CSc 352
Processes

Benjamin Dicken

Source Code

- The human-readable text describing what we want a program to accomplish based on a particular syntax (Say .c or .java)

Executable Program

- The file containing the object code that can be loaded and executed by the CPU of a computer

Process

- An instance of computation that executes a program over some lifespan, depending on how long the program takes to execute

Process

A unit of computation. When we want to run some program:

- Create a new process
- Load the program into the process
- Execute
- Close

A process can have either one or multiple threads of execution

Process Contents

Image

- The executable code / variables / values loaded into memory

Memory

- Memory space to be used for the program stack, heap

OS Descriptors

- For example, open file descriptors

Security Attributes

- Process owner, privileges, etc

Processor State

- Content of registers, memory addressing

https://en.wikipedia.org/wiki/Process_(computing)

CPU

- The CPU is what executes a process.
- The OS manages which processes get run when
- Nowadays, most CPUs are multi-core, but will use both single-core and multi-core in examples.

http://www.cs.rpi.edu/academics/courses/fall04/os/c8/

Multitasking

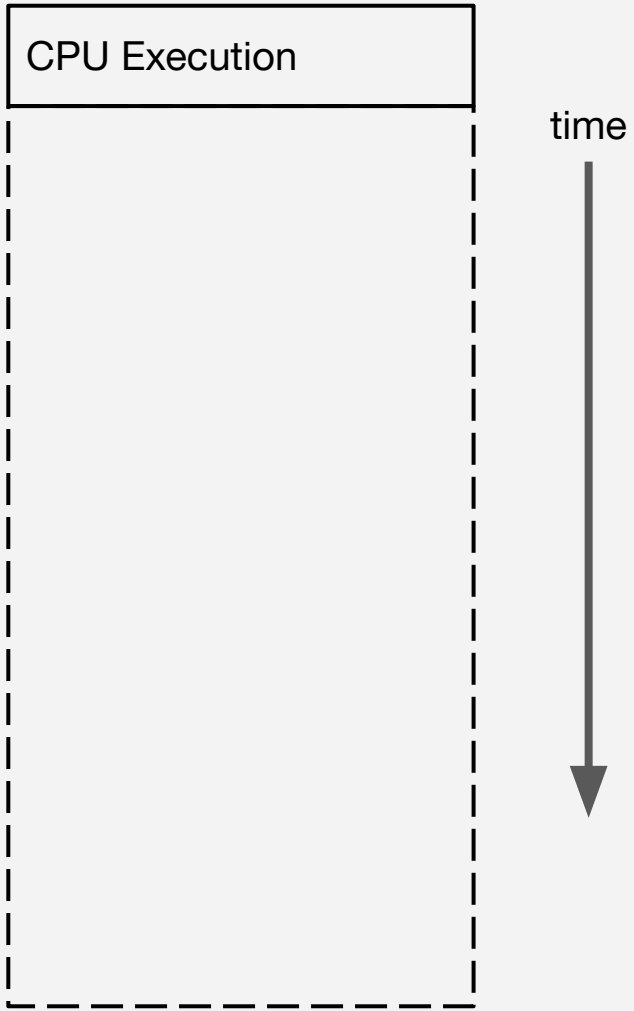
Process A: **bash**

Process B: **grep**

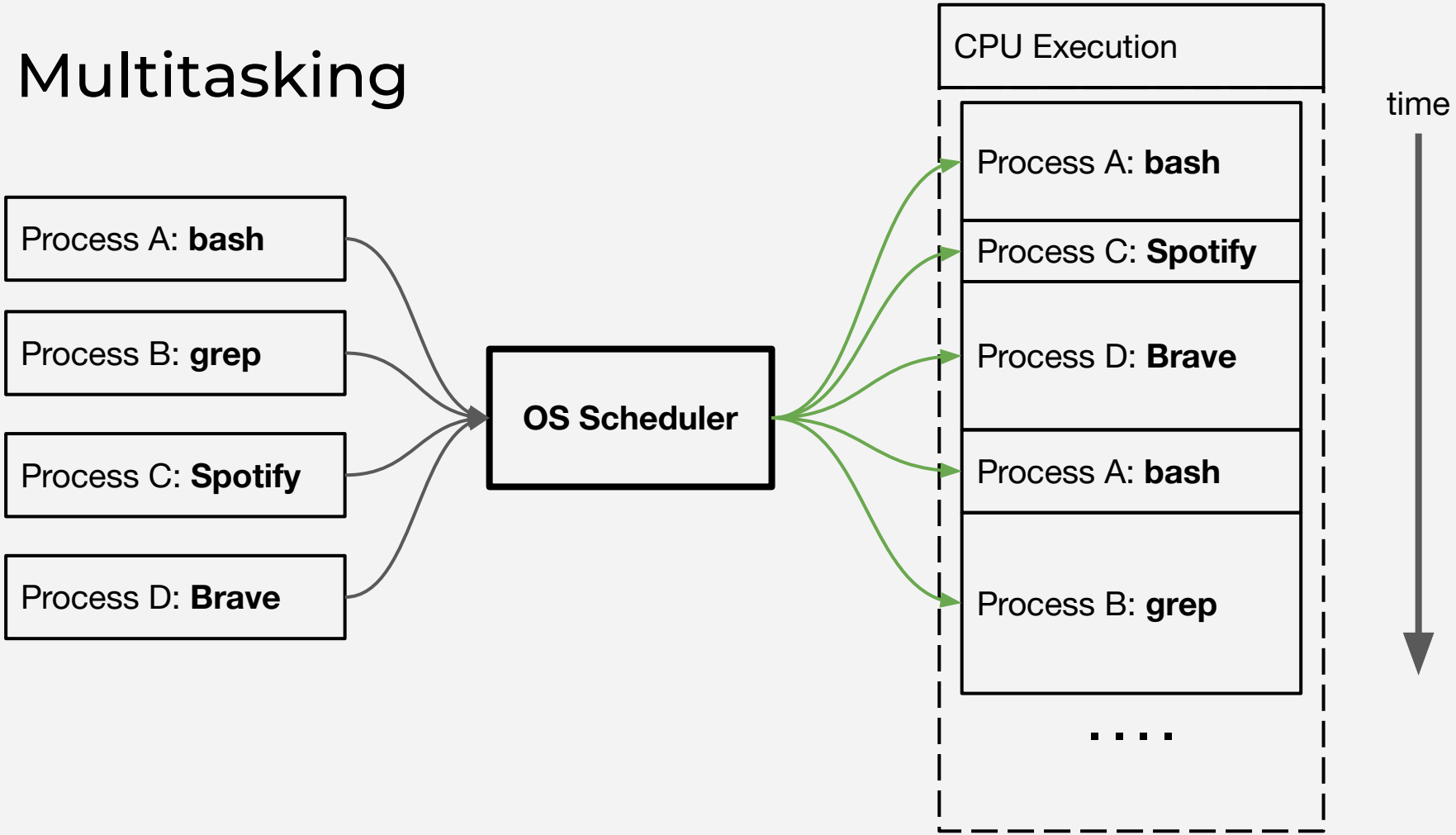
Process C: **Spotify**

Process D: **Brave**

OS Scheduler



Multitasking



Multitasking, multi-core

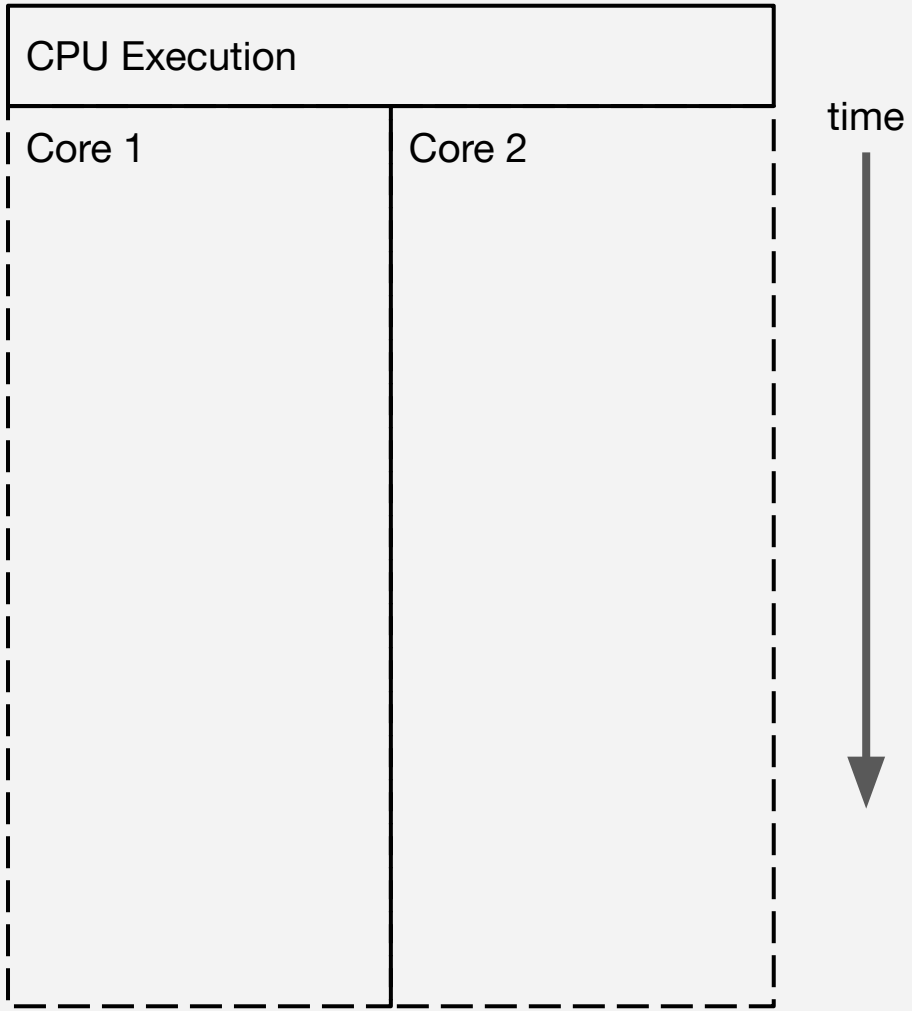
Process A: **bash**

Process B: **grep**

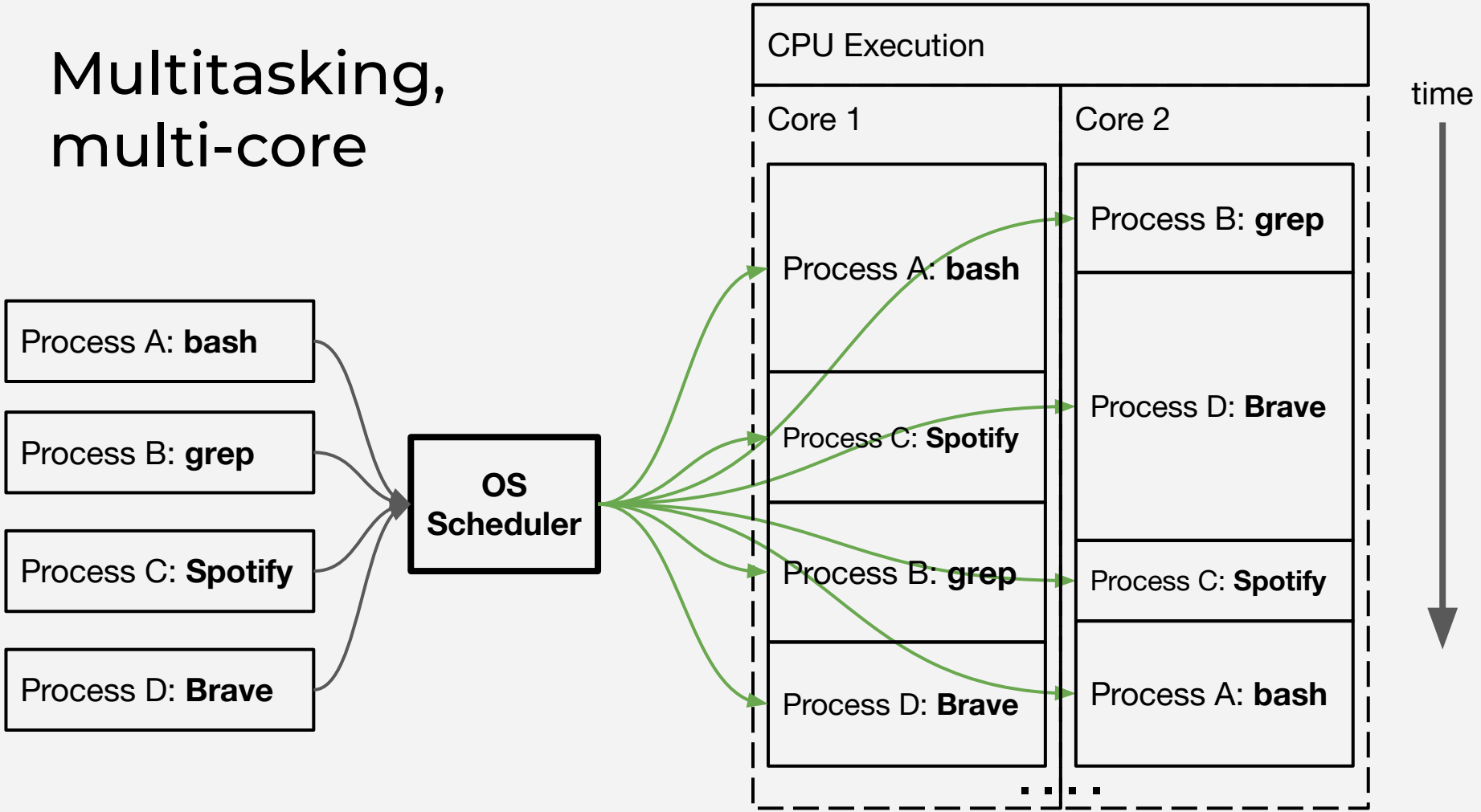
Process C: **Spotify**

Process D: **Brave**

OS Scheduler



Multitasking, multi-core



CPU Scheduler

- A Component of the UNIX OS, manages compute-time of the CPU
- CPU scheduling / switching often happens so fast, things *seem* to be running “at the same time”
- UNIX: Completely Fair Scheduler (CFS)
    - https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

Interacting with Processes

UNIX systems provide a number of commands we can use to view / manage / destroy / create processes.

Let’s look at a few.

Viewing Processes

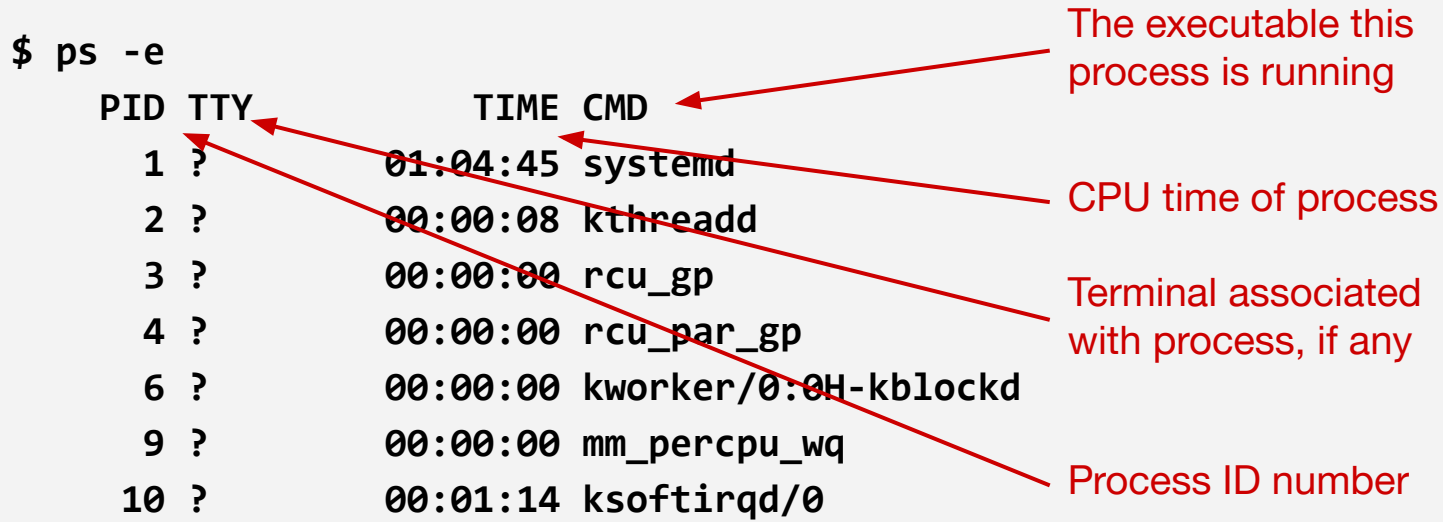
ps allows us to view the ongoing processes on a UNIX system

```
$ ps -e
    PID TTY          TIME CMD
      1 ?        01:04:45 systemd
      2 ?        00:00:08 kthreadd
      3 ?        00:00:00 rcu_gp
      4 ?        00:00:00 rcu_par_gp
      6 ?        00:00:00 kworker/0:0H-kblockd
      9 ?        00:00:00 mm_percpu_wq
     10 ?        00:01:14 ksoftirqd/0
```

Viewing Processes

ps allows us to view the ongoing processes on a UNIX system

```
$ ps -e
    PID TTY          TIME CMD
      1 ?        01:04:45 systemd
      2 ?        00:00:08 kthreadd
      3 ?        00:00:00 rcu_gp
      4 ?        00:00:00 rcu_par_gp
      6 ?        00:00:00 kworker/0:0H-kblockd
      9 ?        00:00:00 mm_percpu_wq
     10 ?        00:01:14 ksoftirqd/0
```



Signalling Processes

kill allows us to send signals to processes (the default is TERM)

```
$ kill -l
$ kill process_id
```

What signals are there?

```
$ man signal
$ man 7 signal
```

Handling Signals

```c
#include <stdio.h>
#include <signal.h>

void handle_signal(int sig) {
  fprintf(stderr, "Fix your broken code!\n");
  fflush(stderr);
}

int main() {
  signal(SIGSEGV, handle_signal);
  char* x = NULL;
  printf("%s\n", x);
  return 0;
}
```

Handling Signals

```c
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void handle_signal(int sig) {
  fprintf(stderr, "Hi there!\n");
  fflush(stderr);
  signal(SIGURG, handle_signal);
}

int main() {
  signal(SIGURG, handle_signal);
  while (1) {
    sleep(1);
  }
  return 0;
}
```

Process Resource Consumption

top allows us to view the resource consumption of processes

```
$ top
```

Like a shell-based version of **System Monitor**

Background Processes

Use the **&** at the end of a command to put it into the background

```
$ ./a.out &
```

Background Processes

Use the **&** at the end of a command to put it into the background

```
$ ./a.out &
```

Use the **fg** / **bg** command to move commands to foreground / background

Use the **jobs** command to view jobs

Background Processes

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
  for (int i = 0; i < 1000; i++) {
    printf("%s\n", argv[1]);
    sleep(2);
  }
  return 0;
}
```