The Processes Abstraction (ch. 4)

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

The Process

A running program

- Lots of processes seemingly running at the same time
- The challenge:
  - Few physical CPUs, illusion of many CPUs

The Process

- **Virtualizing** the CPU
  - Running one process, stopping it, running another, and so forth
  - **Time sharing** of the CPU
  - Illusion that many virtual CPUs exist

The Process

- **Virtualizing** the CPU
  - Running one process, stopping it, running another, and so forth
  - **Time sharing** of the CPU
  - Illusion that many virtual CPUs exist
- **Context switch**
  - Low-level mechanism
  - Stop running one program and start running another
- **Scheduling policy**
  - Algorithm to decide which process should run next
  - By history, workload, performance

Context switch example (xv6-riscv64)

```
1  swtch:
2          sd ra, 0(a0)
3          sd sp, 8(a0)
4          sd s0, 16(a0)
5          sd s1, 24(a0)
6          sd s2, 32(a0)
7          sd s3, 40(a0)
8          sd s4, 48(a0)
9          sd s5, 56(a0)
10         sd s6, 64(a0)
11         sd s7, 72(a0)
12         sd s8, 80(a0)
13         sd s9, 88(a0)
14         sd s10, 96(a0)
15         sd s11, 104(a0)
```

```
1          ld ra, 0(a1)
2          ld sp, 8(a1)
3          ld s0, 16(a1)
4          ld s1, 24(a1)
5          ld s2, 32(a1)
6          ld s3, 40(a1)
7          ld s4, 48(a1)
8          ld s5, 56(a1)
9          ld s6, 64(a1)
10         ld s7, 72(a1)
11         ld s8, 80(a1)
12         ld s9, 88(a1)
13         ld s10, 96(a1)
14         ld s11, 104(a1)
15
16         ret
```

Time and Space Sharing

- **Time sharing**
  - Resource used for a little while by one entity, then a little while by another, and so forth
  - e.g., CPU
- **Space sharing**
  - Resource is divided (in space) among those who wish to use it
  - e.g., memory, disk

Process vs. Program

- **Program**: static code and static data
- **Process**: dynamic instance of the program

- Multiple processes of the same program can exist

What constitutes a process?

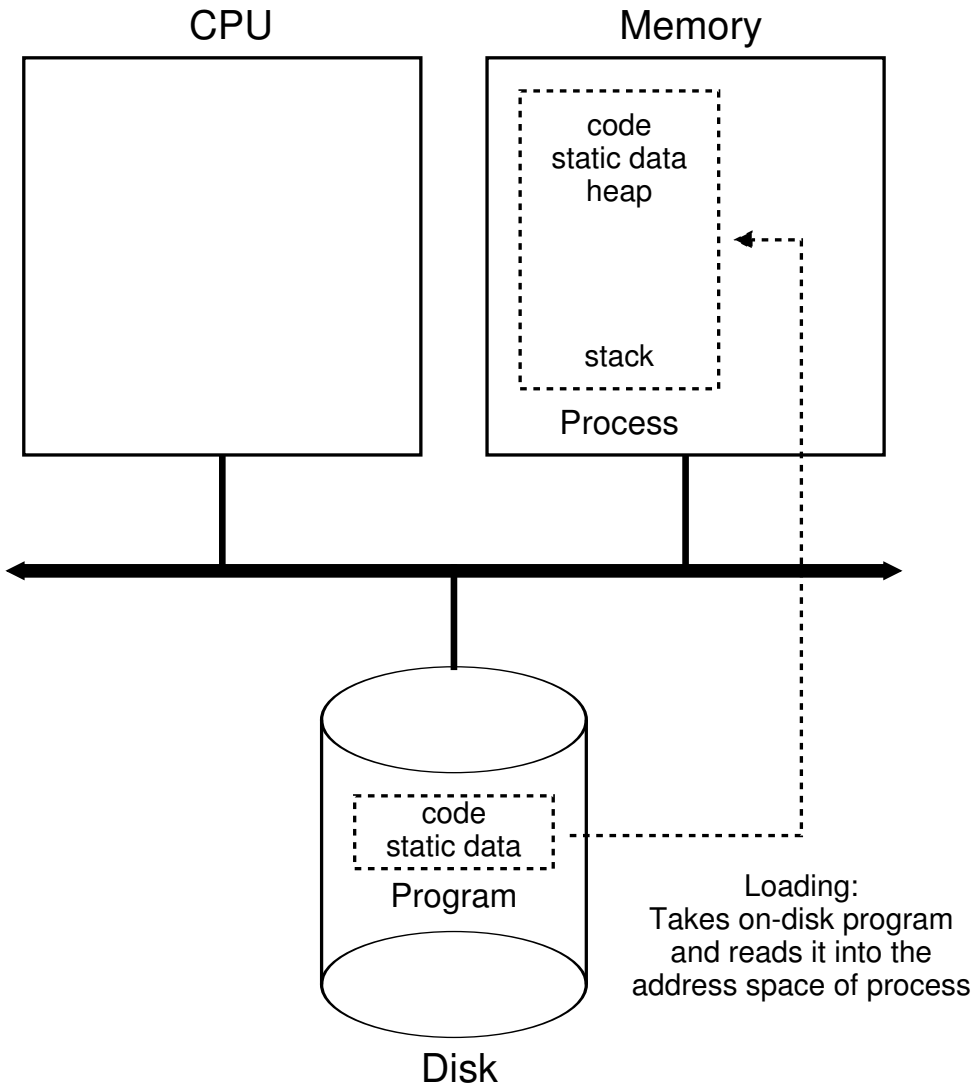
- Memory (**address space**)
  - Instructions (program code)
  - Data (static and dynamic)
  - `cat /proc/<PID>/maps`
- Registers
  - Program counter (PC)
  - Stack pointer
  - etc.
- I/O information
  - e.g., open files
  - `cat /proc/<PID>/fdinfo/*`

Process Creation

- Unix likes OSes: A process is a replica of a currently existing process.
  - There is a way to load an executable file into an existing process.
- Non-Unix like OSes: A process is created with information from an exe file.

Either way, the first process is created by the OS on initialization.

Executable Loading



Executable Loading

- **Load** code and static data into memory
  - Program initially on disk
  - Loading can be done **lazily** (via **paging** and **swapping**)

Executable Loading

- **Load** code and static data into memory
  - Program initially on disk
  - Loading can be done **lazily** (via **paging** and **swapping**)
- Allocate the **stack**
  - Used for local variables, function parameters, return addresses
  - Initialized with `main` arguments: `argc`, `argv`

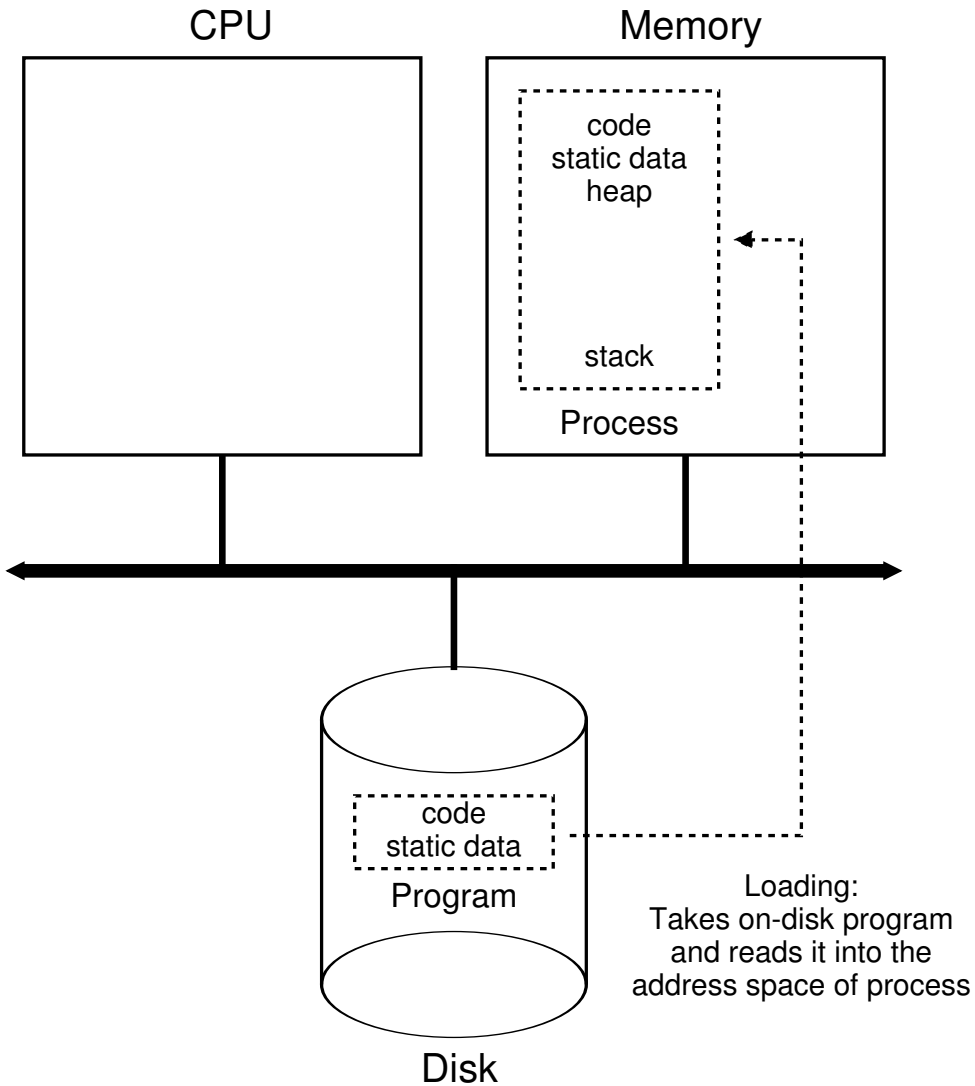
Executable Loading

- **Load** code and static data into memory
  - Program initially on disk
  - Loading can be done **lazily** (via **paging** and **swapping**)
- Allocate the **stack**
  - Used for local variables, function parameters, return addresses
  - Initialized with `main` arguments: `argc`, `argv`
- Allocate the **heap**
  - Used for dynamically-allocated data
  - Request space by calling `malloc`, free it by `free`

Executable Loading

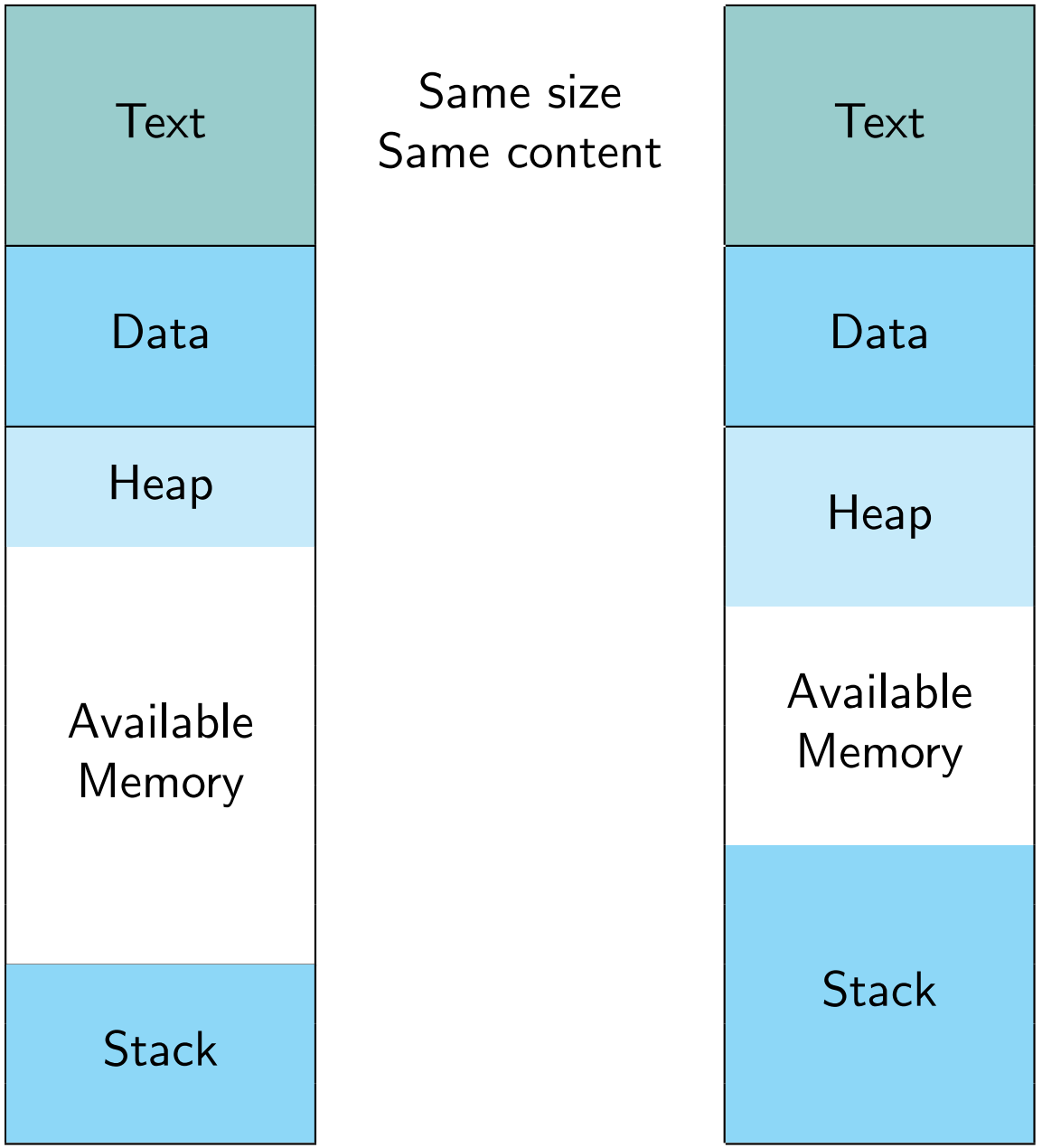
- **Load** code and static data into memory
  - Program initially on disk
  - Loading can be done **lazily** (via **paging** and **swapping**)
- Allocate the **stack**
  - Used for local variables, function parameters, return addresses
  - Initialized with `main` arguments: `argc`, `argv`
- Allocate the **heap**
  - Used for dynamically-allocated data
  - Request space by calling `malloc`, free it by `free`
- Start program at entry point (NOT necessarily `main()`)
  - Transfer control of CPU to the newly-created process

Executable loading

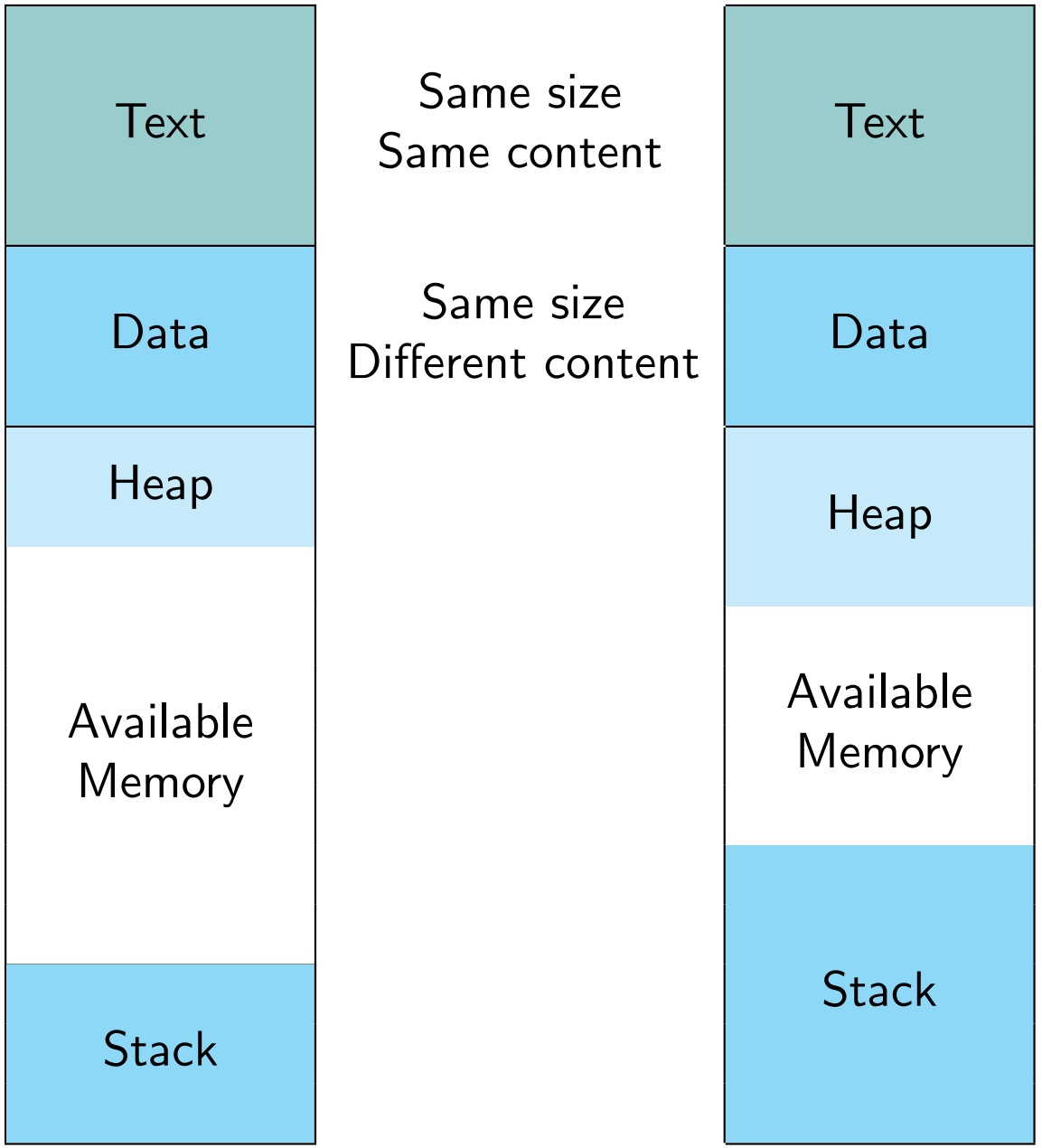


Two Processes for One Program (Virt Mem View)

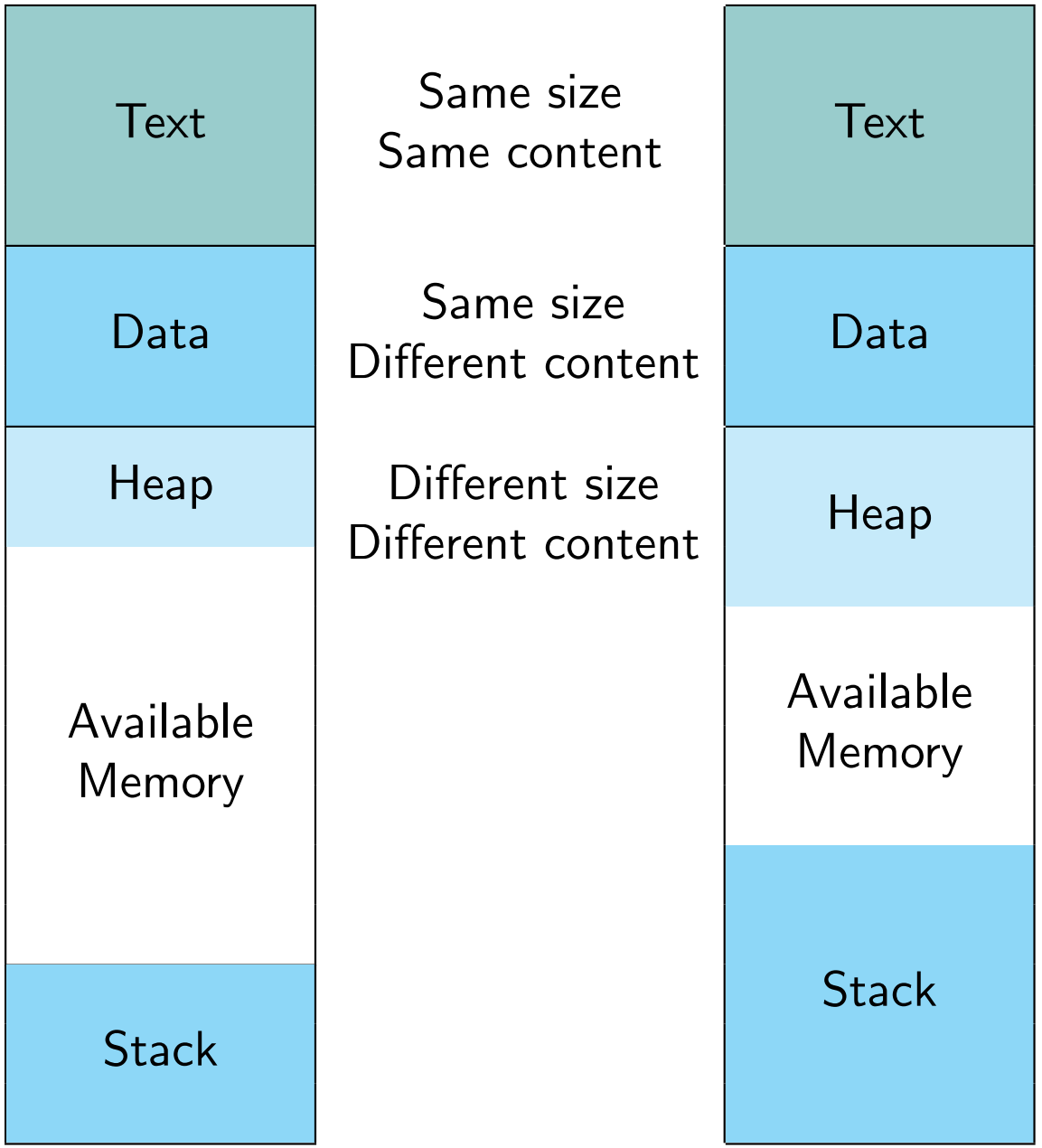
Two Processes for One Program (Virt Mem View)



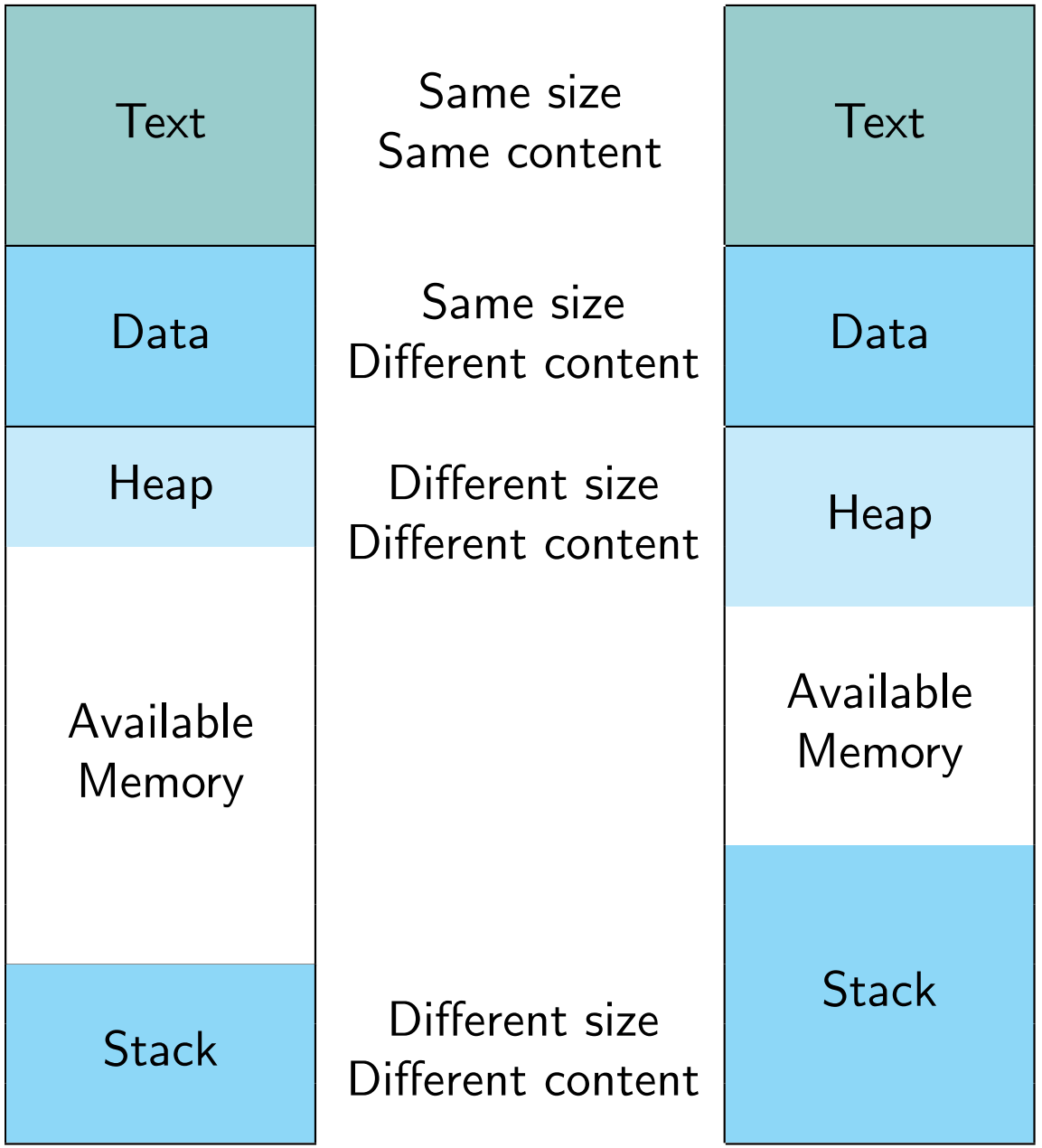
Two Processes for One Program (Virt Mem View)



Two Processes for One Program (Virt Mem View)



Two Processes for One Program (Virt Mem View)



Process Life Cycle

- A process can be in a finite number of **states**
- Events cause **transitions** between states

Process Life Cycle

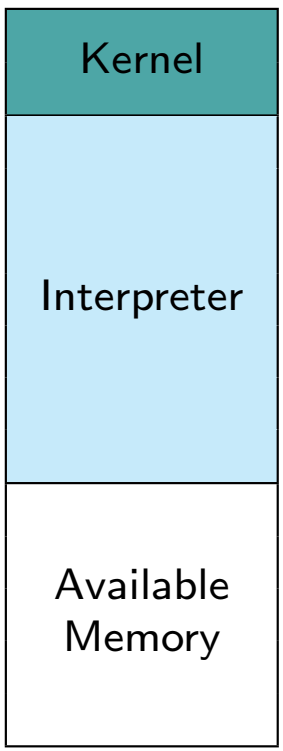
- A process can be in a finite number of **states**
- Events cause **transitions** between states

- **Single-tasking** OS (Ancient, Small phys. memory)
  - Only one process at a time
  - **Interpreter** loaded on boot, overwrites part of itself into process

Process Life Cycle

- A process can be in a finite number of **states**
- Events cause **transitions** between states

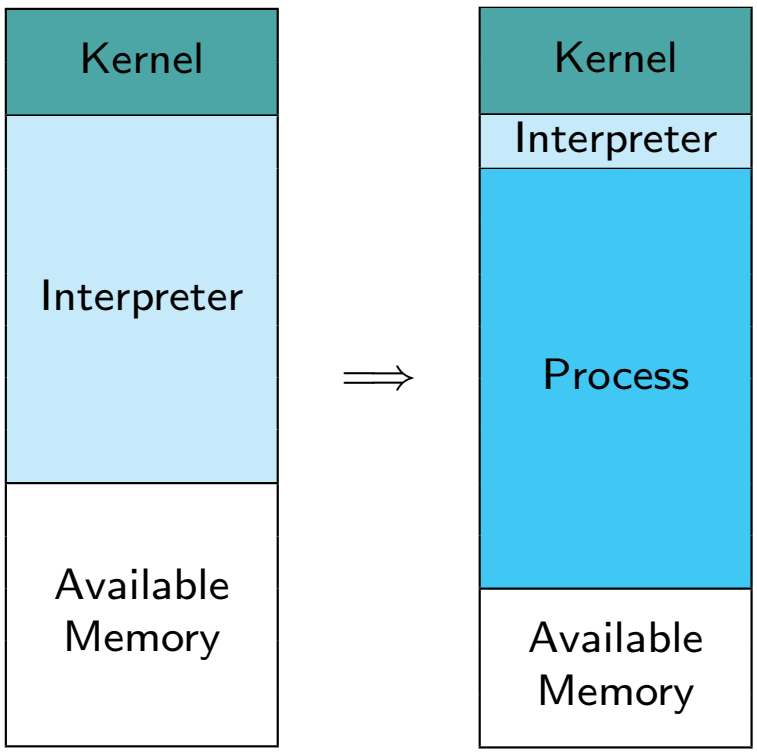
- **Single-tasking** OS (Ancient, Small phys. memory)
  - Only one process at a time
  - **Interpreter** loaded on boot, overwrites part of itself into process



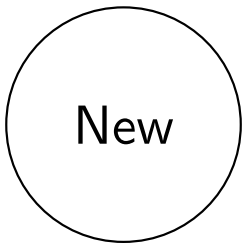
Process Life Cycle

- A process can be in a finite number of **states**
- Events cause **transitions** between states

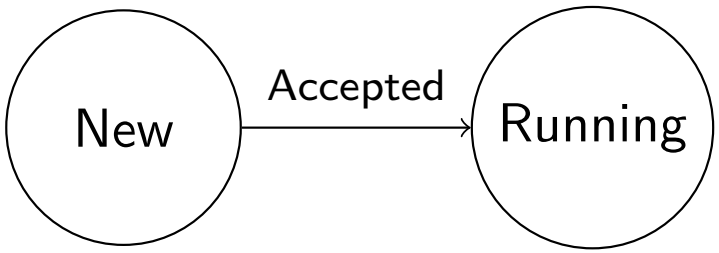
- **Single-tasking** OS (Ancient, Small phys. memory)
  - Only one process at a time
  - **Interpreter** loaded on boot, overwrites part of itself into process



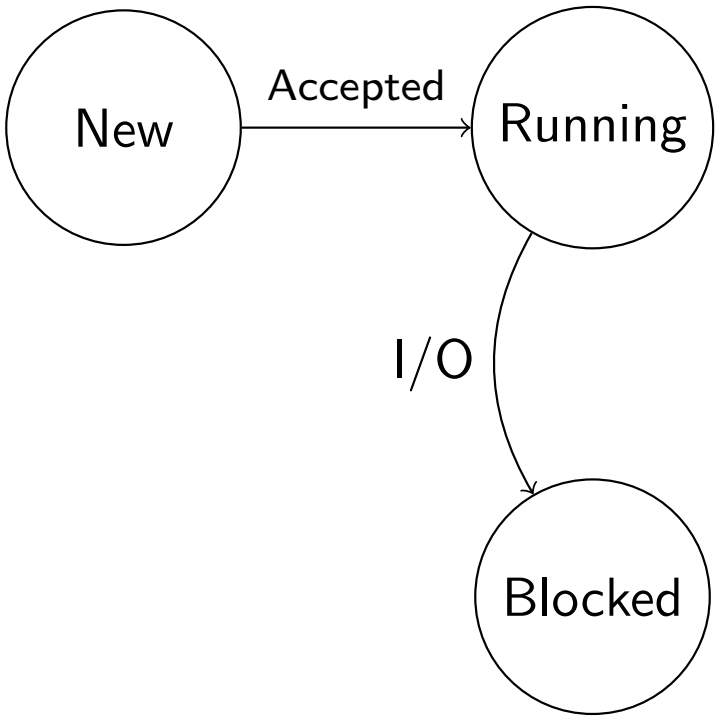
Process Life Cycle - Single Tasking



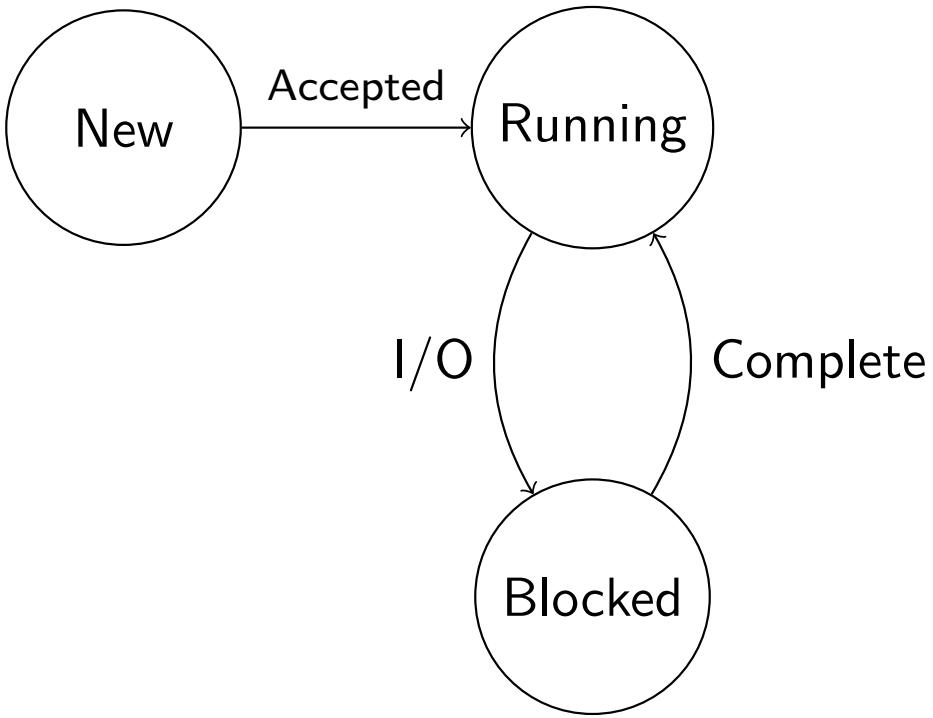
Process Life Cycle - Single Tasking



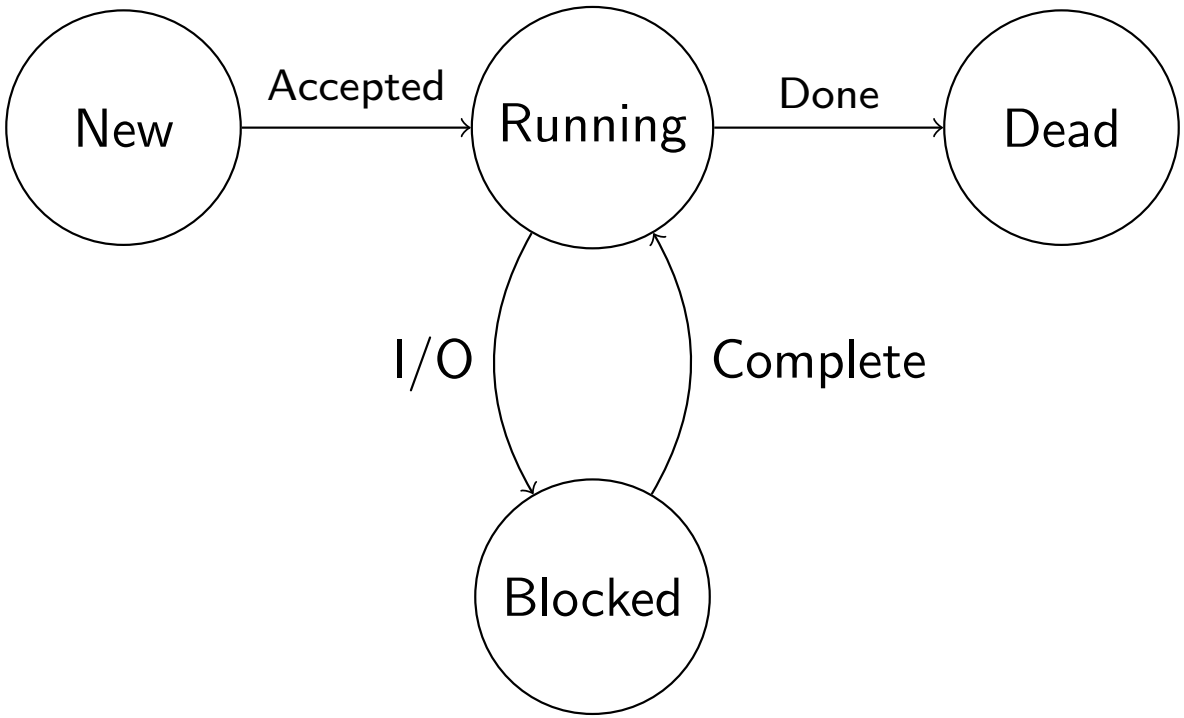
Process Life Cycle - Single Tasking



Process Life Cycle - Single Tasking



Process Life Cycle - Single Tasking



Process Life Cycle

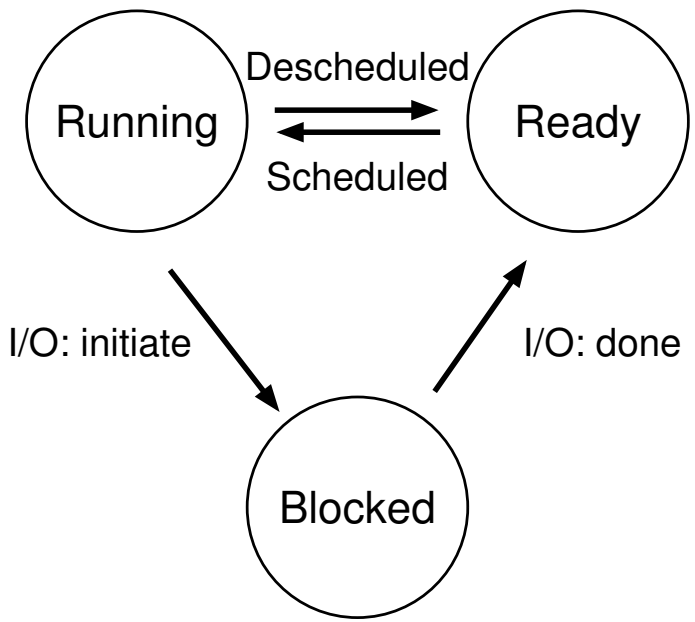
- Modern operating systems: **multi-tasking**
  - Multiple processes co-exist
  - **Cooperative** multi-tasking: `yield`
  - **Preemptive** multi-tasking: **interrupts**

Process Life Cycle

- Modern operating systems: **multi-tasking**
  - Multiple processes co-exist
  - **Cooperative** multi-tasking: `yield`
  - **Preemptive** multi-tasking: **interrupts**
- A process can be **ready** to run, but not running
  - OS schedules a process to run for a while, then deschedules it and picks another process, and so forth
  - A new state: **ready**

Process States

- **Running**: executing on CPU
- **Ready**: ready to run, waiting to be scheduled
- **Blocked**: suspended, waiting for some event



Process States - Example I

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process 0 done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	Process 1 done

Process States - Example II

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	0 initiates I/O
4	Blocked	Running	0 is blocked
5	Blocked	Running	so 1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process 1 done
9	Running	-	
10	Running	-	Process 0 done

Data Structures

- OS maintains a data structure of active processes
  - The **process table**
  - Limited size - `cat /proc/sys/kernel/threads-max`

Data Structures

- OS maintains a data structure of active processes
  - The **process table**
  - Limited size - `cat /proc/sys/kernel/threads-max`
- Process Control Block (**PCB**):
  - Process identifier (**PID**)
  - State
  - Related processes (parent)
  - CPU context, e.g., registers (saved when suspended)
  - Memory locations
  - Open files

Summary (Process Abstraction)

- **Process**: OS abstraction of a running program
- Can be described by:
  - **Address space**
  - CPU registers (inc. **program counter** & **stack pointer**)
  - I/O information (e.g., open files)
- **Process state**: running, ready to run, blocked.
  - transition by different events
- **Process list**: information about all processes in the system
  - **Process control block**: a structure with information about a specific process