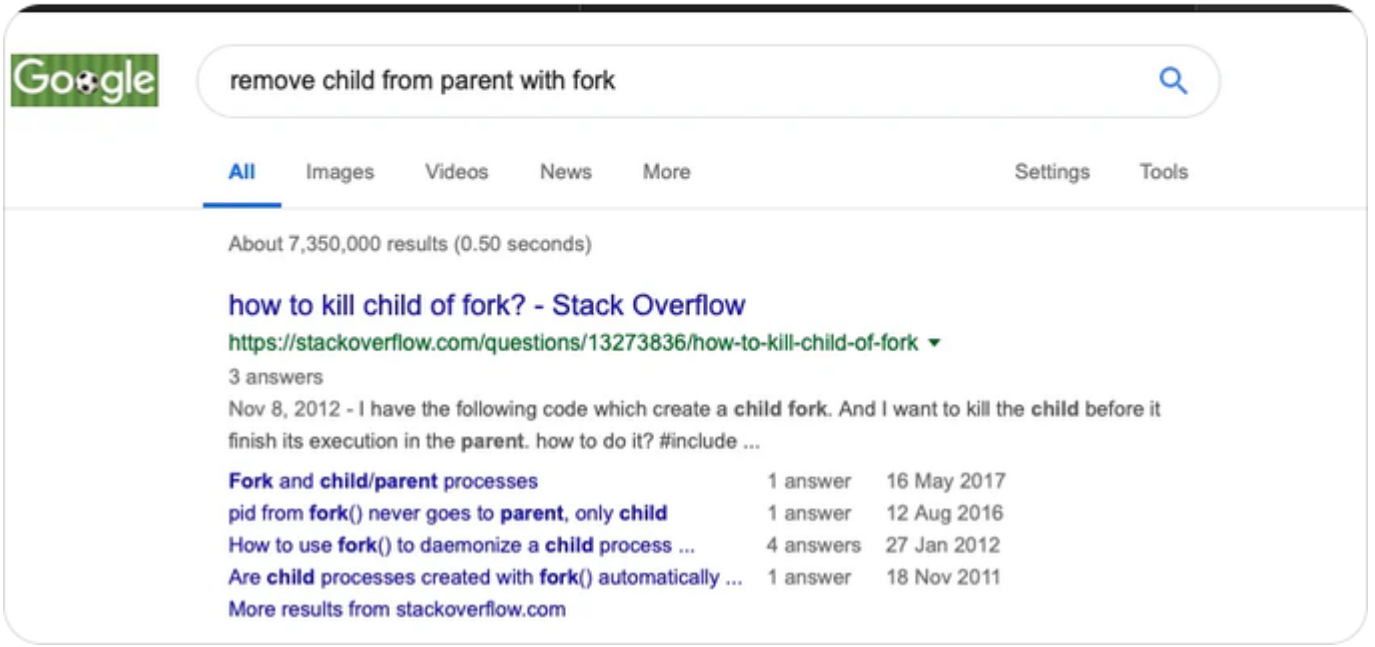
CSCI 2330



One Devloper Army
@OneDevloperArmy

Sometimes, I wonder if non-tech friends wonder about my search history ...



LA hours cancelled today

- Xan is not feeling well



TODO

- Assignment 5 assigned due November 20

Overview

- Lab 5
- Processes

Lab 5

- Infer `krebs`' CPU cache paremeters
- L1 size, L2 size, L3 size
  - Measure how much we can put in each cache until performance drops
- L1 associativity
  - Measure how much we can put in a cache set until performance drops

Lab 5

- Infer `krebs`' CPU cache parameters
- What we as programmers can do until performance plummets

Processes

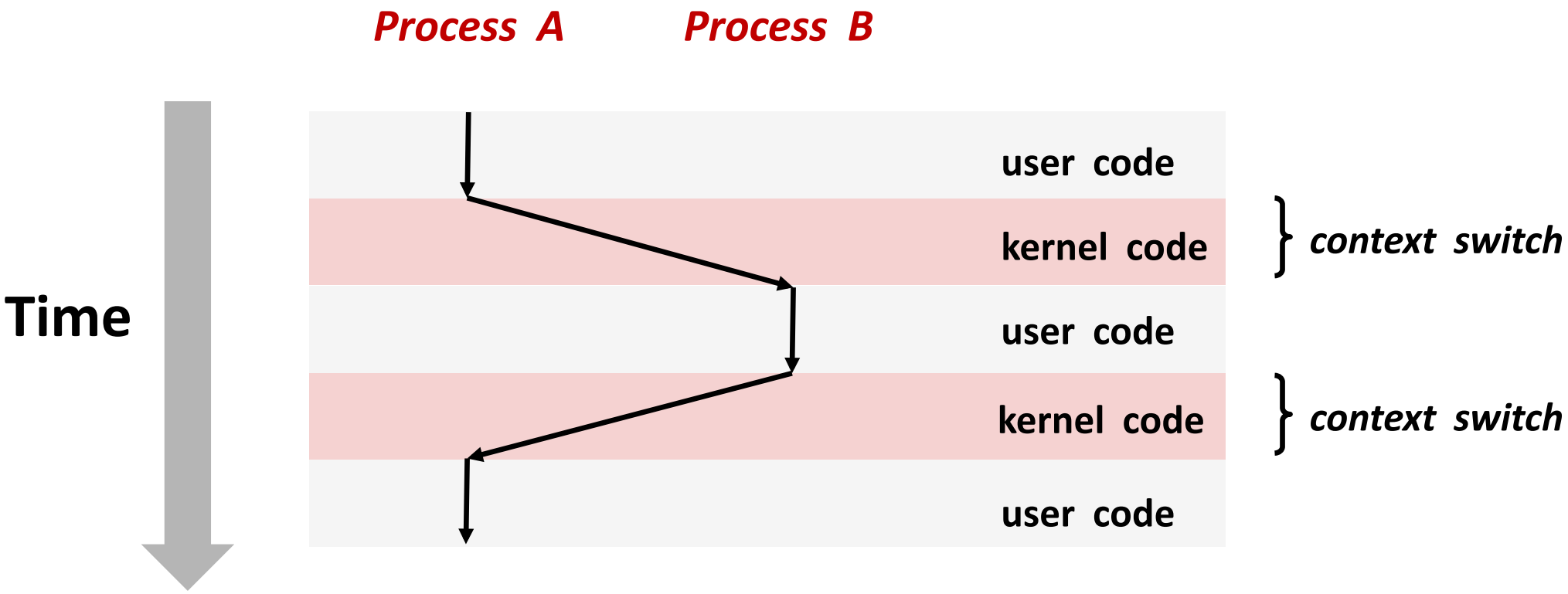
- Our programs have main character syndrome
- We’ve looked at every instruction of our bit bombs
- Not a single instruction acknowledged that any other programs might also be running

Operating system

- The operating system, specifically, the *kernel*, allows the running program to maintain this illusion
- The state of a running program is called a *process*

Context switching

- A single processor can only run one process at once
- The kernel uses *time sharing* to create the appearance that all processes are running simultaneously
- The kernel implements time sharing using context switching – switching processes in and out from execution very quickly



Note: “user” is the opposite of “kernel”, so user code is any non-kernel code

Asynchronous exceptions

- A.k.a. hardware interrupts
- These can happen at any instant
- Caused by a hardware device

Synchronous exceptions

- A.k.a. software interrupts
- Caused by an instruction

Quiz – sync or async?

- Moving a mouse

Quiz – sync or async?

- Moving a mouse
  - Asynchronous

Quiz – sync or async?

- A CPU timer hits zero to indicate its time to context switch

Quiz – sync or async?

- A CPU timer hits zero to indicate its time to context switch
  - Asynchronous

Quiz – sync or async?

- 0 / 0

Quiz – sync or async?

- 0 / 0
  - Synchronous

Quiz – sync or async?

- `int *x = NULL; printf("%d\n", *x);`

Quiz – sync or async?

- `int *x = NULL; printf("%d\n", *x);`
  - Synchronous

Quiz – sync or async?

- A printer signals the print job is finished

Quiz – sync or async?

- A printer signals the print job is finished
  - Asynchronous

Quiz – sync or async?

- Asking the operating system to fork your process

Quiz – sync or async?

- Asking the operating system to fork your process
  - Synchronous
  - A system call!

Where do processes come from?



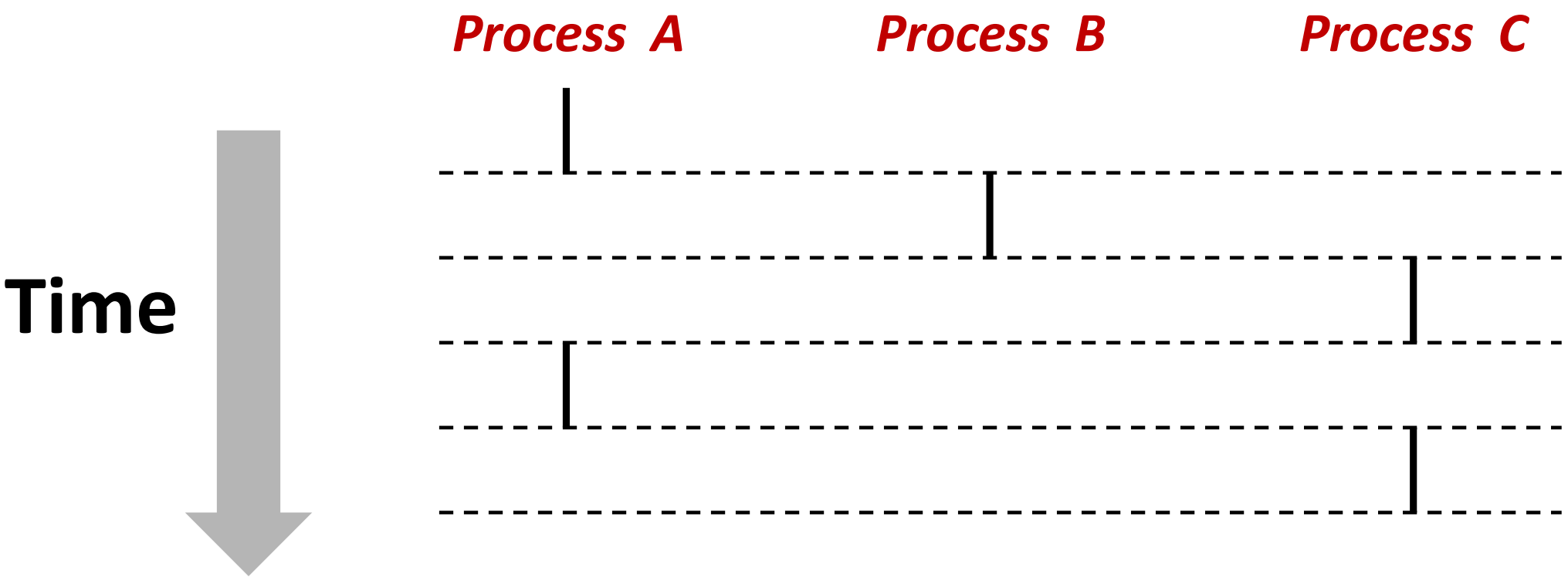
fork system call

- Creates an exact copy of the current process
  - Same memory
  - Same register contents
- With one exception: `%rax`, the fork return value
  - Parent: `fork()` returns PID of newly created process
  - Child: `fork()` returns 0

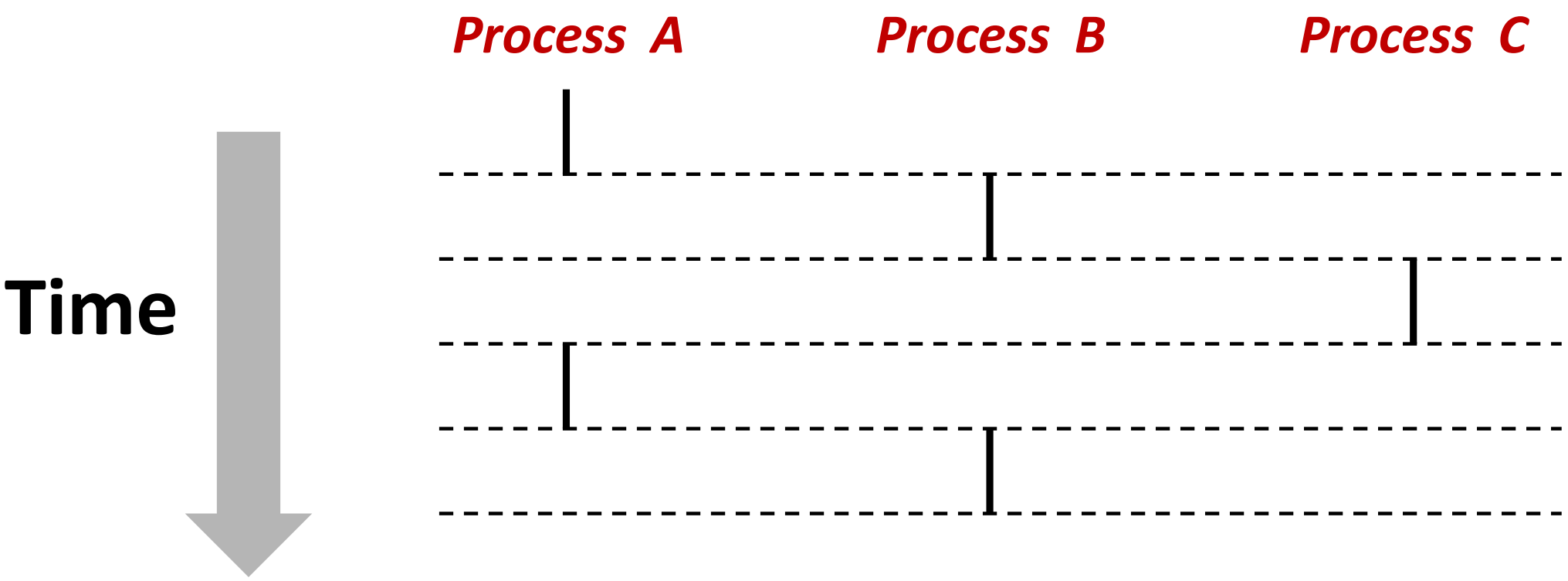
fork system call

- This system call returns *twice* 🍄🍄
- The execution of these two processes is concurrent (the order and timing of how each process will be switched into execution is nondeterministic)…

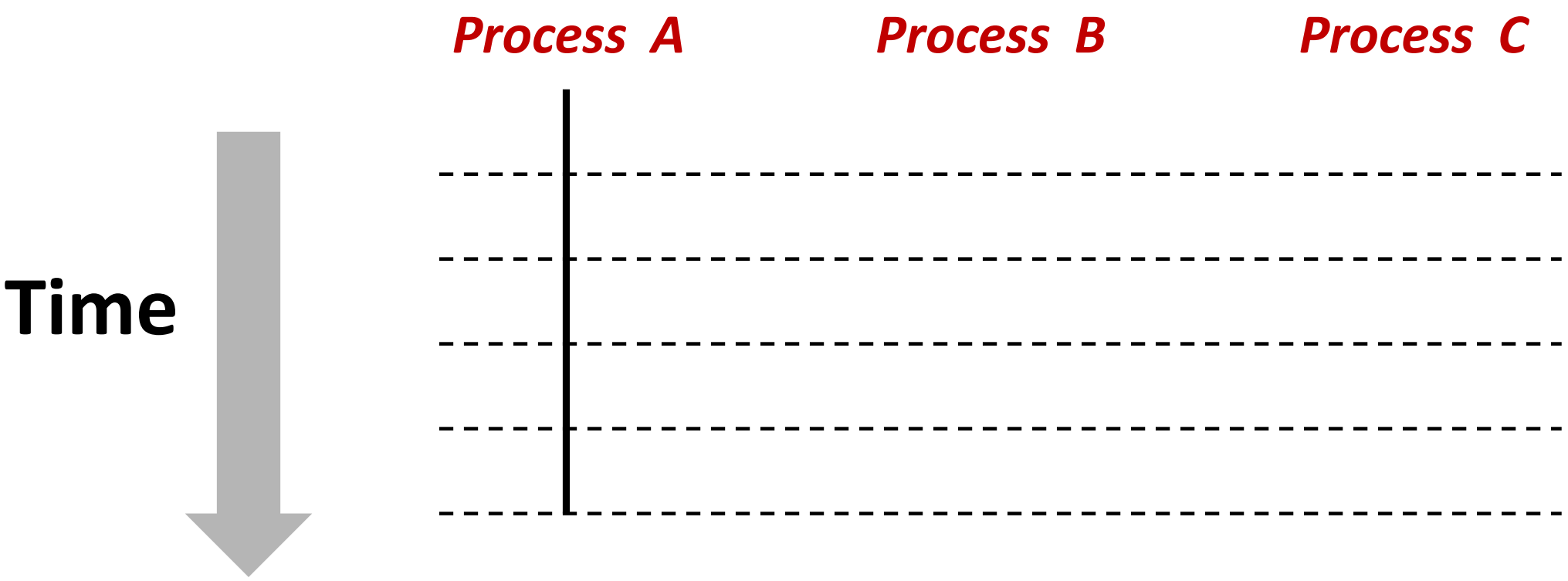
Concurrent processes



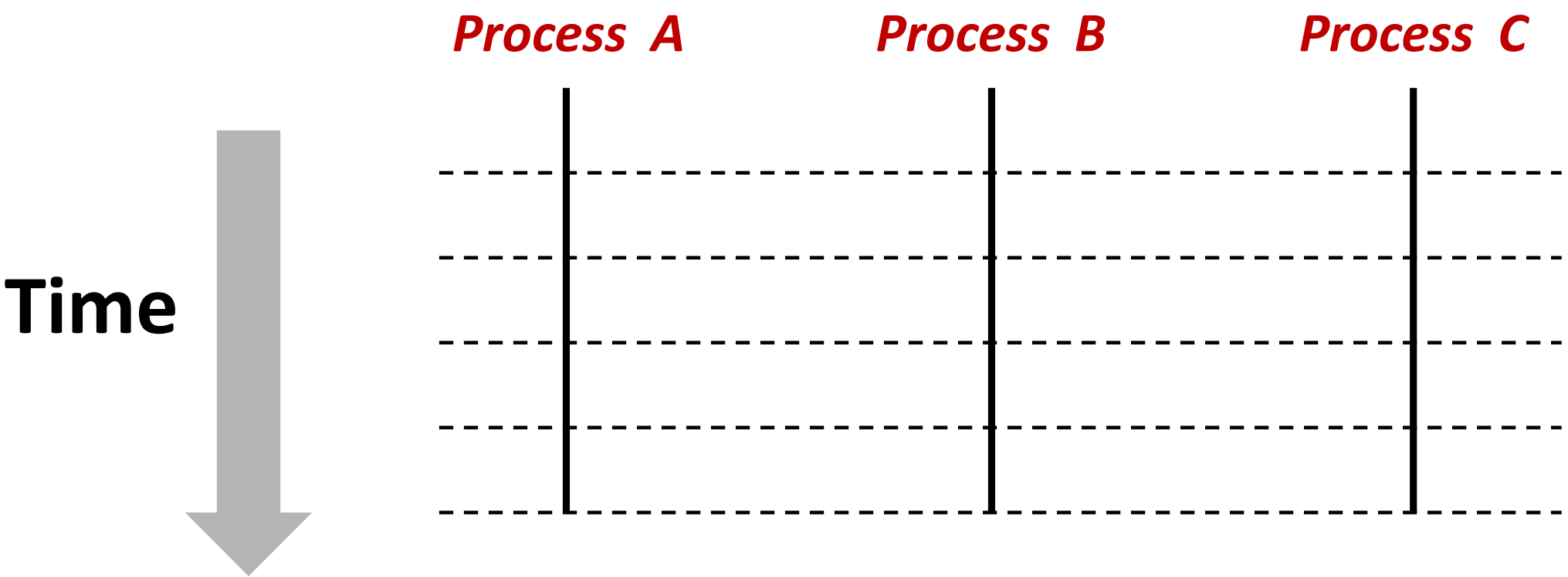
Concurrent processes



Concurrent processes



Simultaneous processes



Fork bomb!

```
while (1) {
    fork();
}
```

Fork exercises

Current limitation

- Our program can have our child execute different instructions than the parent
- But what if we want our child to be a different program all together?
- E.g., you click a dock icon or run a shell command!

Exec system call

- Exec overwrites current program with new program
  - Replaces code with new code
  - Resets all other memory
  - Resets all registers

Exec variants

- `execvp` – takes program name and a NULL-terminated array of arguments
  - `execvp("ls", (char *[]) {"ls", "-l", NULL})`
- `execlp` – takes a program name and a NULL-terminated argument list of arguments
  - `execlp("ls", "ls", "-l", NULL)`
- And others! (See man pages…)

Let’s look at some examples…

- lslong.c
- timedrun.c

But launching a new program?

- So that we don’t wreck the old one?
- Combine fork and exec system calls
- Equilibrium:
  - Fork returns twice 🍄🍄
  - Exec *never* returns
- So fork + exec “returns” once
- Let’s modify forktest.c…

How do we wait for our children to die?

```
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
```

- `pid`: pid of process to wait for
  - If `-1`, wait for *any* process
- `wstatus`: an “output pointer” where waitpid will write the dead child’s exit status
  - Can be `NULL` if we don’t care
- `options`: we will discuss this more later
  - 0 for no options

What is the term for `wait()`ing on a dead process?

Reaping

- When we `wait()` on a process we *reap* it



Let’s put this to practice

```
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
```

- Let’s further enhance forktest.c…
- Note:
  - `fork()` and `execvp()`/`execlp()` require `#include <unistd.h>`
  - `waitpid()` requires `#include <sys/wait.h>`

Who waits for children to die?

- A shell…

```
while (true) {
    Print command prompt.
    Read command line from user.
    Parse command line.
    If command is built-in, execute it.
    Else, fork process
        in child:
            Execute requested command with exec
                                (never returns)
        in parent:
            Wait for child to complete with waitpid
}
```

Preview of what's to come

- The final lab for this course will be implementing a shell
- Writing a shell is an exercise in *systems programming*
  - Not writing user-facing applications
  - Writing the software that makes those possible

What else can shells do?

- Shells can redirect input and output
- Lab 4:
  - `$ ./hex2raw -i 1-hiss.txt > 1-hiss.bytes`
  - `$ ./bytebomb < 1-hiss.bytes`
- May have also seen somewhere:
  - `$ something >> somethingelse.txt`

File modes

- <: open for reading
- >: open for writing
- >>: open for appending

File modes

Mode	Description	Caveats
`"r"` (read)	Opens the file in read-only mode, starting at the beginning	Fails if the file does not exist
`"w"` (write)	Opens the file in write-only mode, starting at the beginning	Creates a new file if it does not exist or resets the file to be empty if it does exist
`"a"` (append)	Opens the file in write-only mode, starting at the end	Creates a new file if it does not exist

You can add a `'+'` to any mode to open the file in read-and-write mode (`"r+"`, `"w+"`, `"a+"`). However, the caveats will still apply.

files.py

```
with open('readme.txt', 'r') as f:
    print(f.fileno())
```

```
$ python3 files.py
3
```

files.py

```
with open('writeme.txt', 'w') as f:
    print(f.fileno())
```

```
$ python3 files.py
3
```

files.py

```
with open('appendtome.txt', 'a') as f:
    print(f.fileno())
```

```
$ python3 files.py
3
```

files.c

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("readme.txt", "r");
    if (file != NULL) {
        printf("%d\n", fileno(file));
        fclose(file);
    }
    return 0;
}
```

```
$ ./foo
3
```

files.c

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("writeme.txt", "w");
    if (file != NULL) {
        printf("%d\n", fileno(file));
        fclose(file);
    }
    return 0;
}
```

```
$ ./foo
3
```

files.c

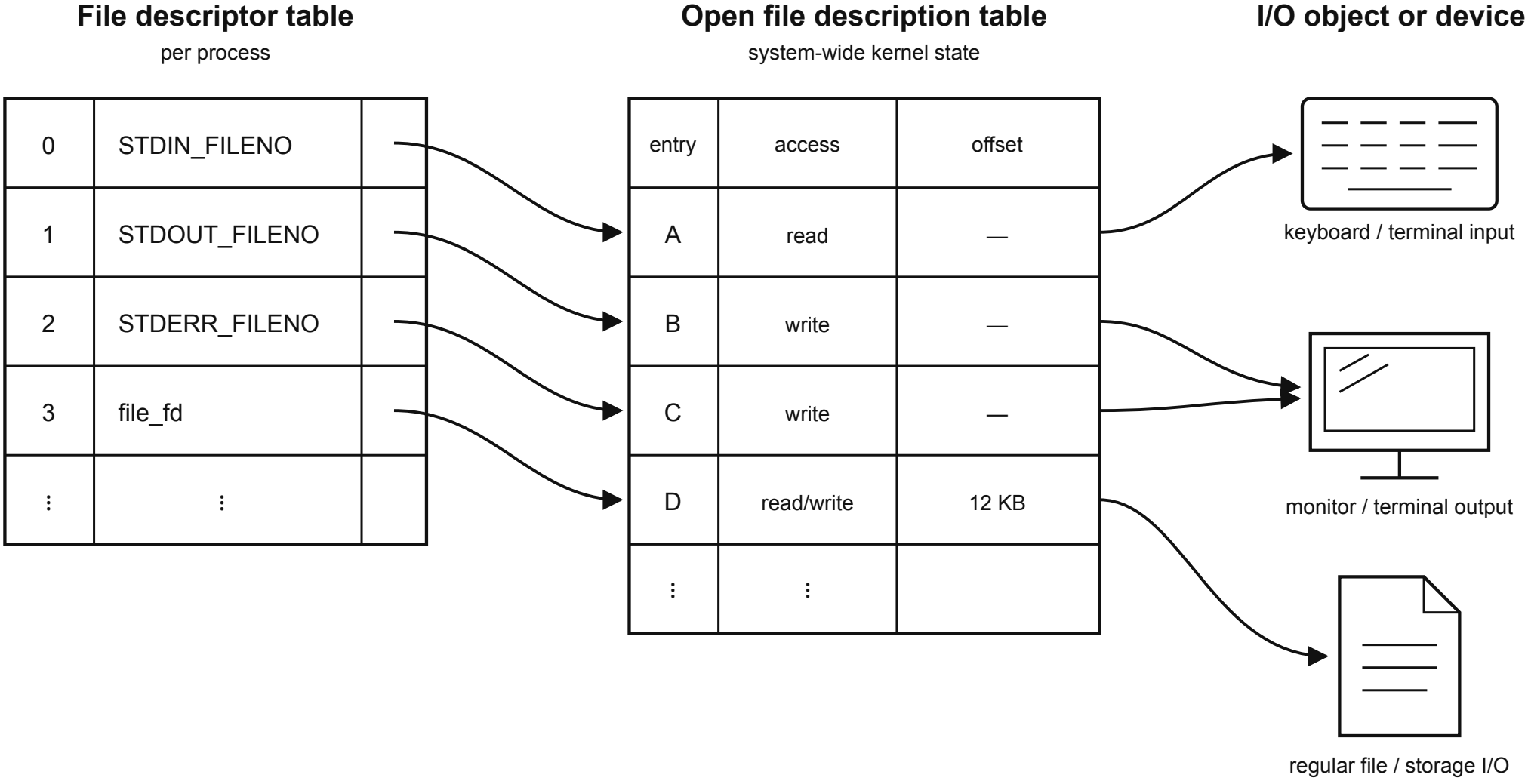
```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("appendtome.txt", "a");
    if (file != NULL) {
        printf("%d\n", fileno(file));
        fclose(file);
    }
    return 0;
}
```

```
$ ./foo
3
```

File handle vs. file descriptor

- File handles are opaque wrappers around file descriptors
- System calls work directly with file descriptors
- File descriptors are lower level: they are ints
- A *file descriptor* is an index into the processes's file descriptor table


File descriptor table
per process
Open file description table
system-wide kernel state
I/O object or device
0
STDIN_FILENO
1
STDOUT_FILENO
2
STDERR_FILENO
3
file_fd
entry
access
offset
A
read
—
B
write
—
C
write
—
D
read/write
12 KB
keyboard / terminal input
monitor / terminal output
regular file / storage I/O

Our working model for a process

- Registers
- Memory
- PID
- File descriptor table 🆕

File handle vs. file descriptor

- Return file handles:
  - In Python, `open()`
  - In C, `fopen()`
- Return file descriptors:
  - In Python, `os.open()`
  - In C, `open()`
- The ones that return file handles wrap the ones that return file descriptors

TODO

- Assignment 5 assigned due November 20