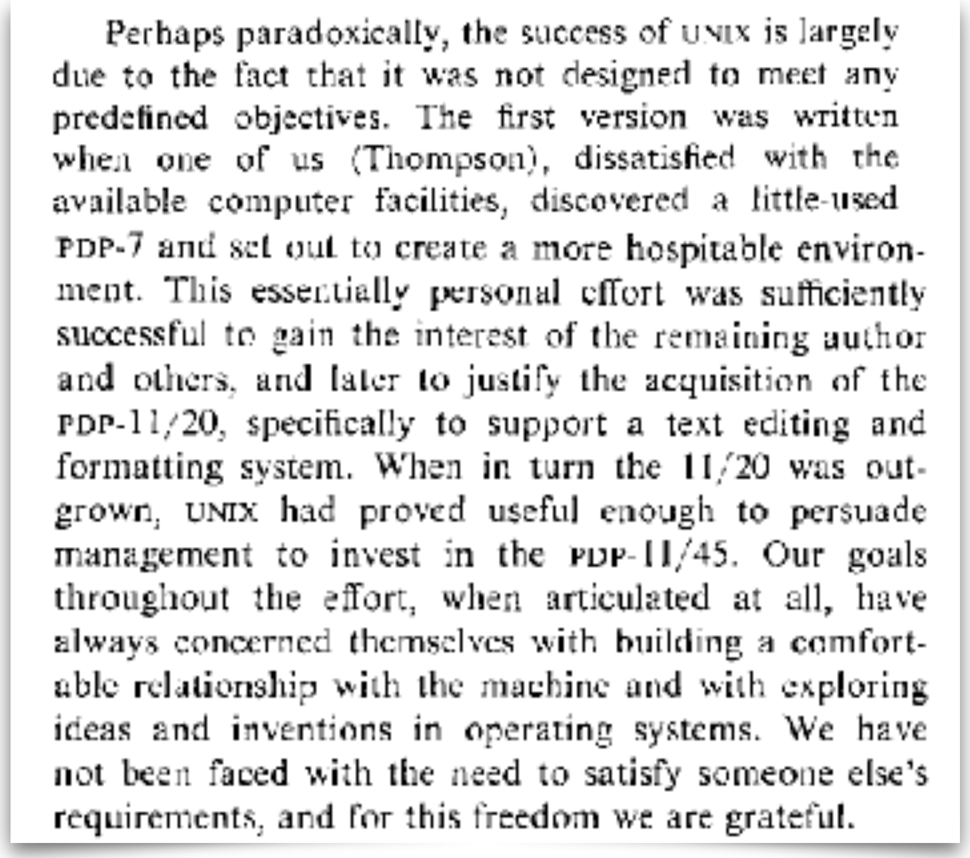
Design philosophy of operating systems (III)

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.

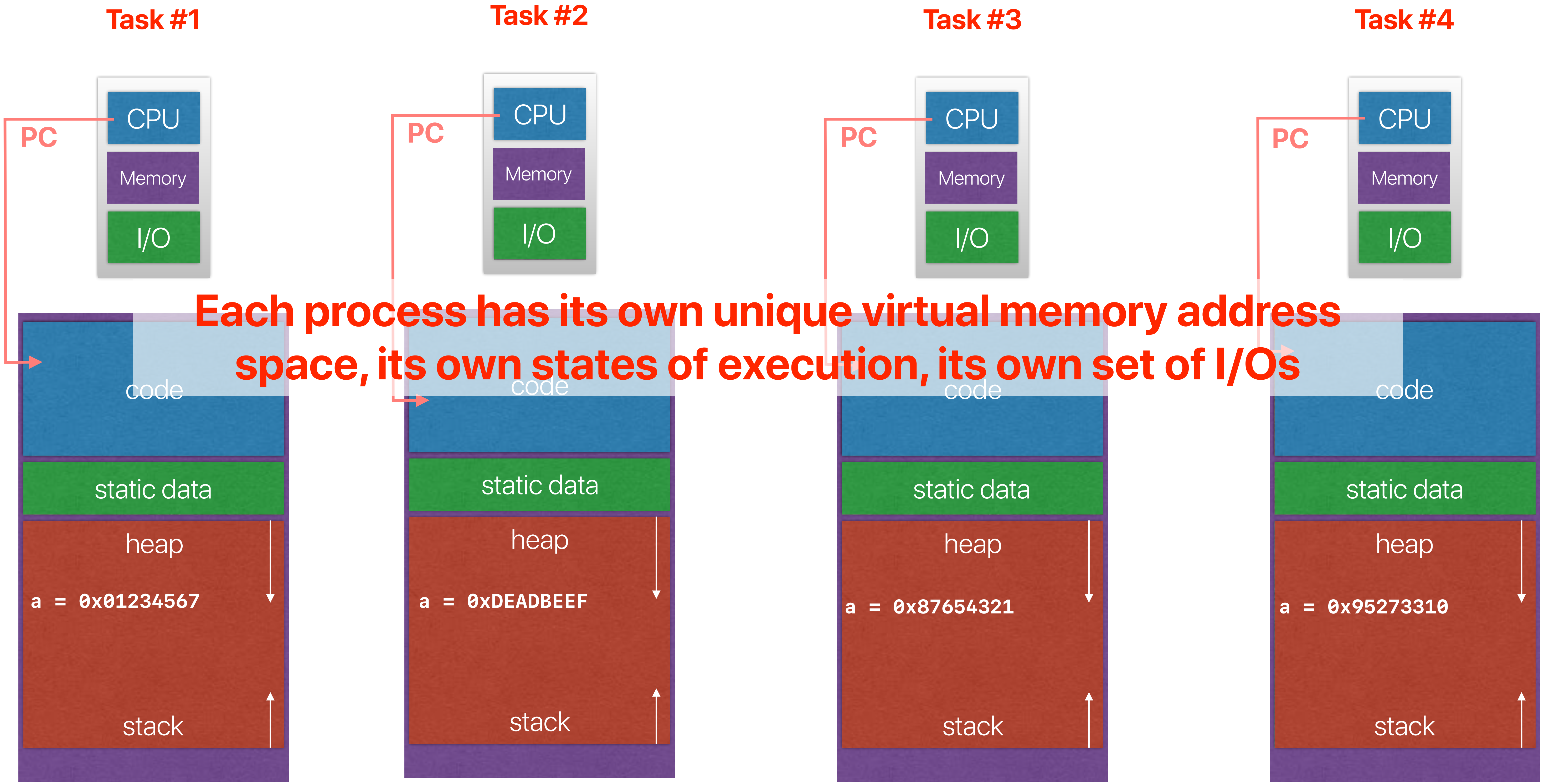
Recap: Protection mechanisms

- 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
- Mach
  - Protection is associated with each “object” — embedded in the memory space of each object — capability
  - Each object can set permissions on functions individually

Outline

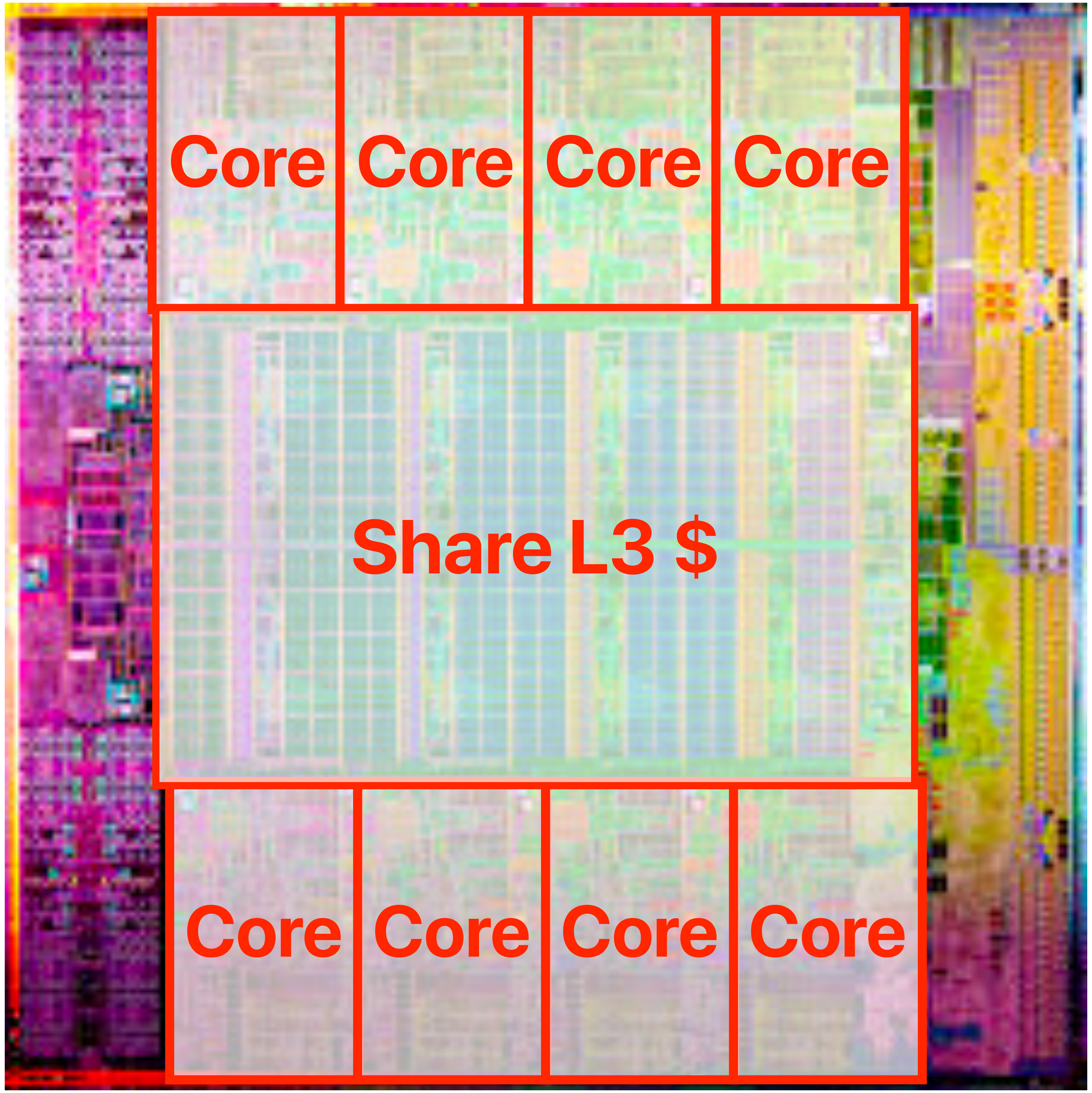
- Mach: A New Kernel Foundation For UNIX Development
- The process interface in UNIX

Tasks/processes

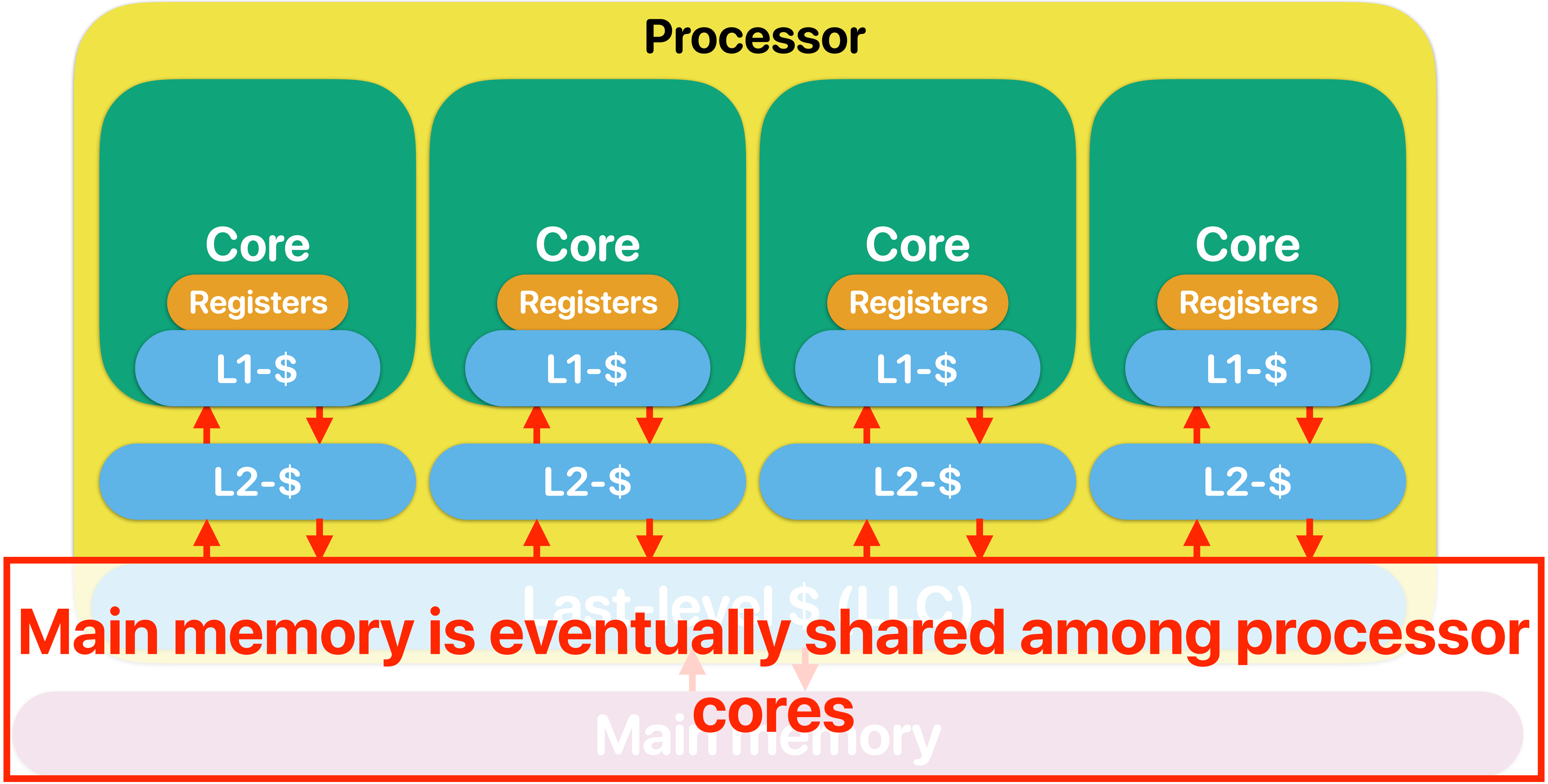


Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os

Intel Sandy Bridge

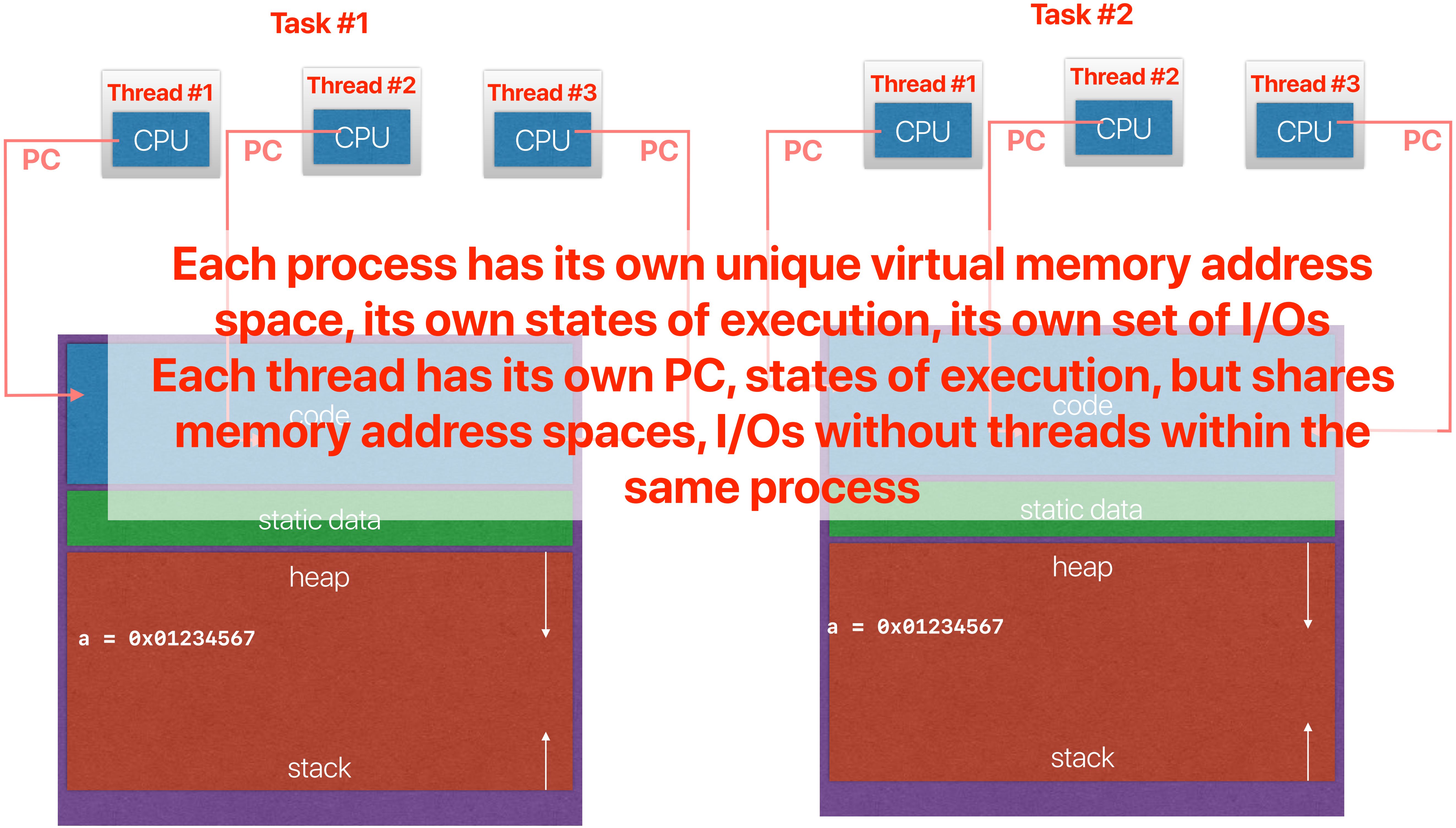


Concept of chip multiprocessors



Main memory is eventually shared among processor cores

Threads



Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os

Each thread has its own PC, states of execution, but shares memory address spaces, I/Os without threads within the same process

Why Threads?

- Process is an abstraction of a computer
  - When you create a process, you duplicate everything
  - However, you only need to duplicate CPU abstraction to parallelize computation tasks
- Threads as lightweight processes
  - Thread is an abstraction of a CPU in a computer
  - Maintain separate execution context
  - Share other resources (e.g. memory)

The cost of creating processes

- Measure process creation overhead using lmbench http://www.bitmover.com/lmbench/

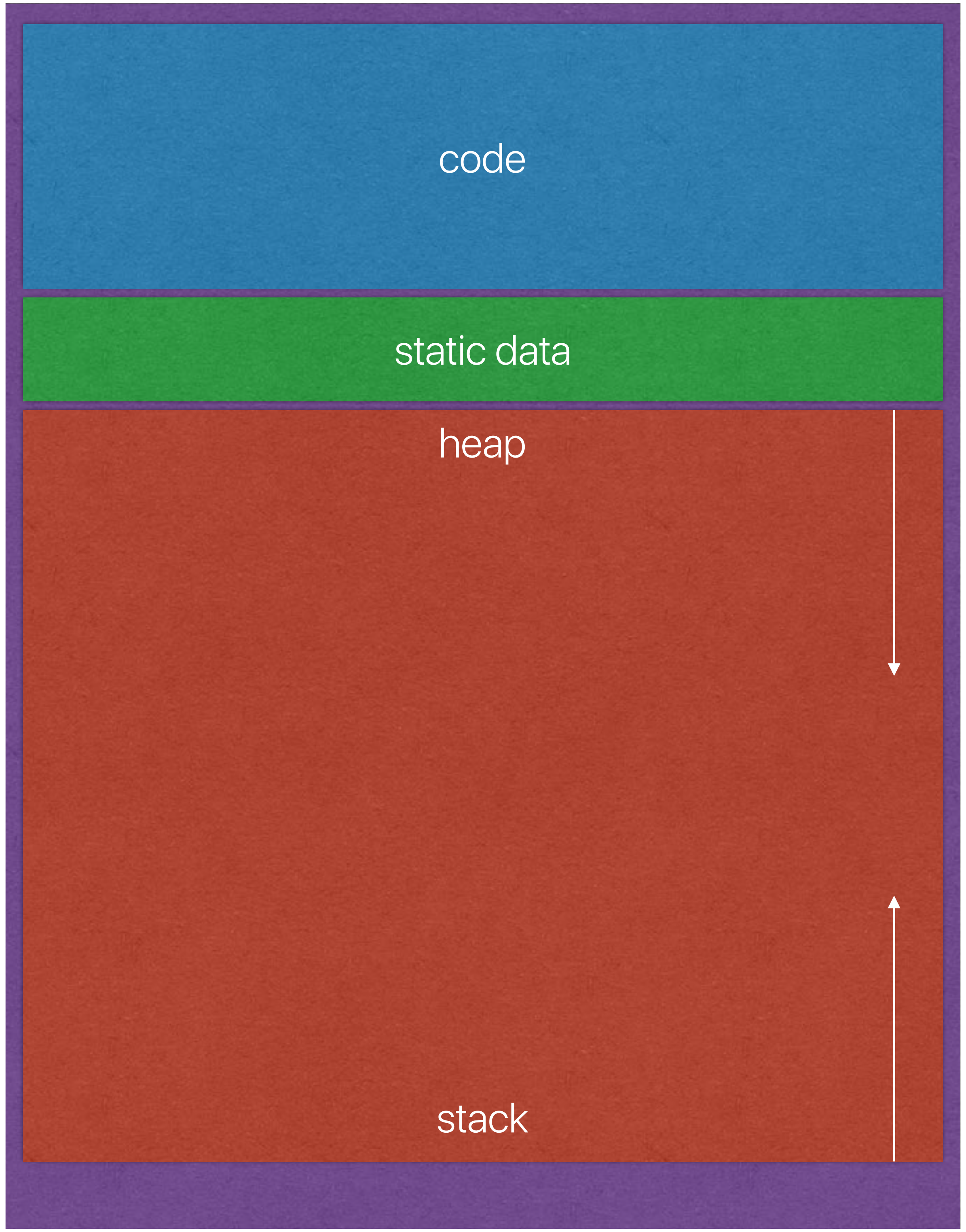
The cost of creating processes

- Measure process creation overhead using lmbench http://www.bitmover.com/lmbench/
- On a 3.7GHz intel Core i5-9600K Processor
  - Process fork+exit ~ 57 microseconds
  - More than 16K cycles

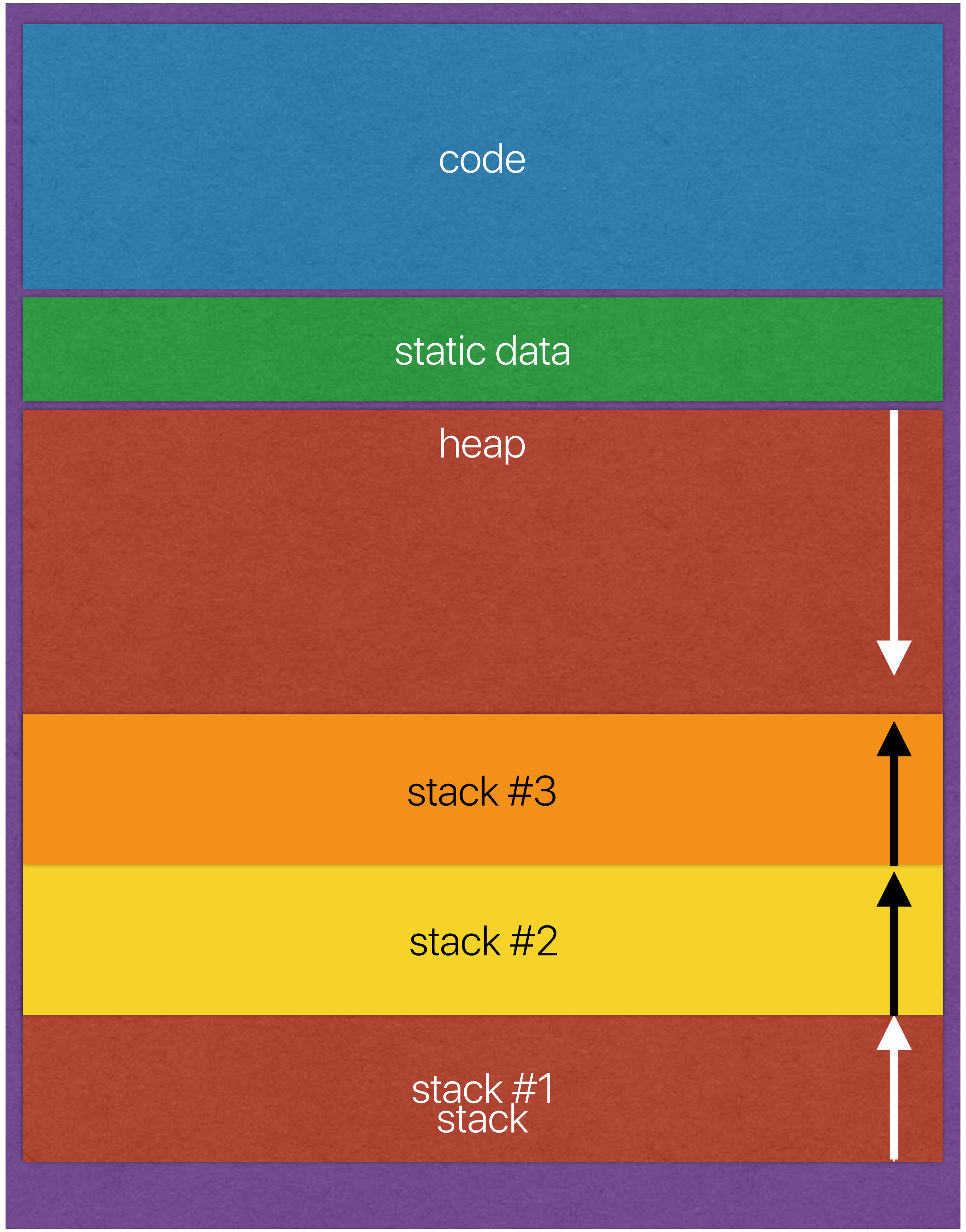
What should threads share?

- How many of the following memory elements should be shared by two threads in the same process?
  - ① Stack section — **each function call, local variables should still be independent**
  - ② Data section — **global variables should be visible to every thread**
  - ③ Text/code section — **each thread are running instructions from the same process**
  - ④ Page table — **all threads share the same address space**
  - A. 0
  - B. 1
  - C. 2
  - [✓ D. 3]
  - E. 4

The virtual memory of single-threaded applications



The virtual memory of multithreaded applications



Tasks/Processes and threads

- How many of the following regarding the comparison of parallelizing computation tasks using processes and threads is/are correct?

① The context switch and creation overhead of processes is higher

— you have to change page tables, warm up TLBs, warm up caches, create a new memory space …

② The overhead of exchanging data among different computing tasks for the same applications is higher in process model

— you cannot directly share data without leveraging other mechanisms

③ The demand of memory usage is higher when using processes

— each process needs its own address space even if most data are potentially identical

④ The security and isolation guarantees are better achieved using processes

— separate address, it's not easy to access data from another process

A. 0

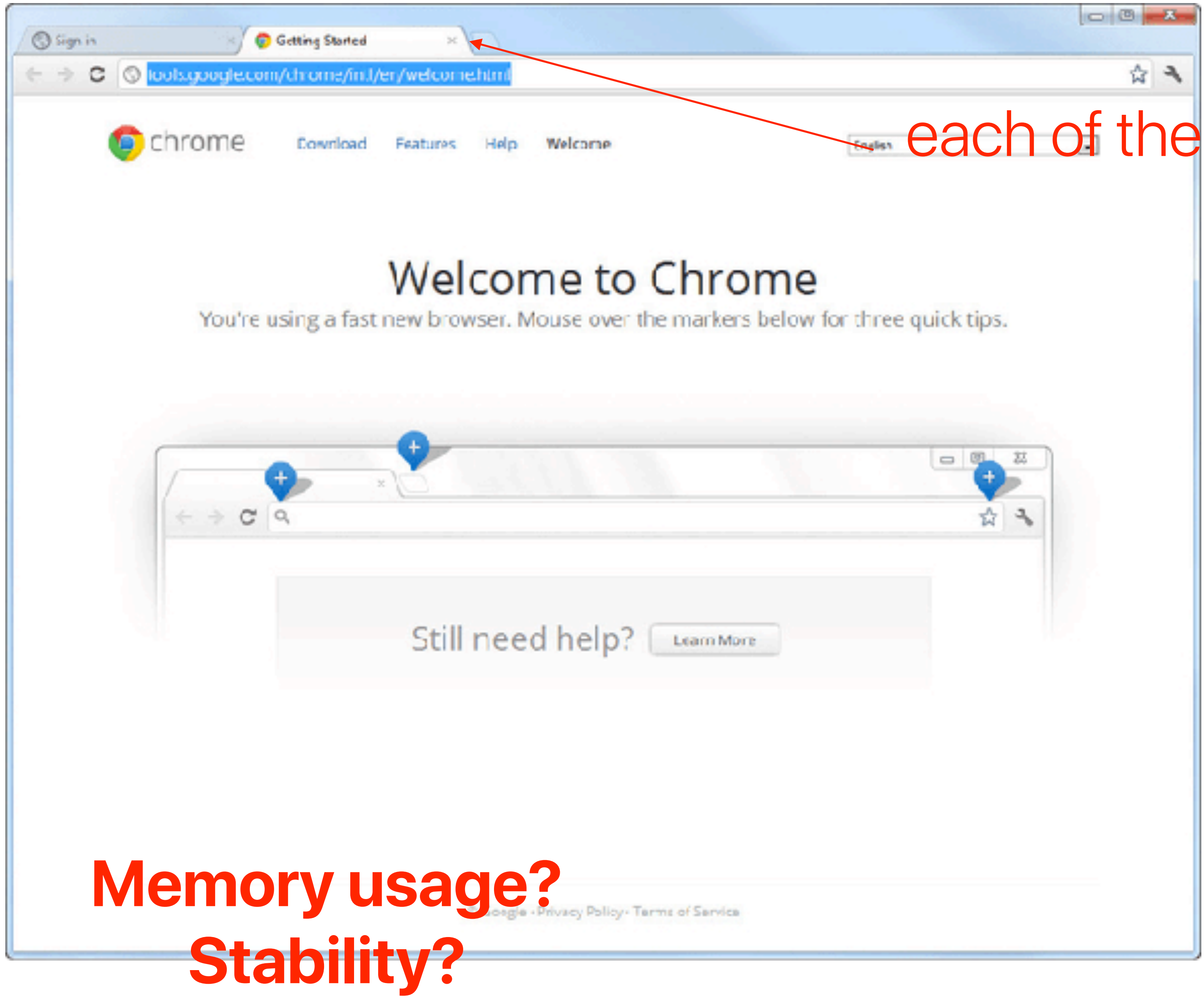
B. 1

C. 2

D. 3

[✓ E. 4]

Case study: Chrome v.s. Firefox



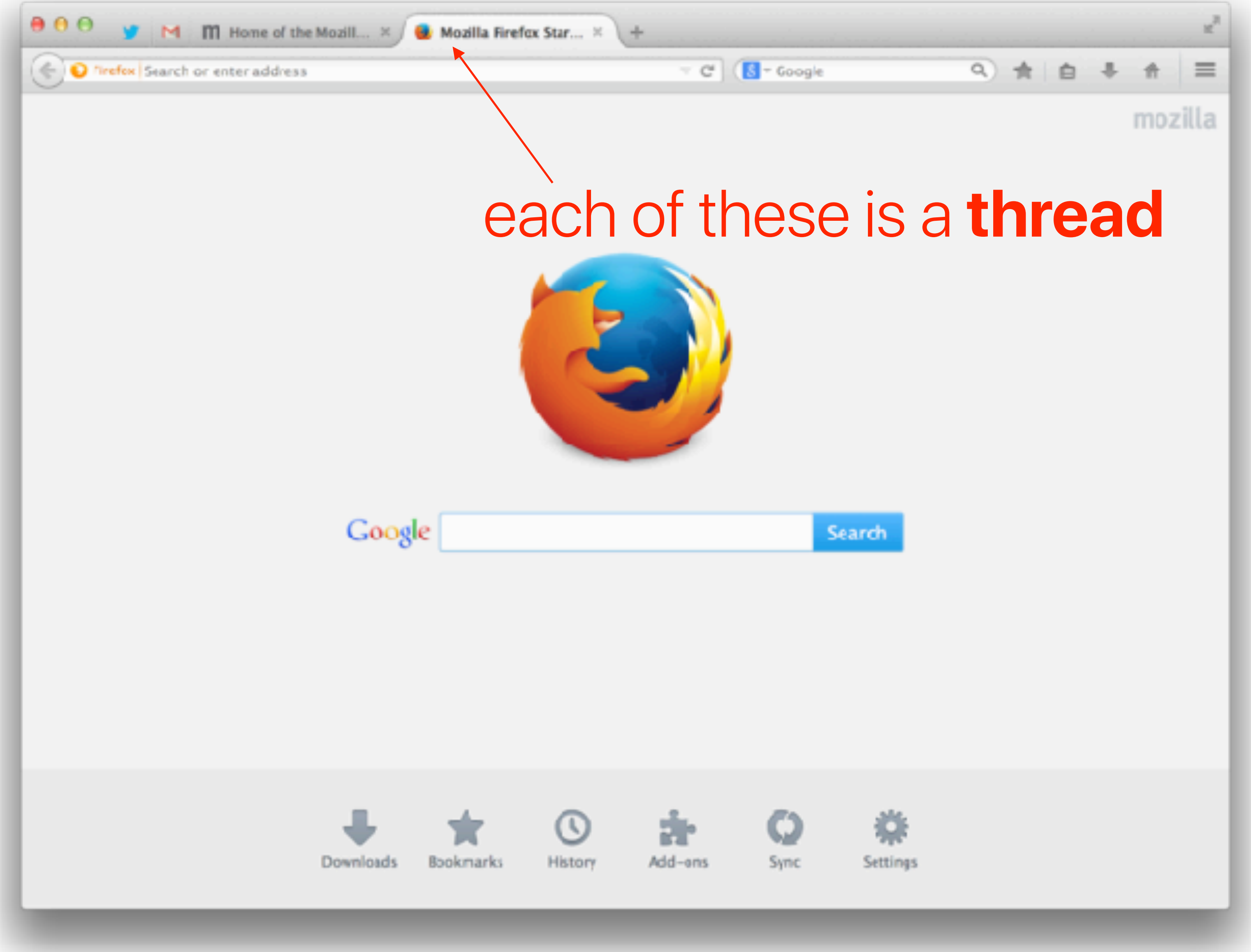
Welcome to Chrome

You're using a fast new browser. Mouse over the markers below for three quick tips.

Still need help?

Google - Privacy Policy - Terms of Service

each of these is a **process**



each of these is a **thread**

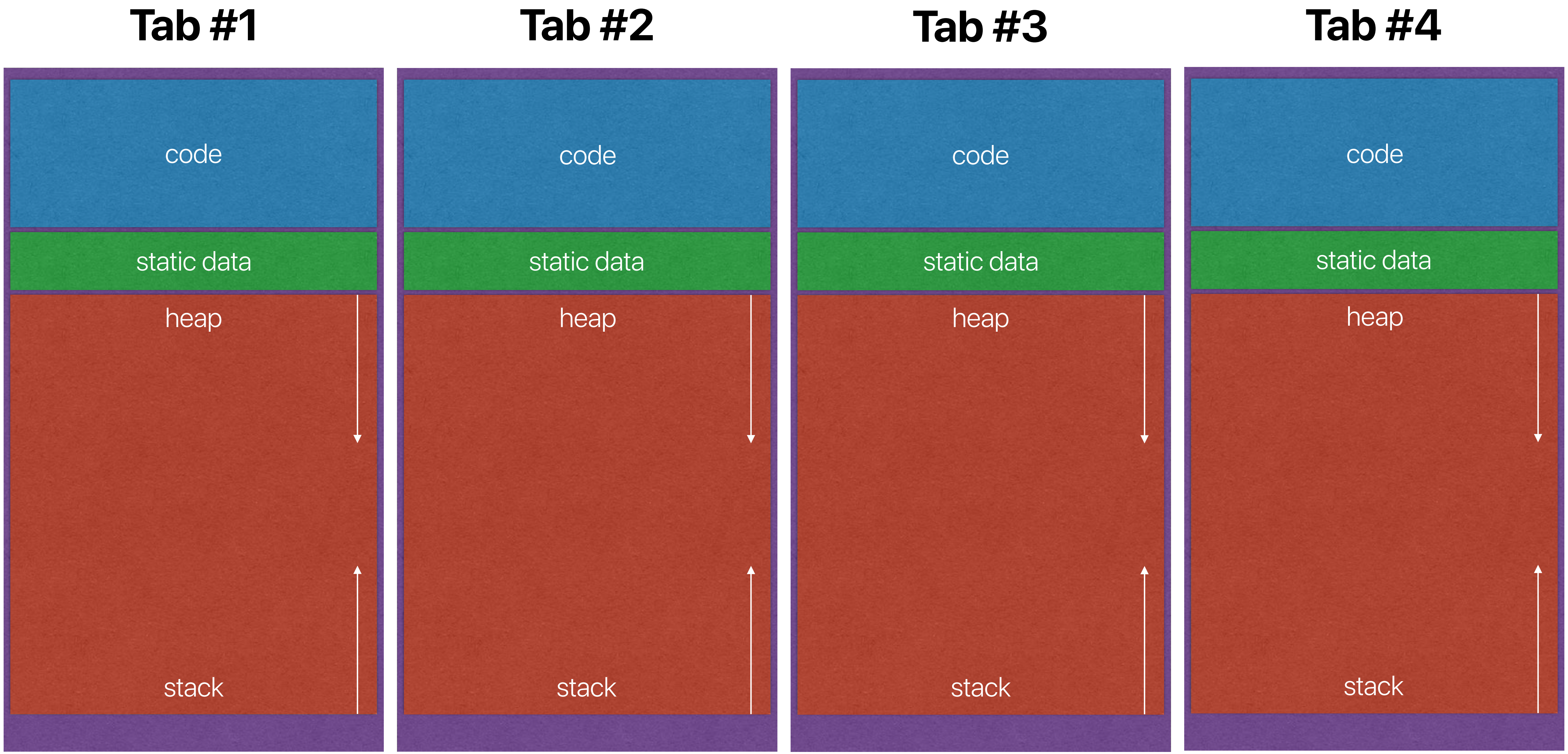
Memory usage?

Stability?

Security?

Latency?

Chrome



Tab #1

code

static data

heap

stack

Tab #2

code

static data

heap

stack

Tab #3

code

static data

heap

stack

Tab #4

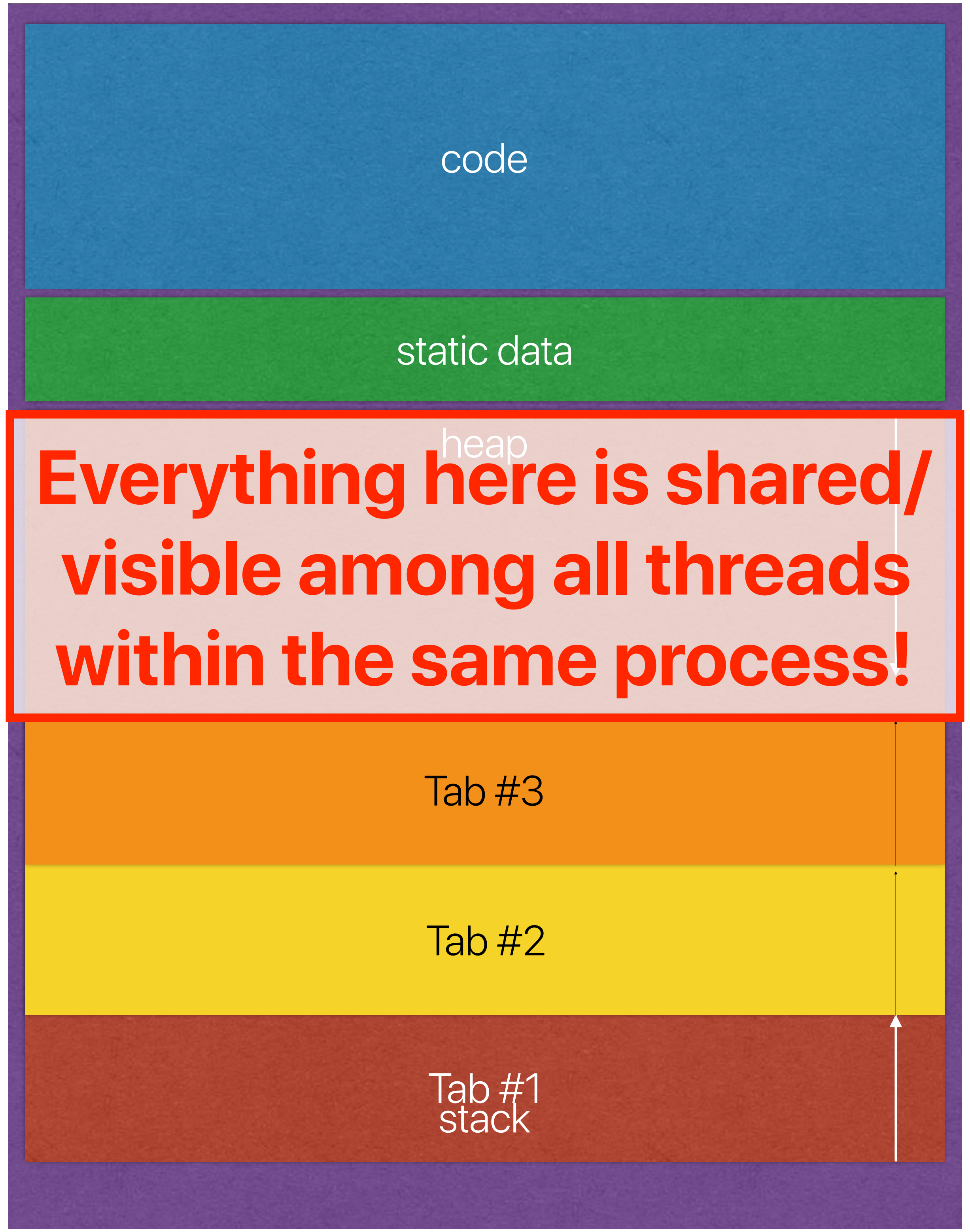
code

static data

heap

stack

Firefox



code

static data

heap

Everything here is shared/ visible among all threads within the same process!

Tab #3

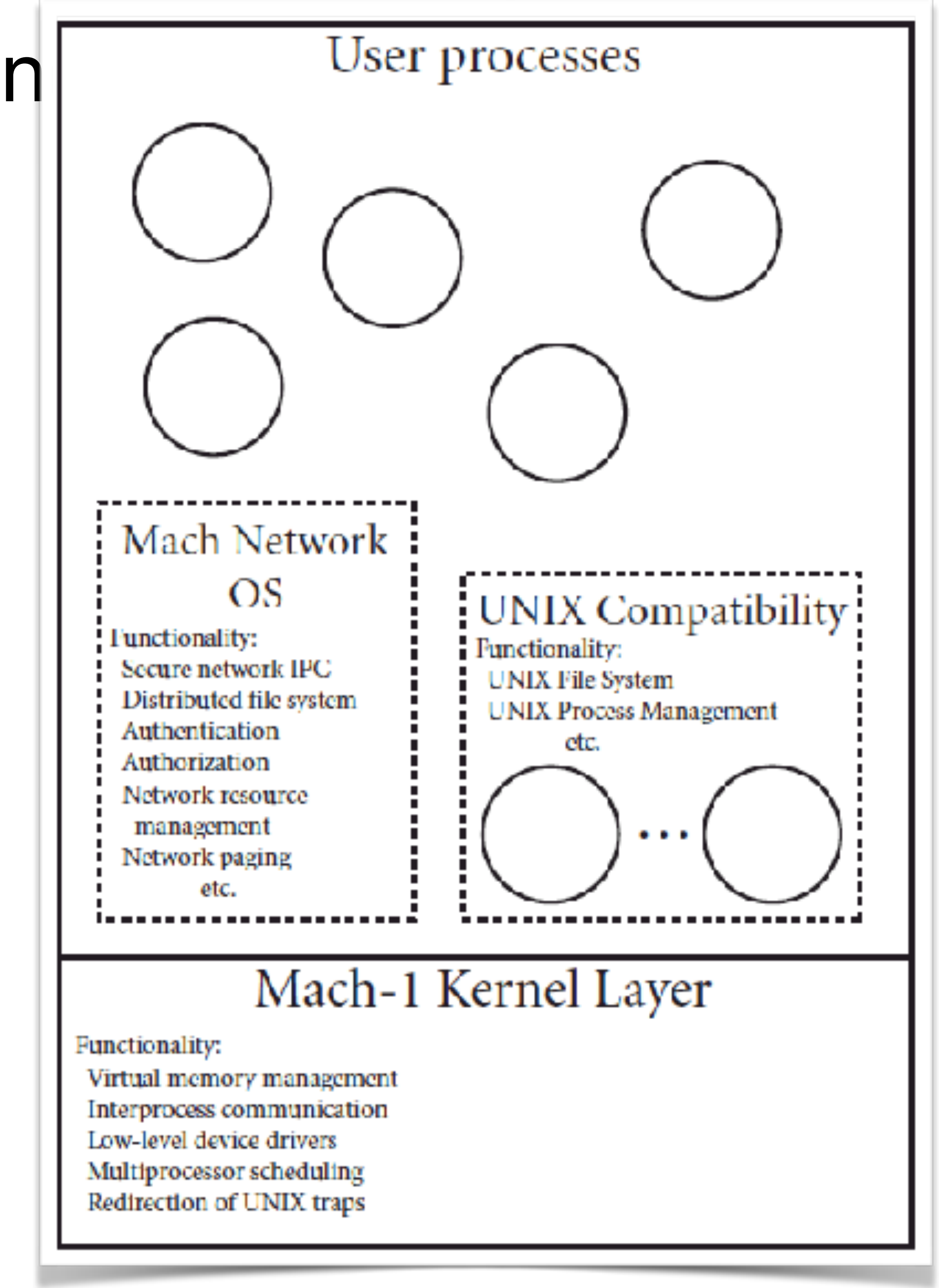
Tab #2

Tab #1

stack

What's in the kernel?

- How many of the following Mach features/fun implemented in the kernel?
  ① I/O device drivers
  ② File system
  ③ Shell
  ④ Virtual memory management
  A. 0
  B. 1
  [✓ C. 2]
  D. 3
  E. 4



Whys v.s. whats

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

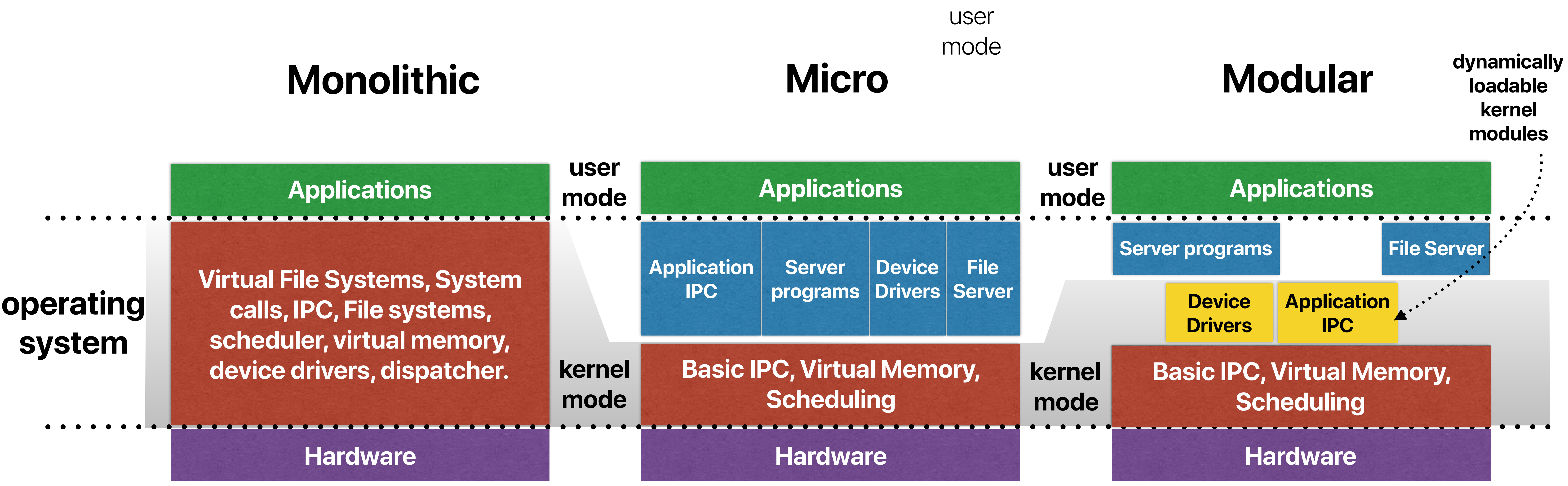
Why		What
(1)	Support for multiprocessors	**Threads**
(2)	Networked computing	**Messages/Ports**
(3)	OS Extensibility	**Microkernel/Object-oriented design**
(4)	Repetitive but confusing mechanisms	**Messages/Ports**

A. 0
B. 1
C. 2
[✓ D. 3]
E. 4

Types of kernels

- What type of kernels does the UNIX described in Dennis M. Ritchie's paper belong to?
  A. Microkernel — the kernel only provides a minimal set of services including memory management, multitasking and inter-process communication **Hydra, Mach**
  B. Monolithic — the kernel implements every function that cannot be in a user-space library: device drivers, scheduler, memory handling, file systems, network stacks **Old UNIX**
  C. Modular — the kernel provides a basic set of functions like microkernels, but allows load/unload kernel modules if necessary **Linux, Windows, MacOS, FreeBSD**
  D. Layered kernel — the kernel follows strict layered design that lower-order module cannot interact with higher-order modules **THE**

Types of Kernels



Original UNIX

Hydra, Mach

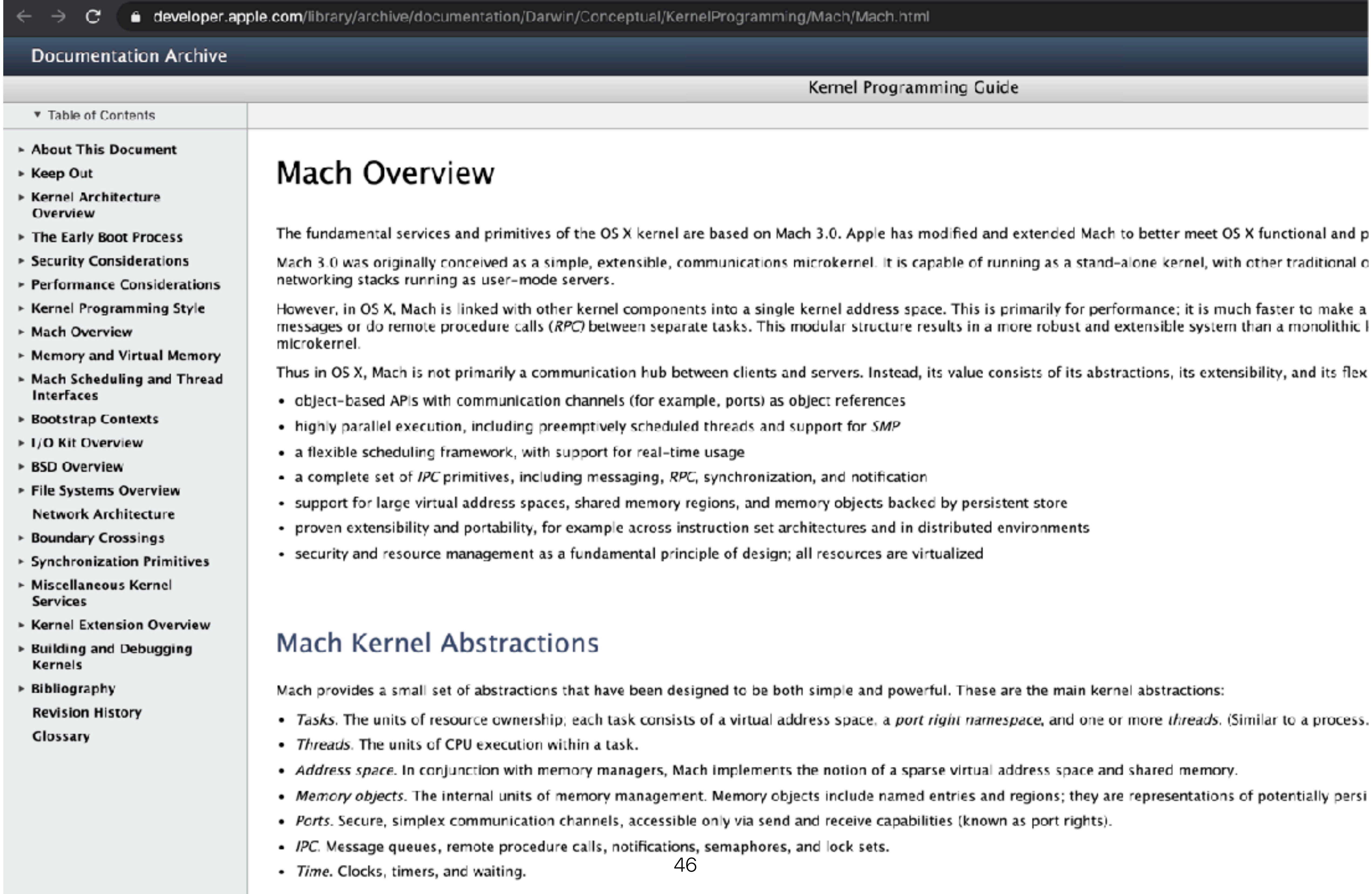
Linux, Windows, MacOS

Why not microkernels?

- Although Mach's design strongly influenced modern operating systems, why most modern operating systems do not adopt the design of microkernels?
  A. Microkernels are more difficult to extend than monolithic kernels
  B. Microkernels are more difficult to maintain than monolithic kernels
  C. Microkernels are less stable than monolithic kernels
  D. Microkernels are not as competitive as monolithic kernels in terms of application performance **Context switches!**
  E. Microkernels are less flexible than monolithic kernels

The impact of Mach

- Threads
- Extensible operating system kernel design
- Strongly influenced modern operating systems
  - Windows NT/2000/XP/7/8/10
  - MacOS



developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html
Documentation Archive
Kernel Programming Guide
▼ Table of Contents

Mach Overview

The fundamental services and primitives of the OS X kernel are based on Mach 3.0. Apple has modified and extended Mach to better meet OS X functional and p

Mach 3.0 was originally conceived as a simple, extensible, communications microkernel. It is capable of running as a stand-alone kernel, with other traditional o networking stacks running as user-mode servers.

However, in OS X, Mach is linked with other kernel components into a single kernel address space. This is primarily for performance; it is much faster to make a messages or do remote procedure calls (*RPC*) between separate tasks. This modular structure results in a more robust and extensible system than a monolithic l microkernel.

Thus in OS X, Mach is not primarily a communication hub between clients and servers. Instead, its value consists of its abstractions, its extensibility, and its flex

- object-based APIs with communication channels (for example, ports) as object references
- highly parallel execution, including preemptively scheduled threads and support for *SMP*
- a flexible scheduling framework, with support for real-time usage
- a complete set of *IPC* primitives, including messaging, *RPC*, synchronization, and notification
- support for large virtual address spaces, shared memory regions, and memory objects backed by persistent store
- proven extensibility and portability, for example across instruction set architectures and in distributed environments
- security and resource management as a fundamental principle of design; all resources are virtualized

Mach Kernel Abstractions

Mach provides a small set of abstractions that have been designed to be both simple and powerful. These are the main kernel abstractions:

- *Tasks*. The units of resource ownership; each task consists of a virtual address space, a *port right namespace*, and one or more *threads*. (Similar to a process.
- *Threads*. The units of CPU execution within a task.
- *Address space*. In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
- *Memory objects*. The internal units of memory management. Memory objects include named entries and regions; they are representations of potentially persi
- *Ports*. Secure, simplex communication channels, accessible only via send and receive capabilities (known as port rights).
- *IPC*. Message queues, remote procedure calls, notifications, semaphores, and lock sets.
- *Time*. Clocks, timers, and waiting.

Experiencing processes and threads

Hung-Wei Tseng

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

What will happen?

- What happens if we execute the following code?

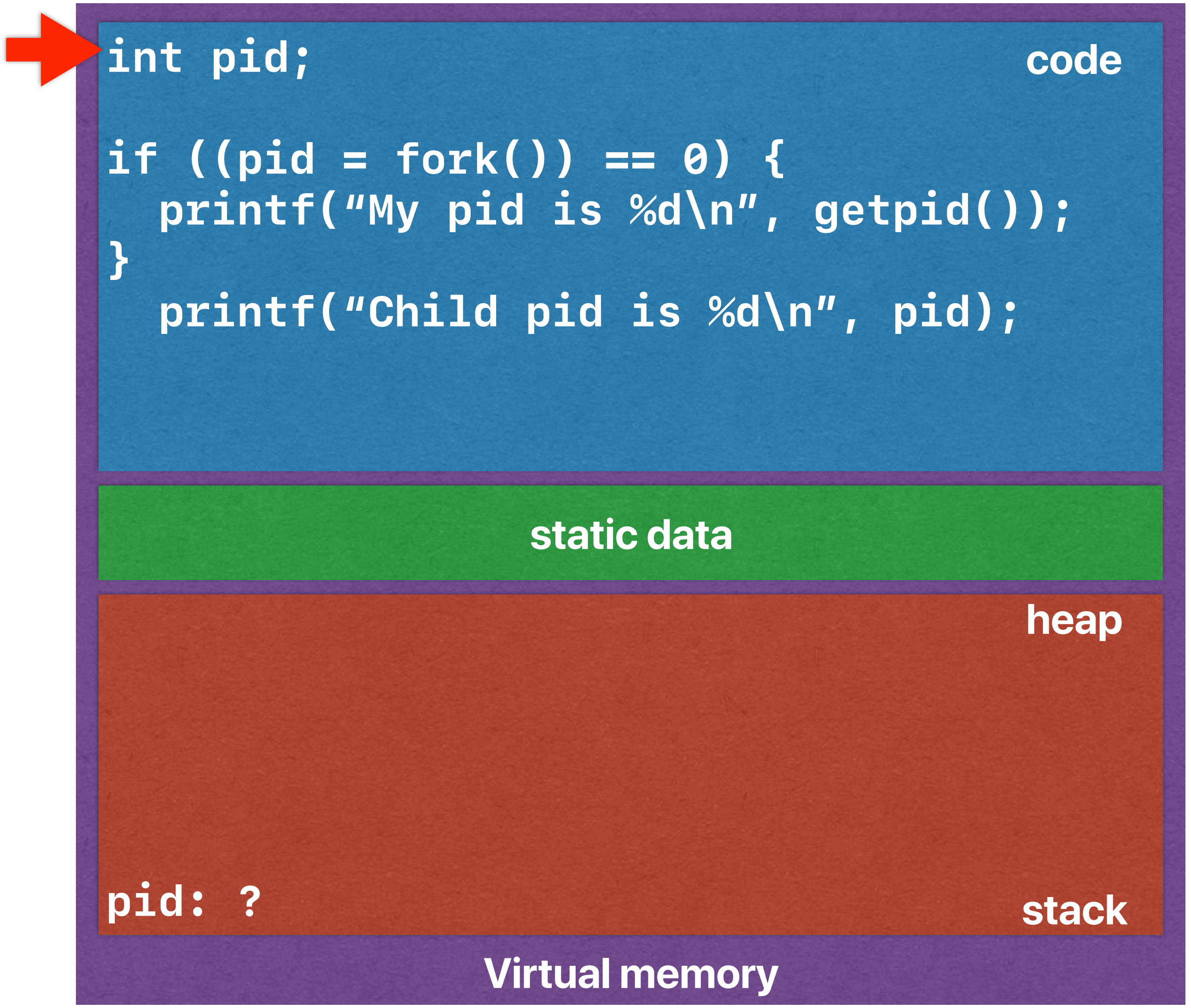
```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf (”My pid is %d\n", getpid());
        }
        printf (”Child pid is %d\n", pid);
        return 0;
}
```

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()

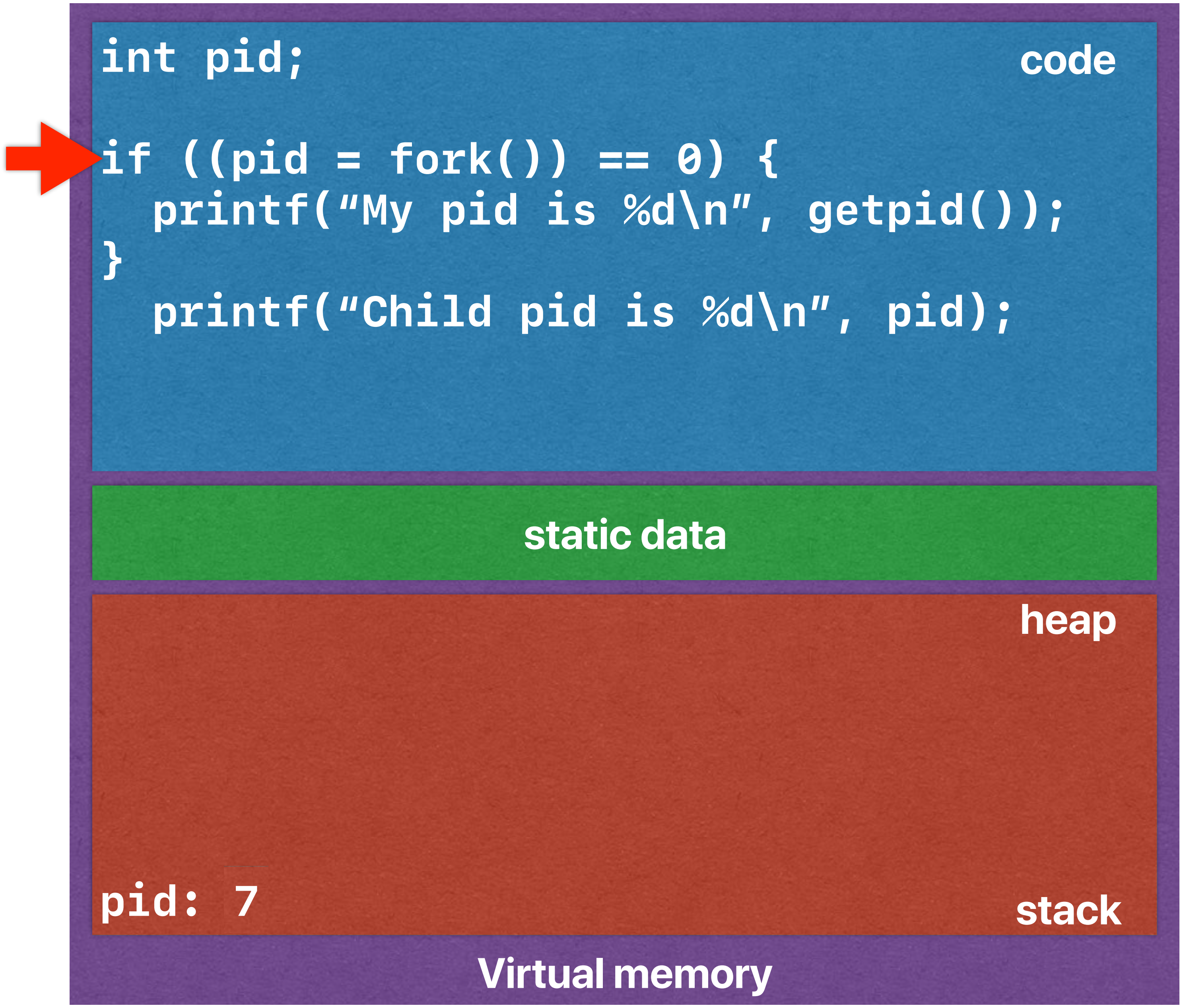


```
int pid;

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

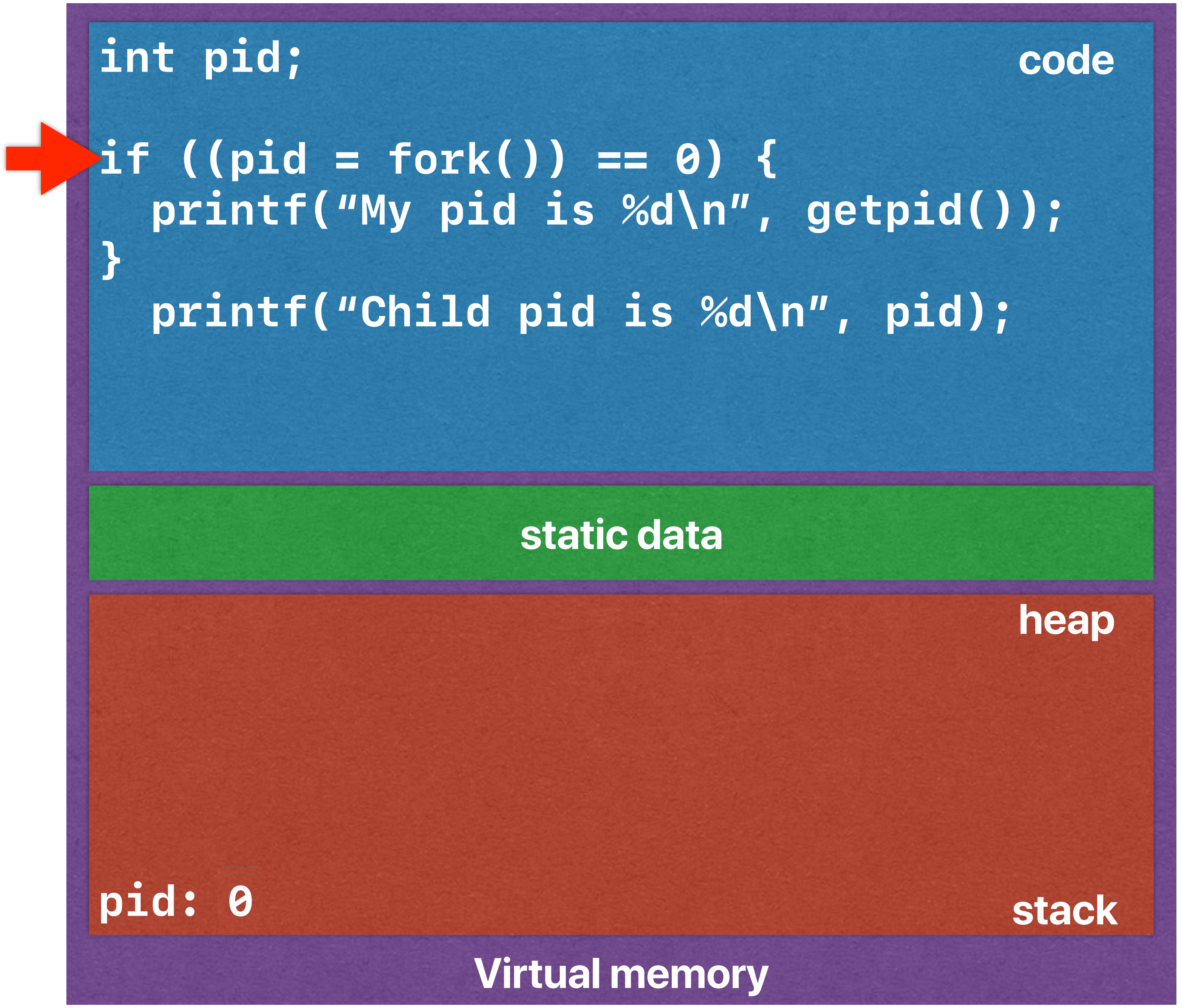
Assume
the parent’s PID is 2;
child’s PID is 7.

fork()



```
int pid;

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



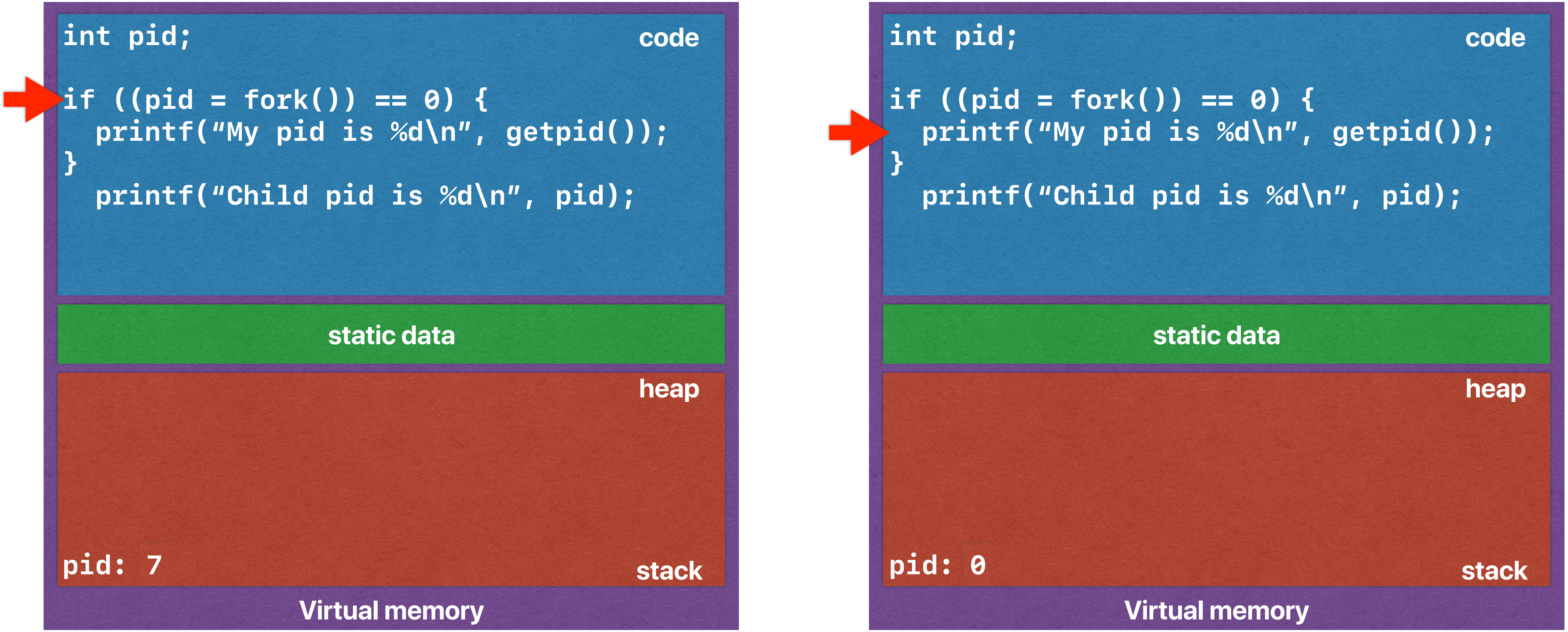
```
int pid;

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);
```

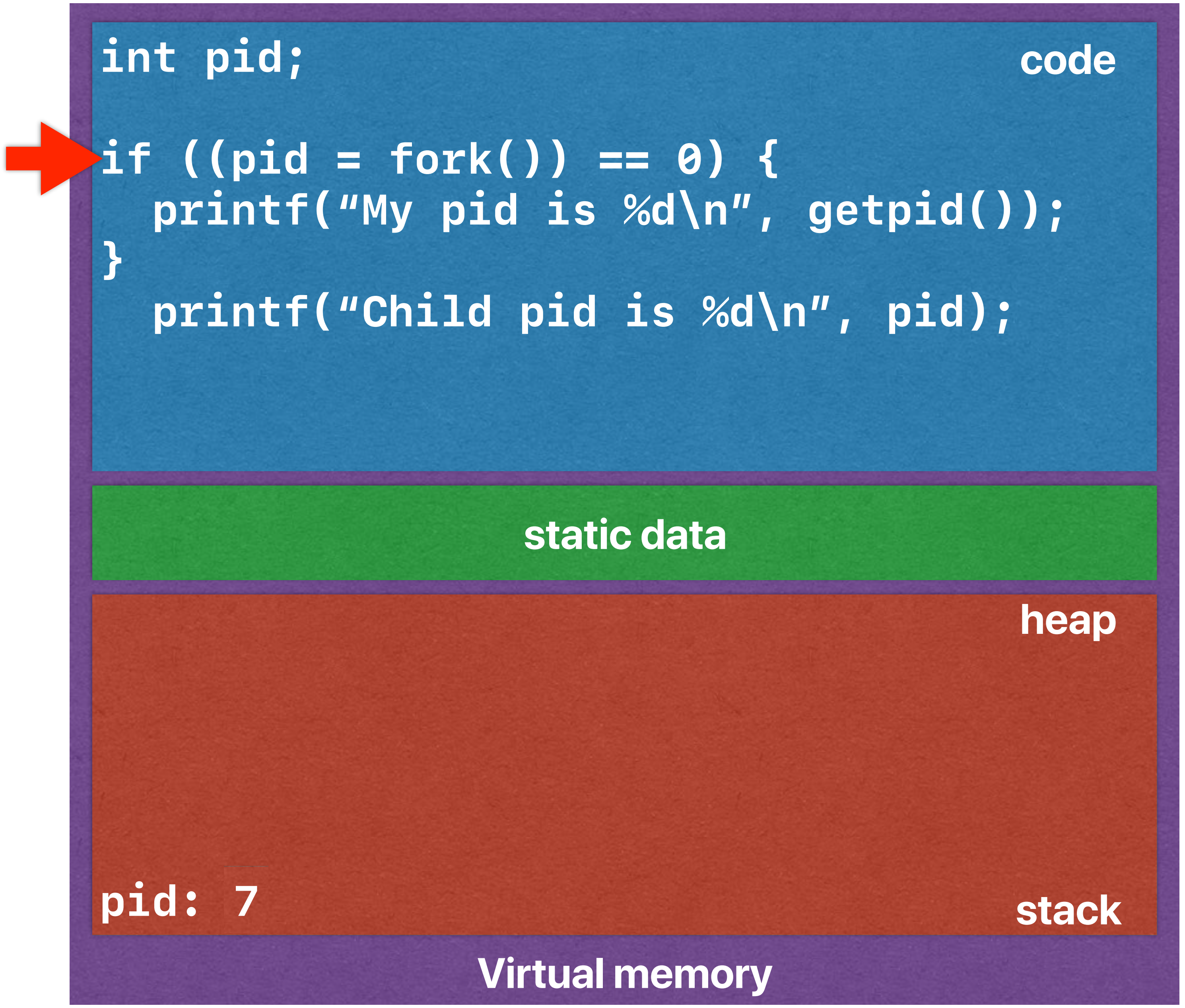
```
int pid;

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



code

```
int pid;

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

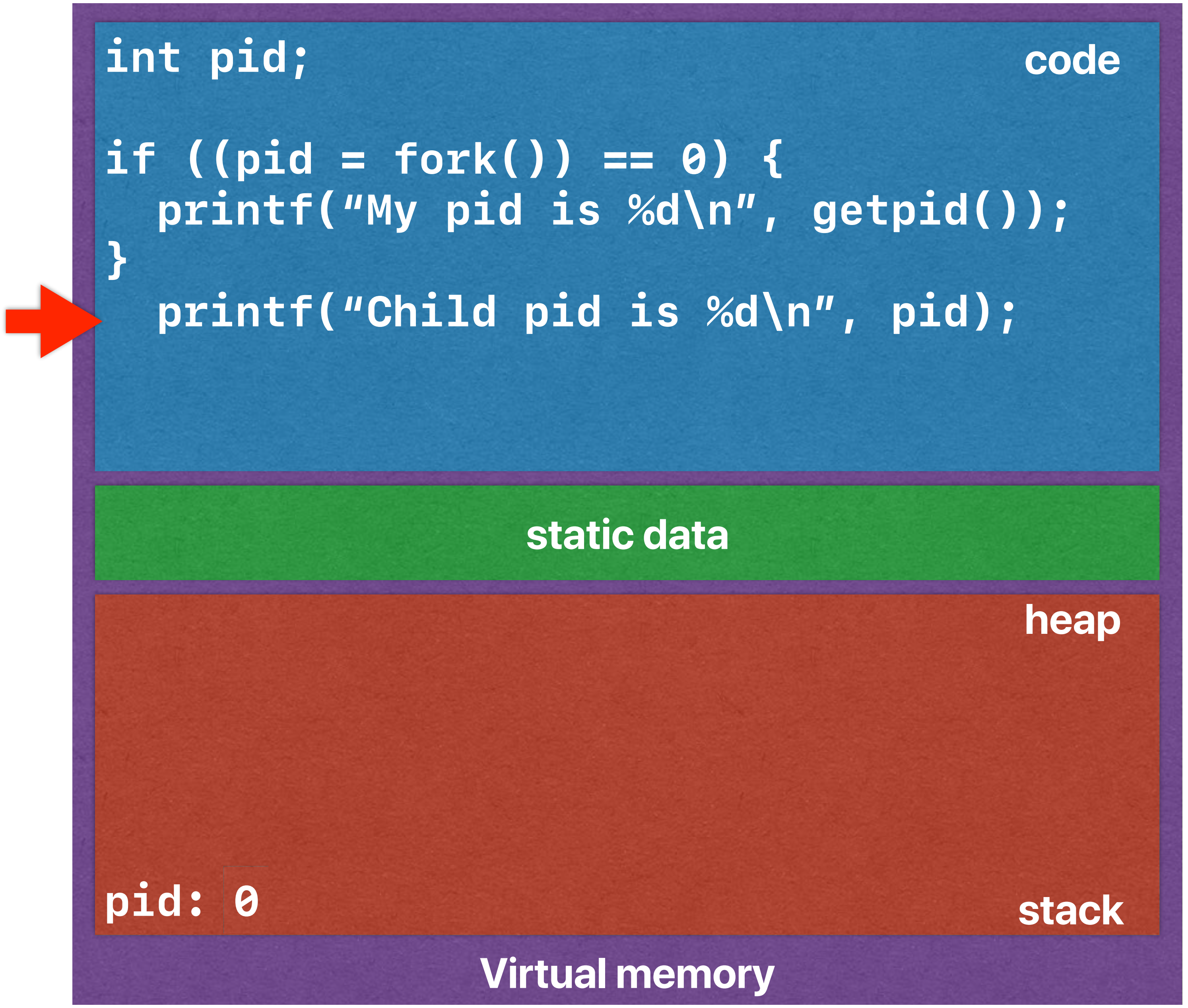
static data

heap

pid: 7

stack

Virtual memory



code

```
int pid;

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

static data

heap

pid: 0

stack

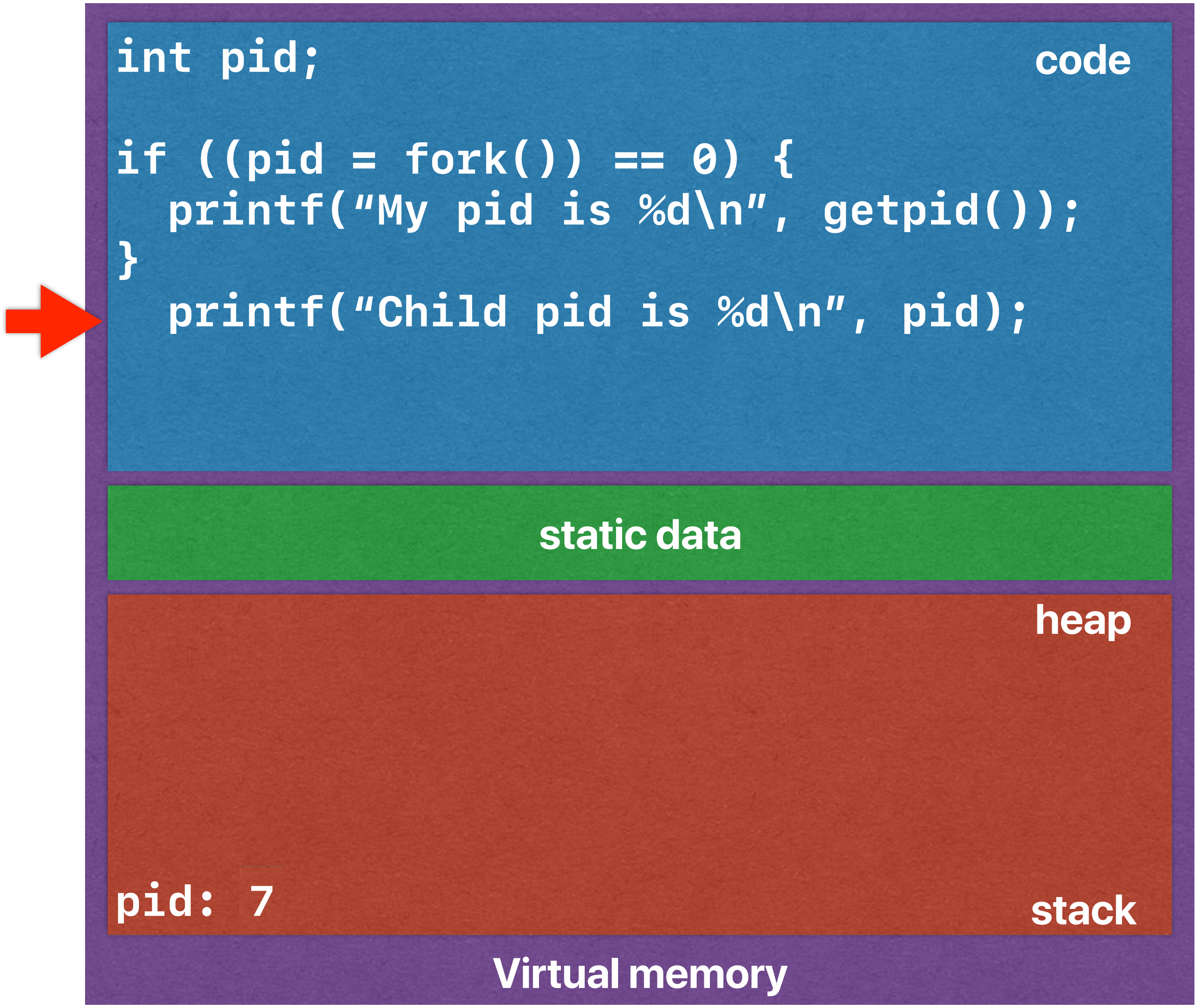
Virtual memory

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

fork()

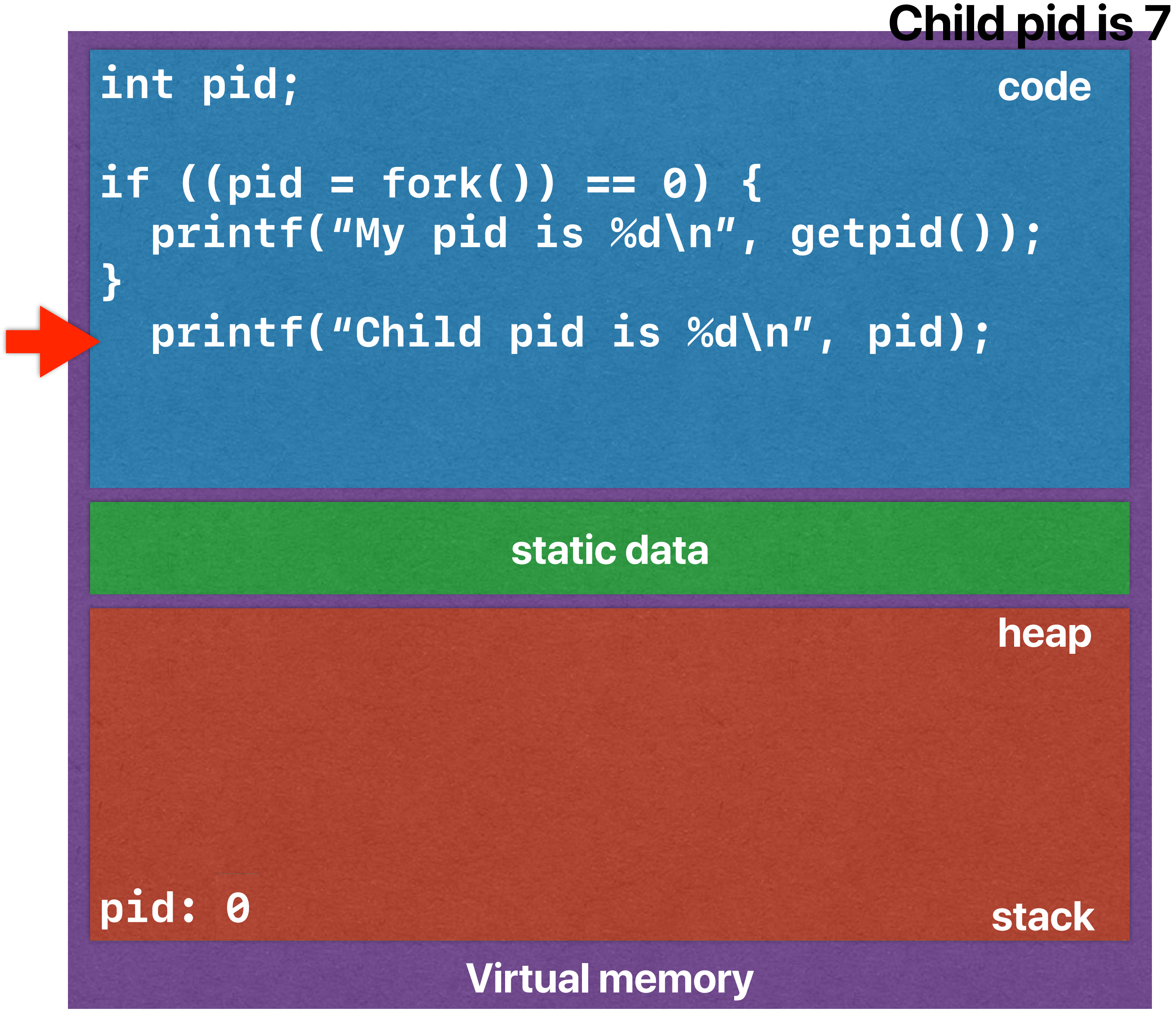
Output:

```
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

Announcement

- Reading quizzes due next Tuesday
  - We do announce after each lecture — you should be aware of that
  - Please also check course webpage for schedule — iLearn doesn’t always generate announcements
- Preview v.s. release slides
  - Preview: uploaded right before the lecture — no PI questions, for note-taking
  - Release slides: upload after the lecture — with complete content
- Check your clicker grades in iLearn
- Podcast is up. Access through iLearn