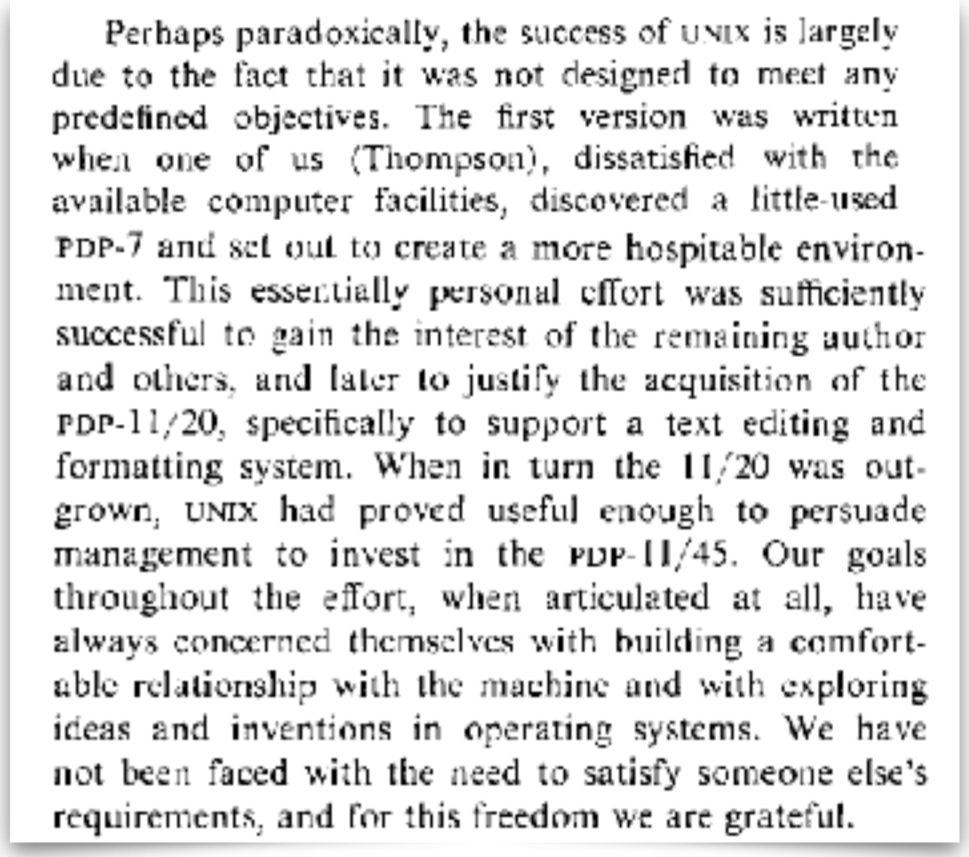
Design philosophy of operating systems (III)

Hung-Wei Tseng

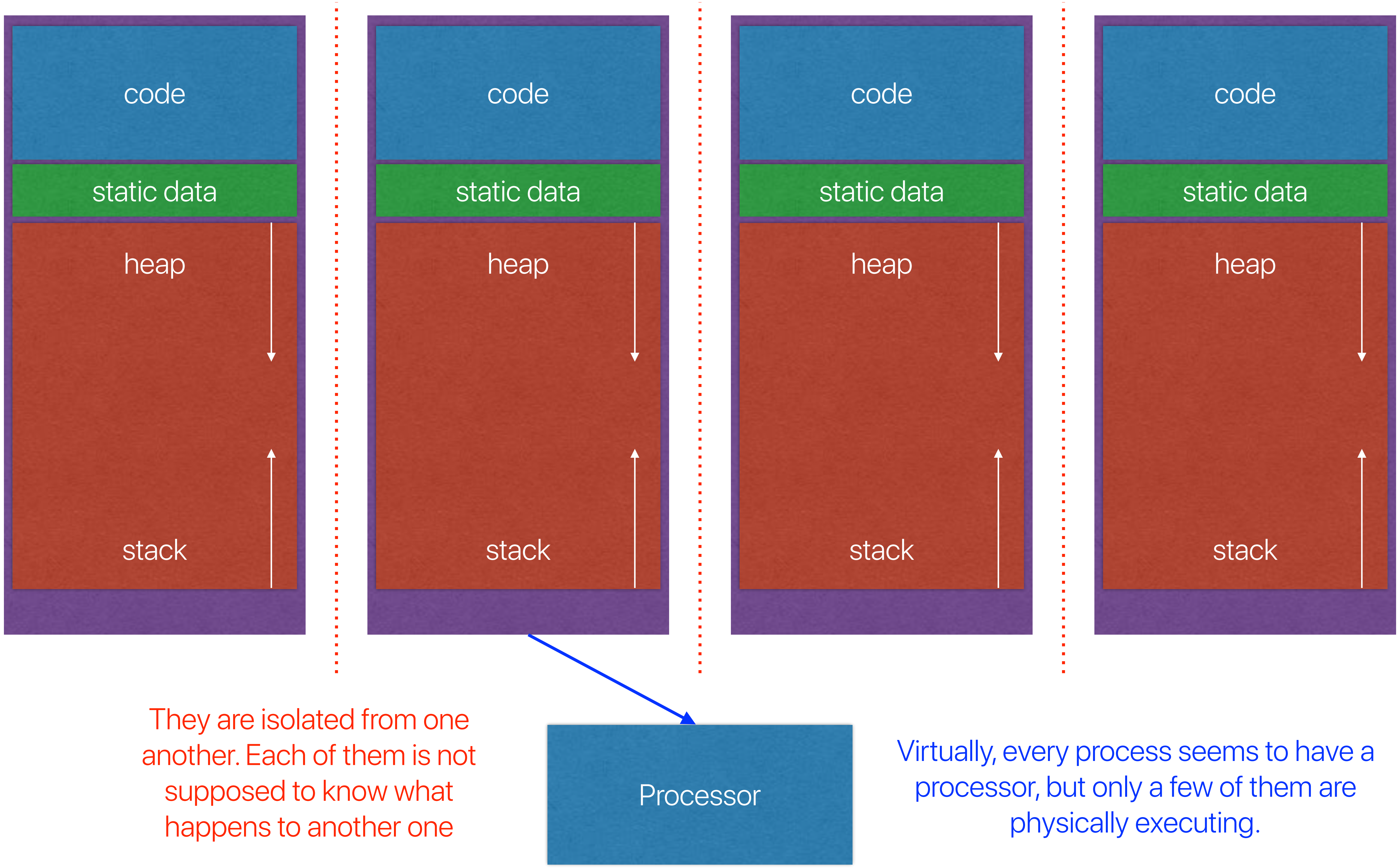
Recap: impact of UNIX

- Clean abstraction — everything as a file
- File system — will discuss in detail after midterm
- Portable OS
    - Written in high-level C programming language
    - The unshakable position of C programming language
- We are still using it!



Perhaps paradoxically, the success of UNIX is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This essentially personal effort was sufficiently successful to gain the interest of the remaining author and others, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, UNIX had proved useful enough to persuade management to invest in the PDP-11/45. Our goals throughout the effort, when articulated at all, have always concerned themselves with building a comfortable relationship with the machine and with exploring ideas and inventions in operating systems. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Recap: Each process has a separate virtual memory space

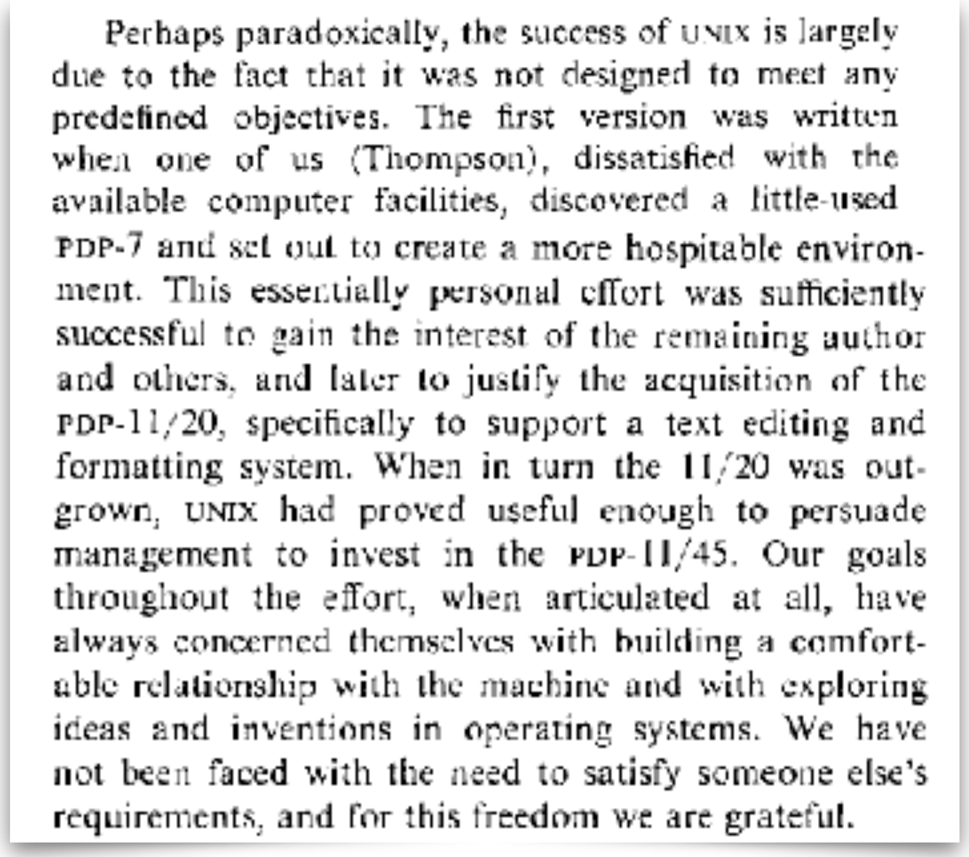


They are isolated from one another. Each of them is not supposed to know what happens to another one

Virtually, every process seems to have a processor, but only a few of them are physically executing.

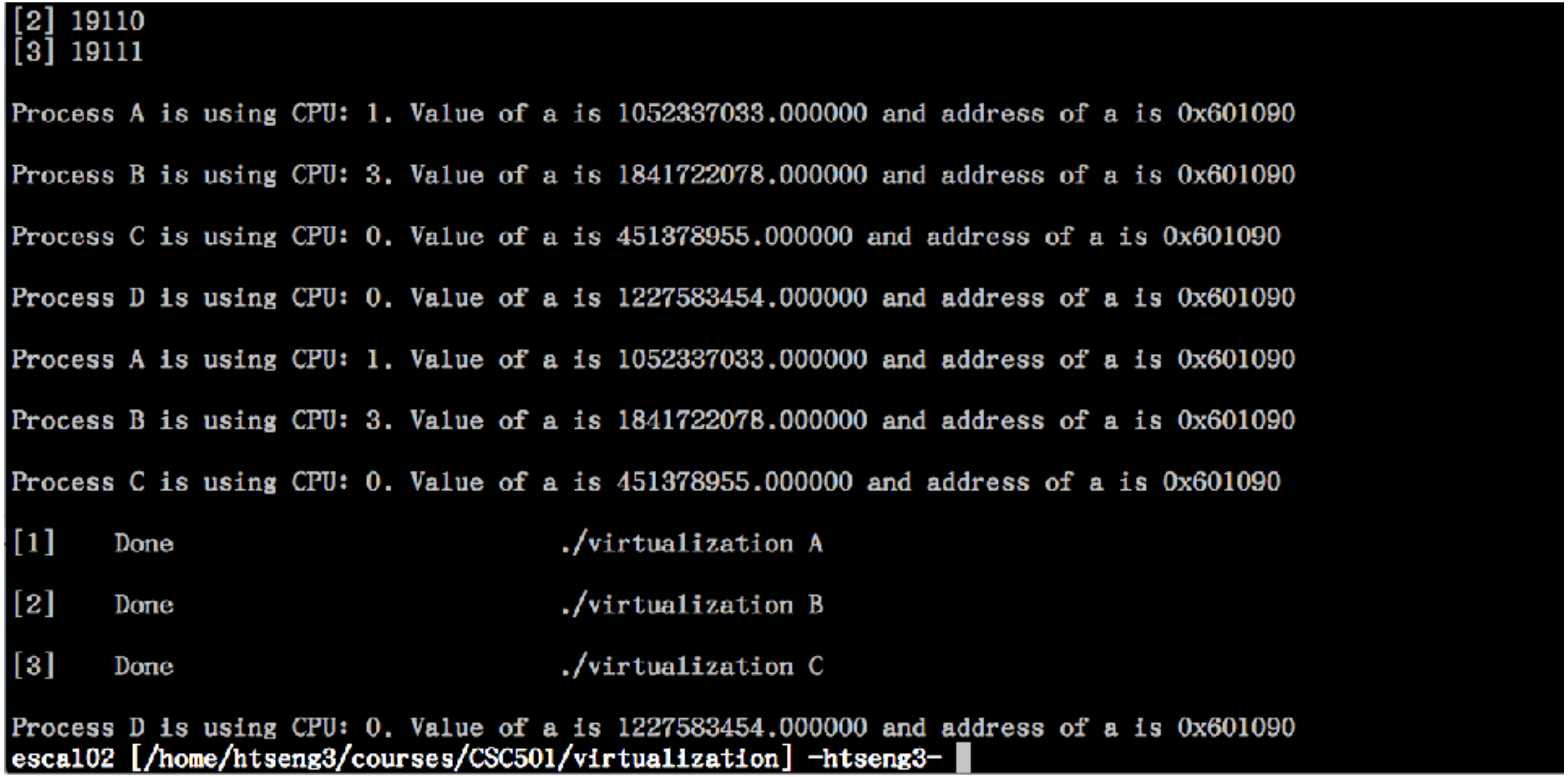
Recap: impact of UNIX

- Clean abstraction — everything as a file
- File system — will discuss in detail after midterm
- Portable OS
  - Written in high-level C programming language
  - The unshakable position of C programming language
- We are still using it!



Perhaps paradoxically, the success of UNIX is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This essentially personal effort was sufficiently successful to gain the interest of the remaining author and others, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, UNIX had proved useful enough to persuade management to invest in the PDP-11/45. Our goals throughout the effort, when articulated at all, have always concerned themselves with building a comfortable relationship with the machine and with exploring ideas and inventions in operating systems. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Recap: Review the first demo



```
[2] 19110
[3] 19111

Process A is using CPU: 1. Value of a is 1052337033.000000 and address of a is 0x601090

Process B is using CPU: 3. Value of a is 1841722078.000000 and address of a is 0x601090

Process C is using CPU: 0. Value of a is 451378955.000000 and address of a is 0x601090

Process D is using CPU: 0. Value of a is 1227583454.000000 and address of a is 0x601090

Process A is using CPU: 1. Value of a is 1052337033.000000 and address of a is 0x601090

Process B is using CPU: 3. Value of a is 1841722078.000000 and address of a is 0x601090

Process C is using CPU: 0. Value of a is 451378955.000000 and address of a is 0x601090

[1]    Done                          ./virtualization A

[2]    Done                          ./virtualization B

[3]    Done                          ./virtualization C

Process D is using CPU: 0. Value of a is 1227583454.000000 and address of a is 0x601090
escal02 [/home/htseng3/courses/CSC501/virtualization] -htseng3-
```

Recap: Protection mechanisms

- UNIX
    - Protection is associated with each file — described in the metadata of a file
    - Each file contains three (only two in the original paper) types of users
    - Each type of users can have read, write, execute permissions
    - setuid to promote right amplifications

Current scoreboard

Red	Blue
3	4

Outline

- The process interface in UNIX
- Mach: A New Kernel Foundation For UNIX Development

The interface of managing processes

The basic process API of UNIX

- `fork`
- `wait`
- `exec`
- `exit`

`fork()`

- `pid_t fork();`
- `fork` used to create processes (UNIX)
- What does `fork()` do?
  - Creates a **new** address space (for child)
  - **Copies** parent’s address space to child’s
  - Points kernel resources to the parent’s resources (e.g. open files)
  - Inserts child process into ready queue
- `fork()` returns twice
  - Returns the child’s PID to the parent
  - Returns “0” to the child

Poll close in 1:30

What will happen?

- What happens if we execute the following code?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf ("My pid is %d\n", getpid());
        }
        printf ("Child pid is %d\n", pid);
         return 0;
}
```

Assume
the parent's PID is 2;
child's PID is 7.

	# of times "my pid" is printed	my pid values printed	# of times "child pid" is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

Poll close in 1:30

What will happen?

- What happens if we execute the following code?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf (”My pid is %d\n", getpid());
        }
        printf (”Child pid is %d\n", pid);
        return 0;
}
```

Assume the parent’s PID is 2; child’s PID is 7.

	# of times “my pid” is printed	my pid values printed	# of times “child pid” is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

What will happen?

- What happens if we execute the following code?

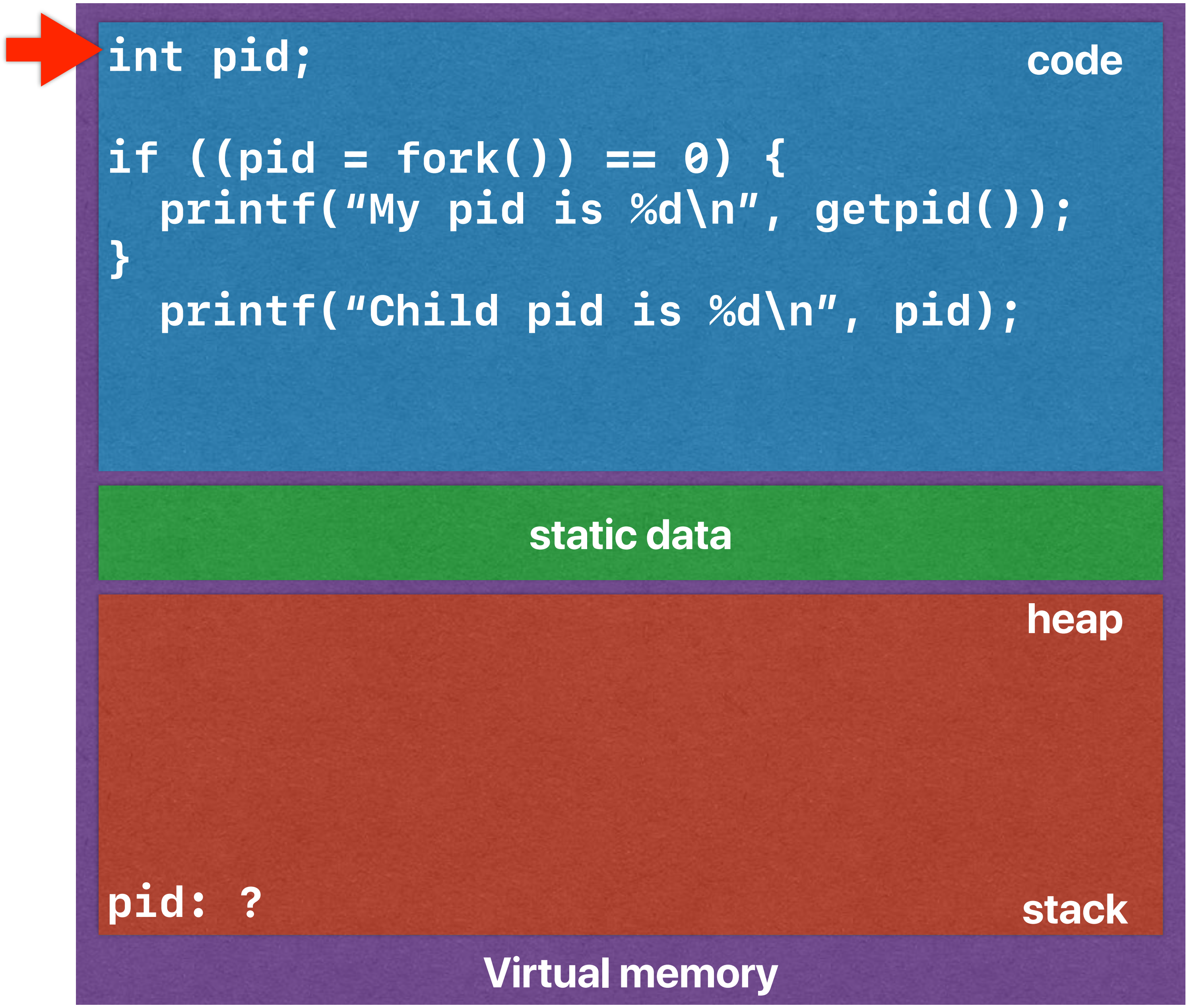
```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf (”My pid is %d\n", getpid());
        }
        printf (”Child pid is %d\n", pid);
        return 0;
}
```

Assume
the parent’s PID is 2;
child’s PID is 7.

	# of times “my pid” is printed	my pid values printed	# of times “child pid” is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

Assume
the parent’s PID is 2;
child’s PID is 7.

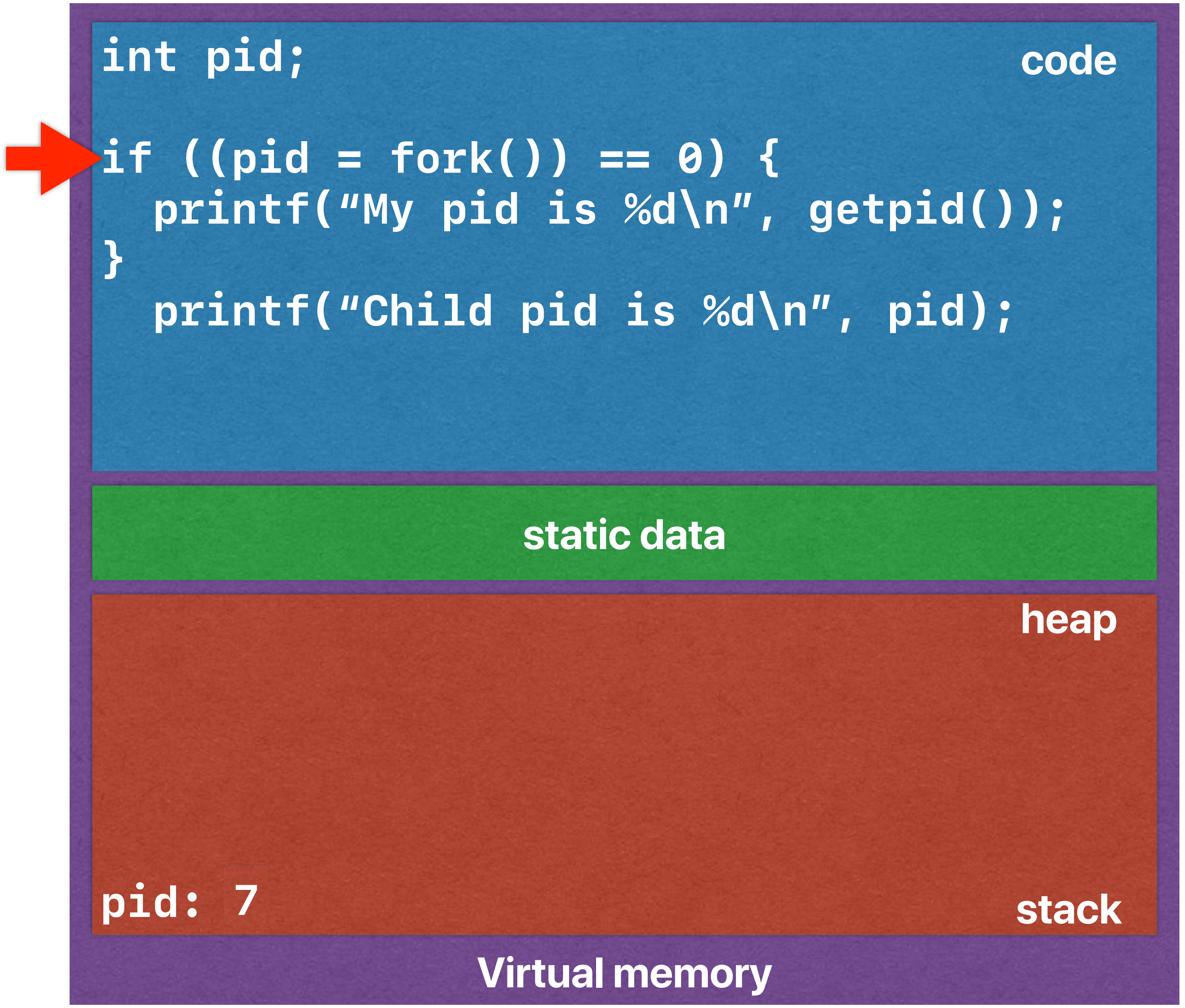
fork()



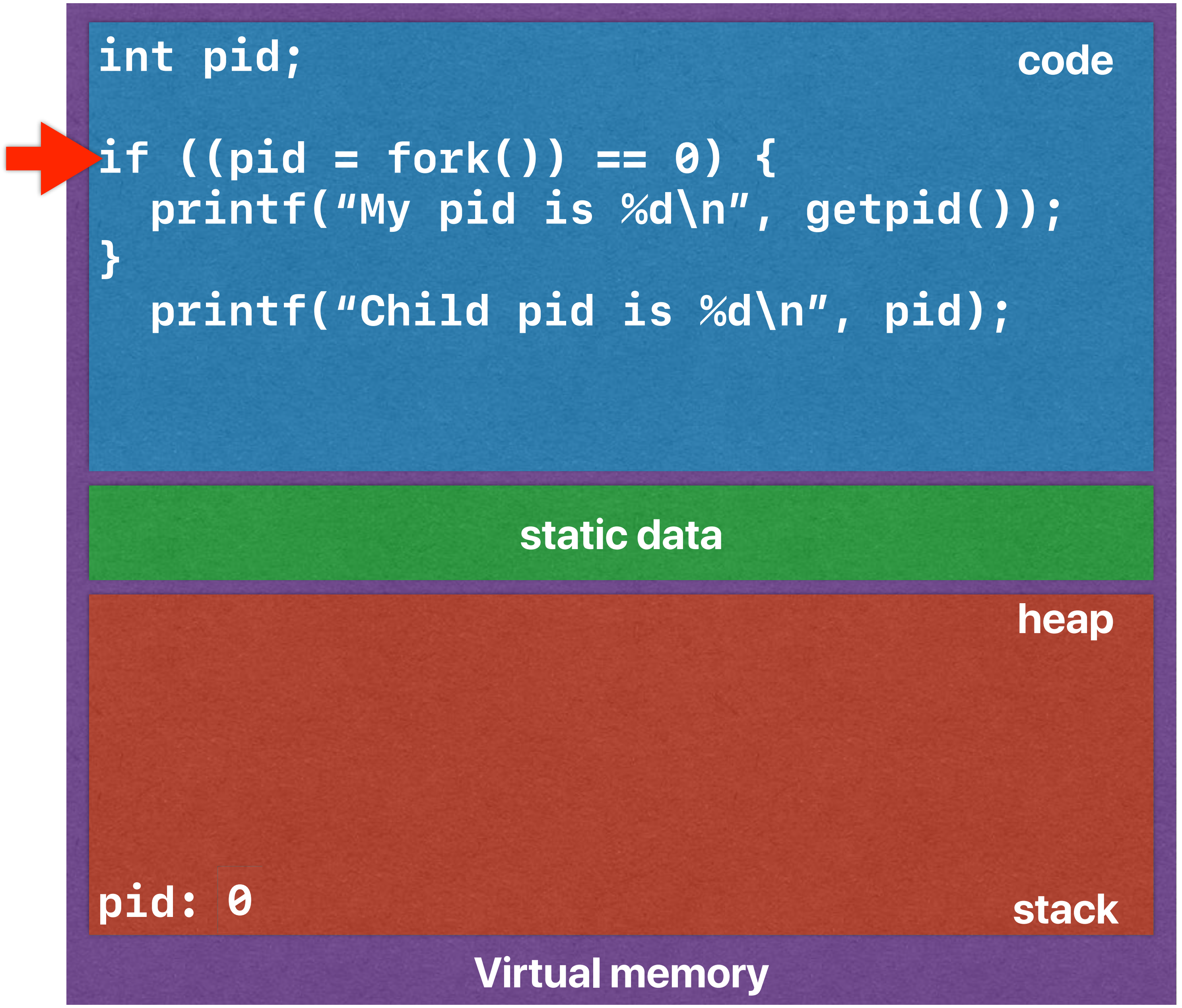
```
int pid;

if ((pid = fork()) == 0) {
  printf(“My pid is %d\n”, getpid());
}
  printf(“Child pid is %d\n”, pid);
```

Assume
the parent's PID is 2;
child's PID is 7.

fork()



```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
}
  printf("Child pid is %d\n", pid);
```

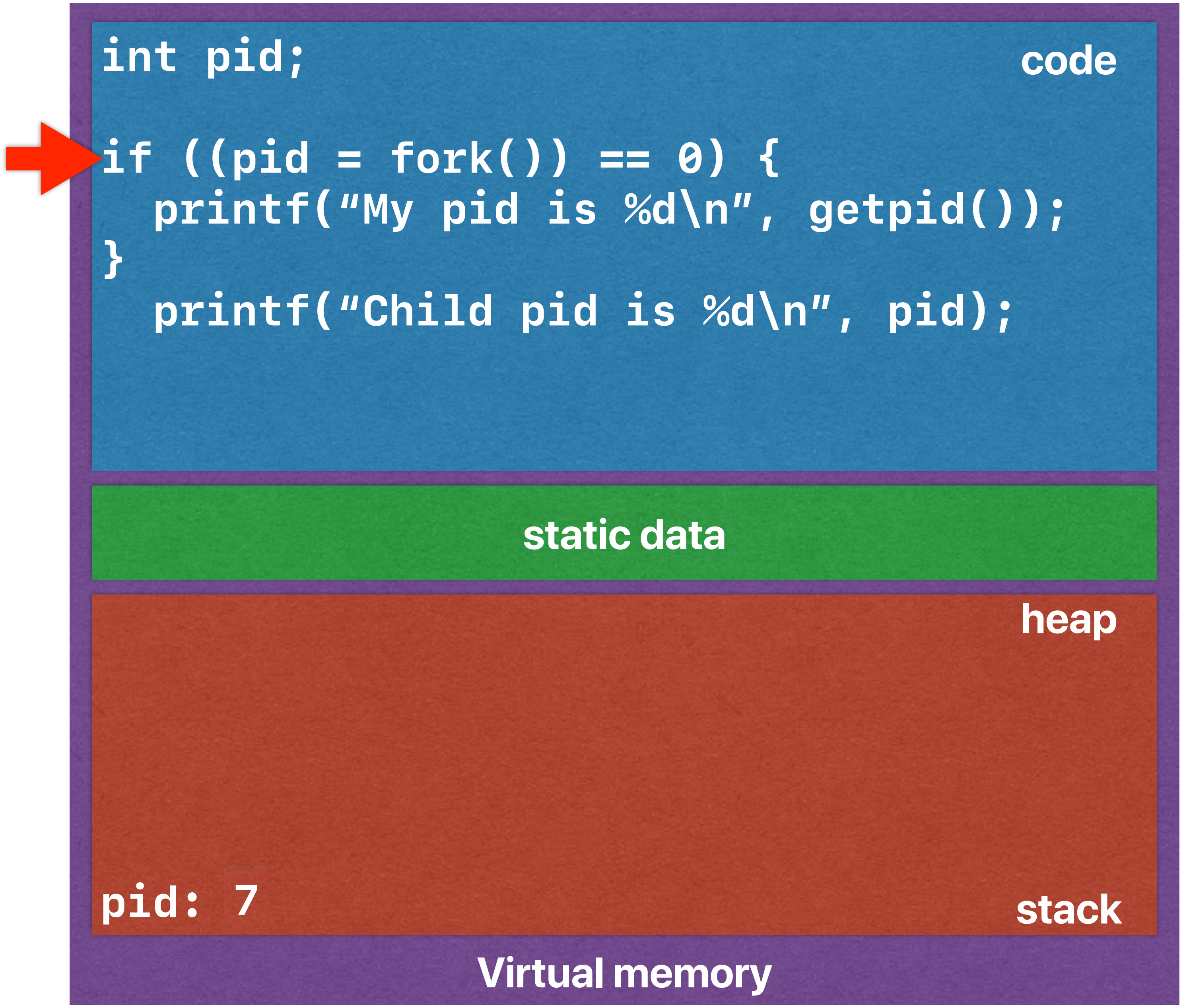


```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
}
  printf("Child pid is %d\n", pid);
```

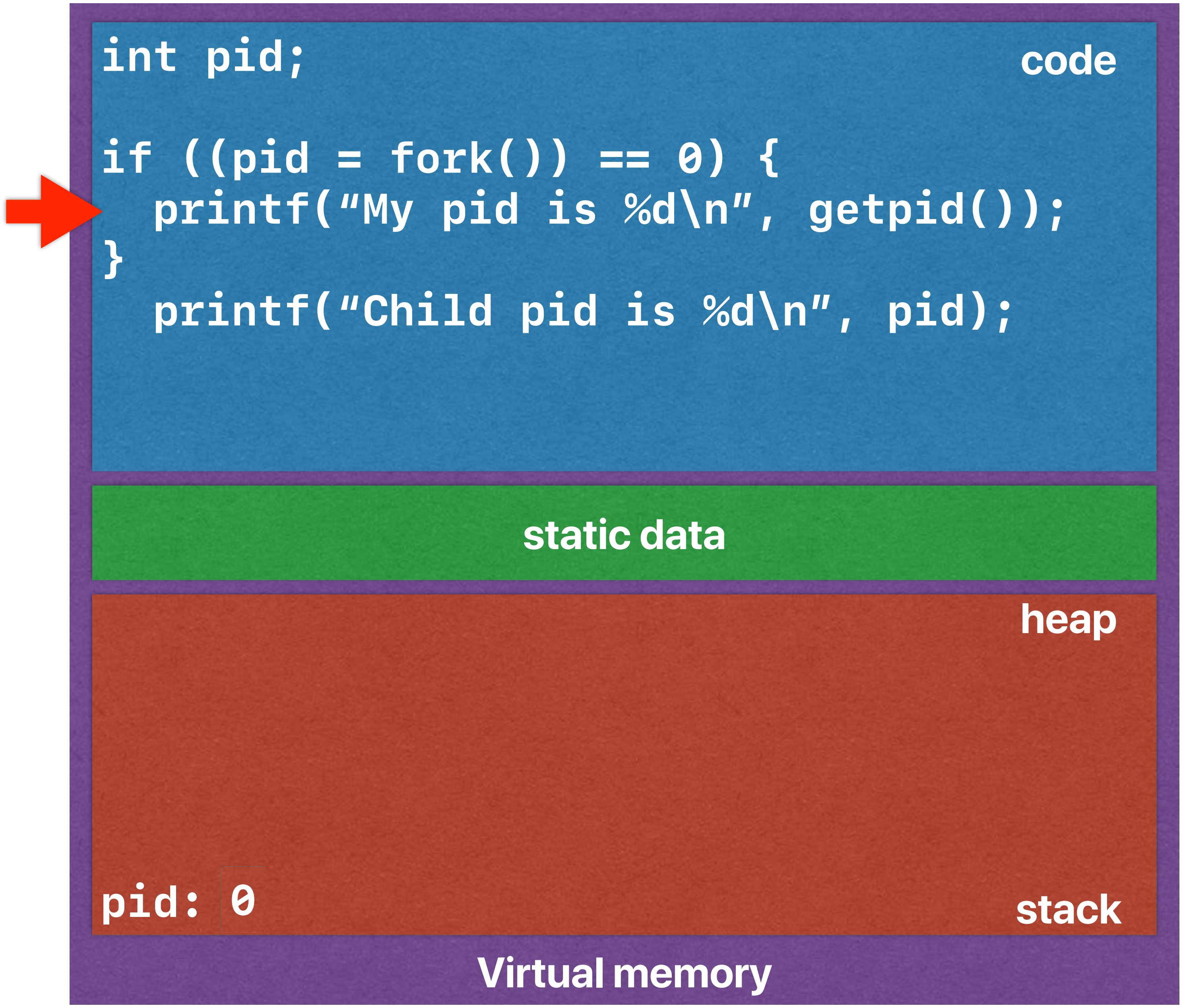
Assume
the parent’s PID is 2;
child’s PID is 7.

fork()

Output:
My pid is 7



```
int pid;

if ((pid = fork()) == 0) {
  printf(“My pid is %d\n”, getpid());
}
  printf(“Child pid is %d\n”, pid);
```



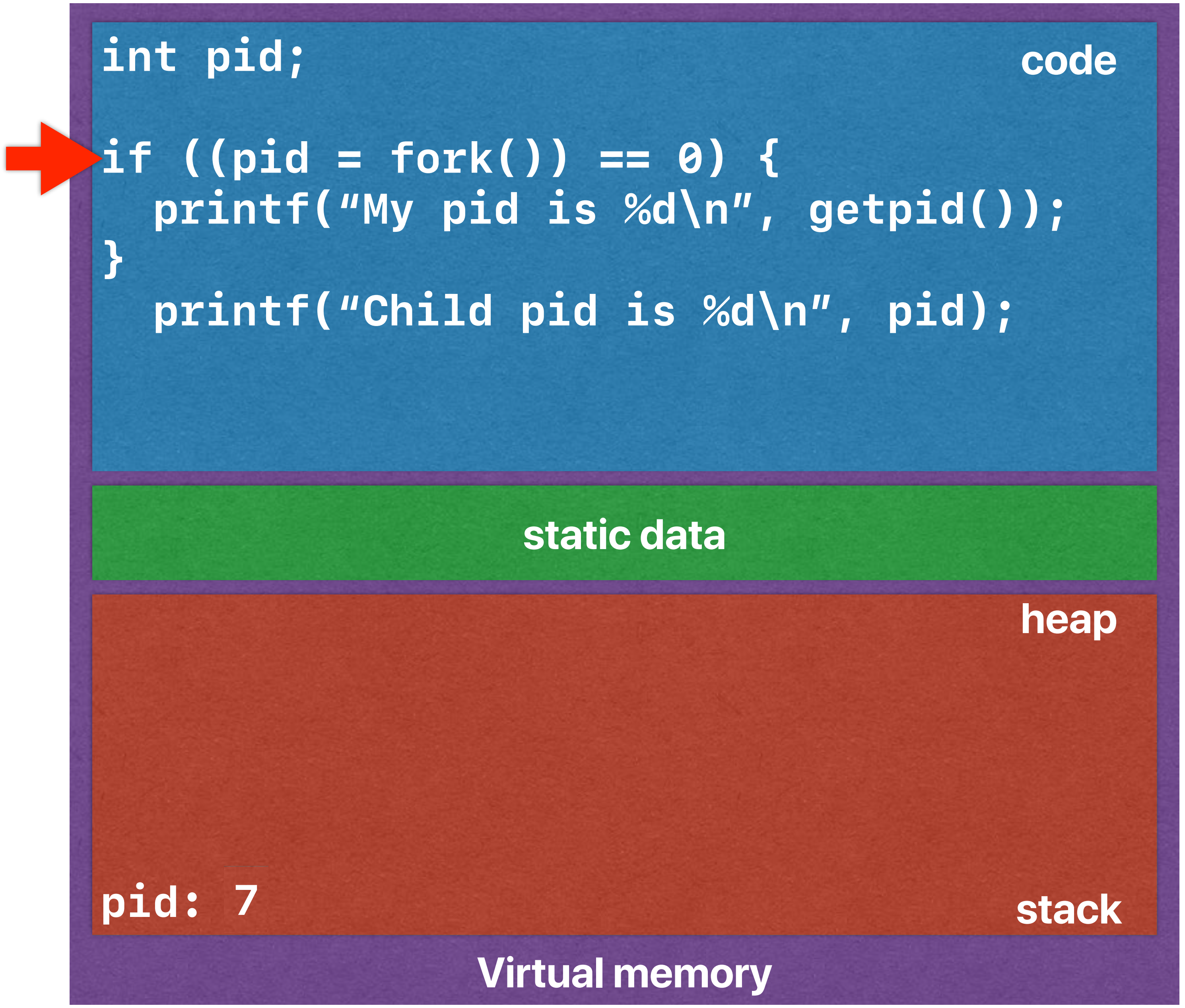
```
int pid;

if ((pid = fork()) == 0) {
  printf(“My pid is %d\n”, getpid());
}
  printf(“Child pid is %d\n”, pid);
```

Assume
the parent's PID is 2;
child's PID is 7.

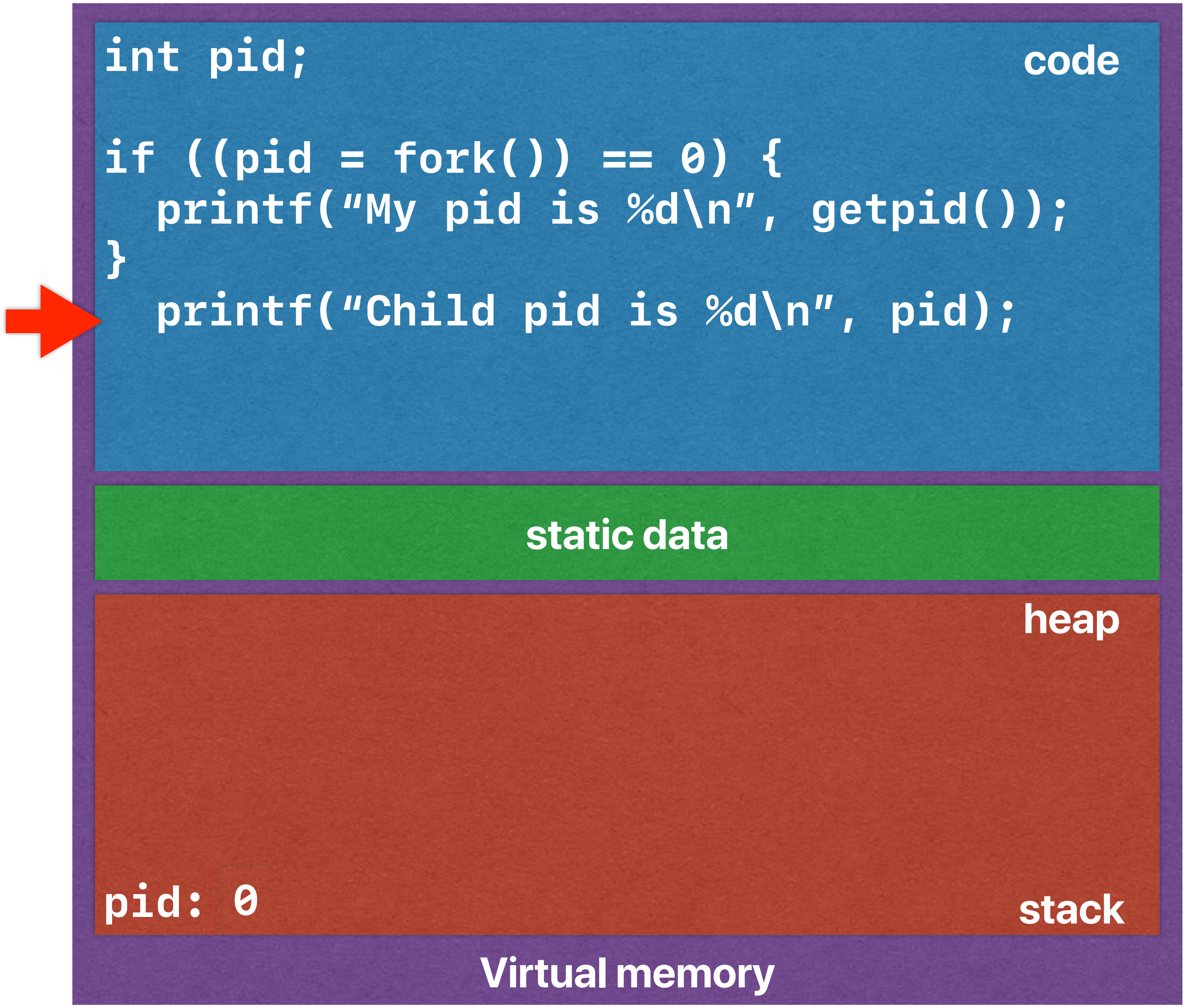
fork()

Output:

```
My pid is 7
Child pid is 0
```



```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
}
  printf("Child pid is %d\n", pid);
```



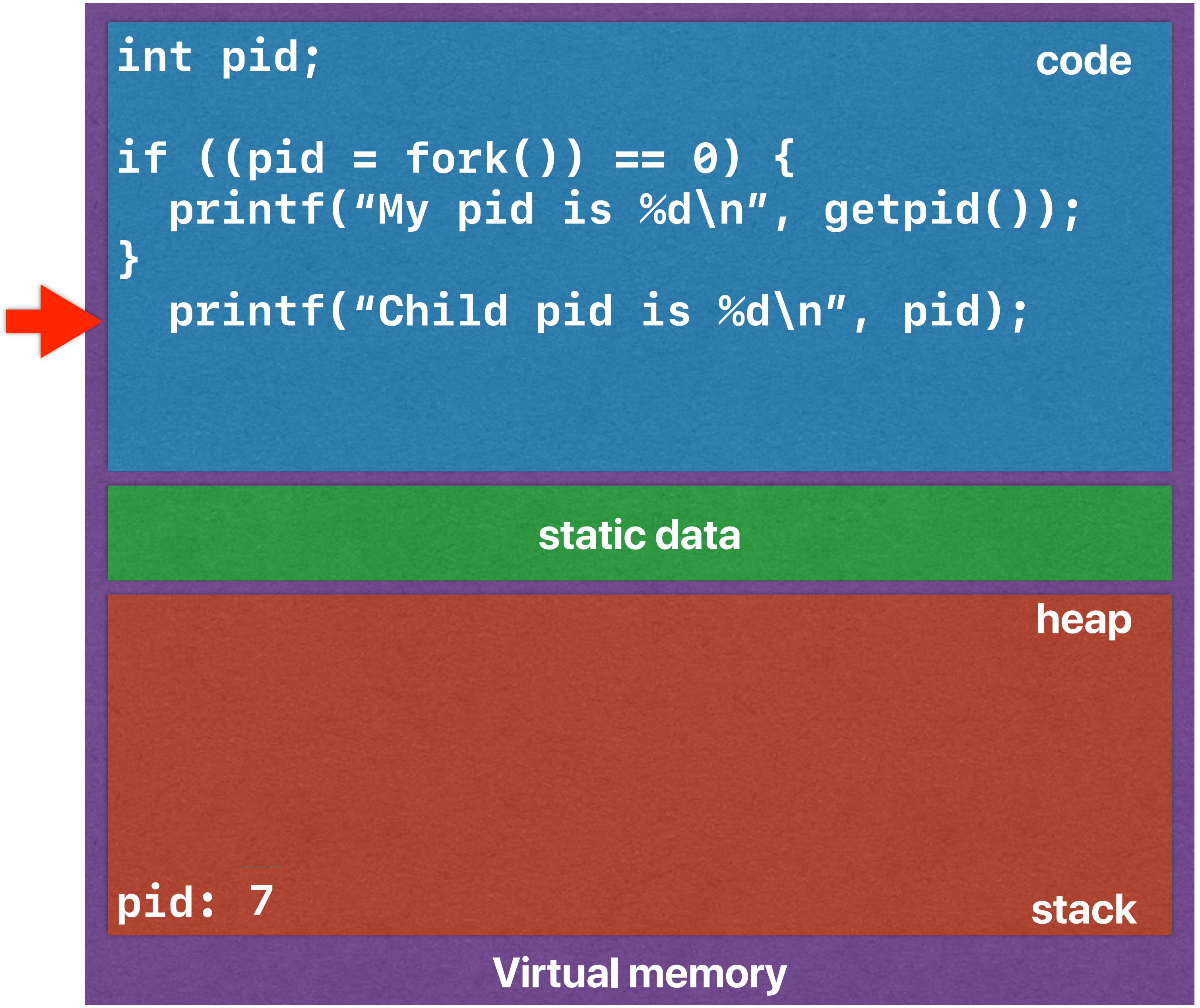
```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
}
  printf("Child pid is %d\n", pid);
```

Assume
the parent’s PID is 2;
child’s PID is 7.

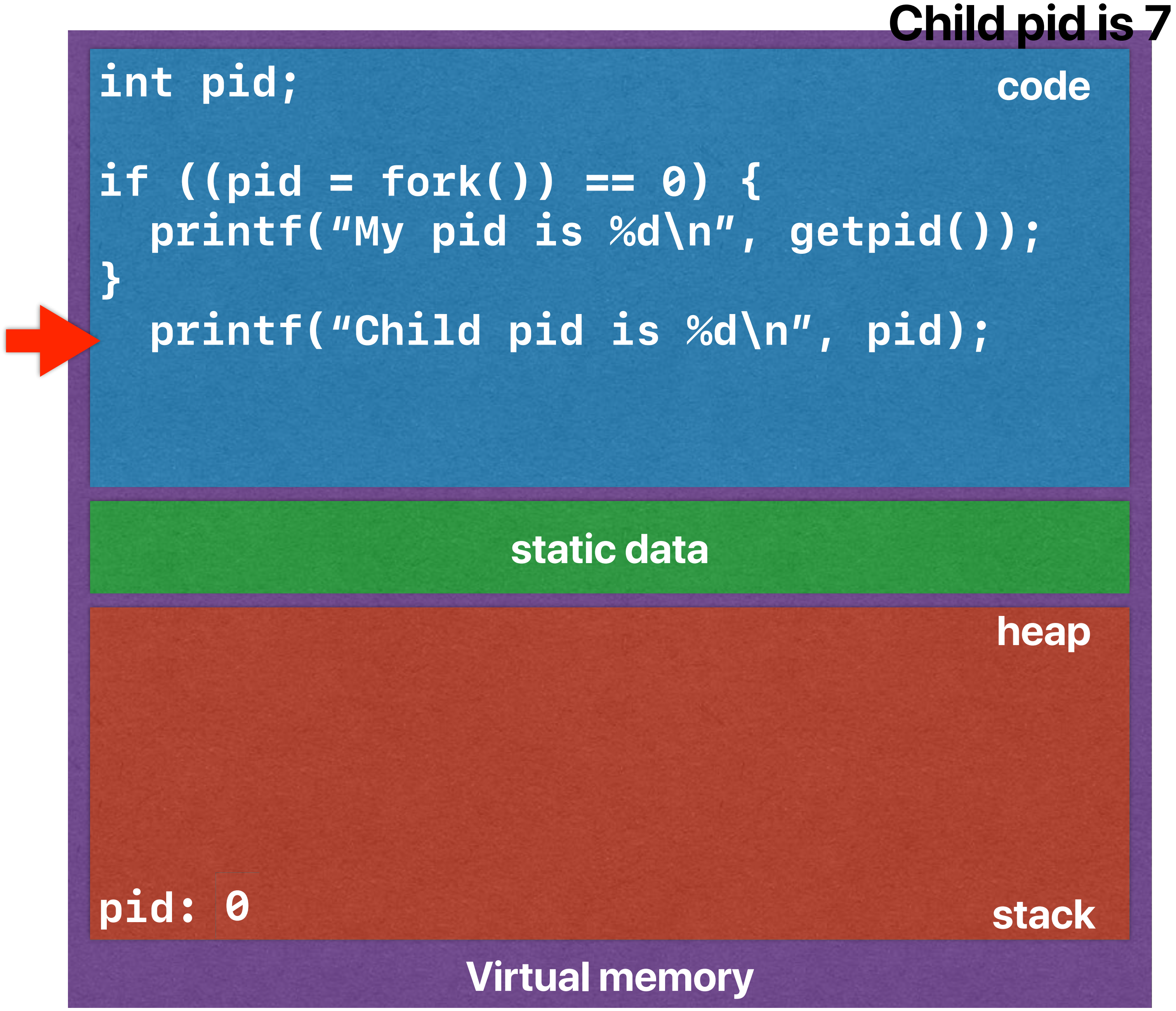
fork()

Output:

```
My pid is 7
Child pid is 0
Child pid is 7
```



```
int pid;

if ((pid = fork()) == 0) {
  printf(“My pid is %d\n”, getpid());
}
  printf(“Child pid is %d\n”, pid);
```



```
int pid;

if ((pid = fork()) == 0) {
  printf(“My pid is %d\n”, getpid());
}
  printf(“Child pid is %d\n”, pid);
```

What will happen?

- What happens if we execute the following code?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf (”My pid is %d\n", getpid());
        }
        printf (”Child pid is %d\n", pid);
        return 0;
}
```

Assume
the parent’s PID is 2;
child’s PID is 7.

	# of times “my pid” is printed	my pid values printed	# of times “child pid” is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

`exit()`

- `void exit(int status)`
- `exit` frees resources and terminates the process
  - Runs an functions registered with atexit
  - Flush and close all open files/streams
  - Releases allocated memory.
  - Remove process from kernel data structures (e.g. queues)
- `status` is passed to parent process
  - By convention, 0 indicates "normal exit"

Poll close in 1:30

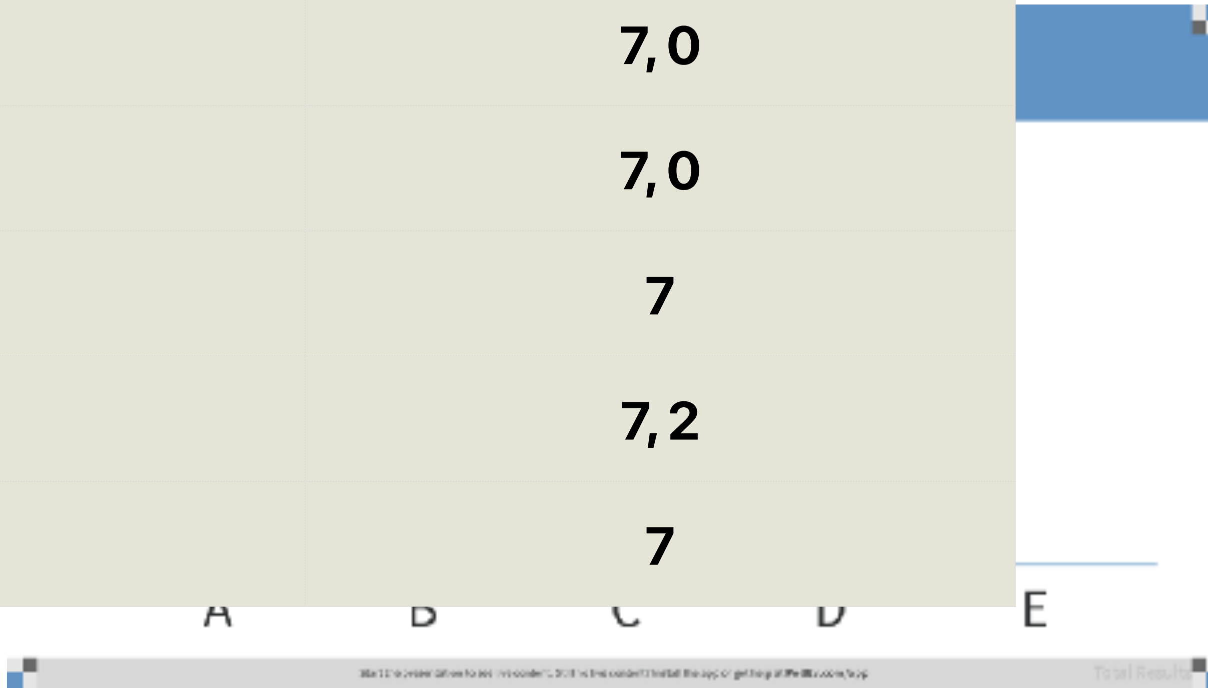
If we add an exit ...

- What happens if we add an exit?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf ("My pid is %d\n", getpid());
            exit(0);
        }
        printf ("Child pid is %d\n", pid);
        return 0;
}
```

Assume
the parent's PID is 2;
child's PID is 7.

	# of times "my pid" is printed	my pid values printed	# of times "child pid" is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7



Poll close in 1:30

If we add an exit ...

- What happens if we add an exit?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf (”My pid is %d\n", getpid());
            exit(0);
        }
        printf (”Child pid is %d\n", pid);
        return 0;
}
```

Assume
the parent’s PID is 2;
child’s PID is 7.

	# of times “my pid” is printed	my pid values printed	# of times “child pid” is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

If we add an exit …

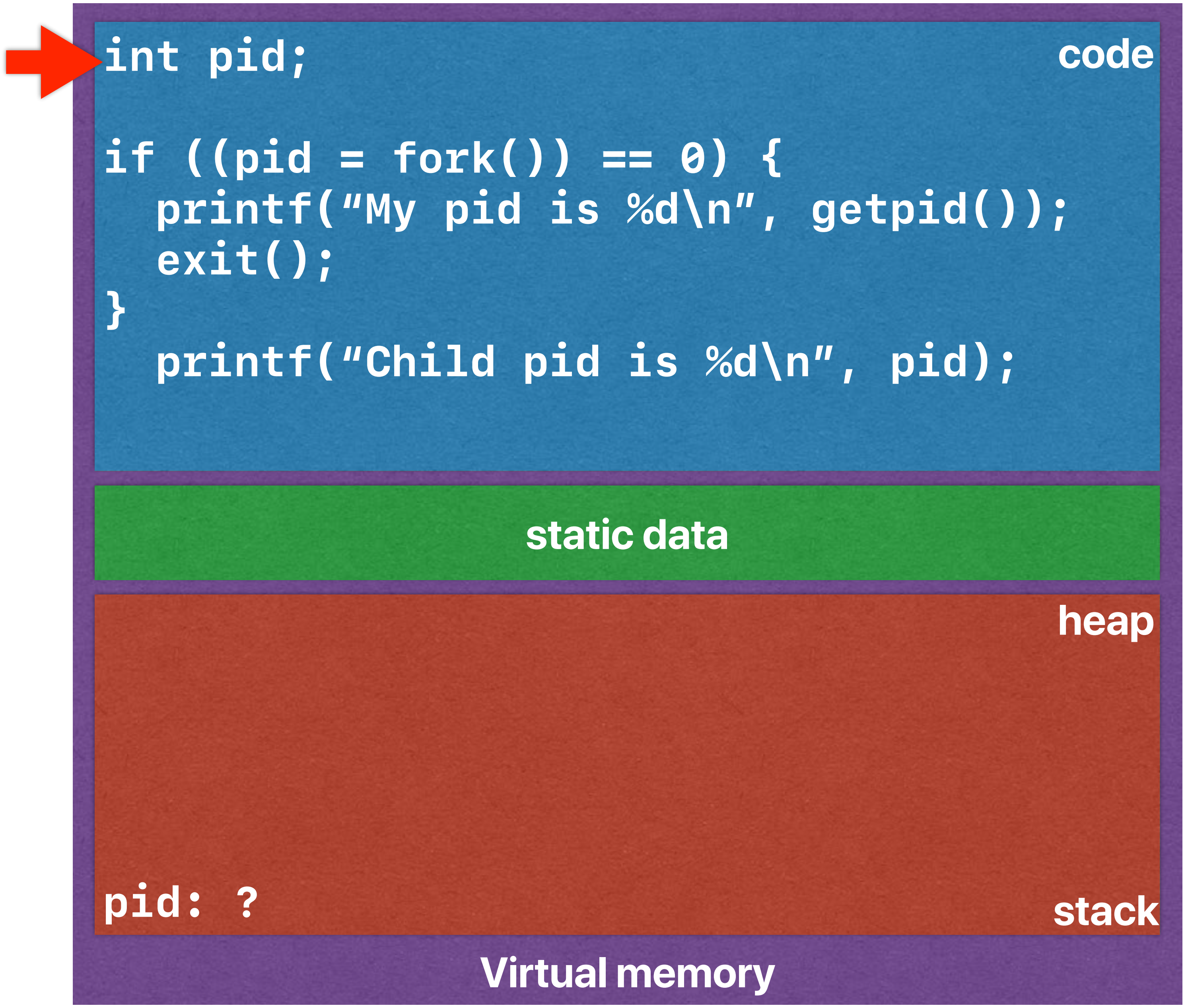
- What happens if we add an exit?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf (″My pid is %d\n", getpid());
            exit(0);
        }
        printf (″Child pid is %d\n", pid);
        return 0;
}
```

Assume
the parent's PID is 2;
child's PID is 7.

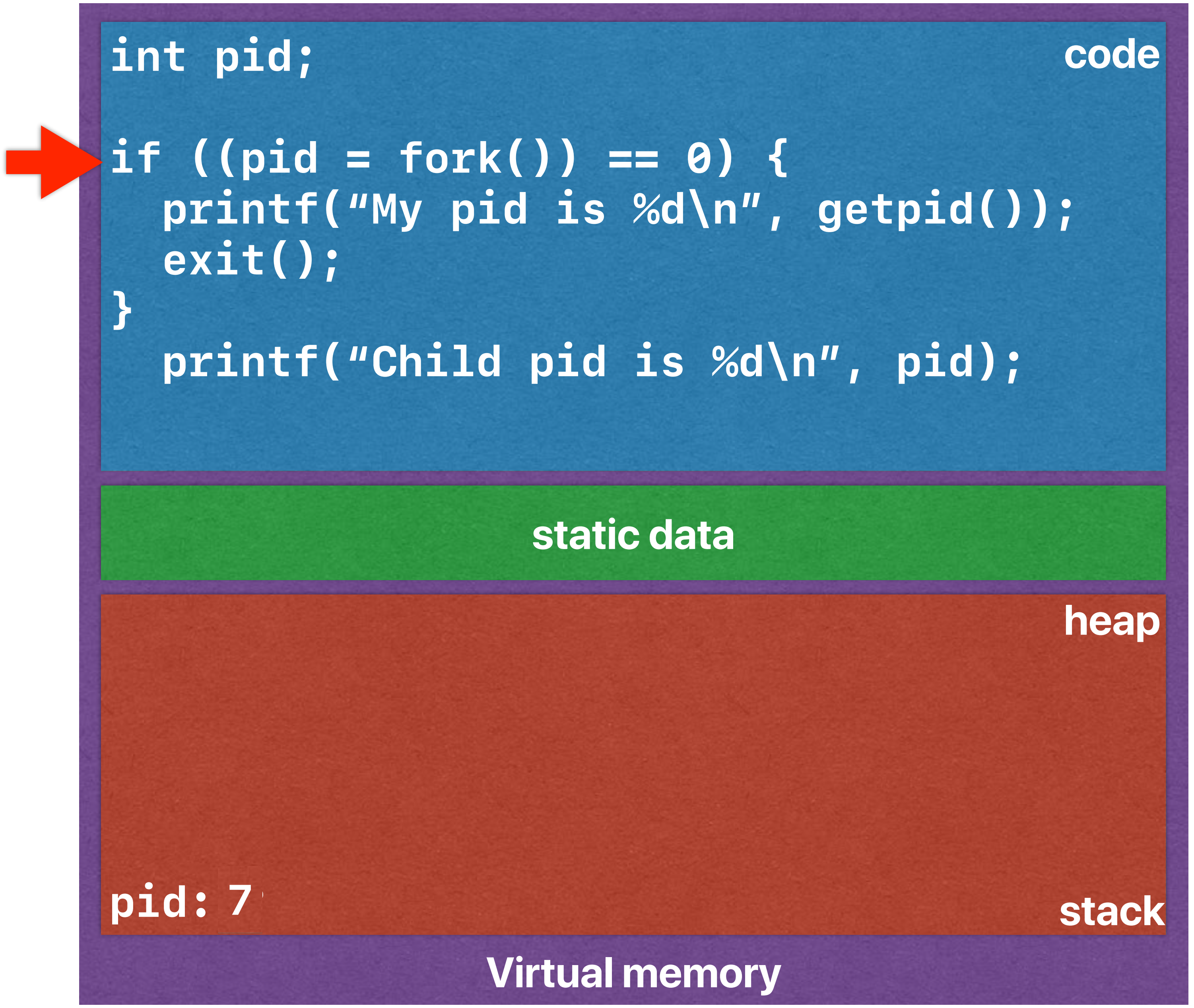
	# of times "my pid" is printed	my pid values printed	# of times "child pid" is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

fork() and exit()

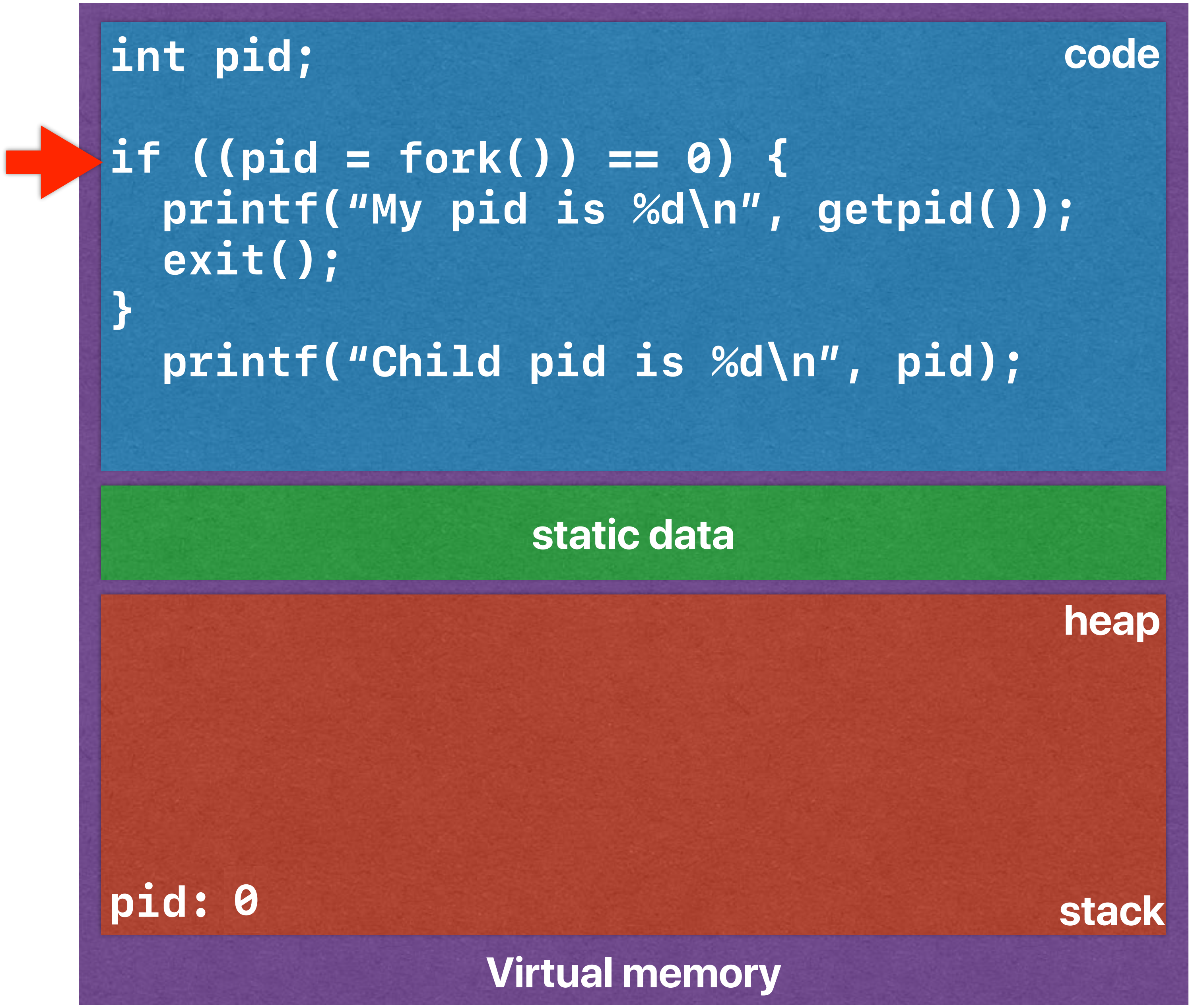


```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```

fork() and exit()



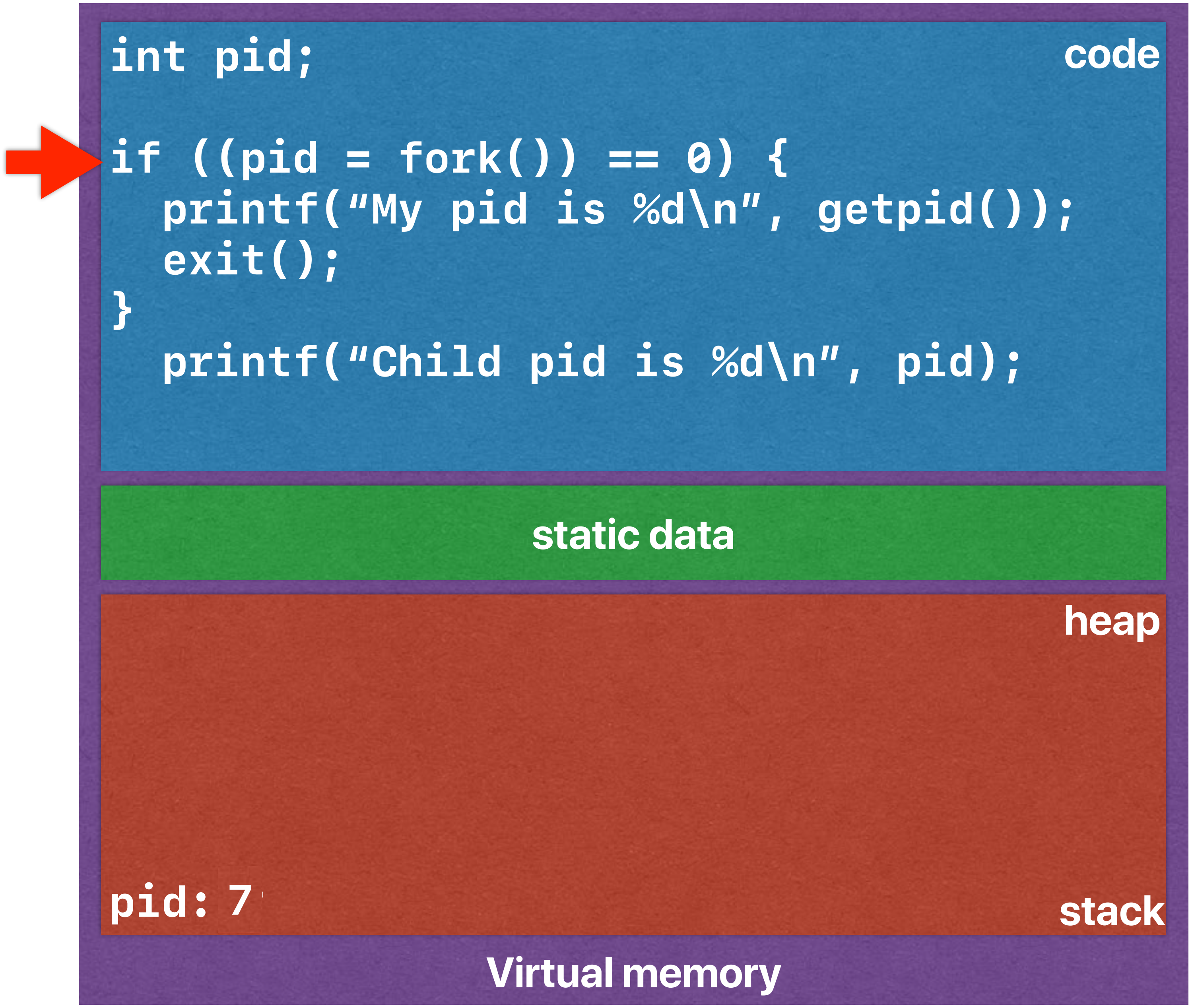
```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```



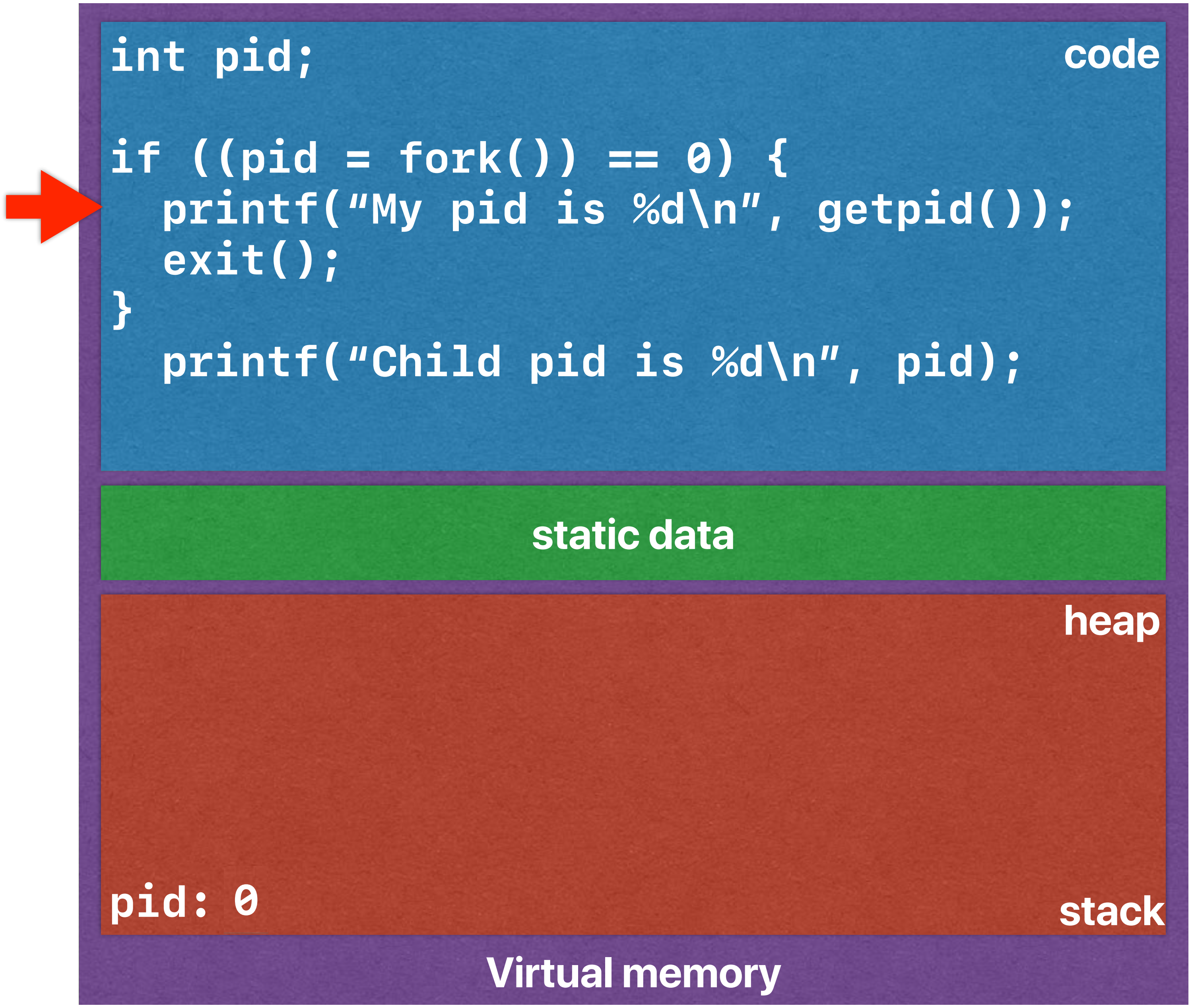
```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```

fork() and exit()

Output:
My pid is 7



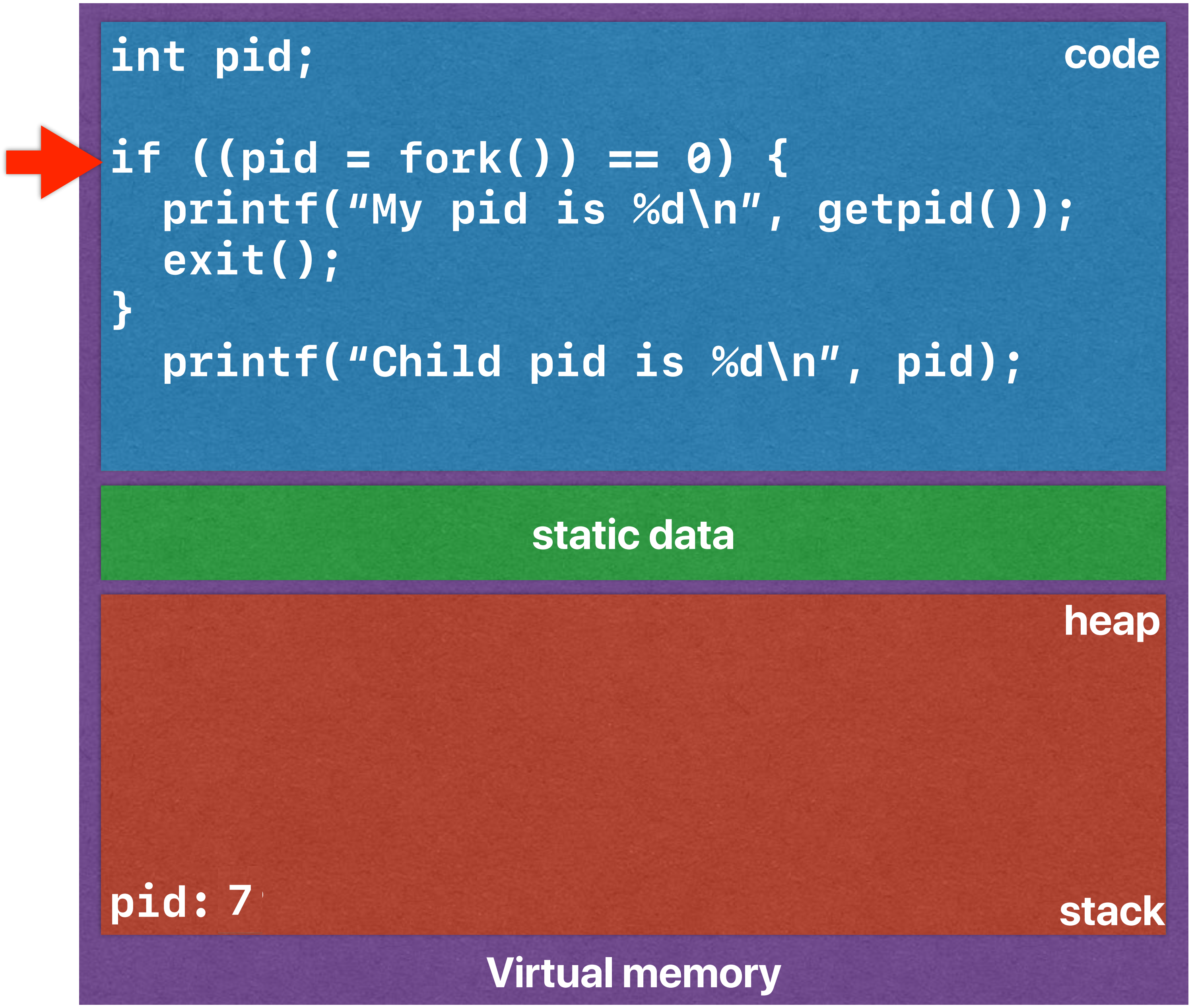
```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```



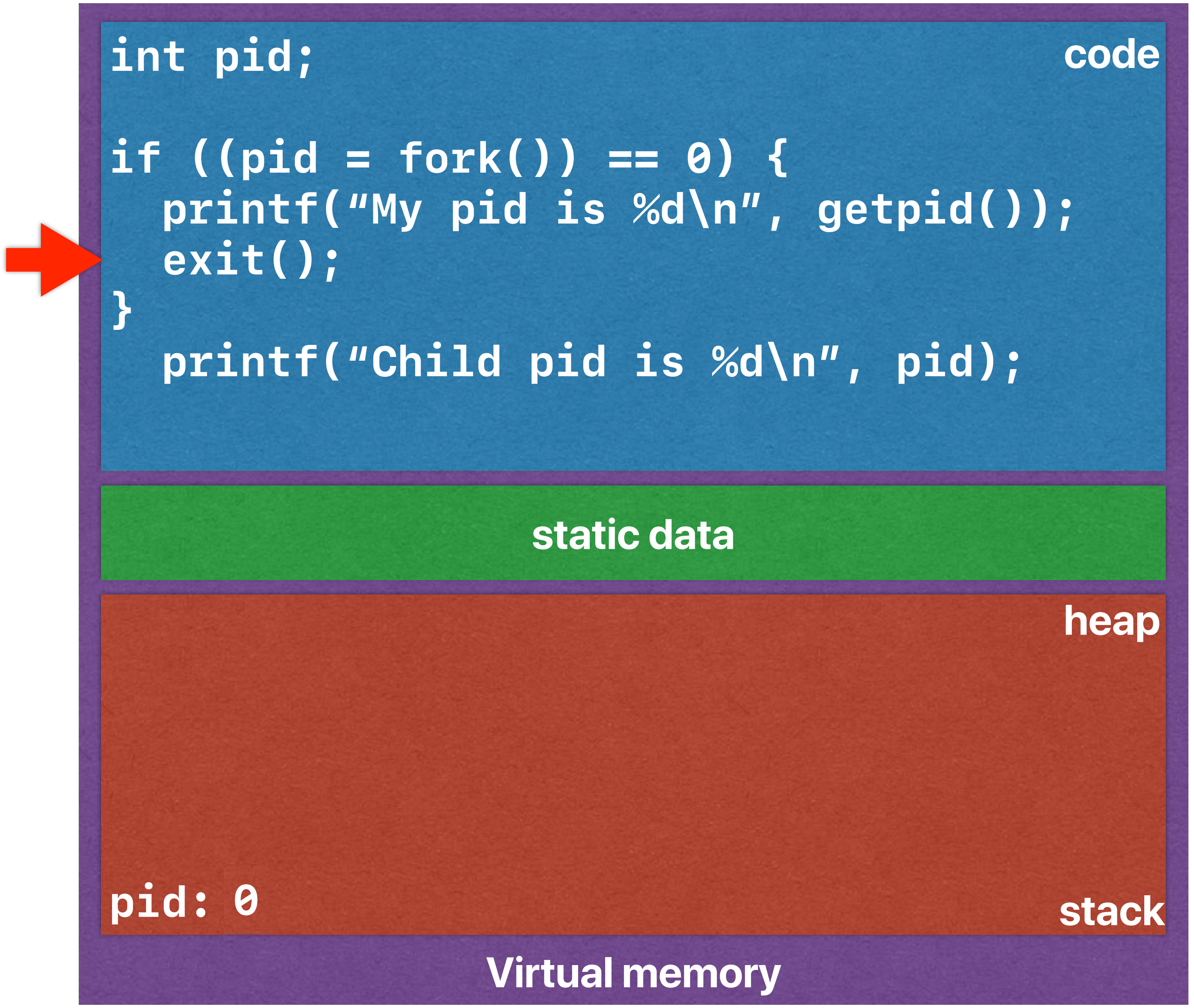
```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```

fork() and exit()

Output:
My pid is 7



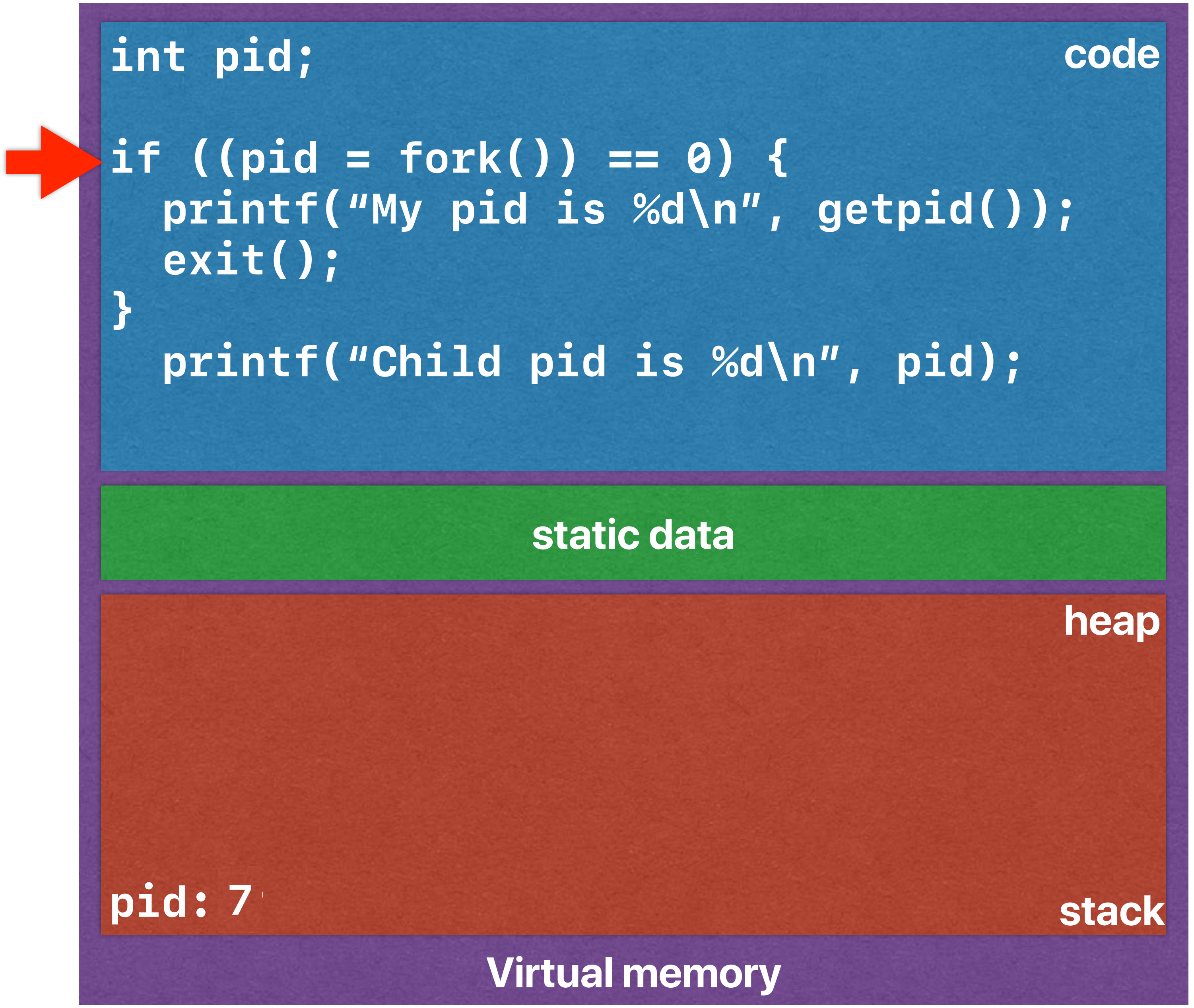
```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```



```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```

fork() and exit()

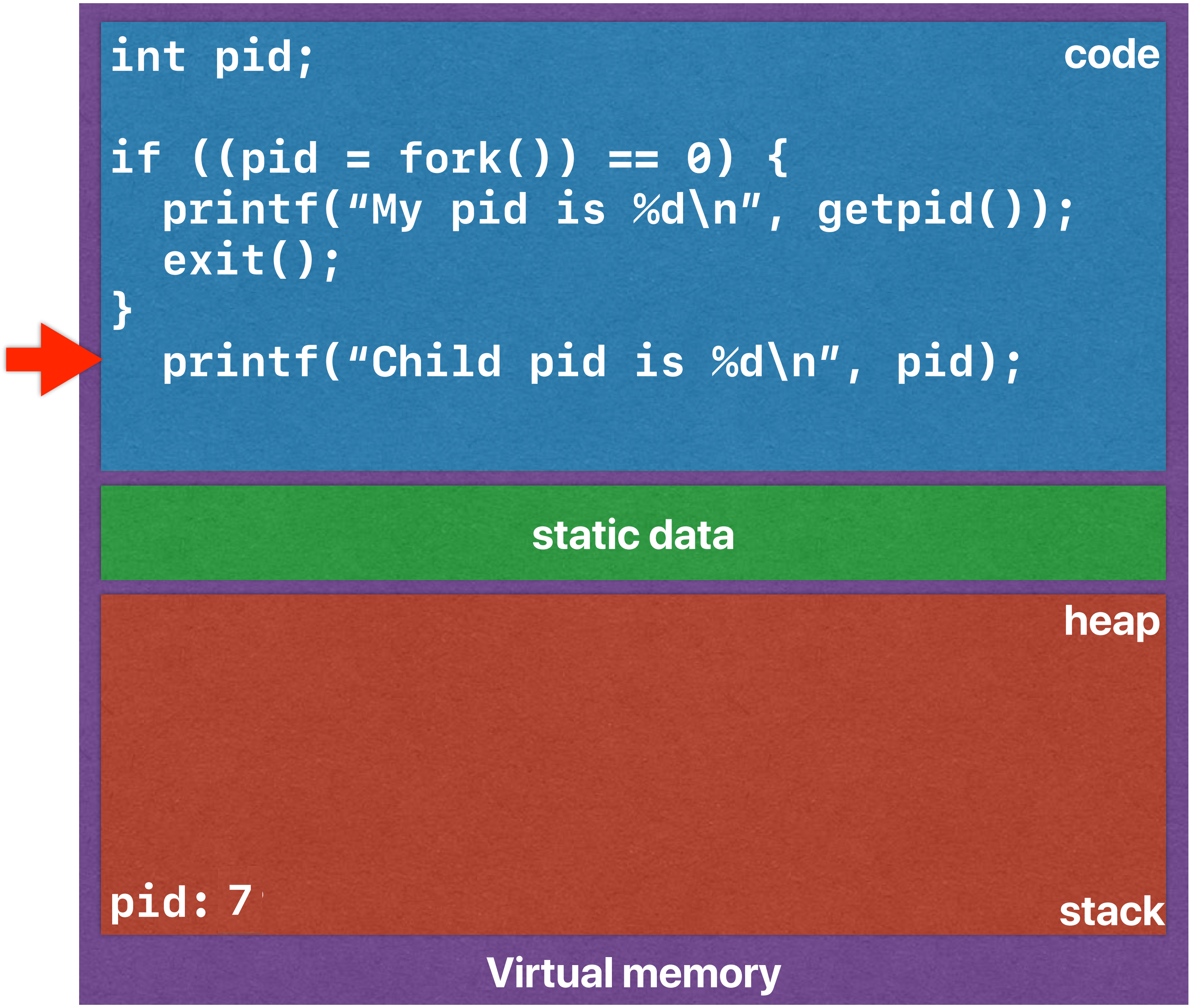
Output:
My pid is 7



```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```

fork() and exit()

Output:
My pid is 7
Child pid is 7



```
int pid;

if ((pid = fork()) == 0) {
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```

If we add an exit ...

- What happens if we add an exit?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf (”My pid is %d\n", getpid());
            exit(0);
        }
        printf (”Child pid is %d\n", pid);
        return 0;
}
```

Assume
the parent’s PID is 2;
child’s PID is 7.

	# of times “my pid” is printed	my pid values printed	# of times “child pid” is printed	child pid values printed
A	1	7	2	7,0
B	1	2	2	7,0
C	2	7,2	1	7
D	1	0	2	7,2
E	1	7	1	7

Poll close in 1:30

More forks

- Consider the following code

```
fork();
printf(“moo\n”);
fork();
printf(“oink\n”);
fork();
printf(“baa\n”);
```

How many animal noises will be printed?

A. 3

B. 6

C. 8

D. 14

E. 24

Poll close in 1:30

More forks

- Consider the following code

```
fork();
printf(“moo\n”);
fork();
printf(“oink\n”);
fork();
printf(“baa\n”);
```

How many animal noises will be printed?

A. 3
B. 6
C. 8
D. 14
E. 24

More forks

- Consider the following code

```
fork();
printf(“moo\n”);     2x
fork();
printf(“oink\n”);    4x
fork();
printf(“baa\n”);     8x
```

How many animal noises will be printed?

A. 3

B. 6

C. 8

[✓ D. 14]

E. 24

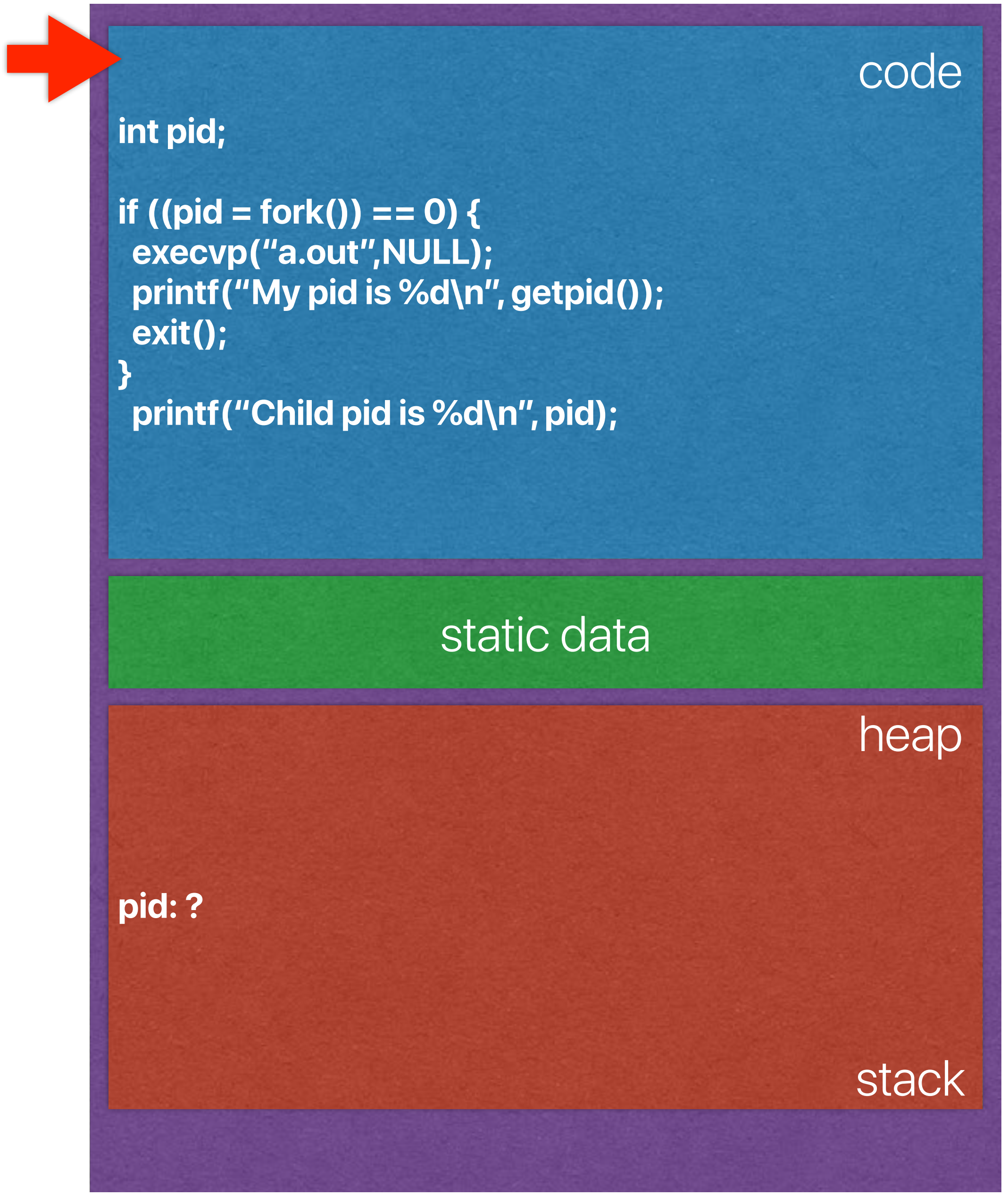
Starting a new program with `execvp()`

- `int execvp(char *prog, char *argv[])`
- `fork` does not start a new program, just duplicates the current program
- What `execvp` does:
  - Stops the current process
  - Overwrites process' address space for the new program
  - Initializes hardware context and args for the new program
  - Inserts the process into the ready queue
- `execvp` does not create a new process

Why separate `fork()` and `exec()`

- Windows only has `exec`
- Flexibility
- Allows redirection & pipe
  - The shell `forks` a new process whenever user invoke a program
  - After `fork`, the shell can setup any appropriate environment variable to before `exec`
  - The shell can easily redirect the output in shell: a.out > file

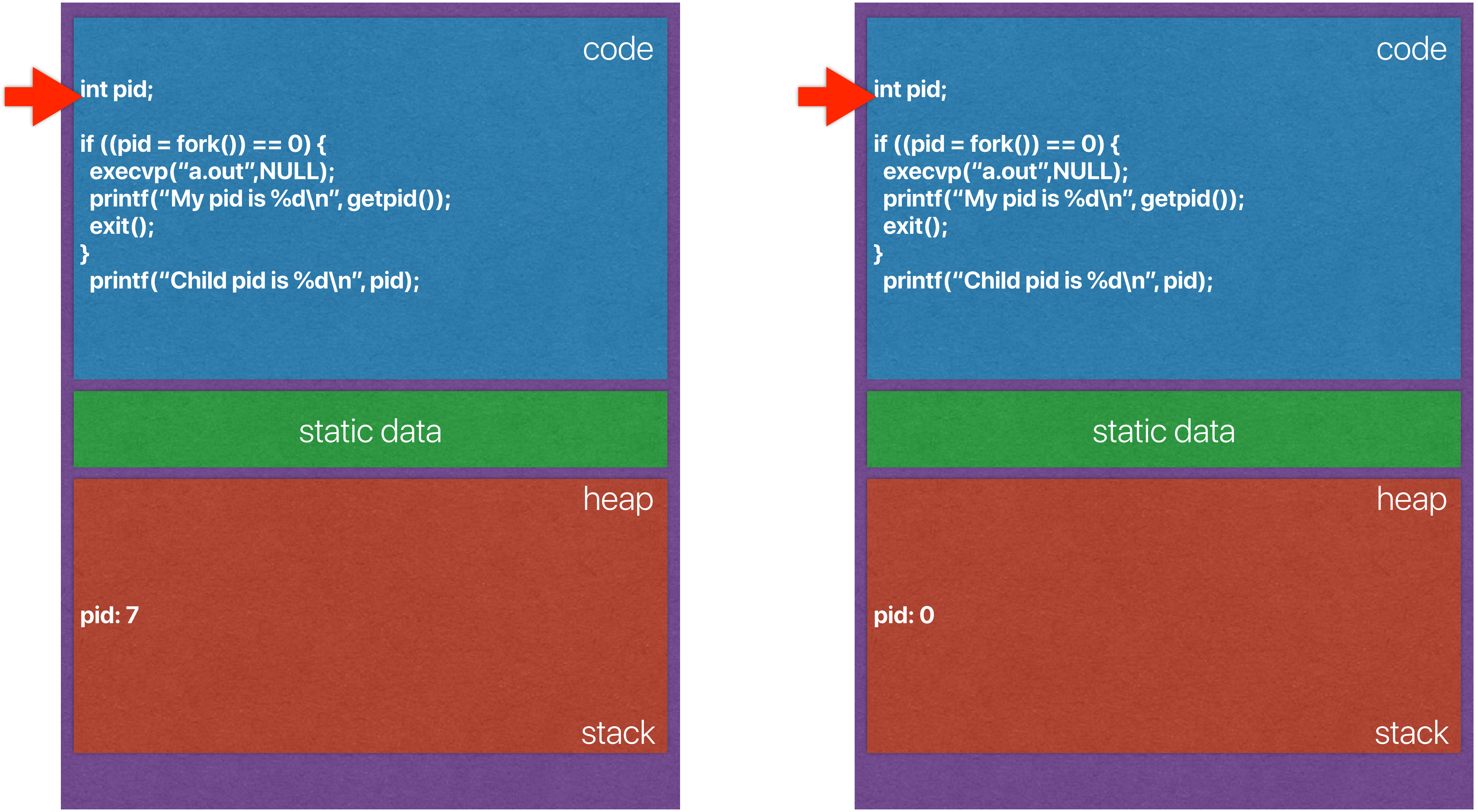
exec()



```
int pid;

if ((pid = fork()) == 0) {
  execvp(“a.out”,NULL);
  printf(“My pid is %d\n”, getpid());
  exit();
}
  printf(“Child pid is %d\n”, pid);
```

pid: ?

exec()

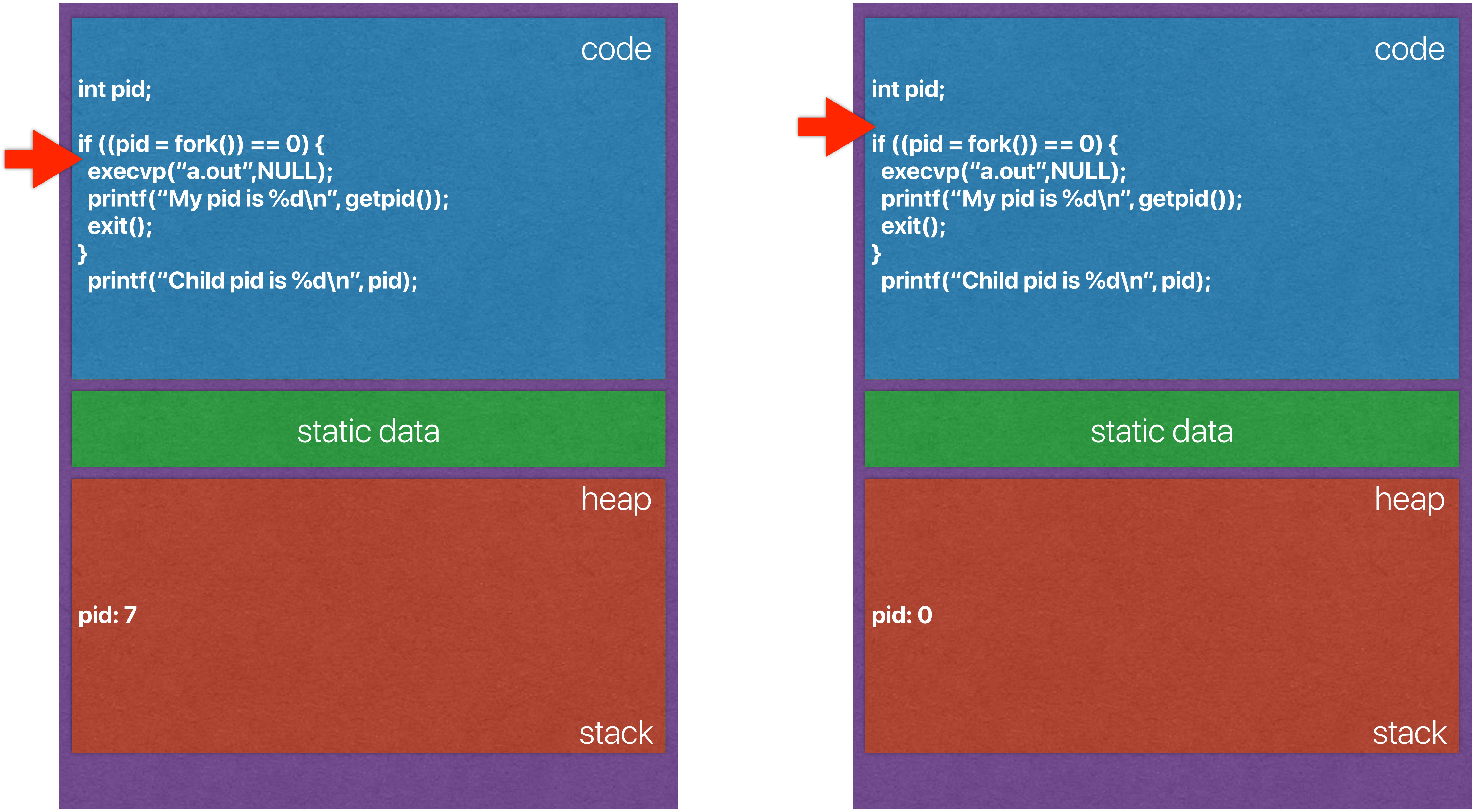


```
int pid;

if ((pid = fork()) == 0) {
  execvp(“a.out”,NULL);
  printf(“My pid is %d\n”, getpid());
  exit();
}
  printf(“Child pid is %d\n”, pid);
```

```
int pid;

if ((pid = fork()) == 0) {
  execvp(“a.out”,NULL);
  printf(“My pid is %d\n”, getpid());
  exit();
}
  printf(“Child pid is %d\n”, pid);
```

exec()

Output:
Child pid is 7

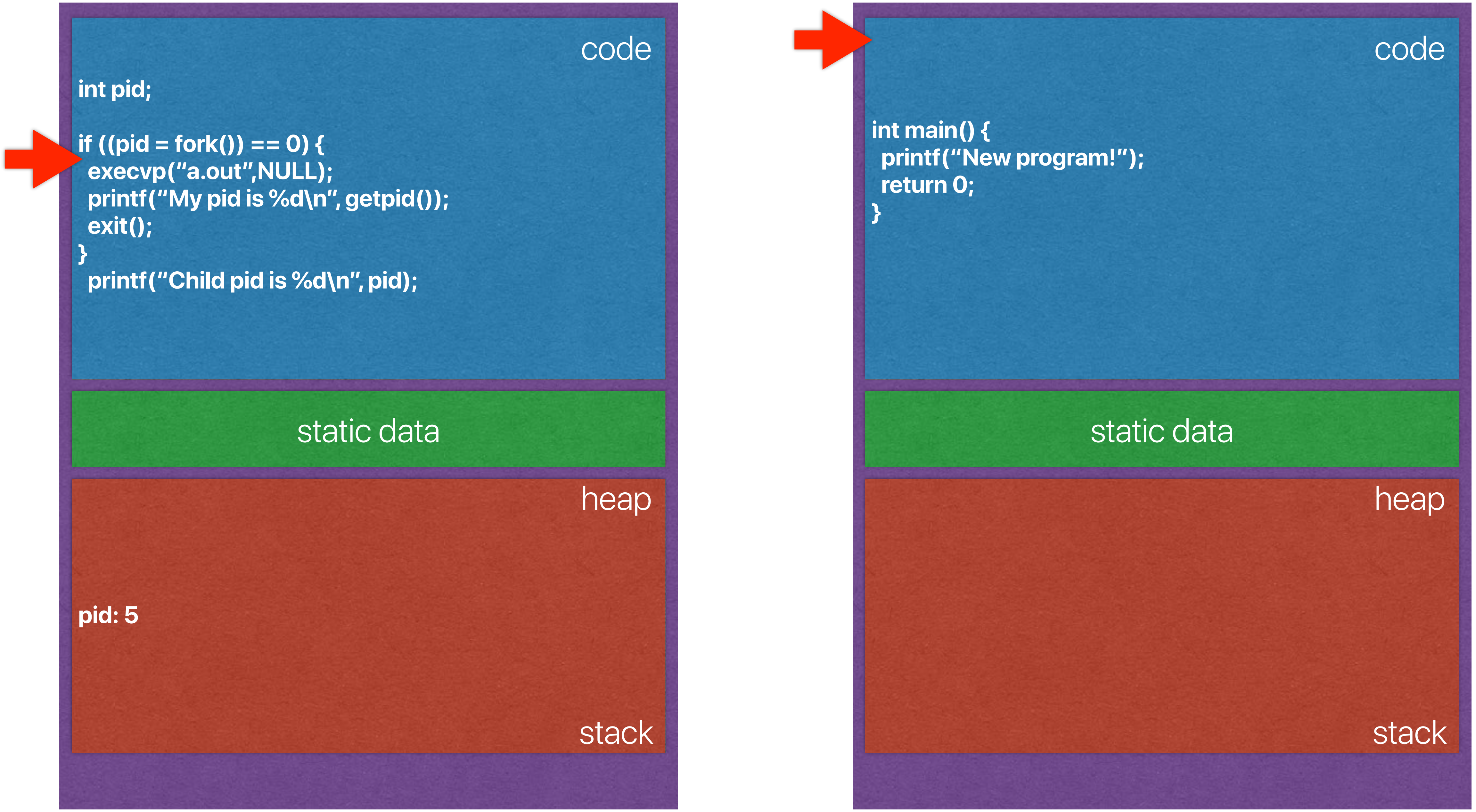


```
int pid;

if ((pid = fork()) == 0) {
  execvp("a.out",NULL);
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```

```
int pid;

if ((pid = fork()) == 0) {
  execvp("a.out",NULL);
  printf("My pid is %d\n", getpid());
  exit();
}
  printf("Child pid is %d\n", pid);
```

exec()

Output:
Child pid is 7
New program!



```
int pid;

if ((pid = fork()) == 0) {
  execvp(“a.out”,NULL);
  printf(“My pid is %d\n”, getpid());
  exit();
}
  printf(“Child pid is %d\n”, pid);
```

pid: 5

```
int main() {
  printf(“New program!”);
  return 0;
}
```

Let’s write our own shells

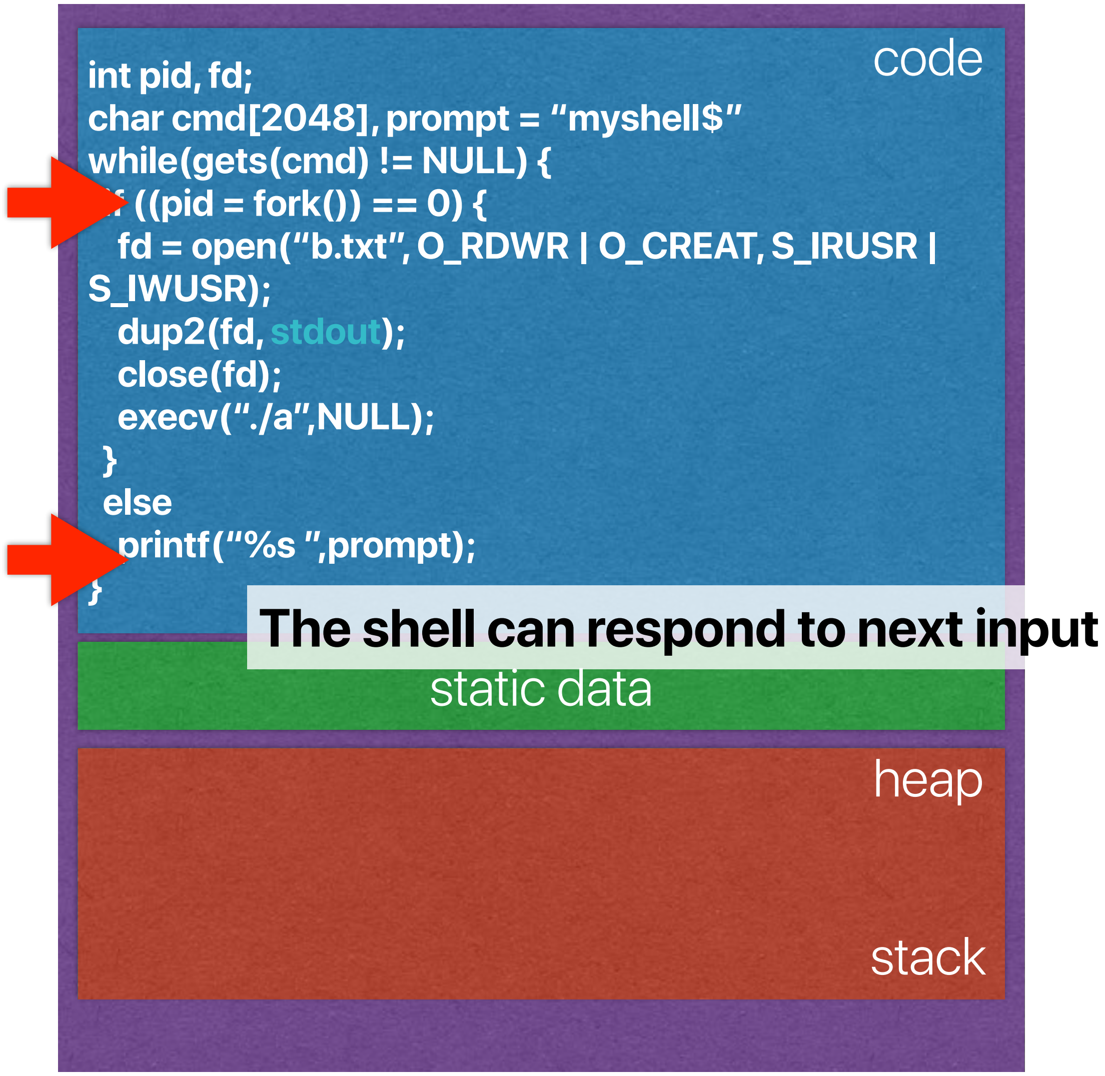
How to implement redirection in shell

- Say, we want to do ./a > b.txt
- fork
- The forked code opens b.txt
- The forked code dup the file descriptor
- The forked code assigns b.txt to stdin/stdout
- The forked code closes b.txt
- exec(“./a”, NULL)

How to implement redirection in shell

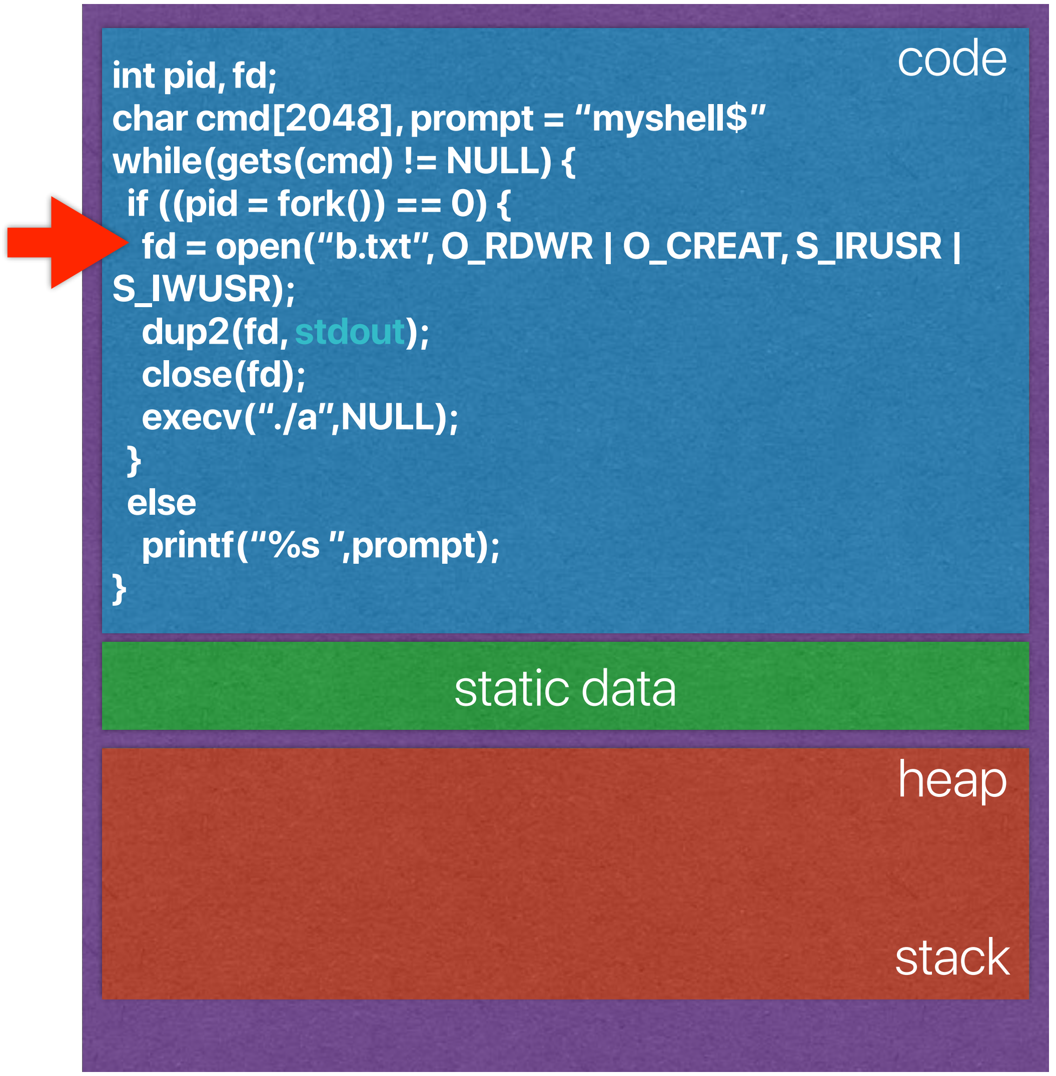
- Say, we want to do ./a > b.txt
- fork
- The forked code opens b.txt
- The forked code dup the file descriptor to stdin/stdout
- The forked code closes b.txt
- exec(“./a”, NULL)

Homework for you:
Think about the case when
your `fork` is equivalent to `fork+exec`()



```
int pid, fd;
char cmd[2048], prompt = “myshell$”
while(gets(cmd) != NULL) {
 if ((pid = fork()) == 0) {
   fd = open(“b.txt”, O_RDWR | O_CREAT, S_IRUSR |
S_IWUSR);
   dup2(fd, stdout);
   close(fd);
   execv(“./a”,NULL);
 }
 else
   printf(“%s ”,prompt);
}
```

The shell can respond to next input



```
int pid, fd;
char cmd[2048], prompt = “myshell$”
while(gets(cmd) != NULL) {
 if ((pid = fork()) == 0) {
   fd = open(“b.txt”, O_RDWR | O_CREAT, S_IRUSR |
S_IWUSR);
   dup2(fd, stdout);
   close(fd);
   execv(“./a”,NULL);
 }
 else
   printf(“%s ”,prompt);
}
```

wait()

- pid_t wait(int *stat)
- pid_t waitpid(pid_t pid, int *stat, int opts)
- wait / waitpid suspends process until a child process ends
  - wait resumes when any child ends
  - waitpid resumes with child with pid ends
  - exit status info 1 is stored in *stat
  - Returns pid of child that ended, or -1 on error
- Unix requires a corresponding wait for every fork

Zombies, Orphans, and Adoption

- Zombie: process that exits but whose parent doesn’t call wait
  - Can’t be killed normally
  - Resources freed but pid remains in use
- Orphan: Process whose parent has exited before it has
  - Orphans are **adopted** by init process, which calls wait periodically

Mach: A New Kernel Foundation For UNIX Development

Mike Accetta , Robert Baron , William Bolosky , David Golub , Richard Rashid , Avadis Tevanian , Michael Young

Computer Science Department, Carnegie Mellon University

Poll close in 1:30

Why is "Mach" proposed?

- How many of the following statements is/are true regarding the motivations of developing Mach in 1986?
  ① Modern UNIX systems do not provide consistent interfaces for system facilities
  ② System level services can only be provided through fully integration of the UNIX kernel
  ③ The process abstraction cannot use multiprocessors efficiently
  ④ Network communication is not protected
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Poll close in 1:30

Why is "Mach" proposed?

- How many of the following statements is/are true regarding the motivations of developing Mach in 1986?
  - ① Modern UNIX systems do not provide consistent interfaces for system facilities
  - ② System level services can only be provided through fully integration of the UNIX kernel
  - ③ The process abstraction cannot use multiprocessors efficiently
  - ④ Network communication is not protected
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Why is “Mach” proposed?

- How many of the following statements is/are true regarding the motivations of developing Mach in 1986?
  - ① Modern UNIX systems do not provide consistent interfaces for system facilities
  - ② System level services can only be provided through fully integration of the UNIX kernel
  - ③ The process abstraction cannot use multiprocessors efficiently
  - ④ Network communication is not protected

  A. 0

  B. 1

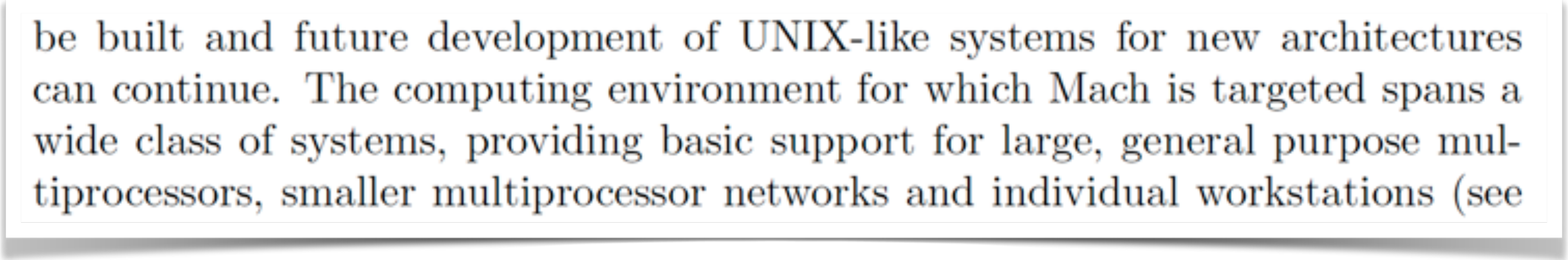
  C. 2

  D. 3

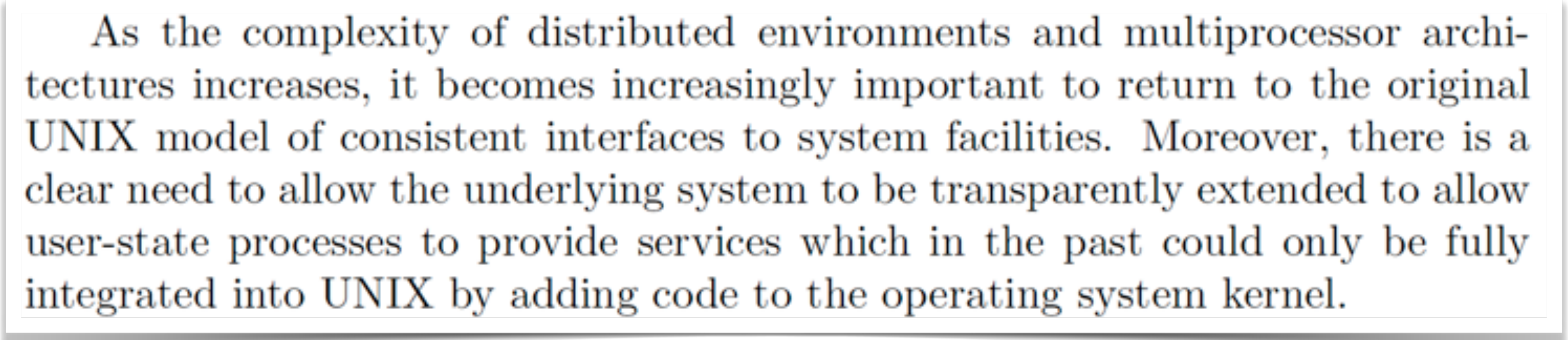
  [✓ E. 4]

Why "Mach"?

- The hardware is changing
  - Multiprocessors
  - Networked computing
- The software
  - The demand of extending an OS easily
  - Repetitive but confusing mechanisms for similar stuffs



be built and future development of UNIX-like systems for new architectures can continue. The computing environment for which Mach is targeted spans a wide class of systems, providing basic support for large, general purpose multiprocessors, smaller multiprocessor networks and individual workstations (see



As the complexity of distributed environments and multiprocessor architectures increases, it becomes increasingly important to return to the original UNIX model of consistent interfaces to system facilities. Moreover, there is a clear need to allow the underlying system to be transparently extended to allow user-state processes to provide services which in the past could only be fully integrated into UNIX by adding code to the operating system kernel.

Make UNIX great again!

Poll close in 1:30

Whys v.s. whats

- How many pairs of the “why” and the “what” in Mach are correct?

Why		What
(1)	Support for multiprocessors	**Threads**
(2)	Networked computing	**Messages/Ports**
(3)	OS Extensibility	**Kernel debugger**
(4)	Repetitive but confusing mechanisms	**Messages/Ports**

A. 0
B. 1
C. 2
D. 3
E. 4

Poll close in 1:30

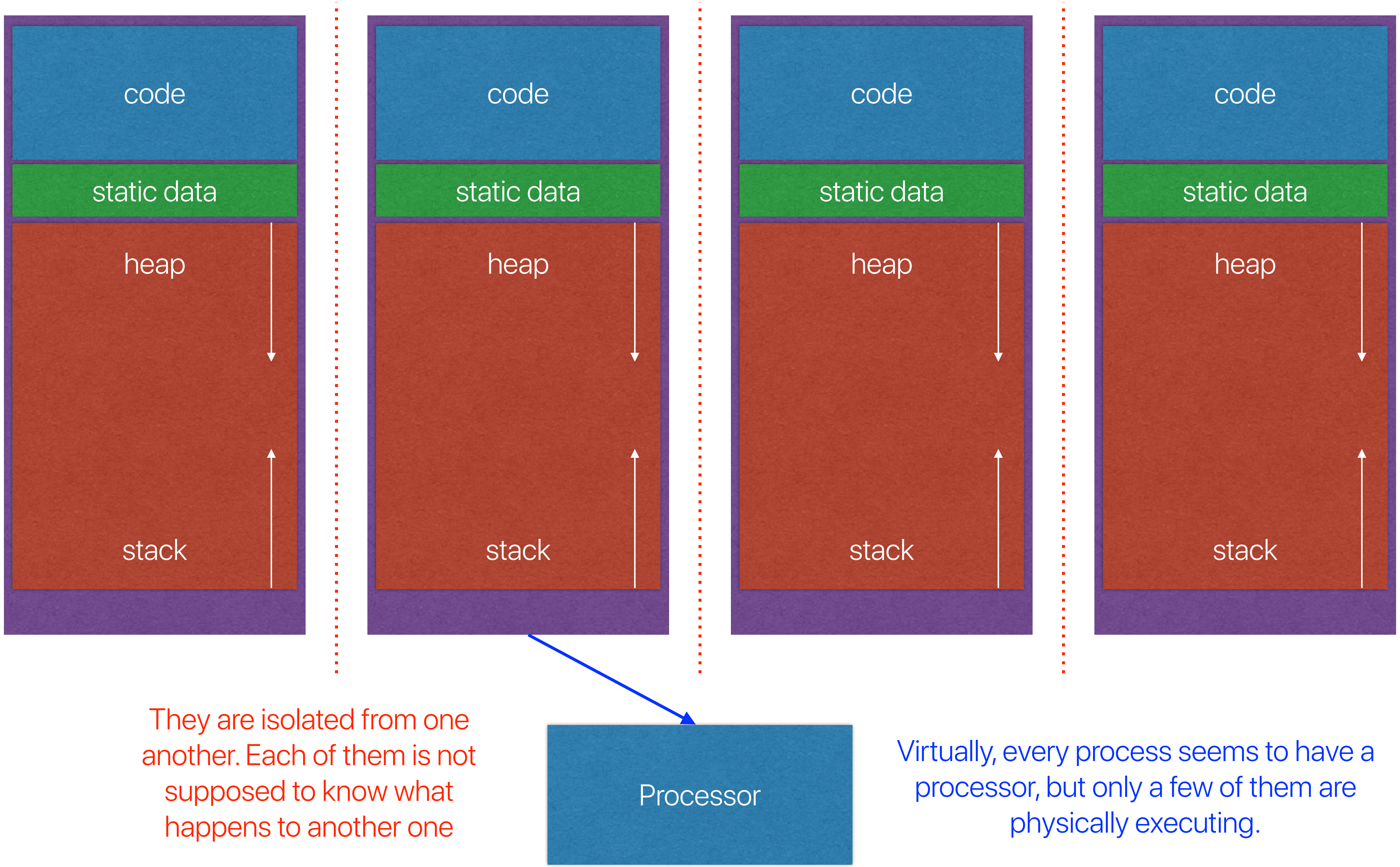
Whys v.s. whats

- How many pairs of the "why" and the "what" in Mach are correct?

	Why	What
(1)	Support for multiprocessors	**Threads**
(2)	Networked computing	**Messages/Ports**
(3)	OS Extensibility	**Kernel debugger**
(4)	Repetitive but confusing mechanisms	**Messages/Ports**

A. 0
B. 1
C. 2
D. 3
E. 4

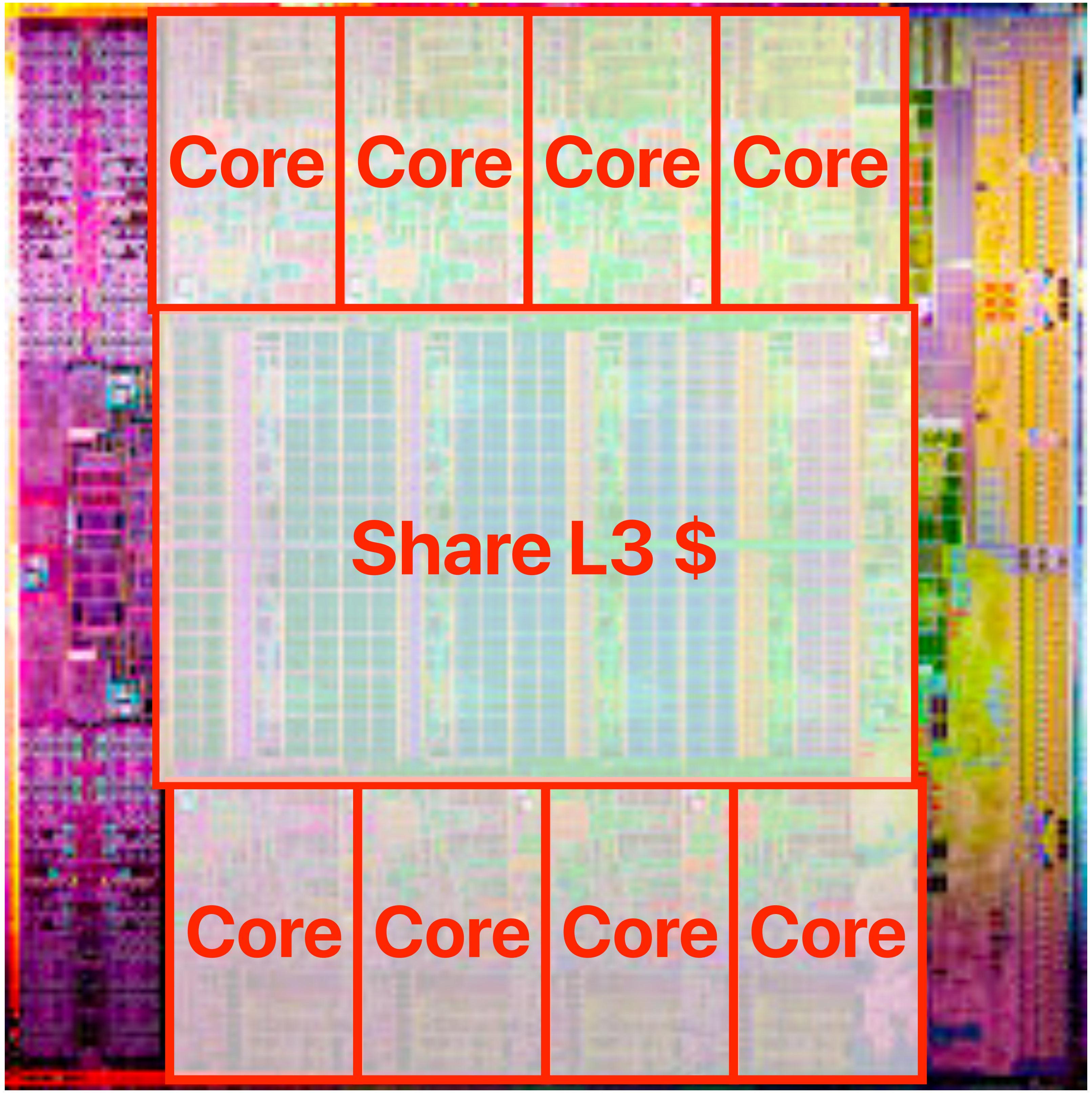
Recap: Each process has a separate virtual memory space



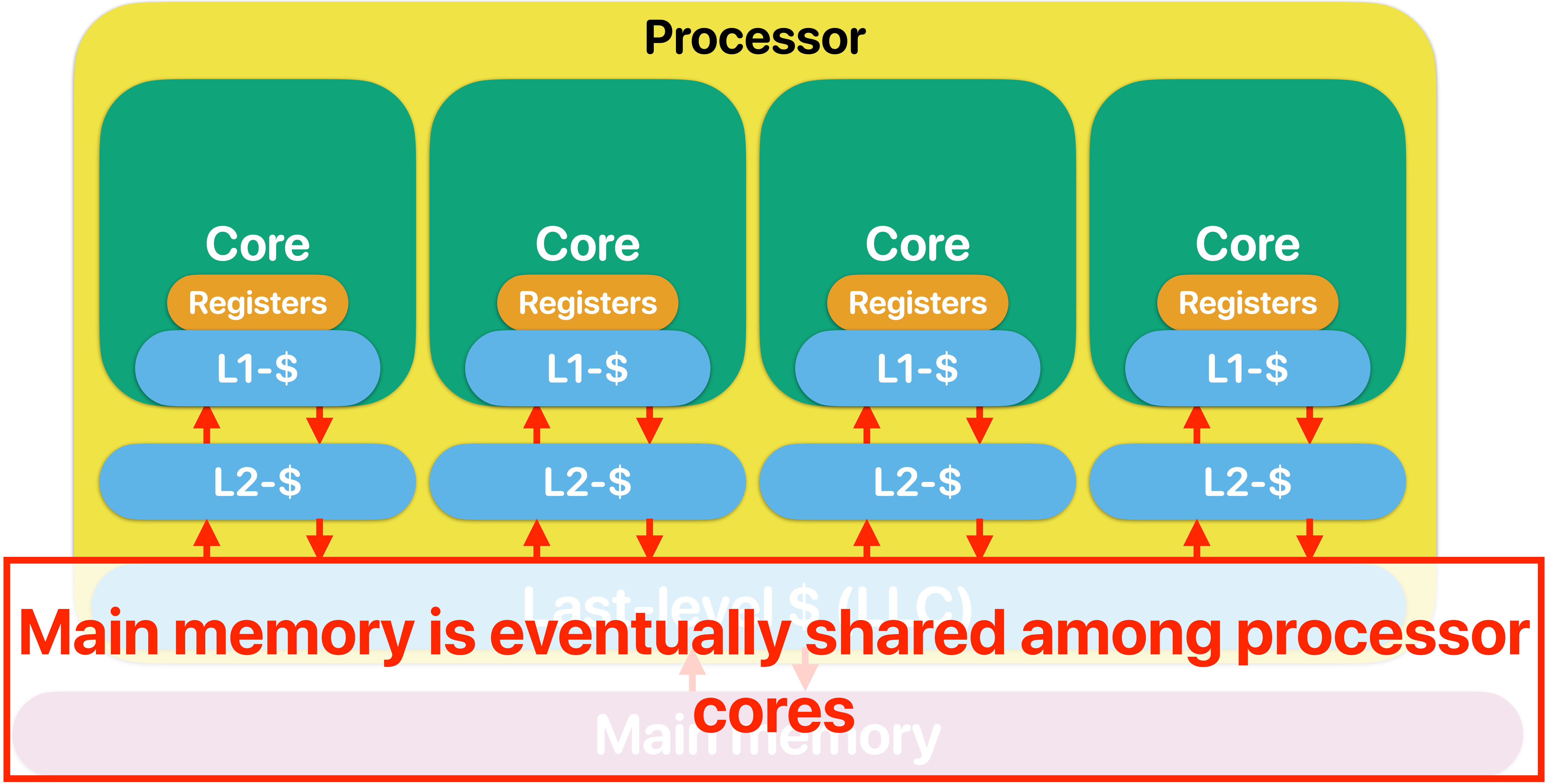
They are isolated from one another. Each of them is not supposed to know what happens to another one

Virtually, every process seems to have a processor, but only a few of them are physically executing.

Intel Sandy Bridge



Concept of chip multiprocessors



Main memory is eventually shared among processor cores

Whys v.s. whats

- How many pairs of the "why" and the "what" in Mach are correct?

Why		What
(1)	Support for multiprocessors	**Threads**
(2)	Networked computing	**Messages/Ports**
(3)	OS Extensibility	**Microkernel/Object-oriented design**
(4)	Repetitive but confusing mechanisms	**Messages/Ports**

A. 0

B. 1

C. 2

[✓ D. 3]

E. 4

Announcement

- Reading quizzes due next Tuesday
- Project groups in 2
  - Will release the project by the end of the week

Computer
Science &
Engineering

202

つづく