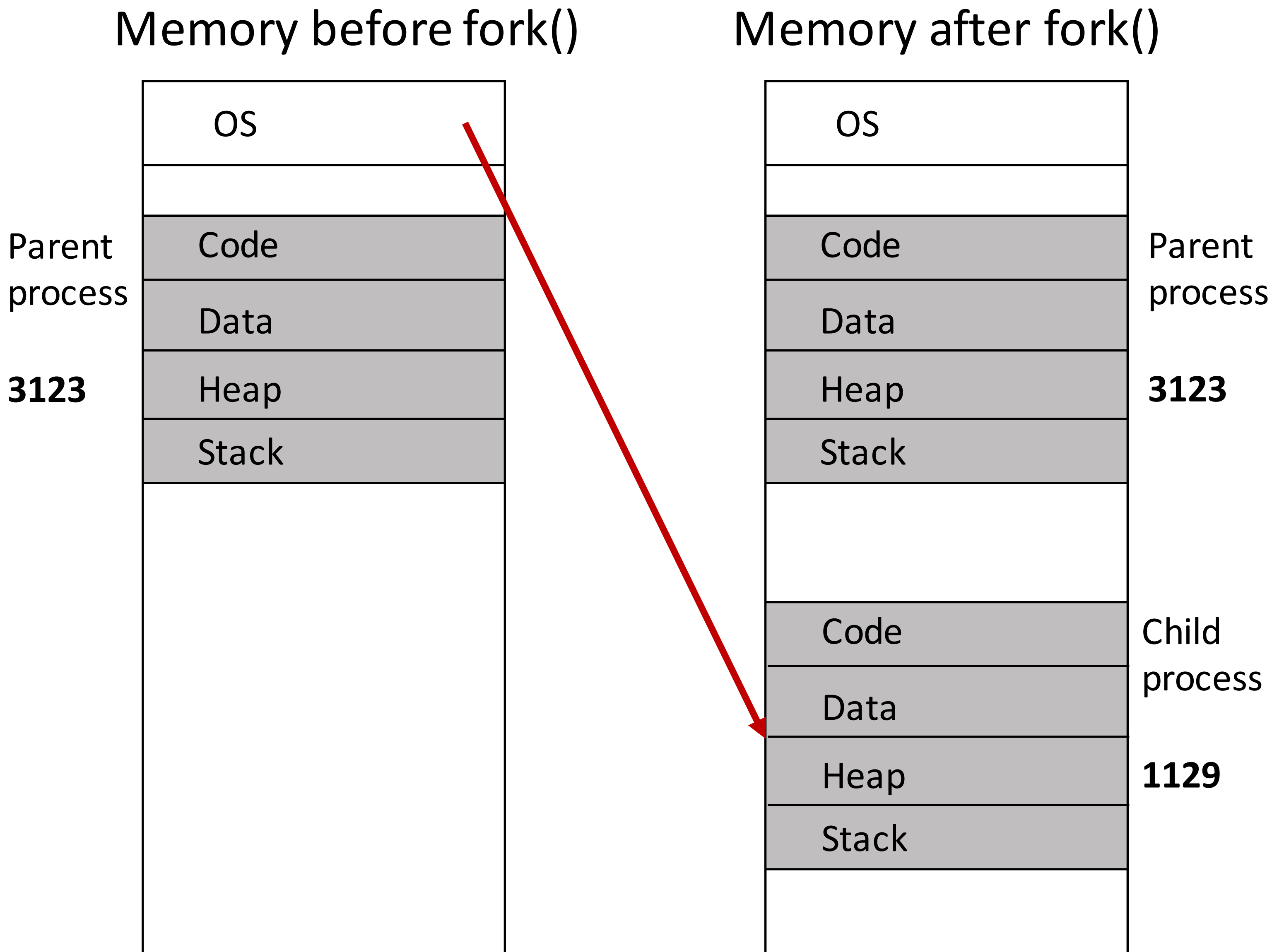


```
int retval = fork();

if (retval == 0) {
    printf("I am the child.");
}
else {
    printf("I am the parent.");
}
```

fork() talks to the operating system to create a new process (child process). Let us assume it gets a process id 1129.

Assume longer arrow for parent process

Forking (7)
(based on Prof. Han's lecture slides Chpt 8.1)



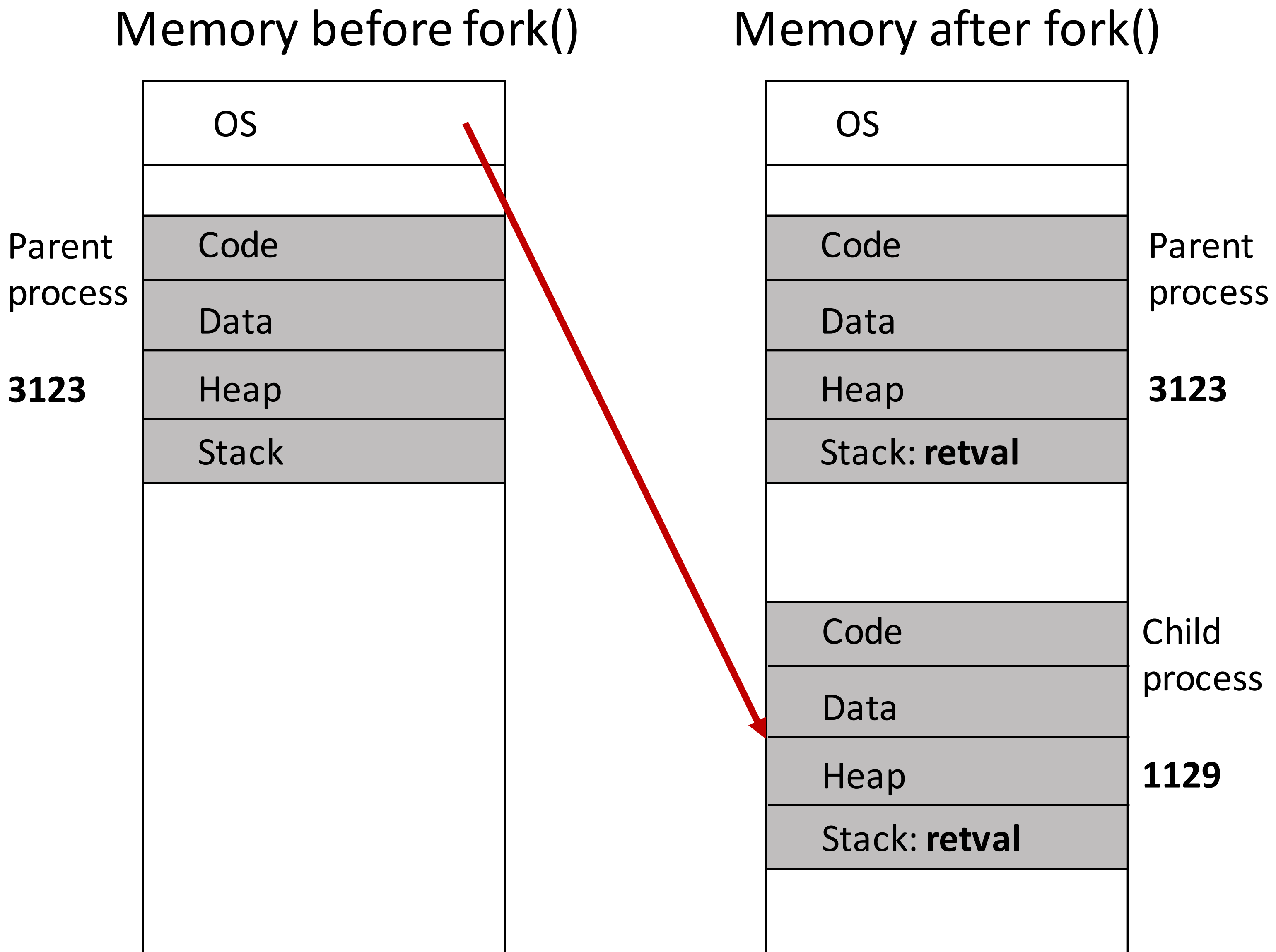
```
int retval = fork();

if (retval == 0) {
   printf("I am the child.");
}
else {
   printf("I am the parent.");
}
```

fork() talks to the operating system to create a new process (child process). Let us assume it gets a process id 1129.

Assume longer arrow for parent process

Note: Unlike threads, processes do not share memory (hence they are called heavyweight).

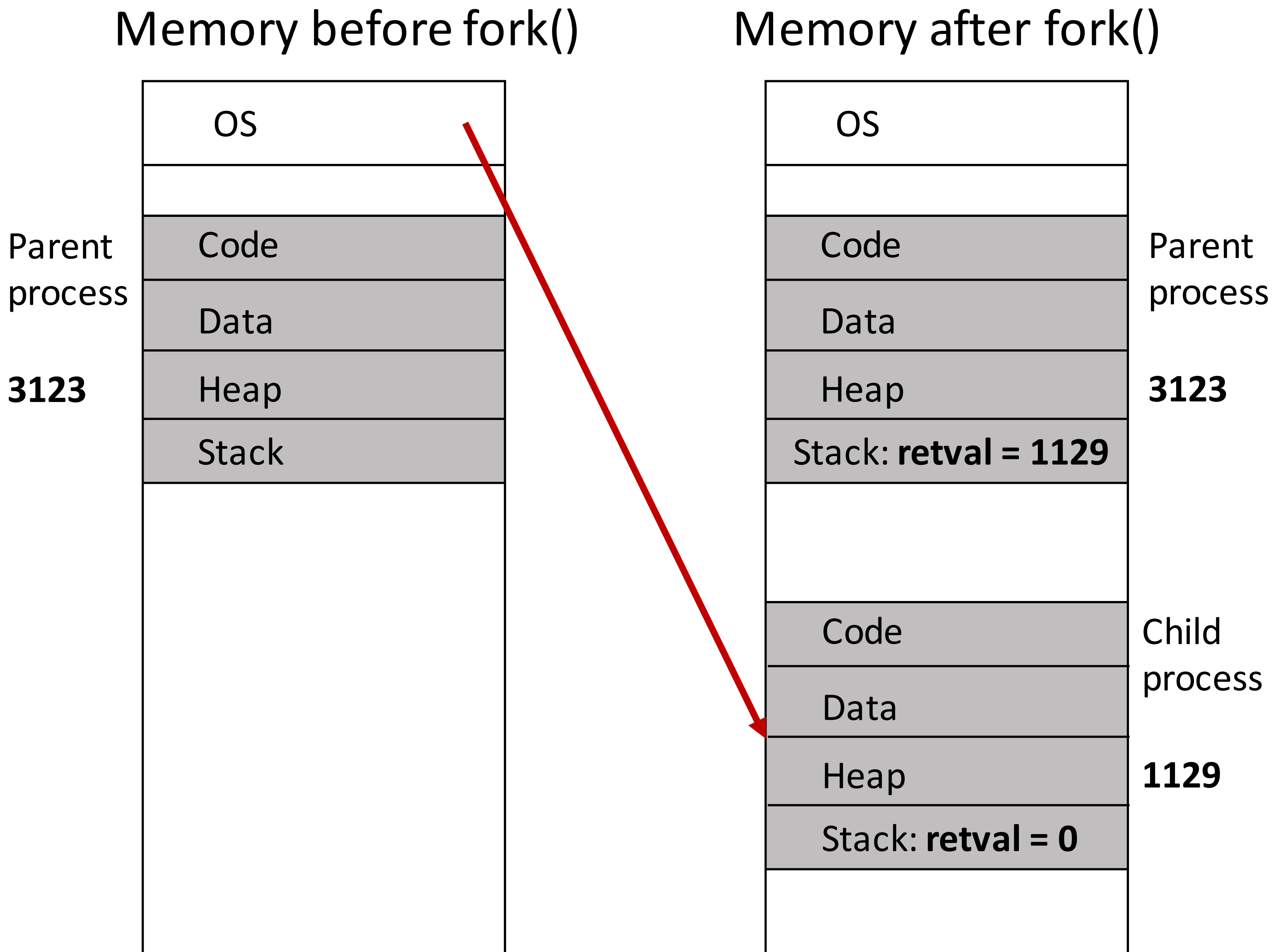
Forking (8)
(based on Prof. Han's lecture slides Chpt 8.1)



```
int retval = fork();

if (retval == 0) {
    printf("I am the child.");
}
else {
    printf("I am the parent.");
}
```

The fork() function needs to return a value to both parent and child processes.

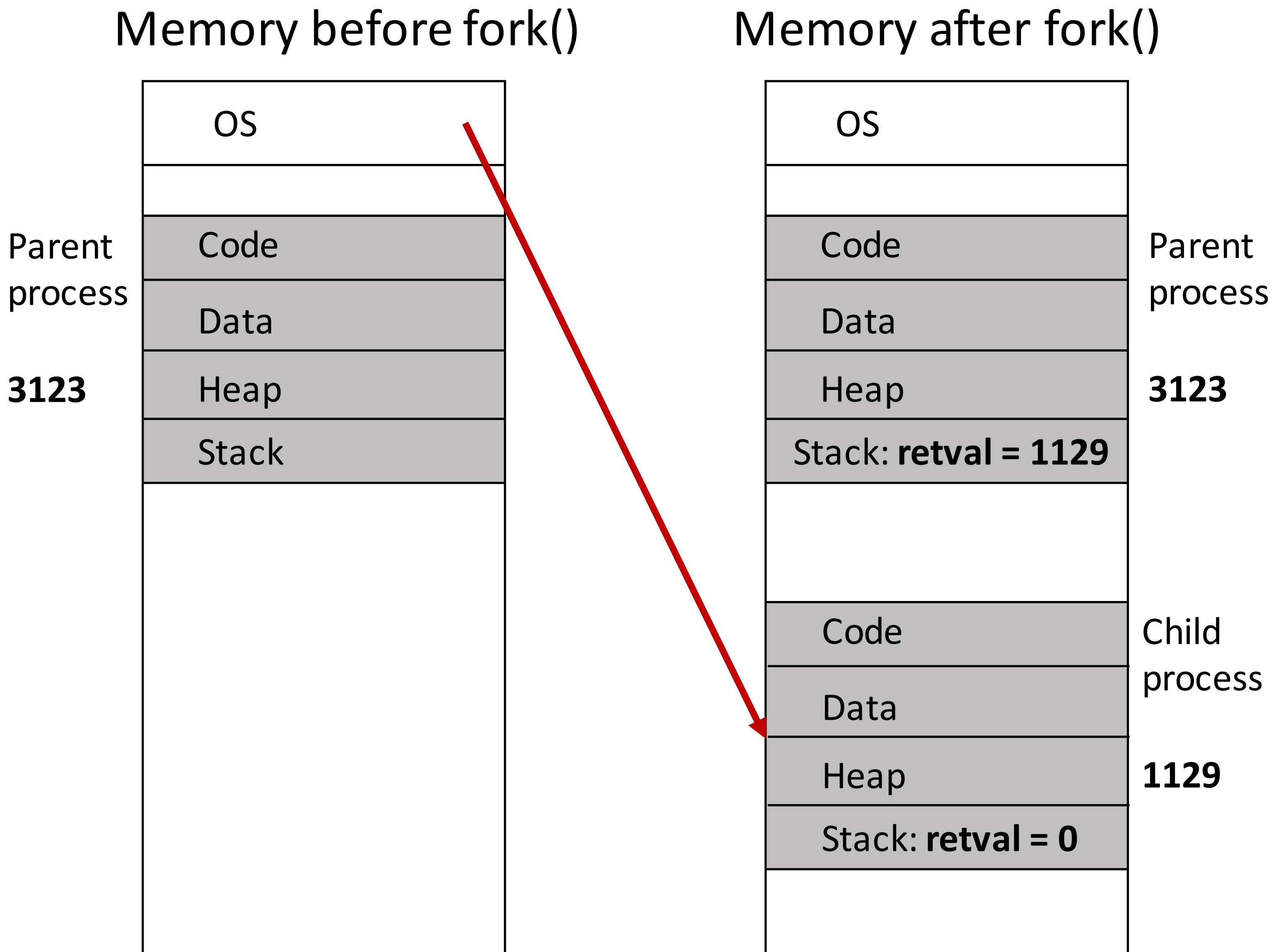
Forking (9)
(based on Prof. Han's lecture slides Chpt 8.1)



```
int retval = fork();

if (retval == 0) {
    printf("I am the child.");
}
else {
    printf("I am the parent.");
}
```

Child process gets 0.
Parent process gets 1129 (i.e., child's process id).

Forking (10)
(based on Prof. Han’s lecture slides Chpt 8.1)

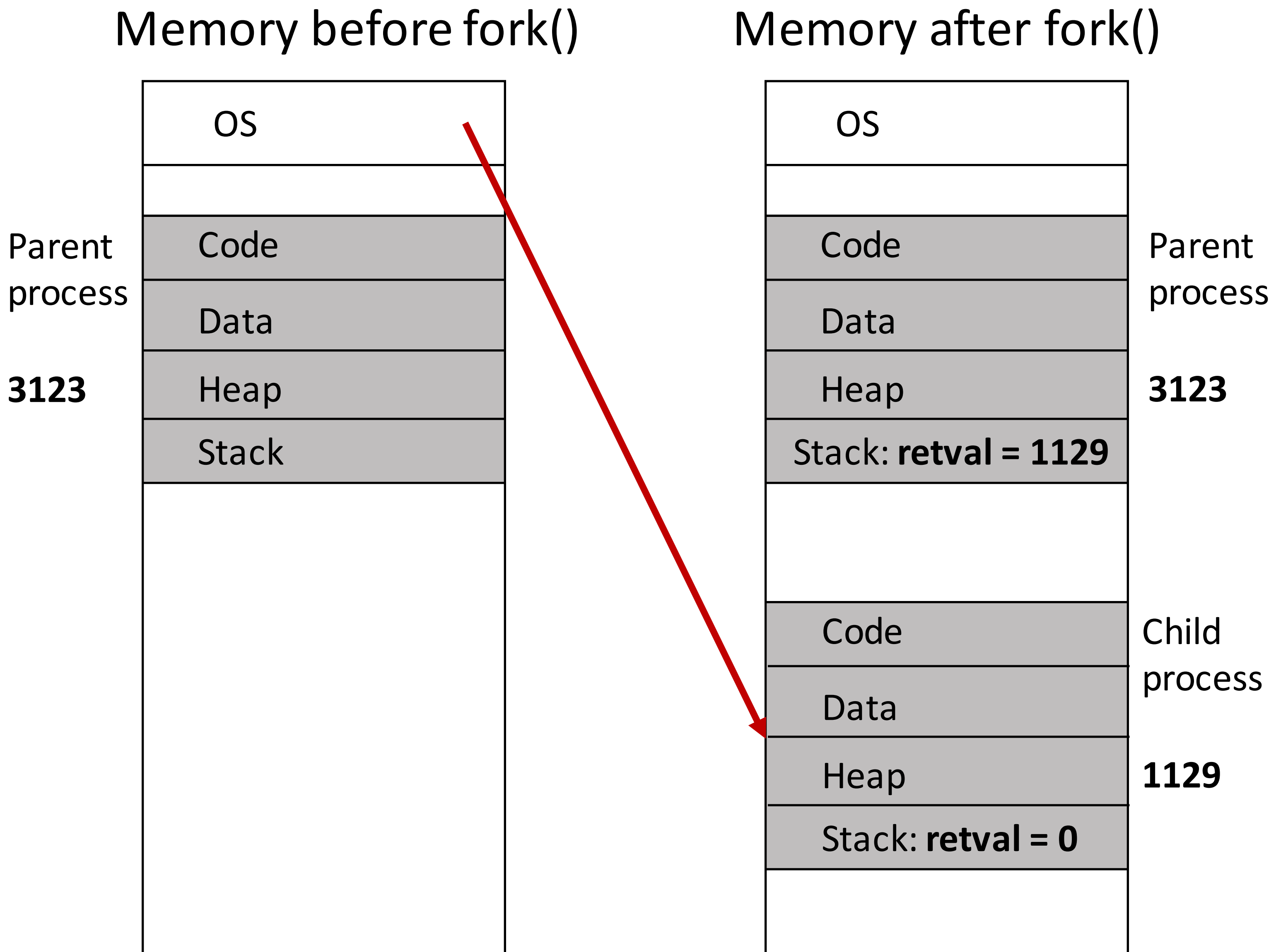


```
int retval = fork();

if (retval == 0) {
    printf(“I am the child.”);
}
else {
    printf(“I am the parent.”);
}
```

Thus, the if statement is true only for the child and NOT the parent.

This distinction in return value allows us to programmatically separate the parent from the child process.

Forking (11) (based on Prof. Han’s lecture slides Chpt 8.1)



```
int retval = fork();

if (retval == 0) {
    printf(“I am the child.”);
}
else {
    printf(“I am the parent.”);
}
```

The parent and child print the respective printf statements. Since they execute concurrently, we cannot guarantee the ordering of these printf statements.

Forking: How many printfs get executed?

```
#include "csapp.h"
void doit() {
Fork();
Fork();
printf("hello\n");
return;
}

int main()
{
doit();
printf("hello\n");
exit(0);
}
```

Forking: How many printfs get executed?

```
  #include "csapp.h"
  void doit() {
2 Fork();
3 Fork();
4 printf("hello\n");
5 return;
  }

  int main()
  {
1 doit();
6 printf("hello\n");
7 exit(0);
  }
```

The numbers mark the time ordering of the lines of code.

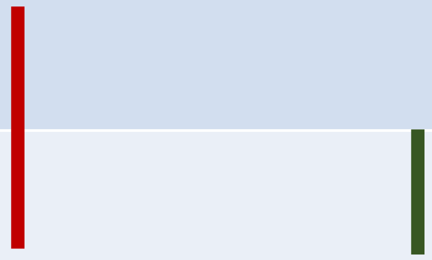
Forking: How many printfs?

```
  #include "csapp.h"
  void doit() {
2 Fork();
3 Fork();
4 printf("hello\n");
5 return;
  }

  int main()
  {
→1 doit();
6 printf("hello\n");
7 exit(0);
  }
```

Time	Processes
1	│
2	
3	
4	
5	
6	
7	

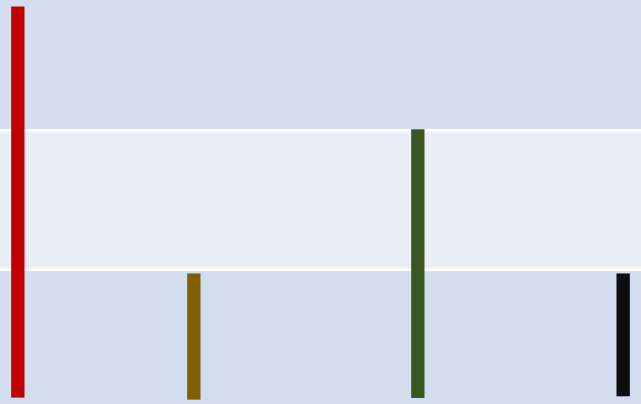
Forking: How many printfs?

```
  #include "csapp.h"
  void doit() {
→ 2 Fork();
  3 Fork();
  4 printf("hello\n");
  5 return;
  }

  int main()
  {
→ 1 doit();
  6 printf("hello\n");
  7 exit(0);
  }
```

Time	Processes
1	
2	
3	
4	
5	
6	
7	

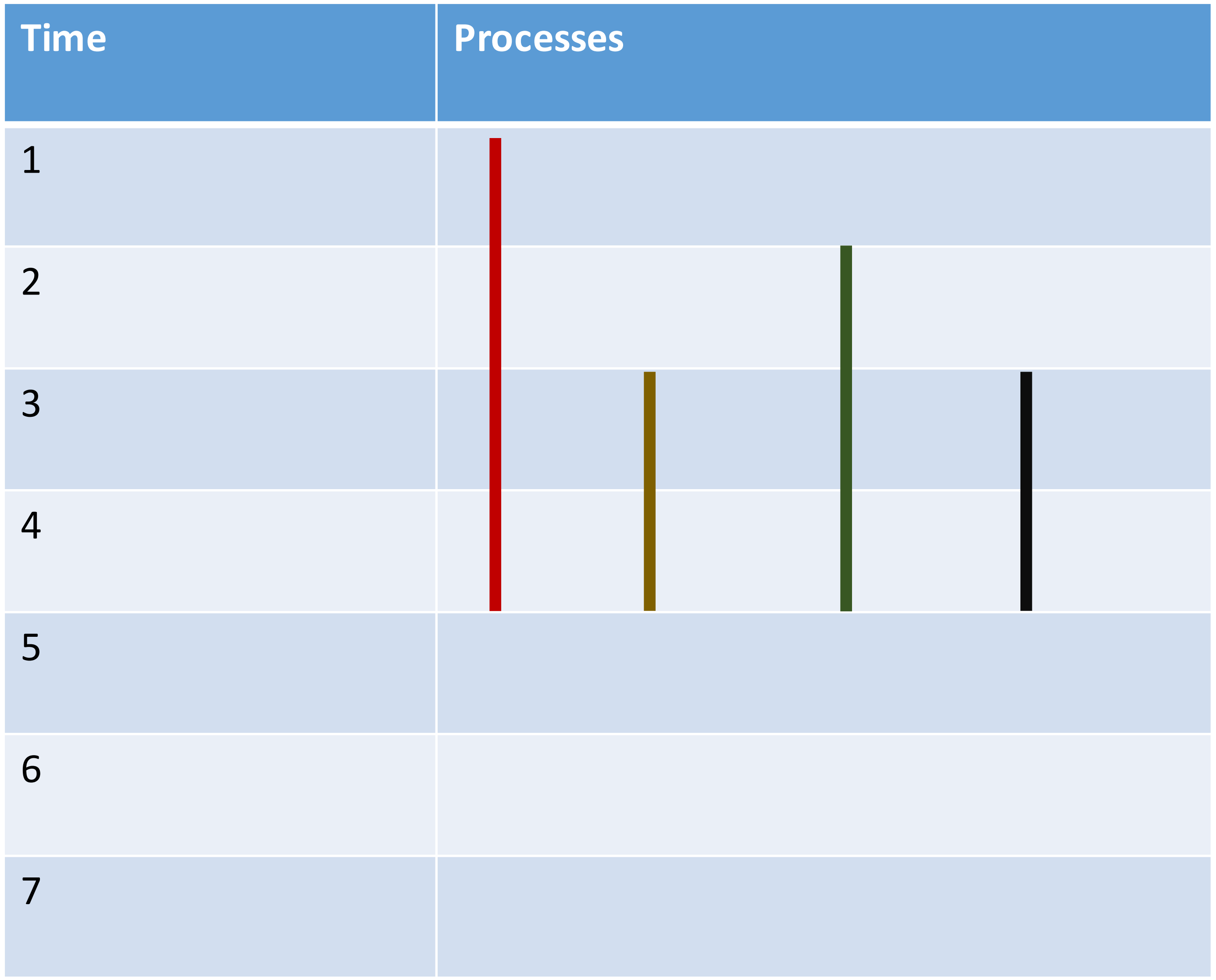
Forking: How many printfs?

```
    #include "csapp.h"
    void doit() {
→ 2 Fork();
→ 3 Fork();
  4 printf("hello\n");
  5 return;
    }

    int main()
    {
→ 1 doit();
  6 printf("hello\n");
  7 exit(0);
    }
```

Time	Processes
1	
2	
3	
4	
5	
6	
7	

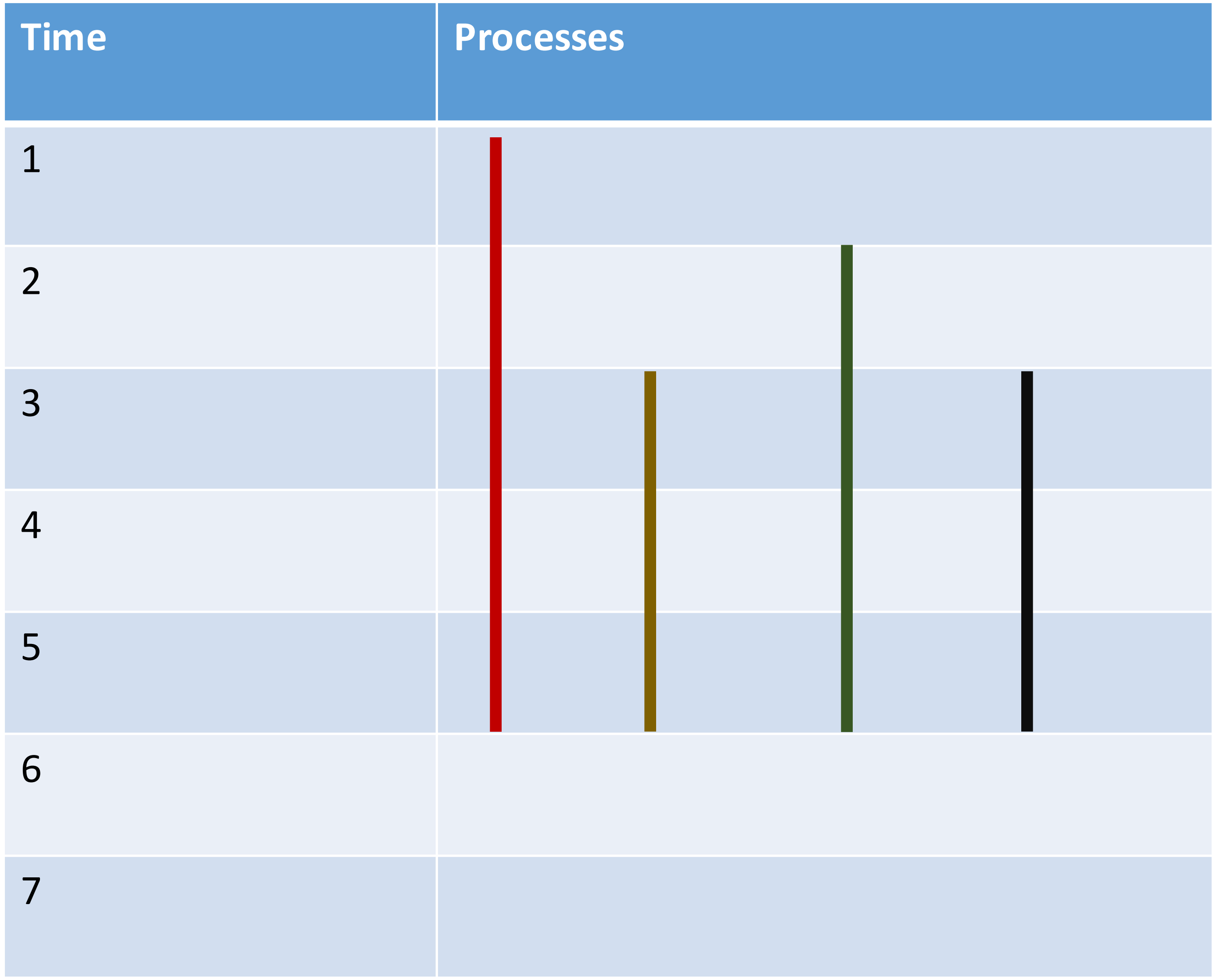
Forking: How many printfs? **4**

```
     #include "csapp.h"
     void doit() {
→ 2  Fork();
→ 3  Fork();
→ 4  printf("hello\n");
  5  return;
     }

     int main()
     {
→ 1  doit();
  6  printf("hello\n");
  7  exit(0);
     }
```



Forking: How many printfs? **4**

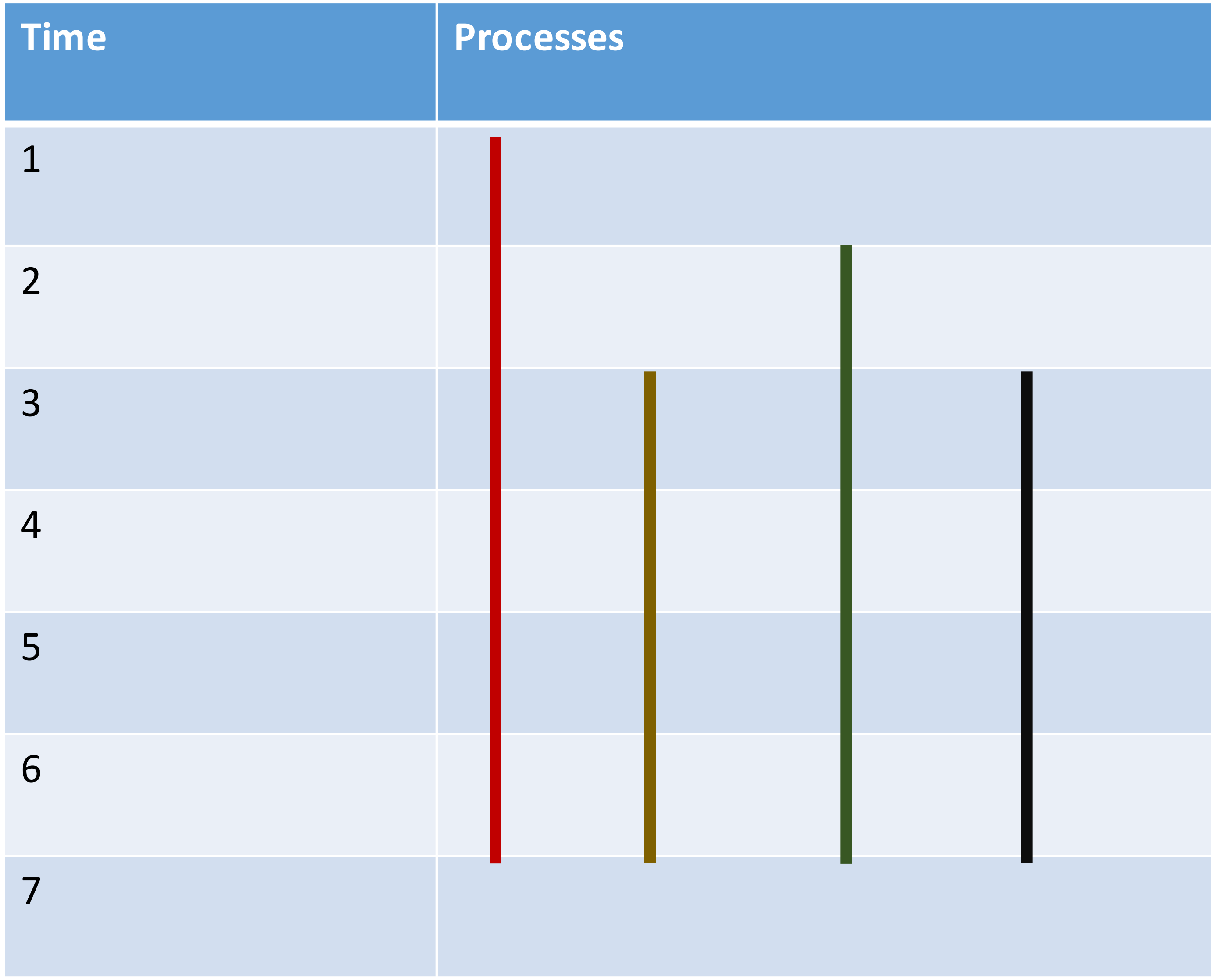
```
       #include "csapp.h"
       void doit() {
→ 2    Fork();
→ 3    Fork();
→ 4    printf("hello\n");
→ 5    return;
       }

       int main()
       {
→ 1    doit();
  6    printf("hello\n");
  7    exit(0);
       }
```



Time	Processes
1	
2	
3	
4	
5	
6	
7	

Forking: How many printfs? **4 + 4**

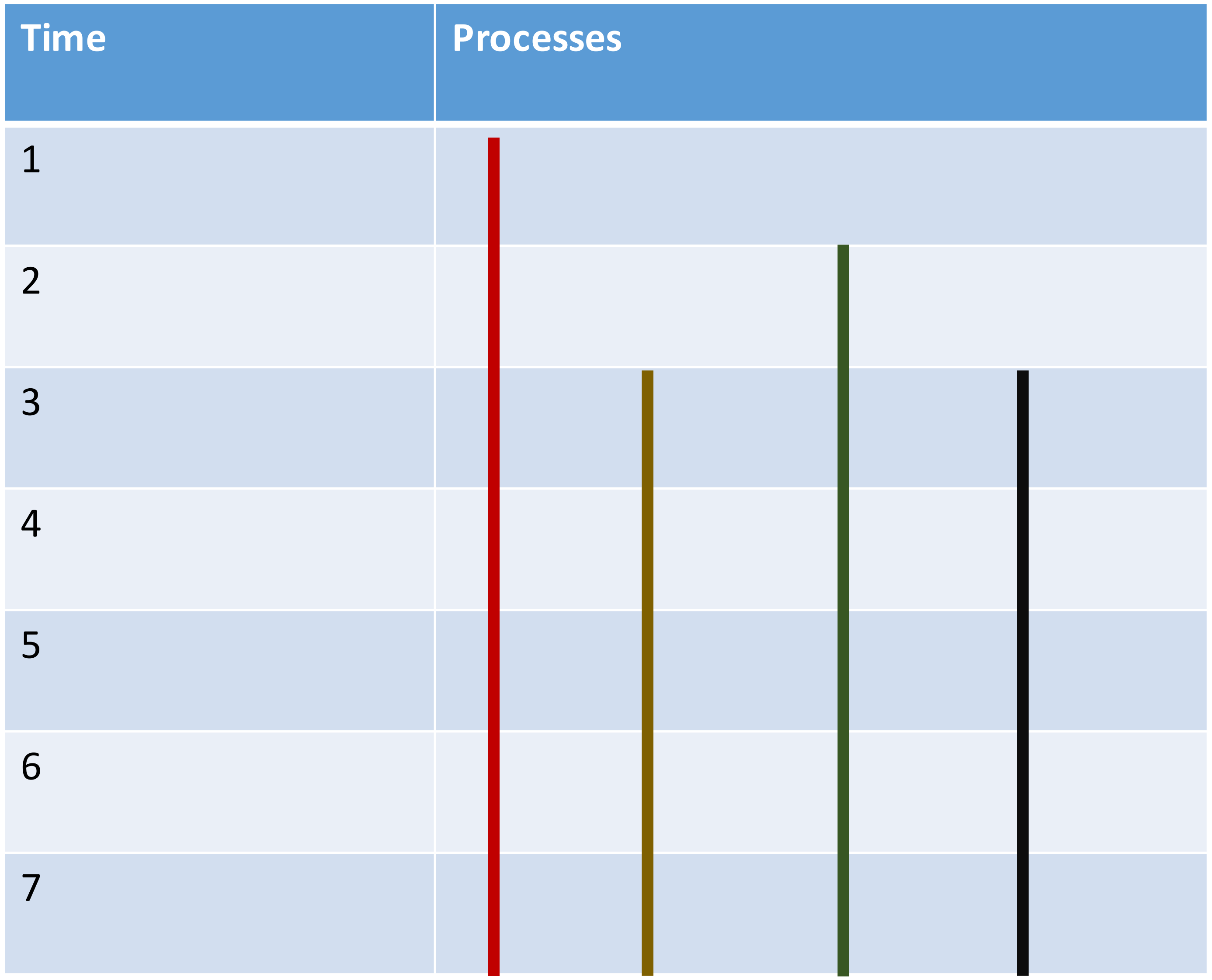
```
  #include "csapp.h"
  void doit() {
2   Fork();
3   Fork();
4   printf("hello\n");
5   return;
  }

  int main()
  {
1   doit();
6   printf("hello\n");
7   exit(0);
  }
```



Time	Processes
1	
2	
3	
4	
5	
6	
7	

Forking: How many printfs? **8**

```
        #include "csapp.h"
        void doit() {
→ 2     Fork();
→ 3     Fork();
→ 4     printf("hello\n");
→ 5     return;
        }

        int main()
        {
→ 1     doit();
→ 6     printf("hello\n");
→ 7     exit(0);
        }
```



Time
Processes
1
2
3
4
5
6
7