Design philosophy of operating systems (III)

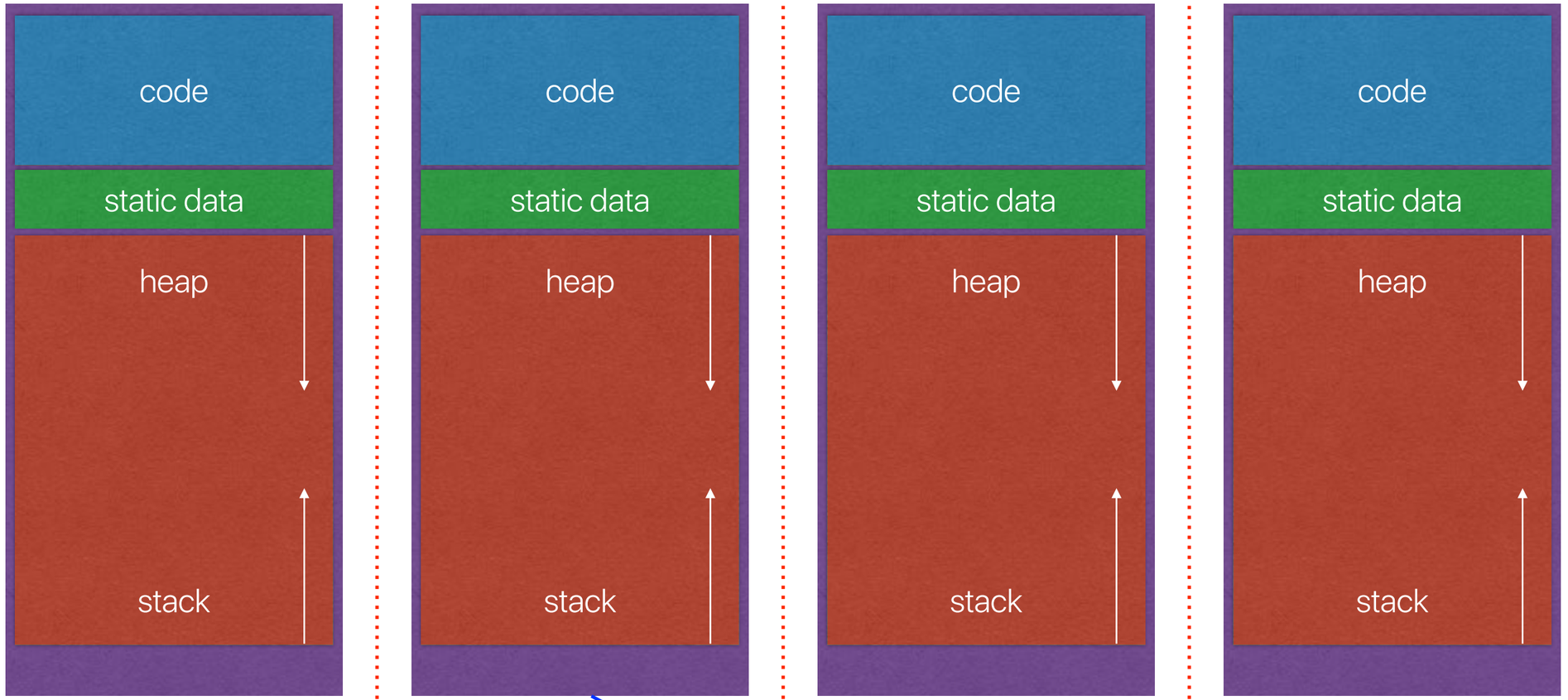
Hung-Wei Tseng

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

Processor

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
esca102 [/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

The basic process API of UNIX

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

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

UNIX’s interface of managing processes

If we add an exit …

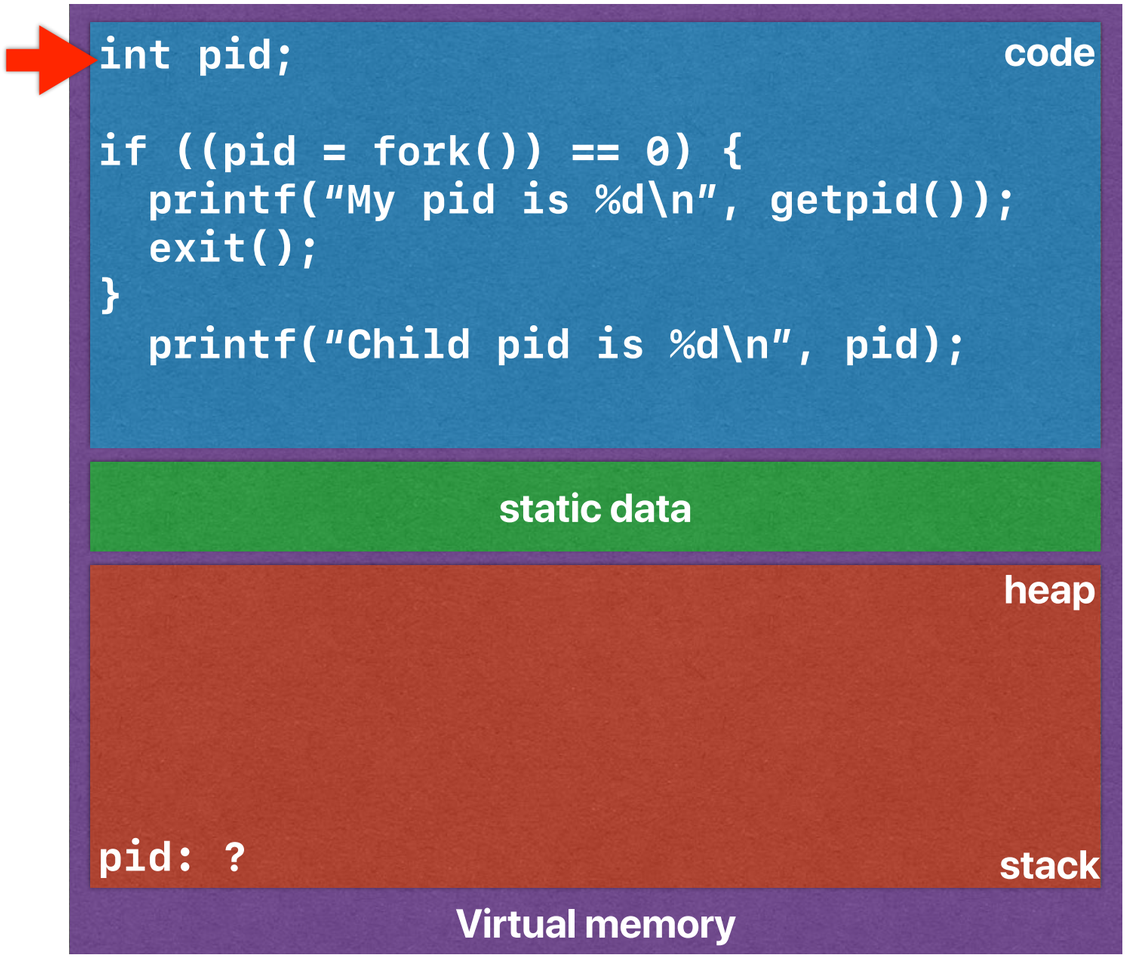
- What happens if we add an exit?

```
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;
}
```

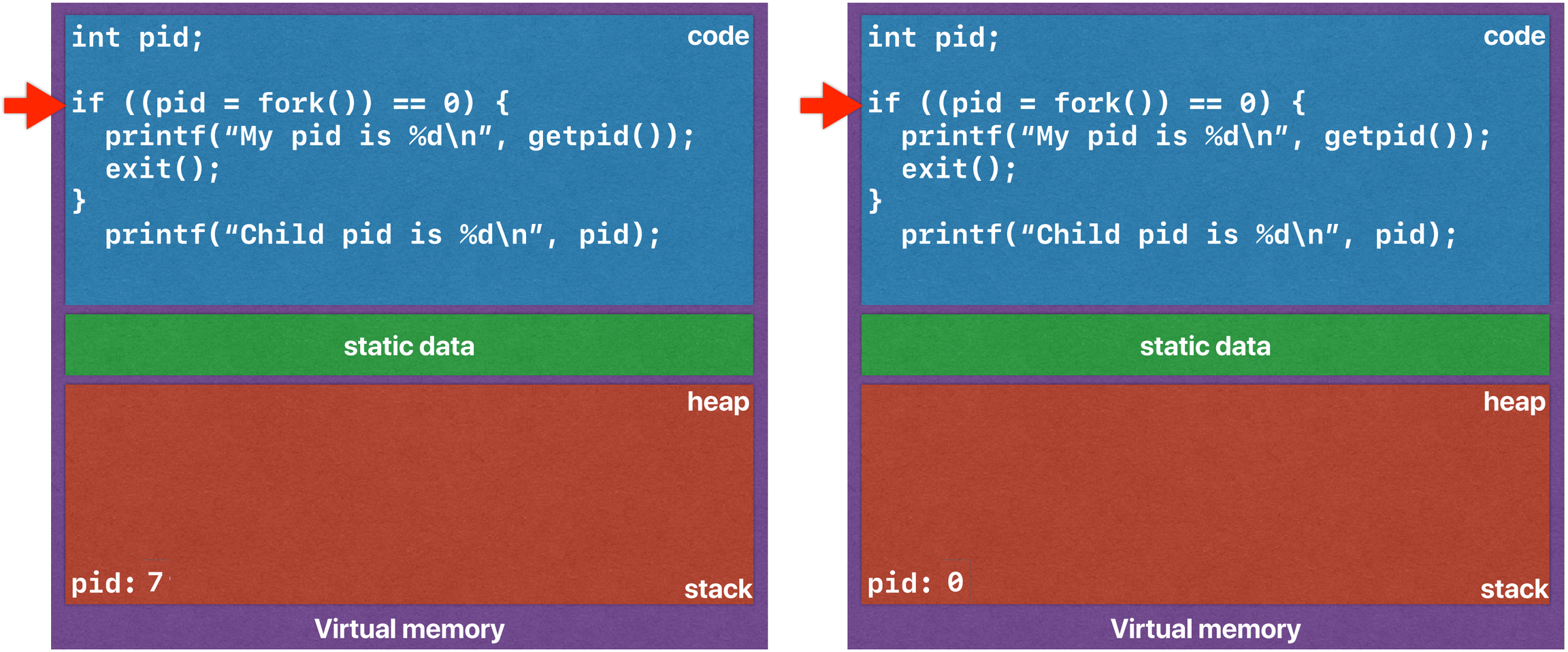
Assume
the parent's PID is 2;
child's PID is 7.

	# 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()

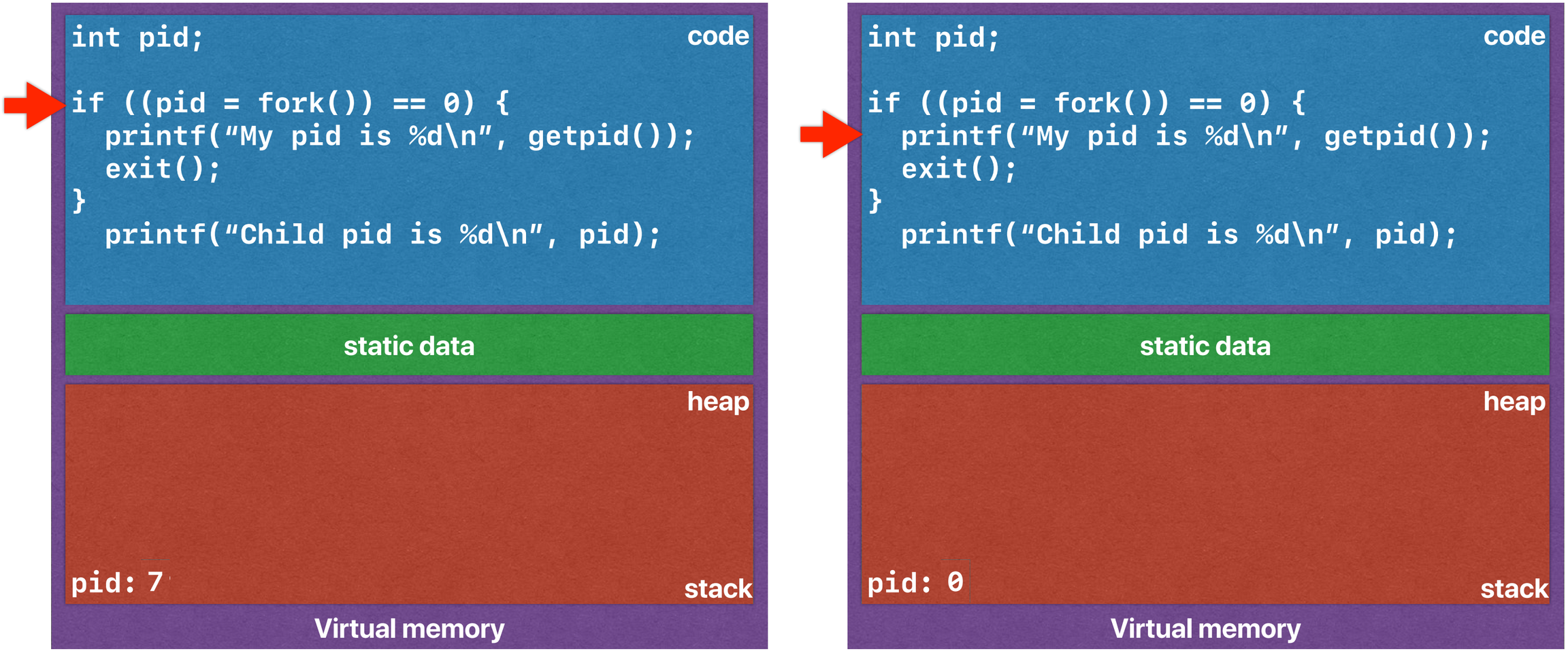


fork() and exit()



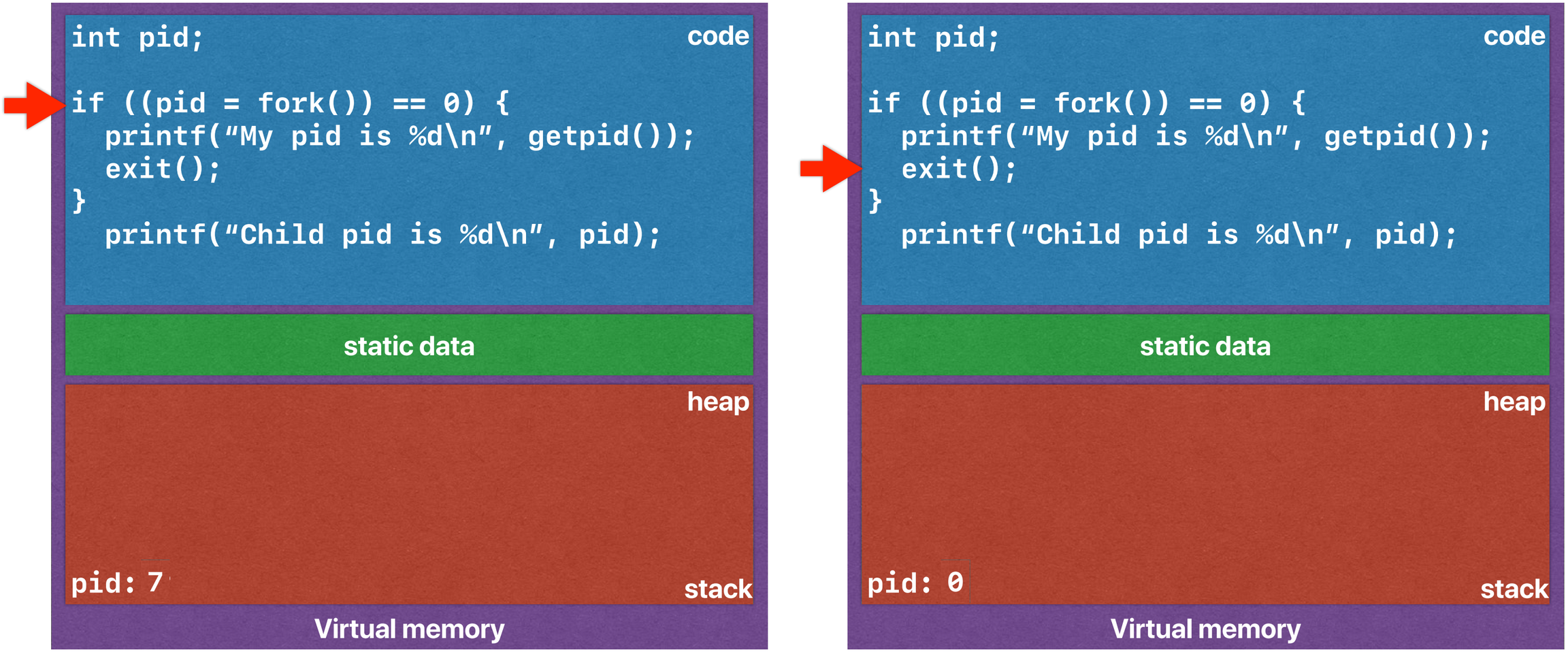
fork() and exit()

Output:
My pid is 7



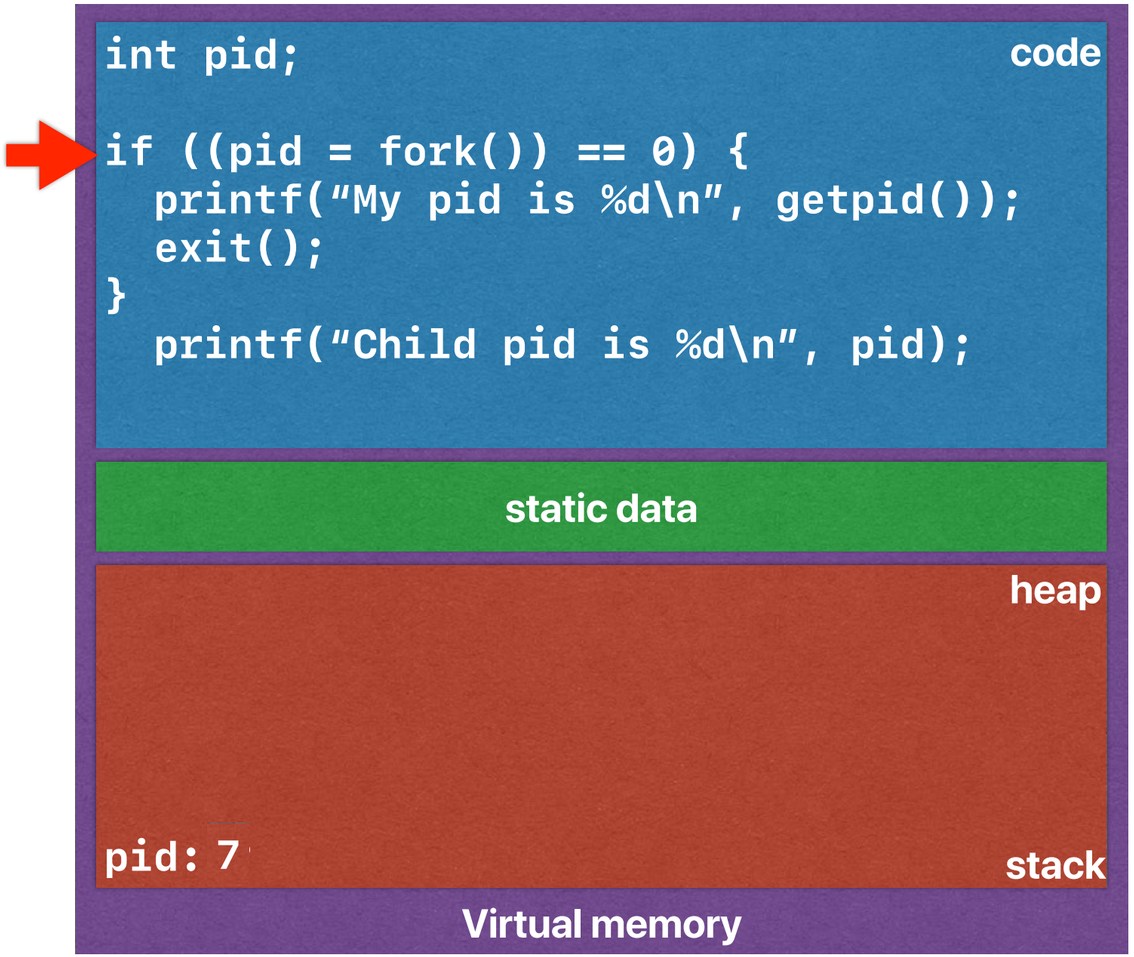
fork() and exit()

Output:
My pid is 7



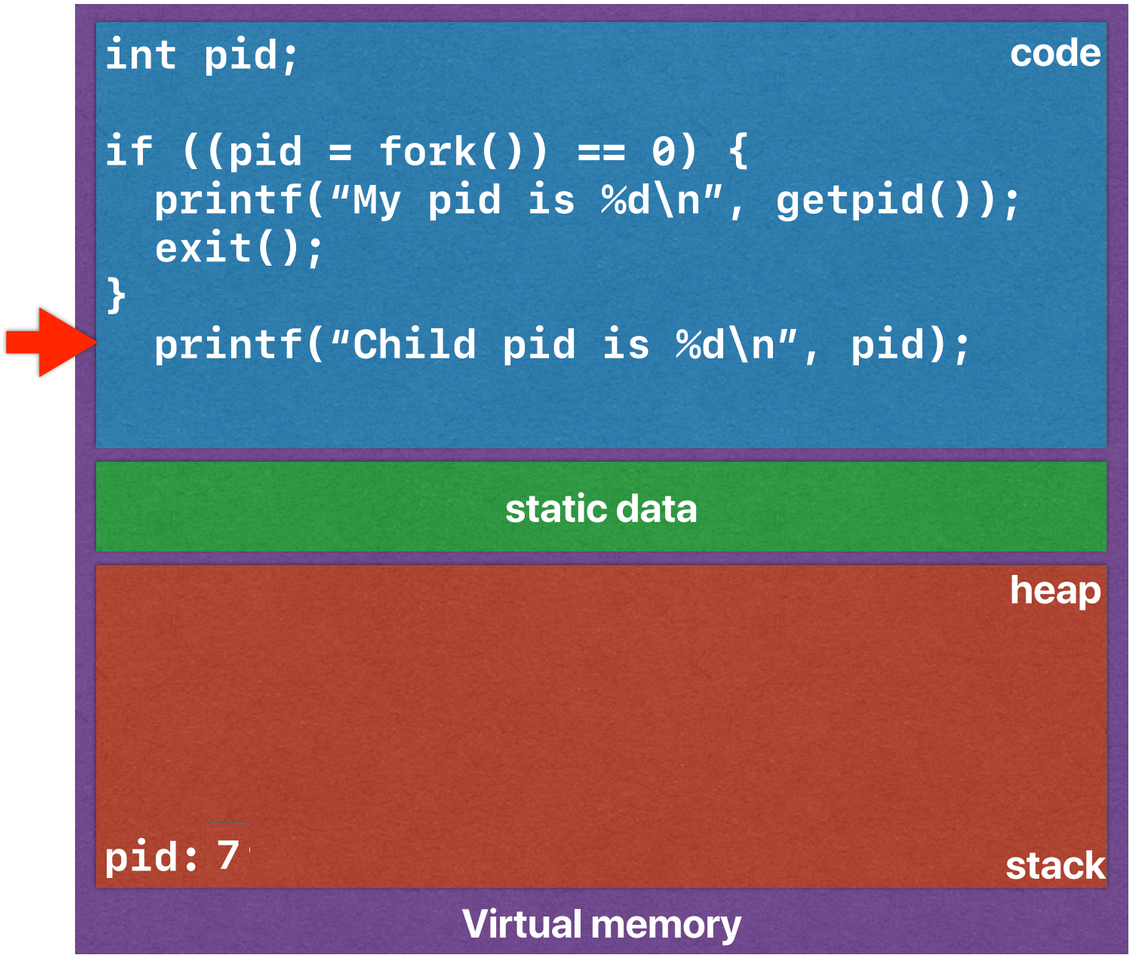
fork() and exit()

Output:
My pid is 7



fork() and exit()

Output:
My pid is 7
Child pid is 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

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");
fork();
printf("oink\n");
fork();
printf("baa\n");
```

2x

4x

8x

How many animal noises will be printed?

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

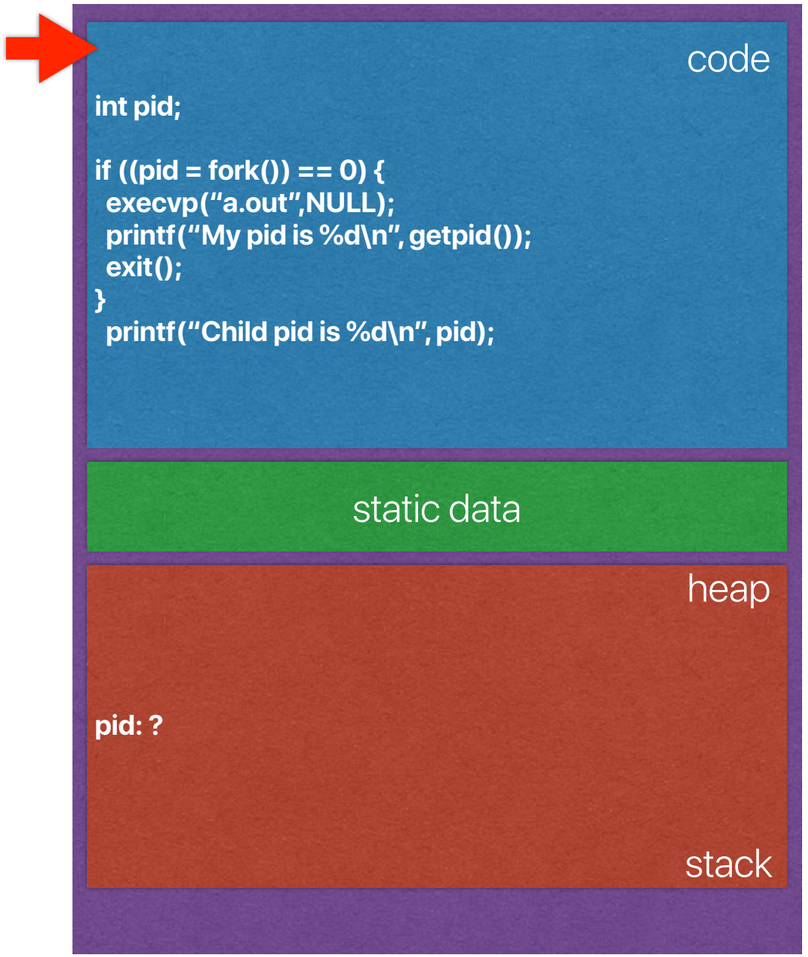
Starting a new program with `execvp()`

- `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

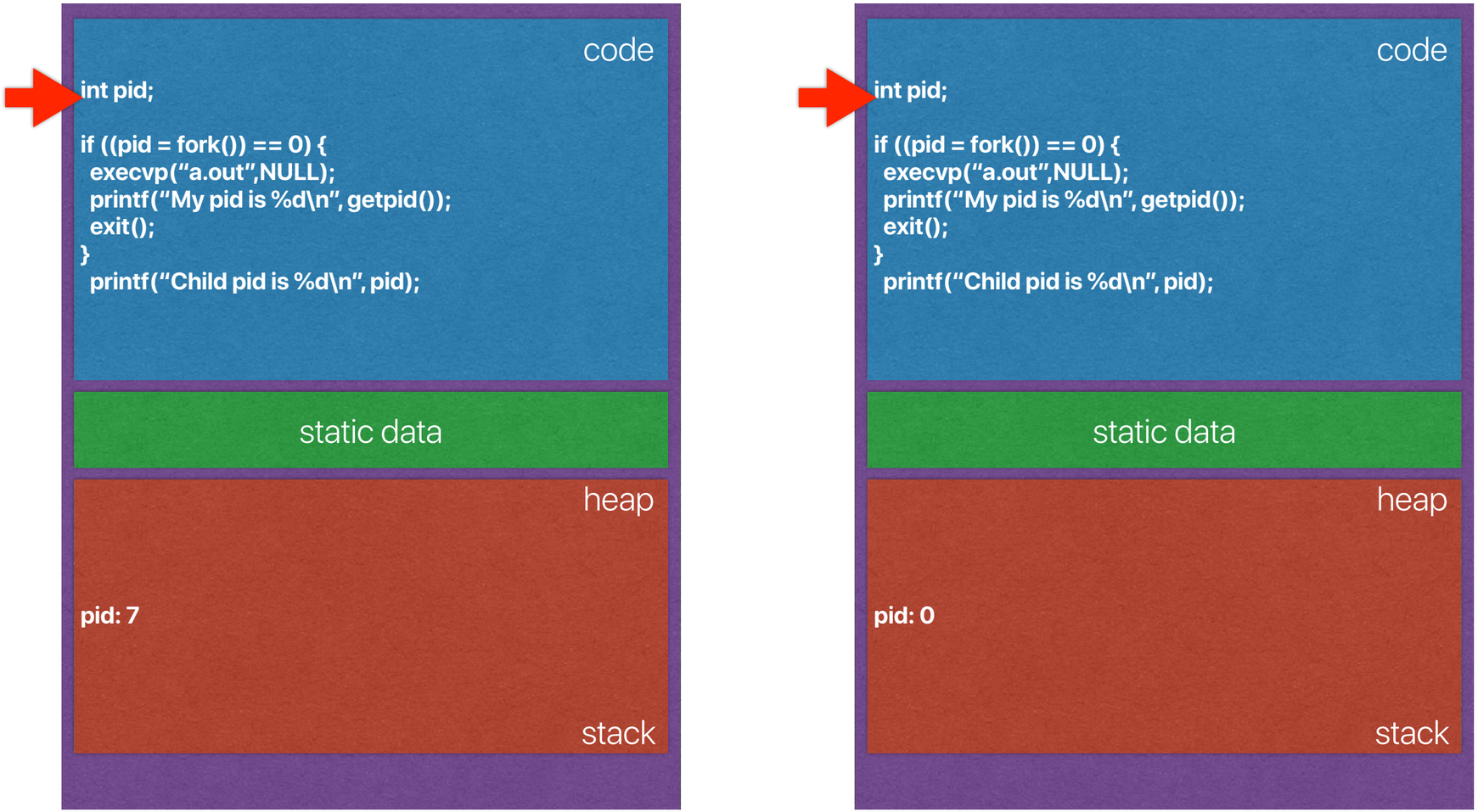
Why separate `fork()` and `exec()`

- 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()

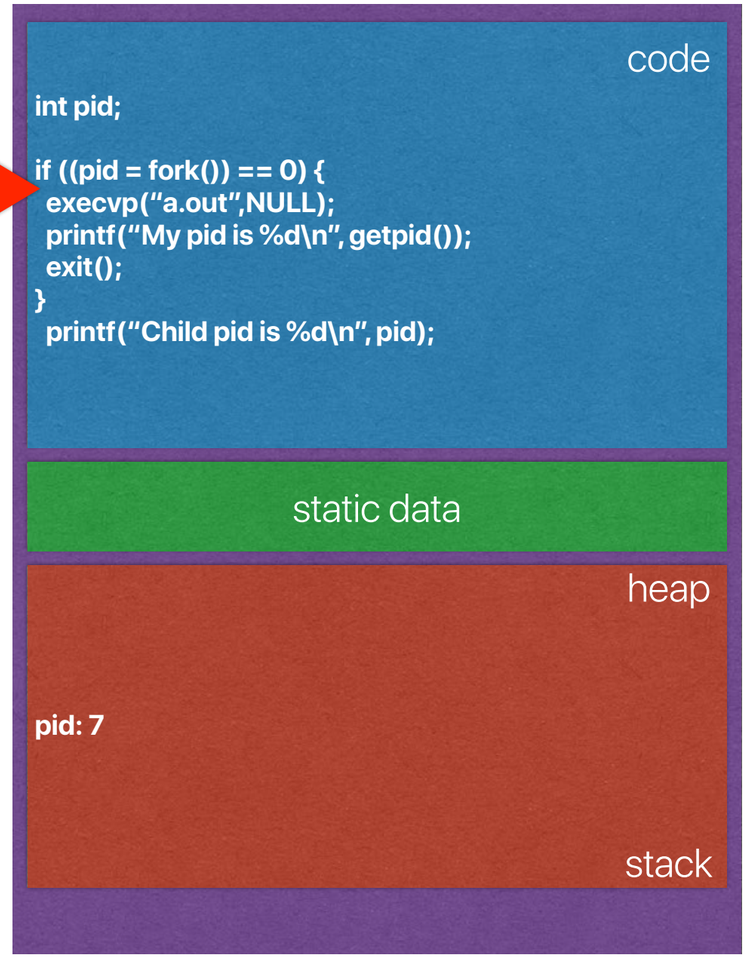

code
int pid;
if ((pid = fork()) == 0) {
execvp("a.out",NULL);
printf("My pid is %d\n", getpid());
exit();
}
printf("Child pid is %d\n", pid);
static data
heap
pid: ?
stack

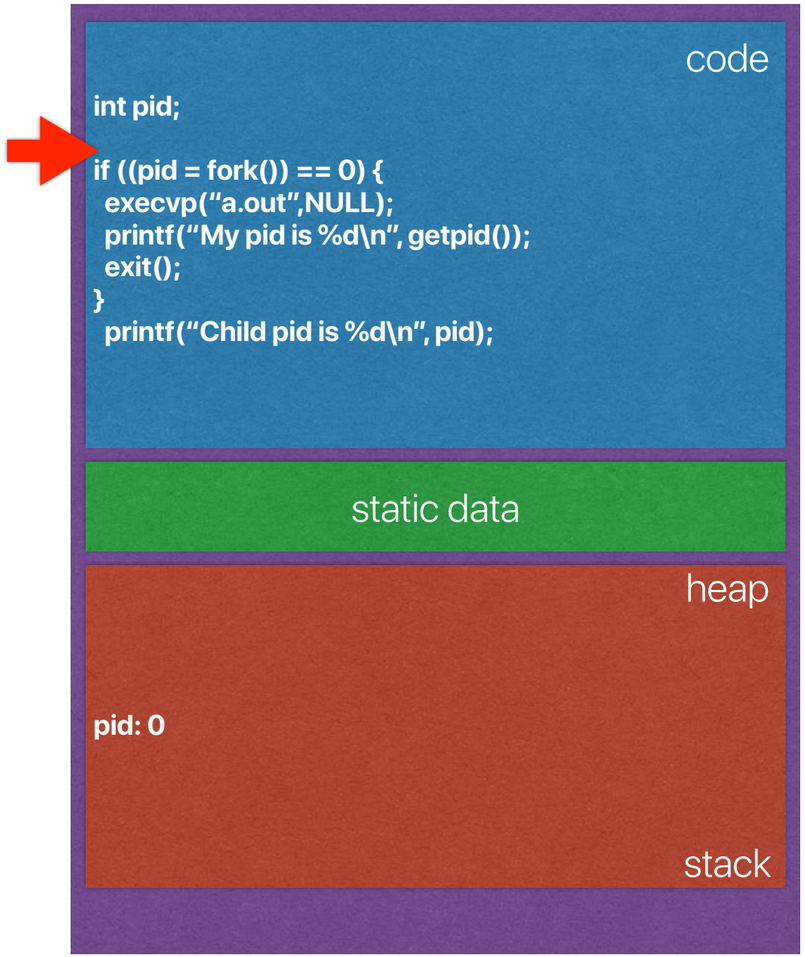
exec()


code
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);
static data
heap
pid: 7
stack
code
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);
static data
heap
pid: 0
stack

exec( )

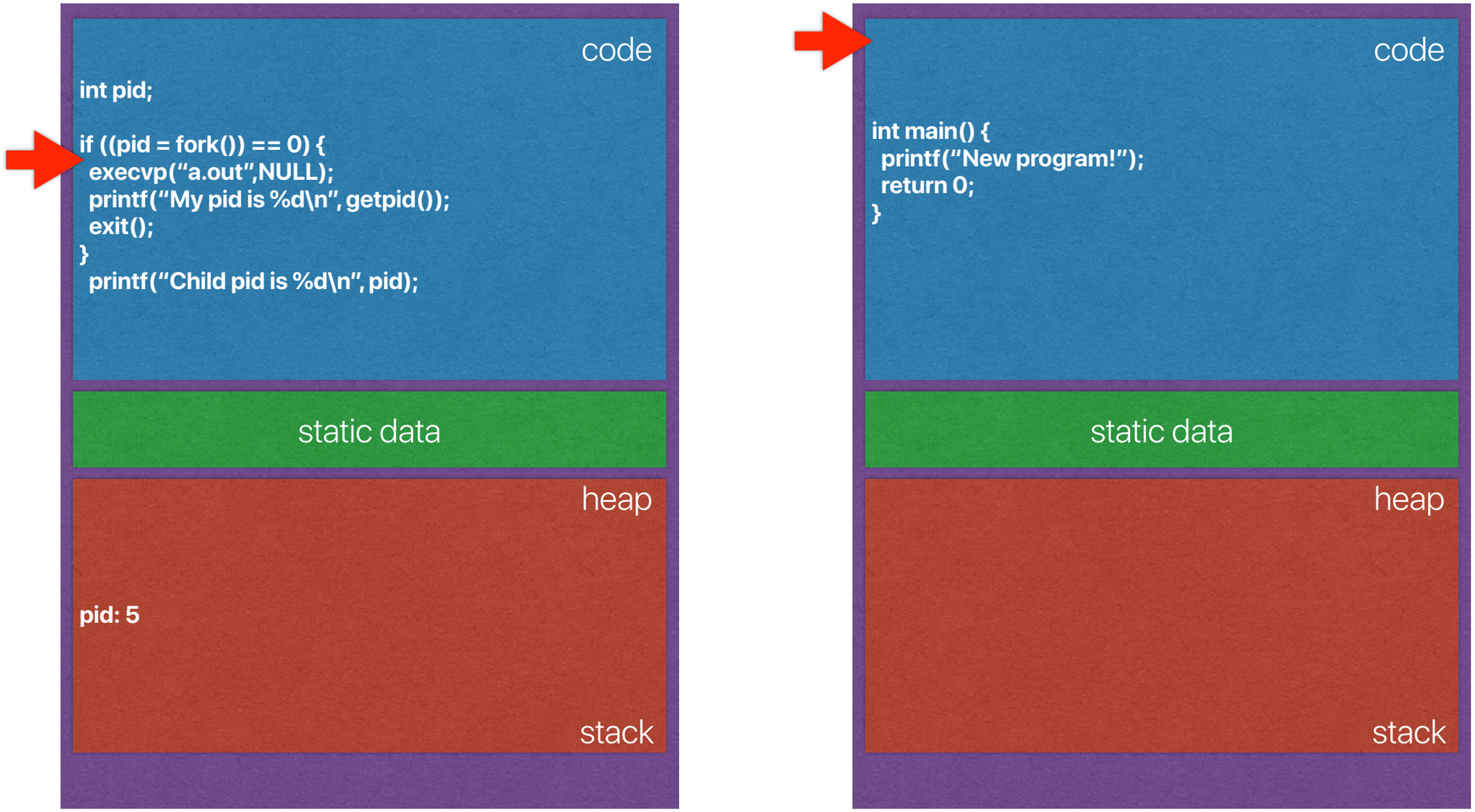
Output:
Child pid is 7





exec()

Output:
Child pid is 7
New program!



Poll close in 1:30

What's in the kernel?

- How many of the following UNIX features/functions are 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

Poll close in 1:30

What's in the kernel?

- How many of the following UNIX features/functions are 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

Let’s write our own shells

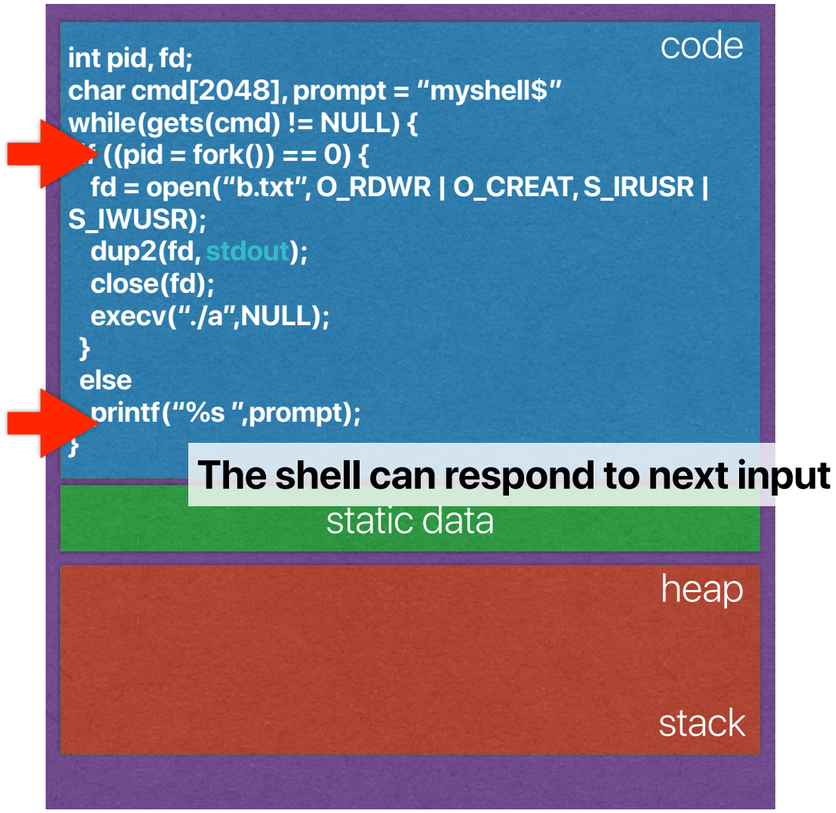
How to implement redirection in shell

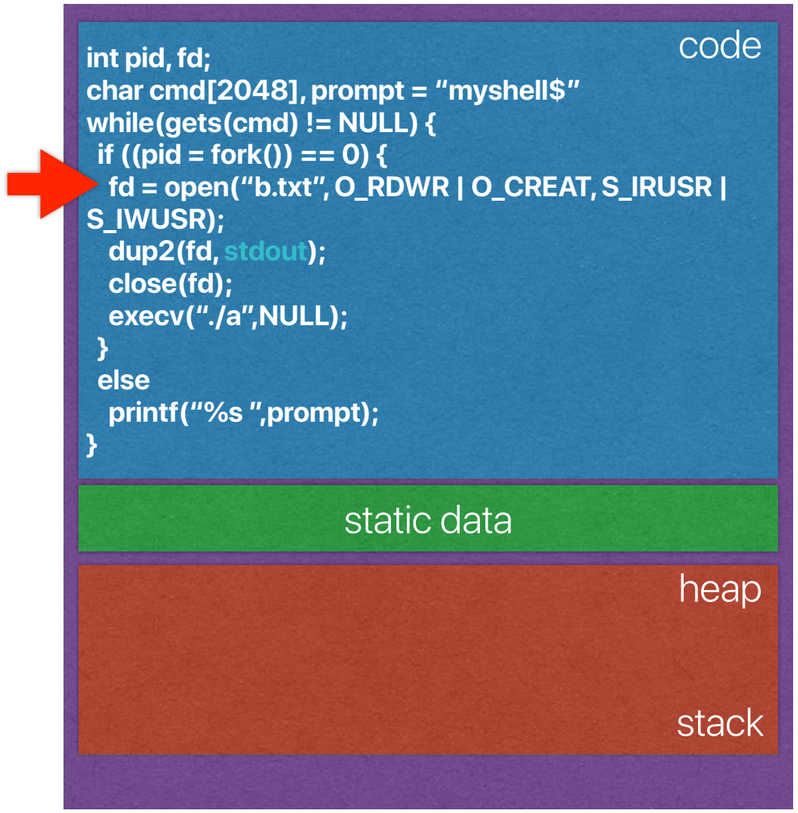
- Say, we want to do ./a > b.txt
- fork
- The forked code opens b.txt
- The forked code dup the file descriptor
- The forked code assigns b.txt to stdin/stdout
- The forked code closes b.txt
- exec(“./a”, NULL)

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()`



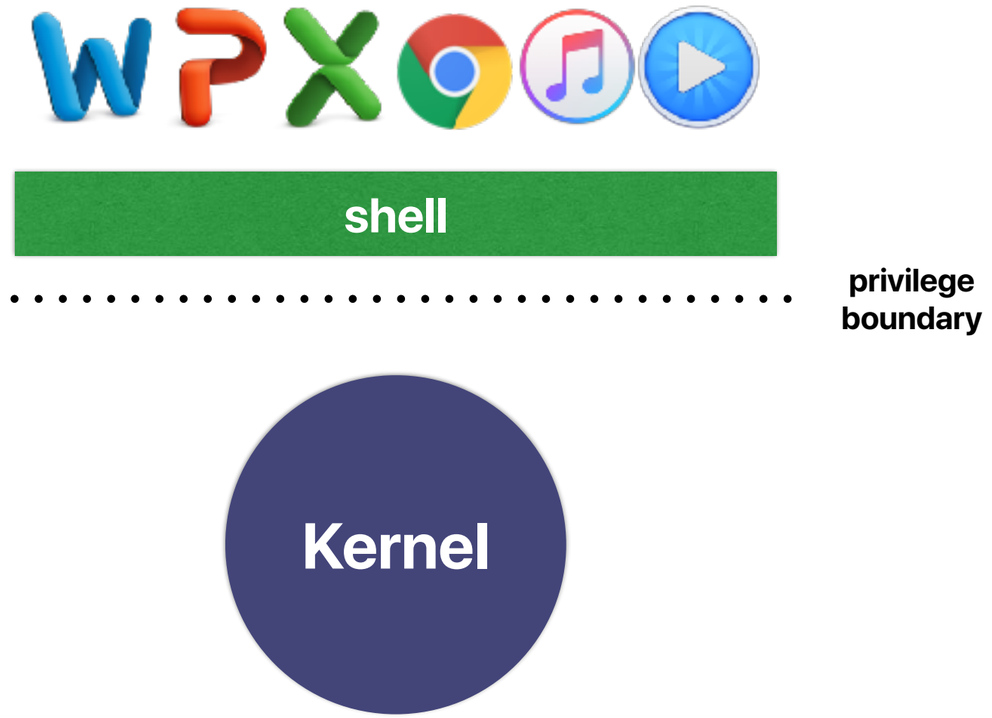


wait()

- `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`

What's in the kernel?

- How many of the following UNIX features/functions are 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



Shell

- A user program provides an interactive UI
- Interprets user command into OS functions
- Basic semantics:
  command argument_1 argument_2 …
- Advanced semantics
  - Redirection
    - >
    - <
  - Pipe
    - |
  - Multitasking
    - &

The impact of UNIX

- Clean abstraction
- 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.

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

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 3.2GHz intel Core i5-6500 Processor
  - Process fork+exit: 53.5437 microseconds
  - More than 16K cycles

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

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.

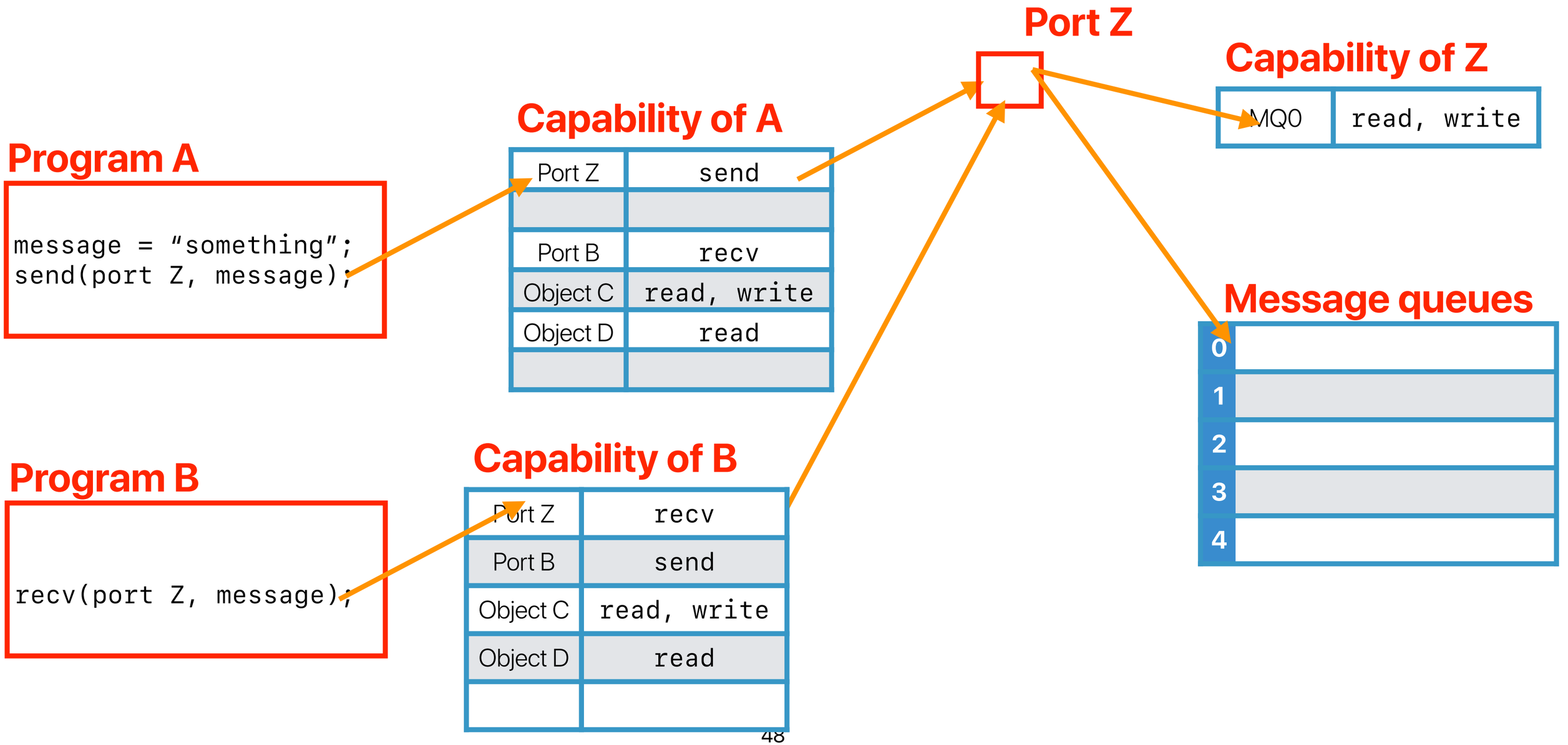
Interprocess communication

- UNIX provides a variety of mechanisms
  - Pipes
  - Pty's
  - Signals
  - Sockets
- No protection
- No consistency
- Location dependent

Ports/Messages

- Port is an abstraction of:
  - Message queues
  - Capability
- What do ports/messages promote?
  - Location independence — everything is communicating with ports/messages, no matter where it is

Ports/Messages



What is capability? — Hydra

- An access control list associated with an object
- Contains the following:
  - A reference to an object
  - A list of access rights
- Whenever an operation is attempted:
  - The requester supplies a capability of referencing the requesting object — like presenting the boarding pass
  - The OS kernel examines the access rights
    - Type-independant rights
    - Type-dependent rights

Poll close in 1:30

Tasks/Processes and threads

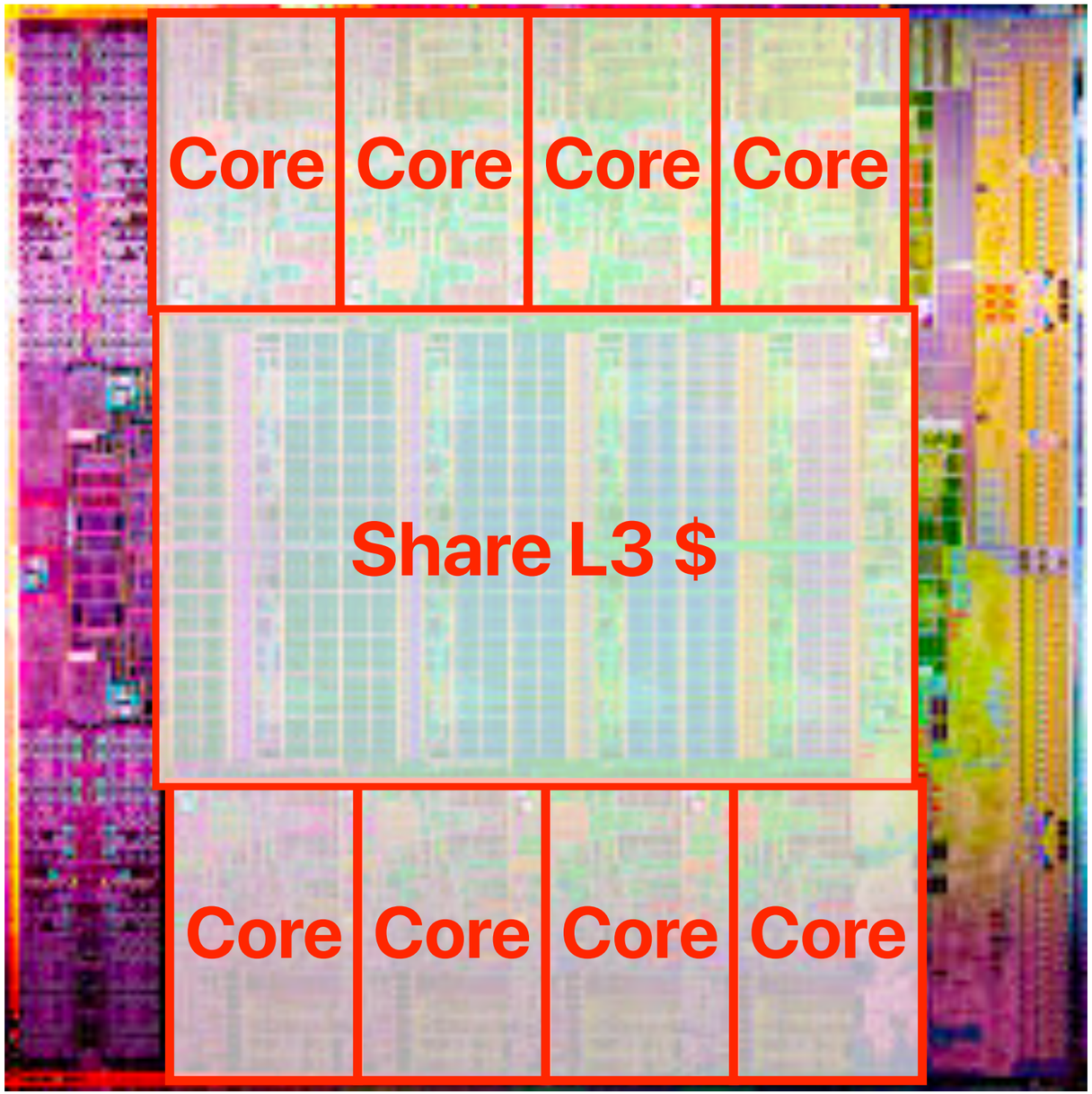
- How many of the following regarding the comparison of parallelizing computation tasks using processes and threads is/are correct?
  - ① The context switch and creation overhead of processes is higher
  - ② The overhead of exchanging data among different computing tasks for the same applications is higher in process model
  - ③ The demand of memory usage is higher when using processes
  - ④ The security and isolation guarantees are better achieved using processes
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Poll close in 1:30

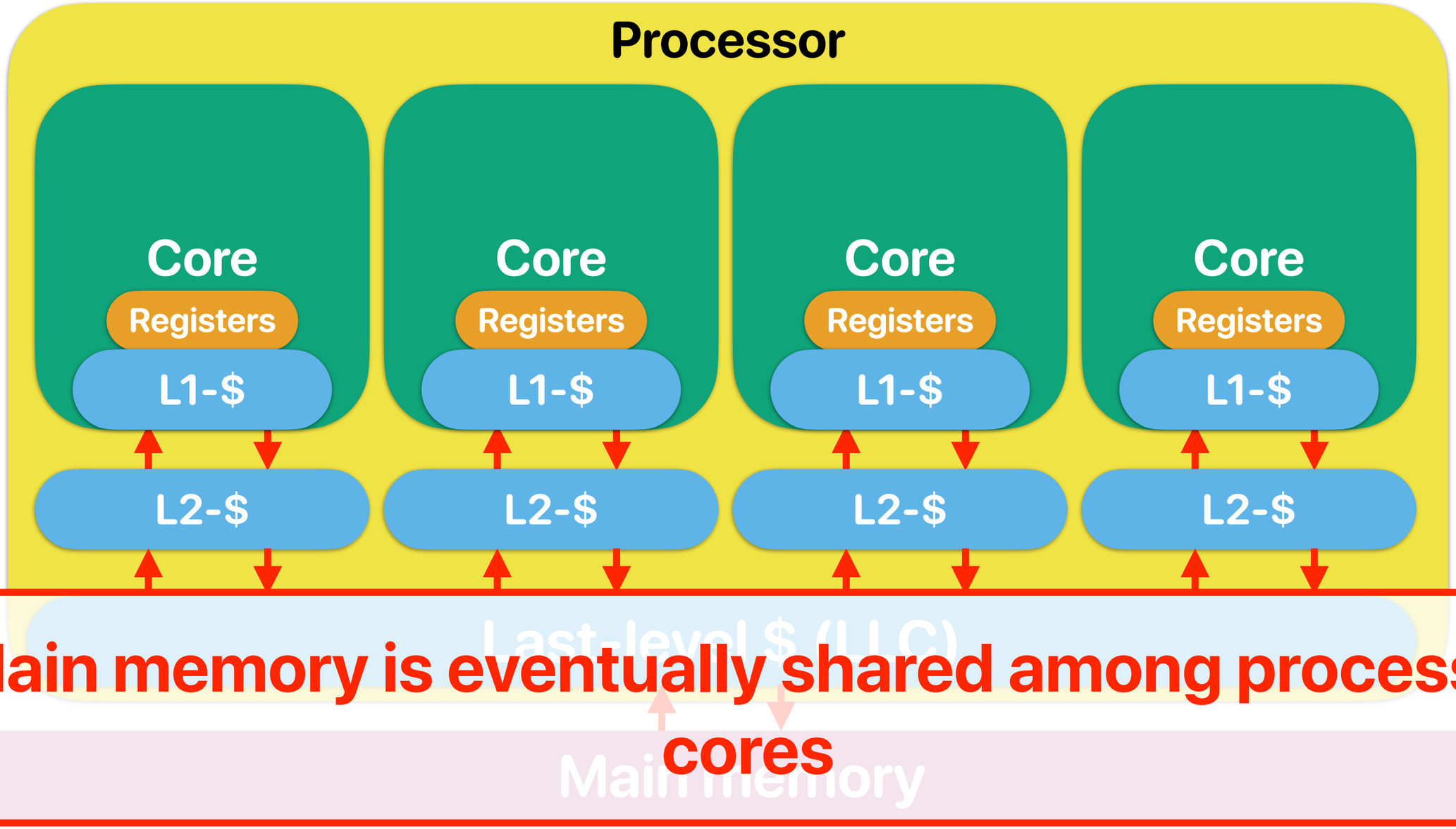
Tasks/Processes and threads

- 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
  - ② The overhead of exchanging data among different computing tasks for the same applications is higher in process model
  - ③ The demand of memory usage is higher when using processes
  - ④ The security and isolation guarantees are better achieved using processes
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Intel Sandy Bridge

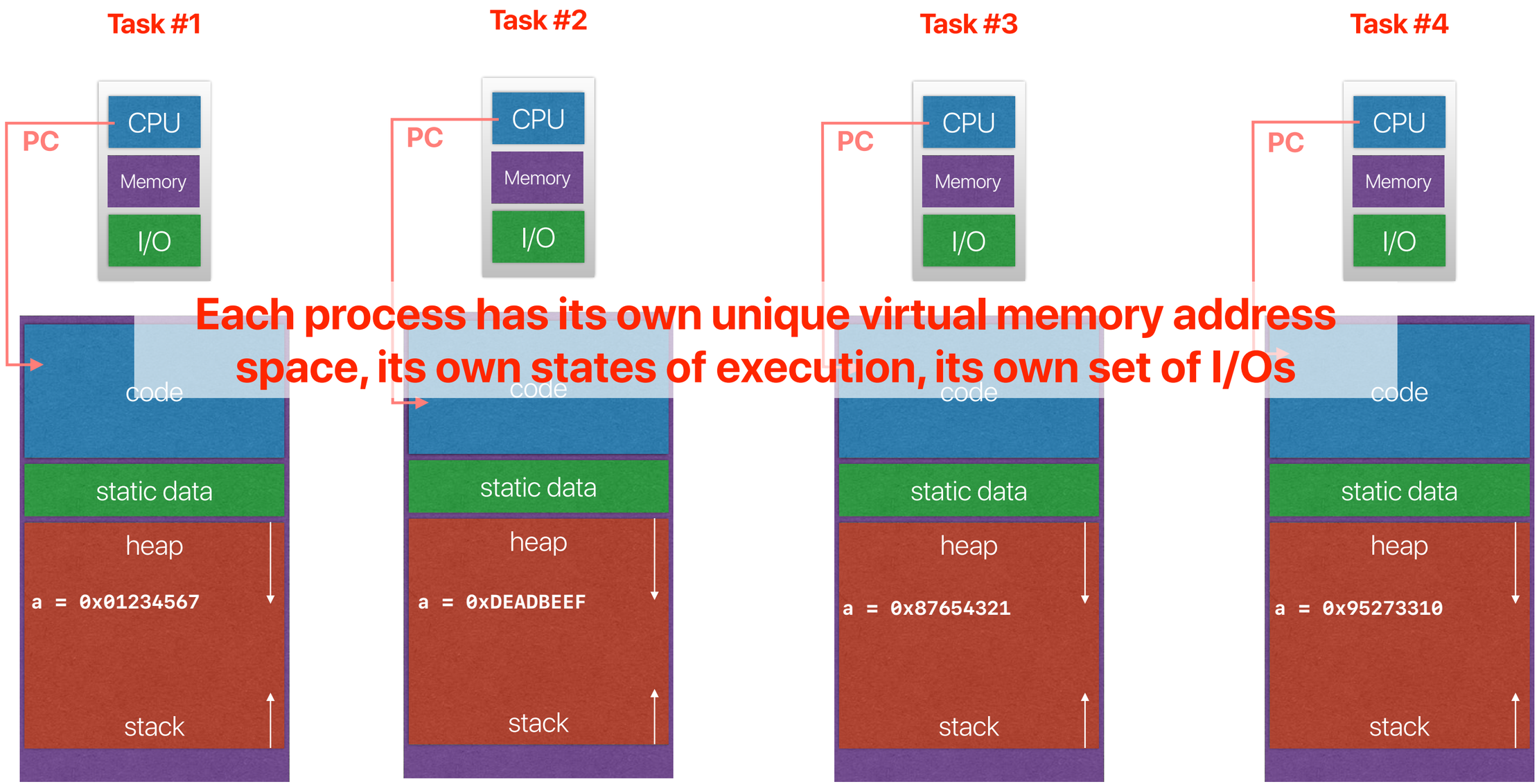


Concept of chip multiprocessors



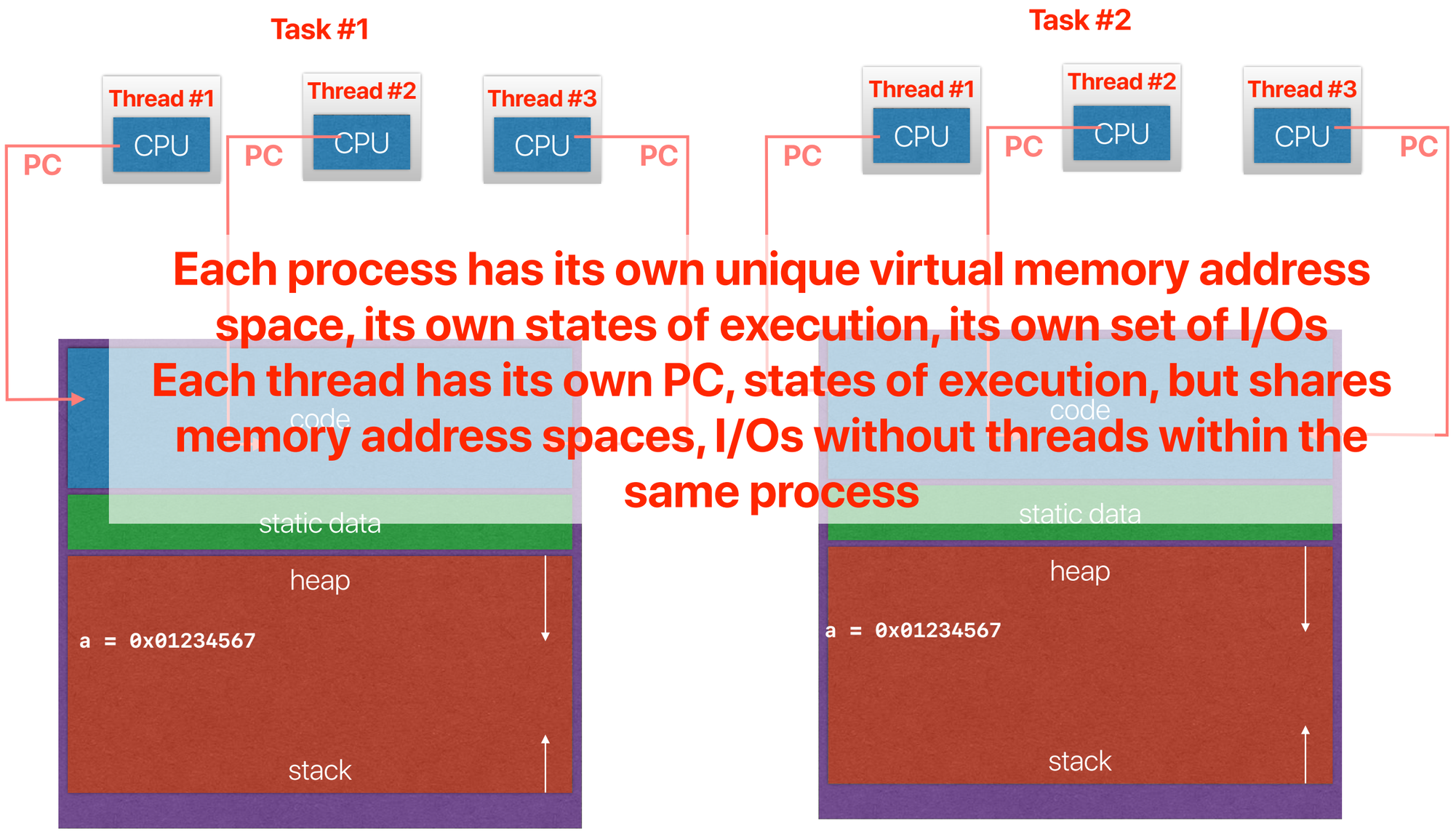
Main memory is eventually shared among processor cores

Tasks/processes



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

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

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
  - ② The overhead of exchanging data among different computing tasks for the same applications is higher in process model
  - ③ The demand of memory usage is higher when using processes
  - ④ The security and isolation guarantees are better achieved using processes
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

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

— you cannot directly share data without leveraging other mechanisms

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

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

Announcement

- Reading quizzes due next Tuesday
  - Welcome new friends! — will drop a total of 6 reading quizzes for the quarter
  - Attendance count as 4 reading quizzes
  - We plan to have a total of 11 reading quizzes
- Office Hour links are inside Google Calendar events
  - https://calendar.google.com/calendar/u/0/r?cid=ucr.edu_b8u6dvkretn6kq6igunlc6bldg@group.calendar.google.com
  - Different links from lecture ones
  - We cannot share through any public channels so that we can better avoid Zoom bombing
- We will make both midterm and final exams online this quarter
  - Avoid the uncertainty of COVID-19
  - Avoid high-density in the classroom (only sits 60 and we have 59 for now) during examines

Computer
Science &
Engineering

202

つづく