# Operating Systems

## Lecture 2: The Process

Nipun Batra

# Concurrency

# Concurrency

1. We discussed previously how OS juggles between multiple processes

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops
    1. Different answers?

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops
   1. Different answers?
   2. Counter  UNEQUAL TO  N * # Loops. Why?

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops
    1. Different answers?
    2. Counter  UNEQUAL TO  N * # Loops. Why?
    3. Update operation:

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops
    1. Different answers?
    2. Counter UNEQUAL TO N * # Loops. Why?
    3. Update operation:
        1. Load Counter into register

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops
    1. Different answers?
    2. Counter  UNEQUAL TO  N * # Loops. Why?
    3. Update operation:
        1. Load Counter into register
        2. Update Counter

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops
   1. Different answers?
   2. Counter  UNEQUAL TO  N * # Loops. Why?
   3. Update operation:
      1. Load Counter into register
      2. Update Counter
      3. Store Counter

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops
    1. Different answers?
    2. Counter  UNEQUAL TO  N * # Loops. Why?
    3. Update operation:
        1. Load Counter into register
        2. Update Counter
        3. Store Counter
    4. Non-atomic update!

# Concurrency

1. We discussed previously how OS juggles between multiple processes
2. Run python thread.py #loops
    1. Different answers?
    2. Counter  UNEQUAL TO  N * # Loops. Why?
    3. Update operation:
        1. Load Counter into register
        2. Update Counter
        3. Store Counter
    4. Non-atomic update!

# Persistence

# Persistence

1. RAM is volatile —> need persistent storage

# Persistence

1. RAM is volatile —> need persistent storage
2. Earlier HDD and now mostly SSD

# Persistence

1. RAM is volatile —> need persistent storage
2. Earlier HDD and now mostly SSD
3. OS has a component called filesystem:

# Persistence

1. RAM is volatile —> need persistent storage
2. Earlier HDD and now mostly SSD
3. OS has a component called filesystem:
    1. Storing and reading files

# Persistence

1. RAM is volatile —> need persistent storage
2. Earlier HDD and now mostly SSD
3. OS has a component called filesystem:
    1. Storing and reading files
    2. Maintains data structure for file access

# Persistence

1. RAM is volatile —> need persistent storage
2. Earlier HDD and now mostly SSD
3. OS has a component called filesystem:
    1. Storing and reading files
    2. Maintains data structure for file access

# Persistence

1. RAM is volatile —> need persistent storage
2. Earlier HDD and now mostly SSD
3. OS has a component called filesystem:
    1. Storing and reading files
    2. Maintains data structure for file access

# Design Goals

# Design Goals

1. High performance -> Minimize OS overheads

# Design Goals

1. High performance -> Minimize OS overheads
    1. Extra memory

# Design Goals

1. High performance -> Minimize OS overheads
    1. Extra memory
    2. Extra CPU

# Design Goals

1. High performance -> Minimize OS overheads
    1. Extra memory
    2. Extra CPU
    3. Extra disk

# Design Goals

1. High performance -> Minimize OS overheads
    1. Extra memory
    2. Extra CPU
    3. Extra disk
2. Protecting applications from one harming another and the OS -> Isolation

# Design Goals

1. High performance -> Minimize OS overheads
   1. Extra memory
   2. Extra CPU
   3. Extra disk
2. Protecting applications from one harming another and the OS -> Isolation
3. Reliability

# Design Goals

1. High performance -> Minimize OS overheads
    1. Extra memory
    2. Extra CPU
    3. Extra disk
2. Protecting applications from one harming another and the OS -> Isolation
3. Reliability
    1. Imagine sitting in a flight and the OS crashing!

# Design Goals

1. High performance -> Minimize OS overheads
   1. Extra memory
   2. Extra CPU
   3. Extra disk
2. Protecting applications from one harming another and the OS -> Isolation
3. Reliability
   1. Imagine sitting in a flight and the OS crashing!
   2. Or, dispensing cash in an ATM and the OS crashing!

# Design Goals

1. High performance -> Minimize OS overheads
    1. Extra memory
    2. Extra CPU
    3. Extra disk
2. Protecting applications from one harming another and the OS -> Isolation
3. Reliability
    1. Imagine sitting in a flight and the OS crashing!
    2. Or, dispensing cash in an ATM and the OS crashing!
    3. Or, the MRI scan machine OS reboots on its own!

# Design Goals

1. High performance -> Minimize OS overheads
    1. Extra memory
    2. Extra CPU
    3. Extra disk
2. Protecting applications from one harming another and the OS -> Isolation
3. Reliability
    1. Imagine sitting in a flight and the OS crashing!
    2. Or, dispensing cash in an ATM and the OS crashing!
    3. Or, the MRI scan machine OS reboots on its own!
4. Energy efficiency (esp. for mobile systems!)

# Process

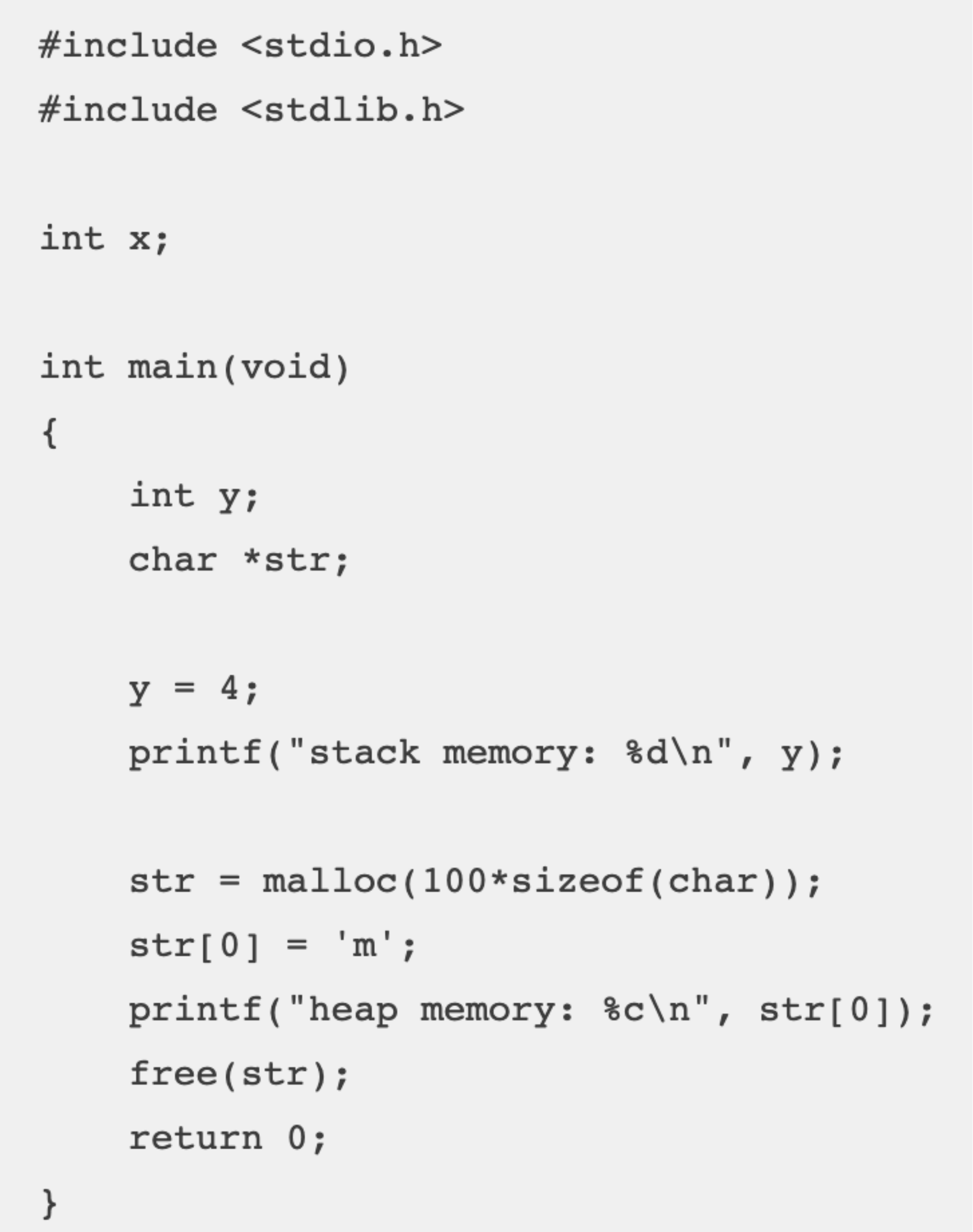
- Process = Running program
- Review example from previous lecture
    - Output of top
    - Activity monitor

# Memory in C

– **static**: global variable storage, permanent for the entire run of the program.
– **stack**: local variable storage (automatic, continuous memory).
– **heap**: dynamic storage (large pool of memory, not allocated in contiguous order).

https://craftofcoding.wordpress.com/2015/12/07/memory-in-c-the-stack-the-heap-and-static/

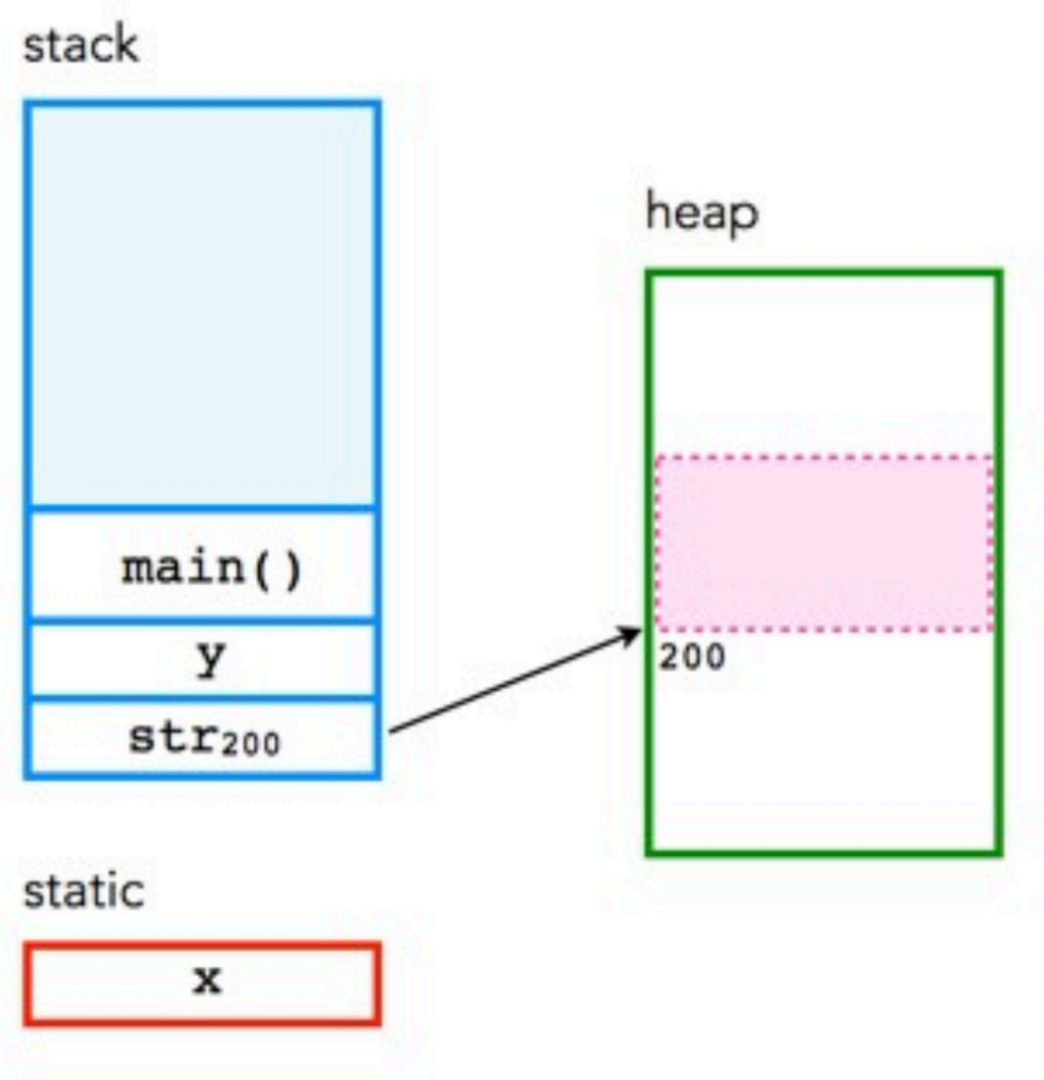
# Memory in C



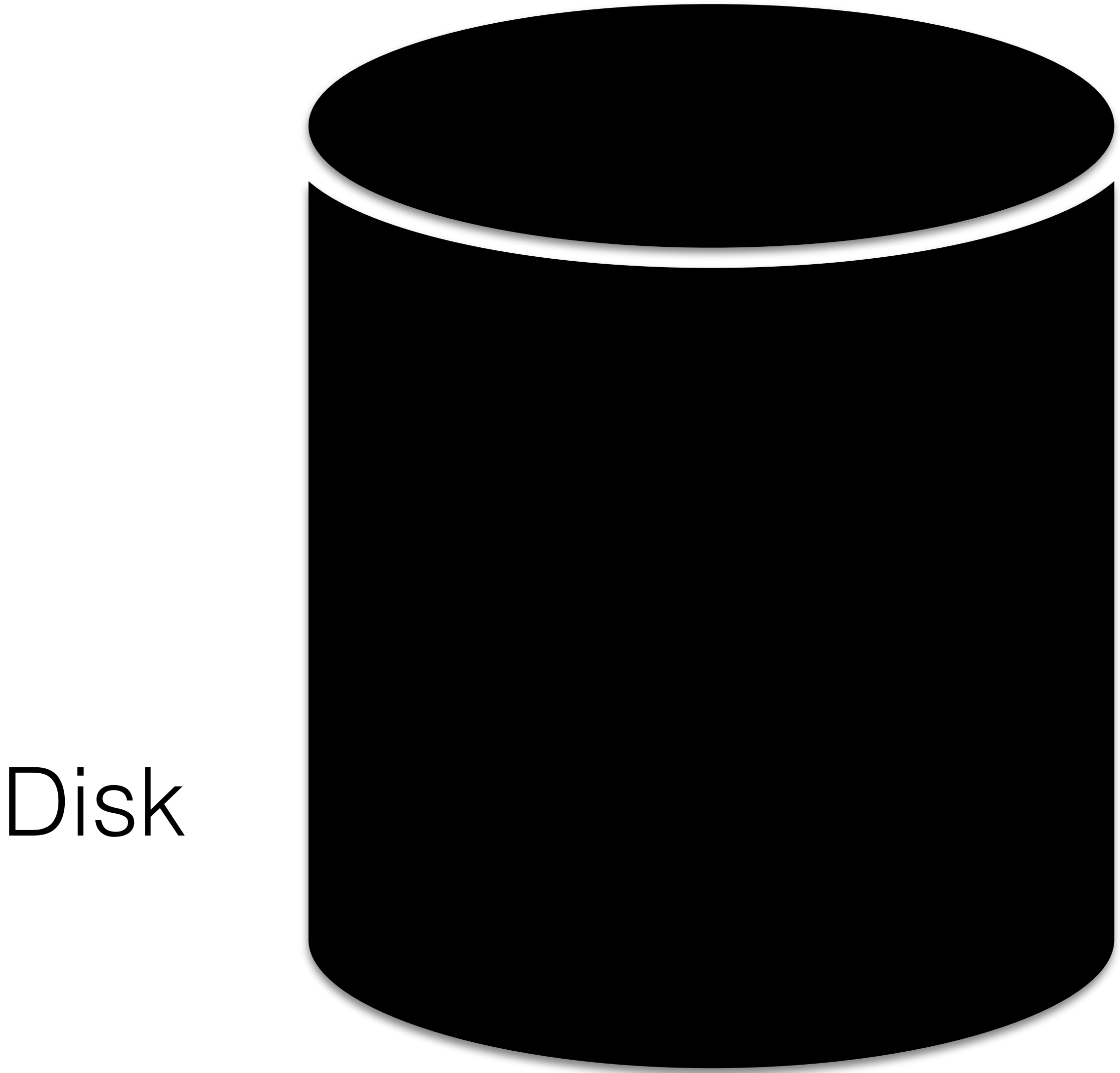
```
#include <stdio.h>
#include <stdlib.h>

int x;

int main(void)
{
    int y;
    char *str;

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```

# Memory in C

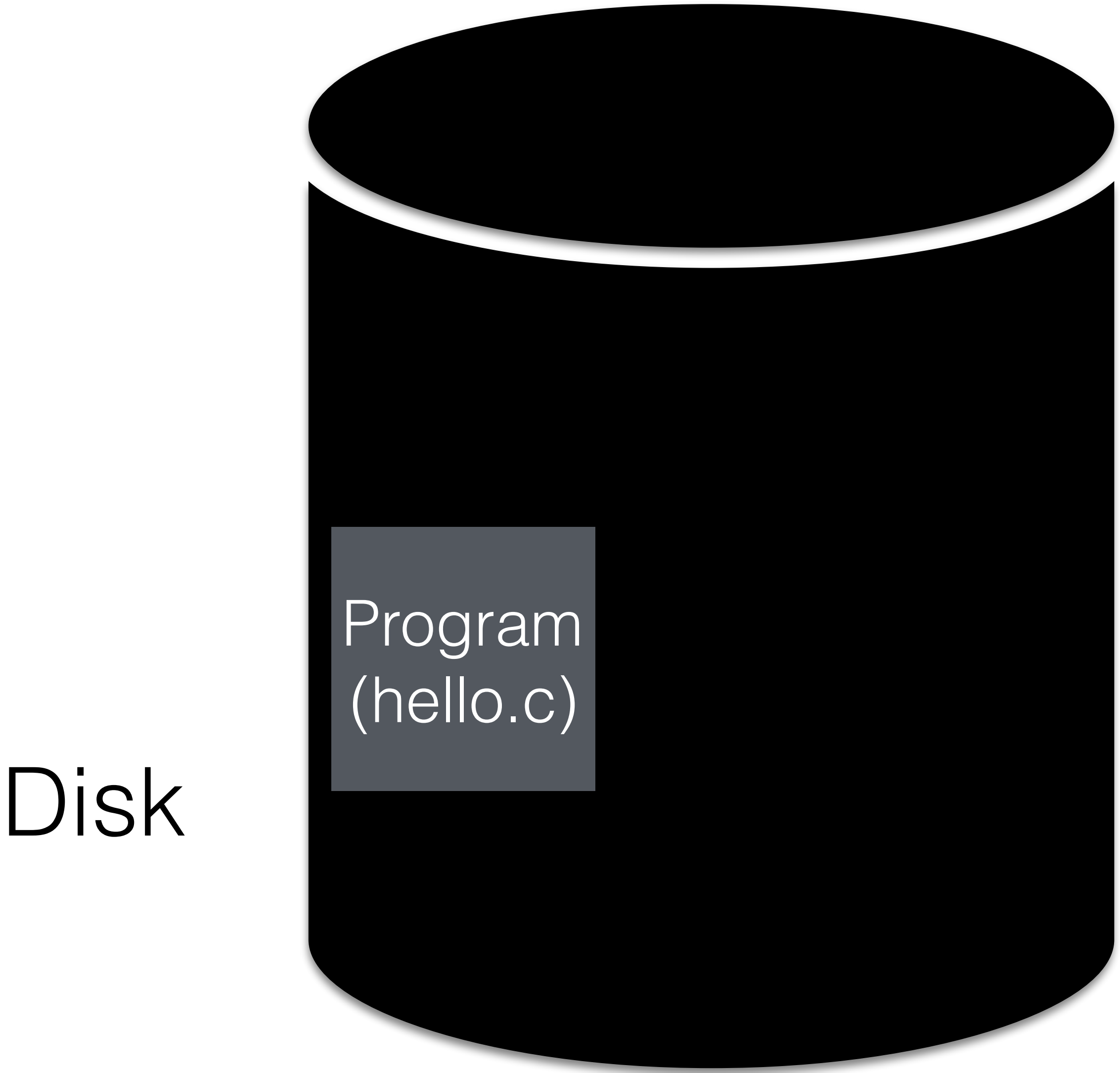
```
#include <stdio.h>
#include <stdlib.h>

int x;

int main(void)
{
    int y;
    char *str;

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```



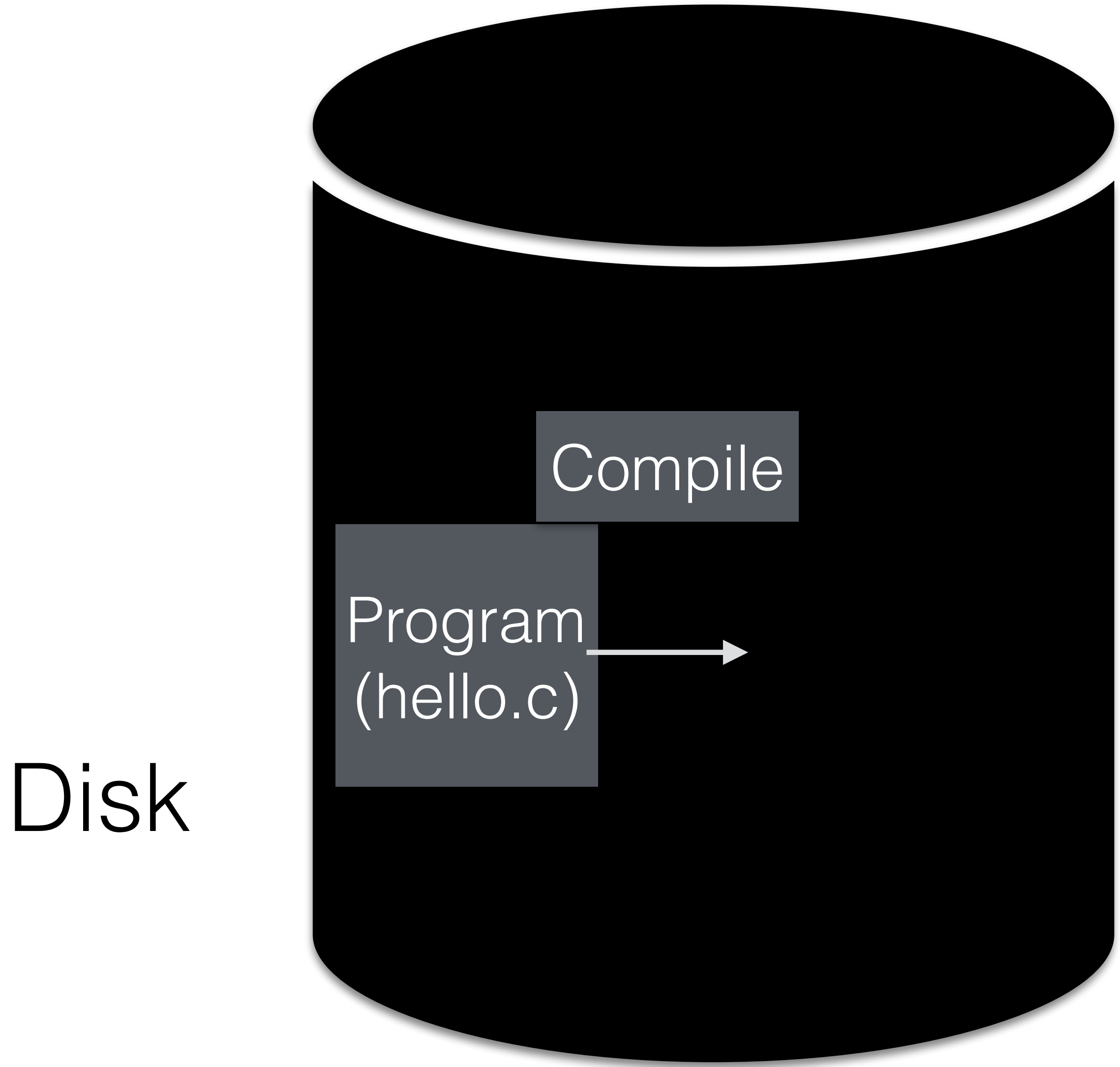
# Process Execution



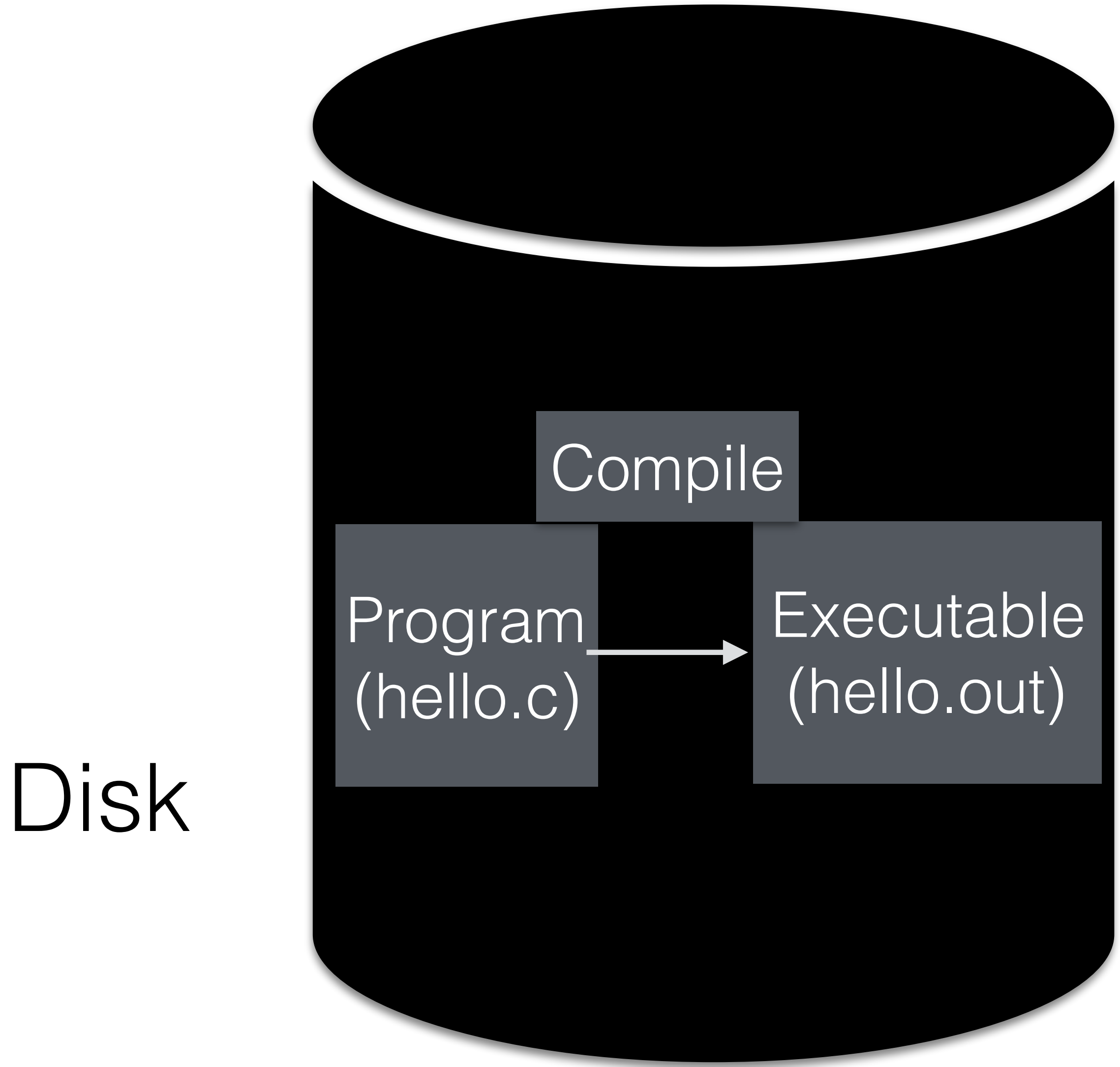
# Process Execution



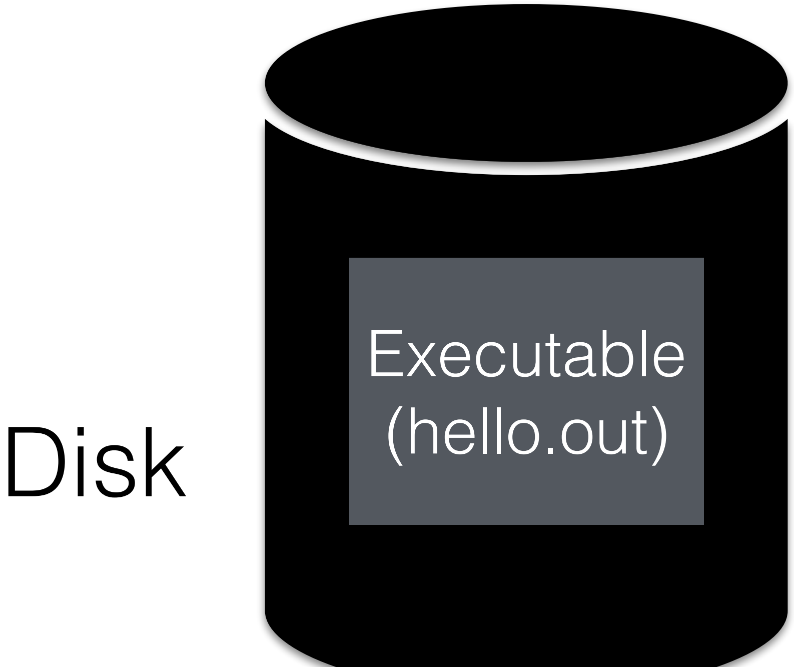
# Process Execution



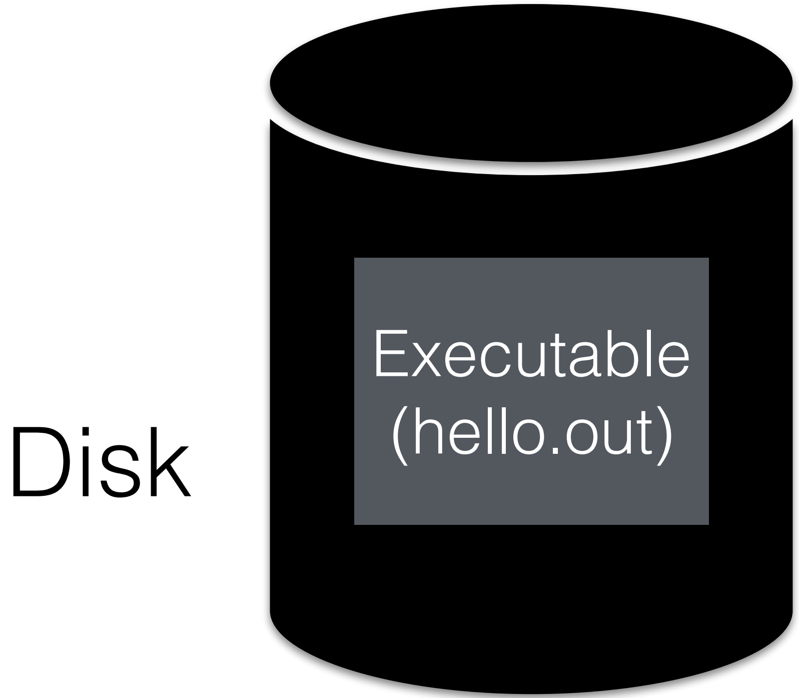
# Process Execution



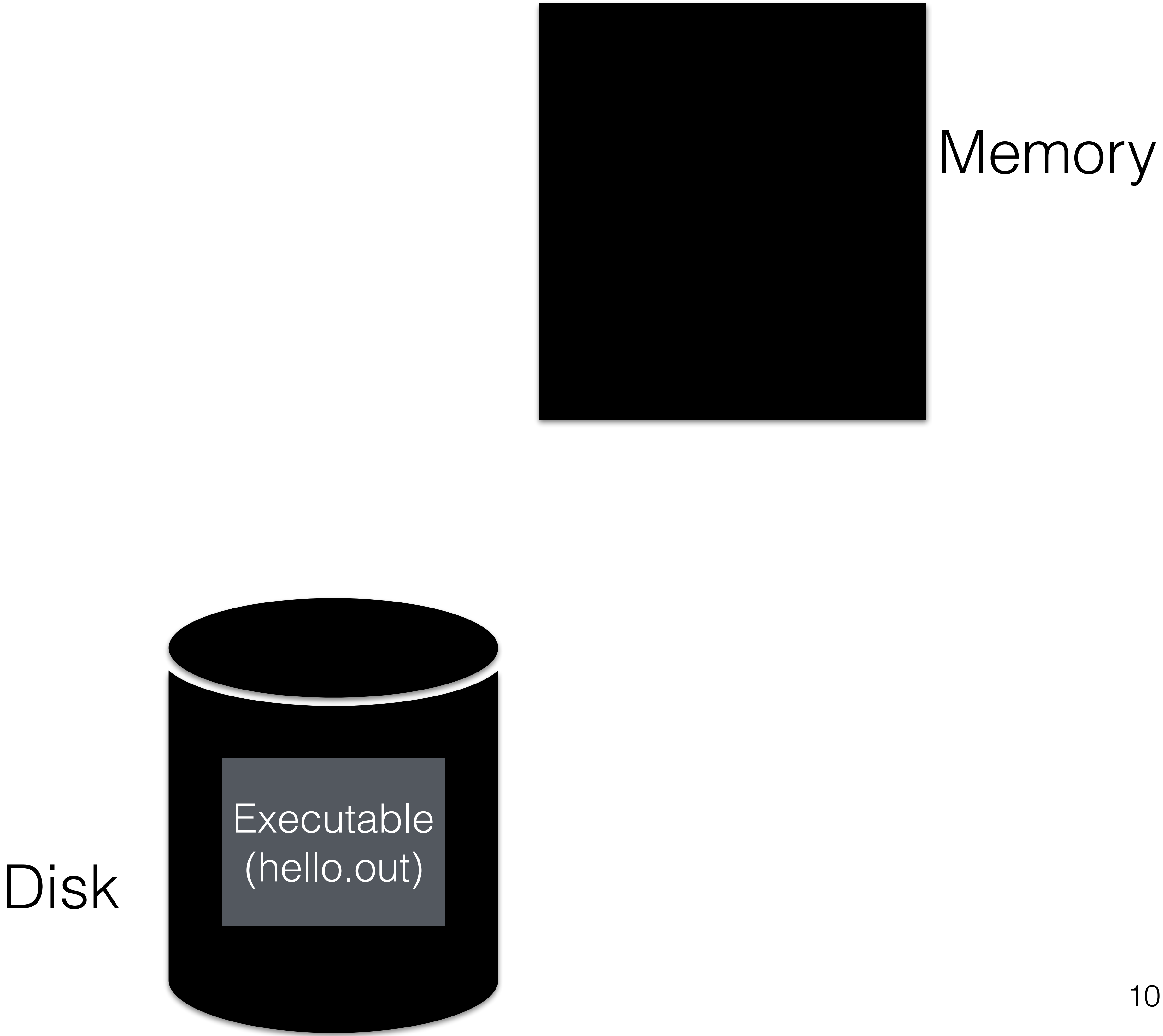
# Process Execution



# Process Execution



# Process Execution

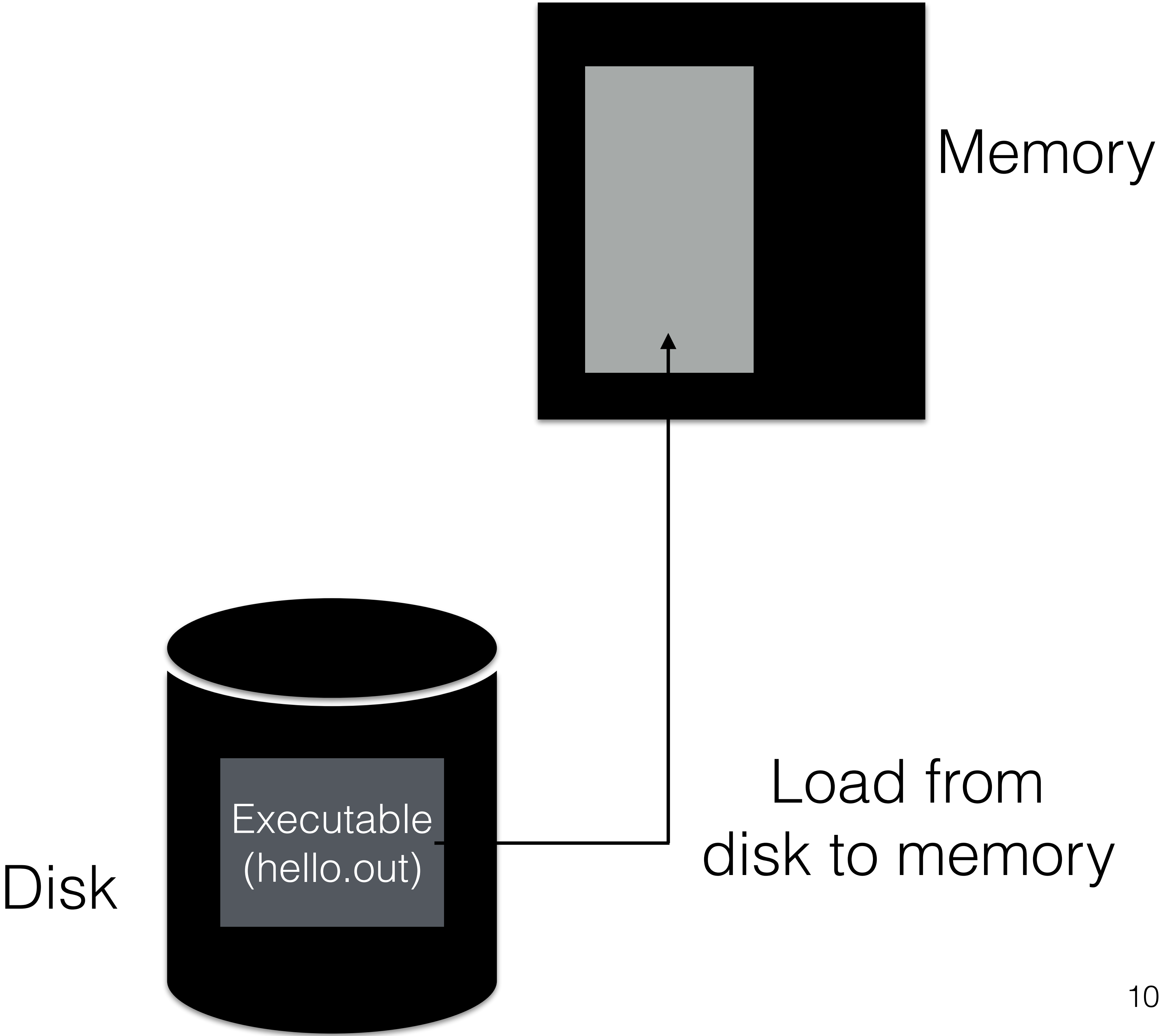


Memory

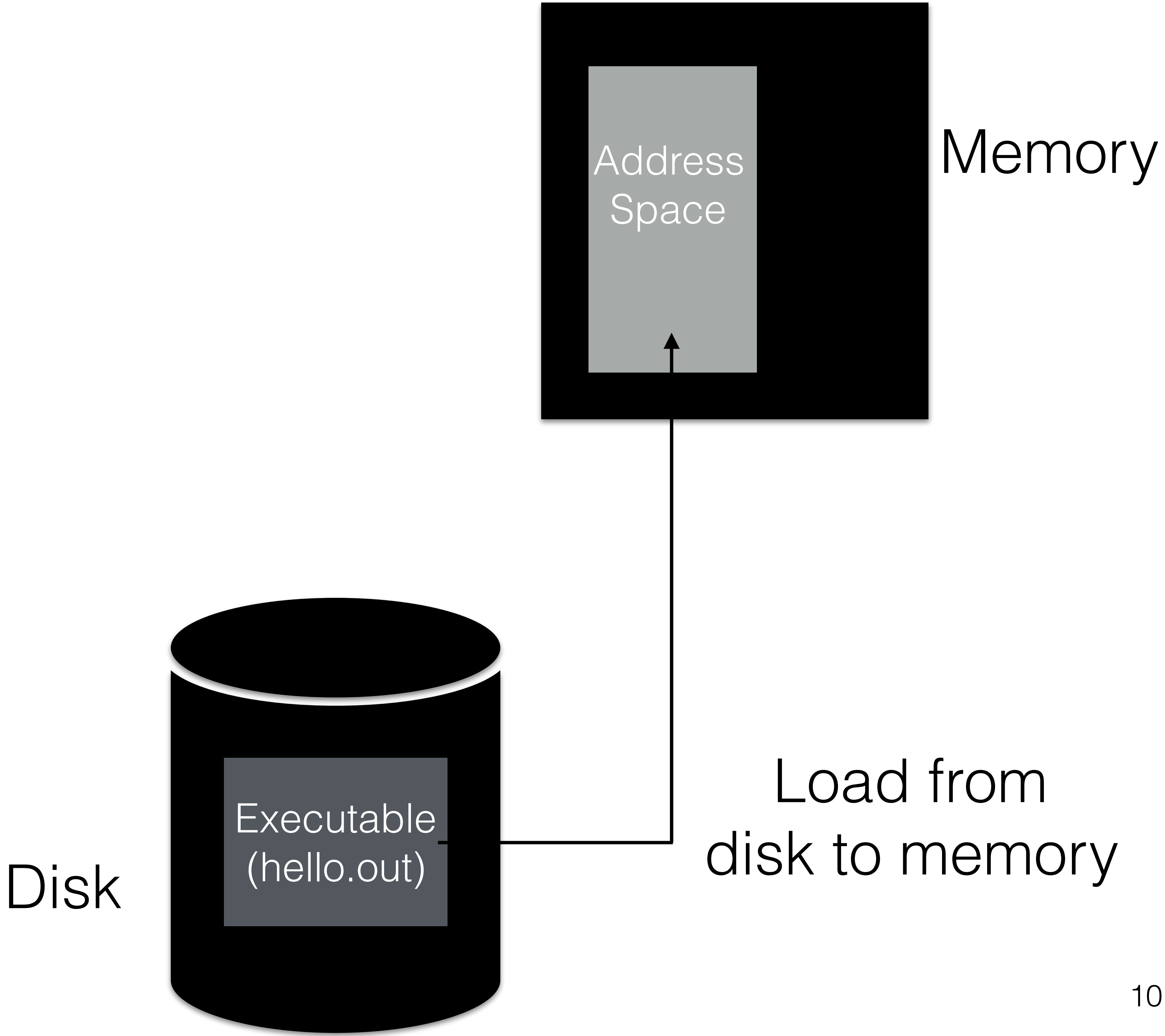
Disk

Executable
(hello.out)

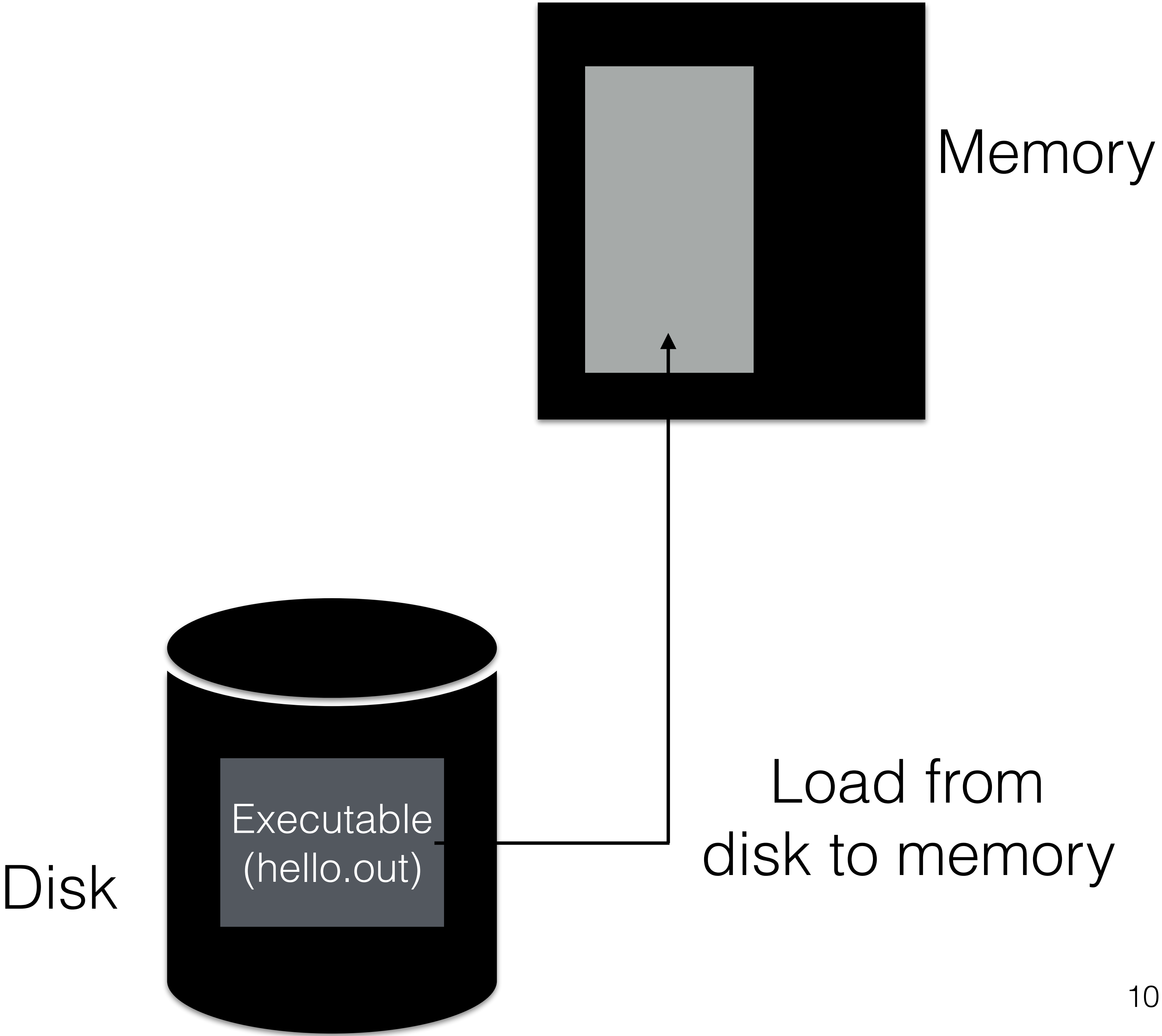
# Process Execution



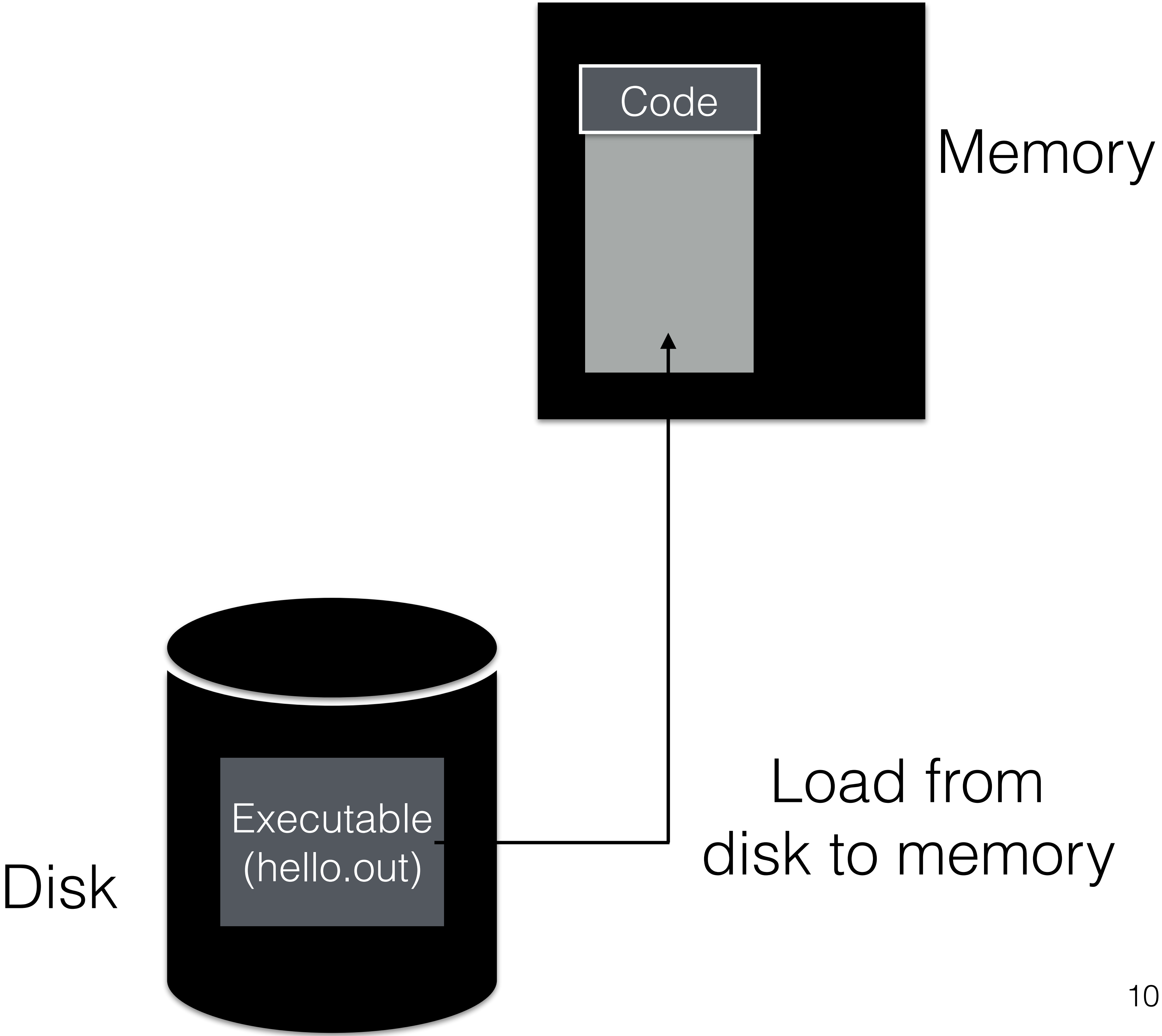
# Process Execution



# Process Execution

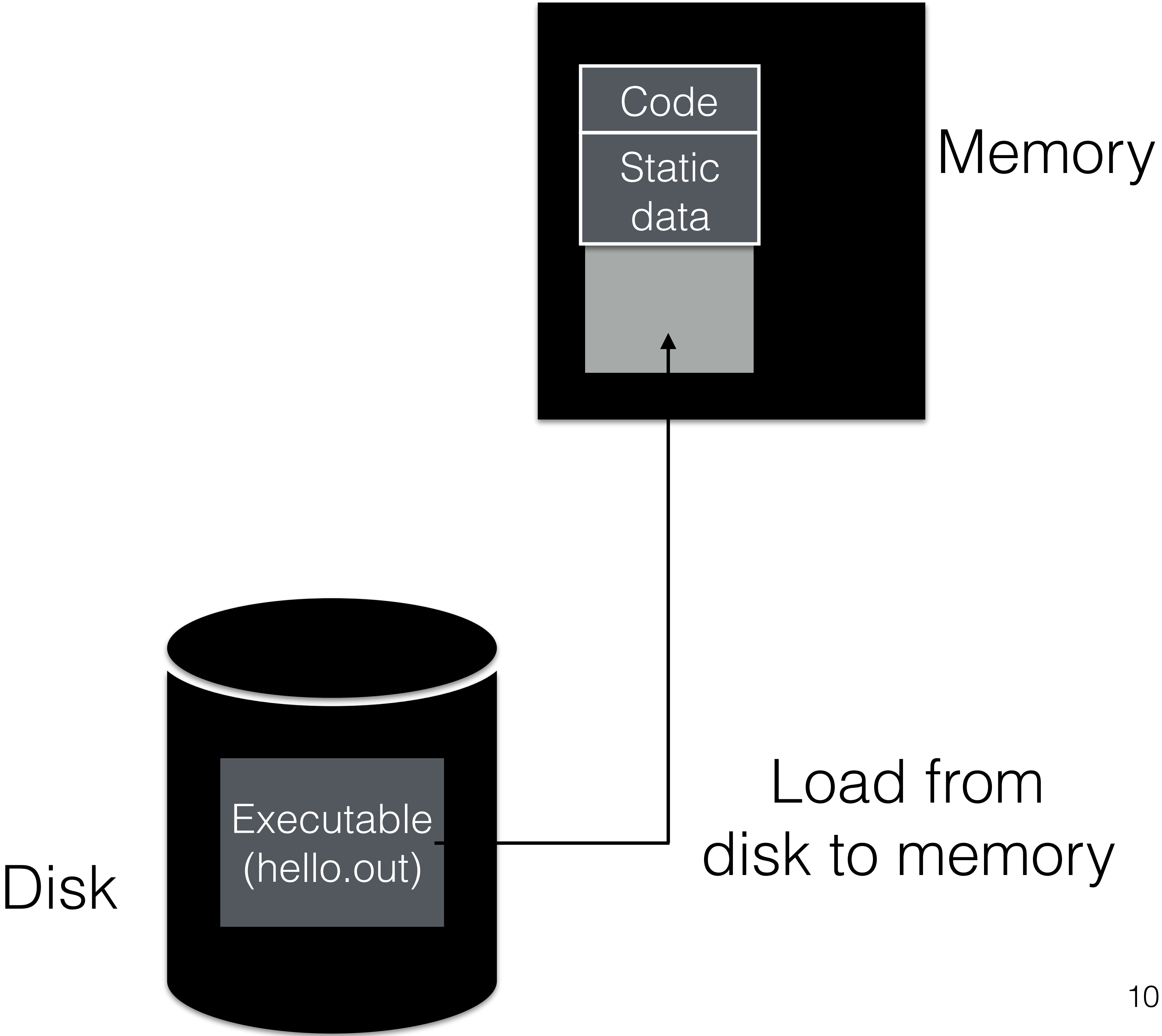


# Process Execution

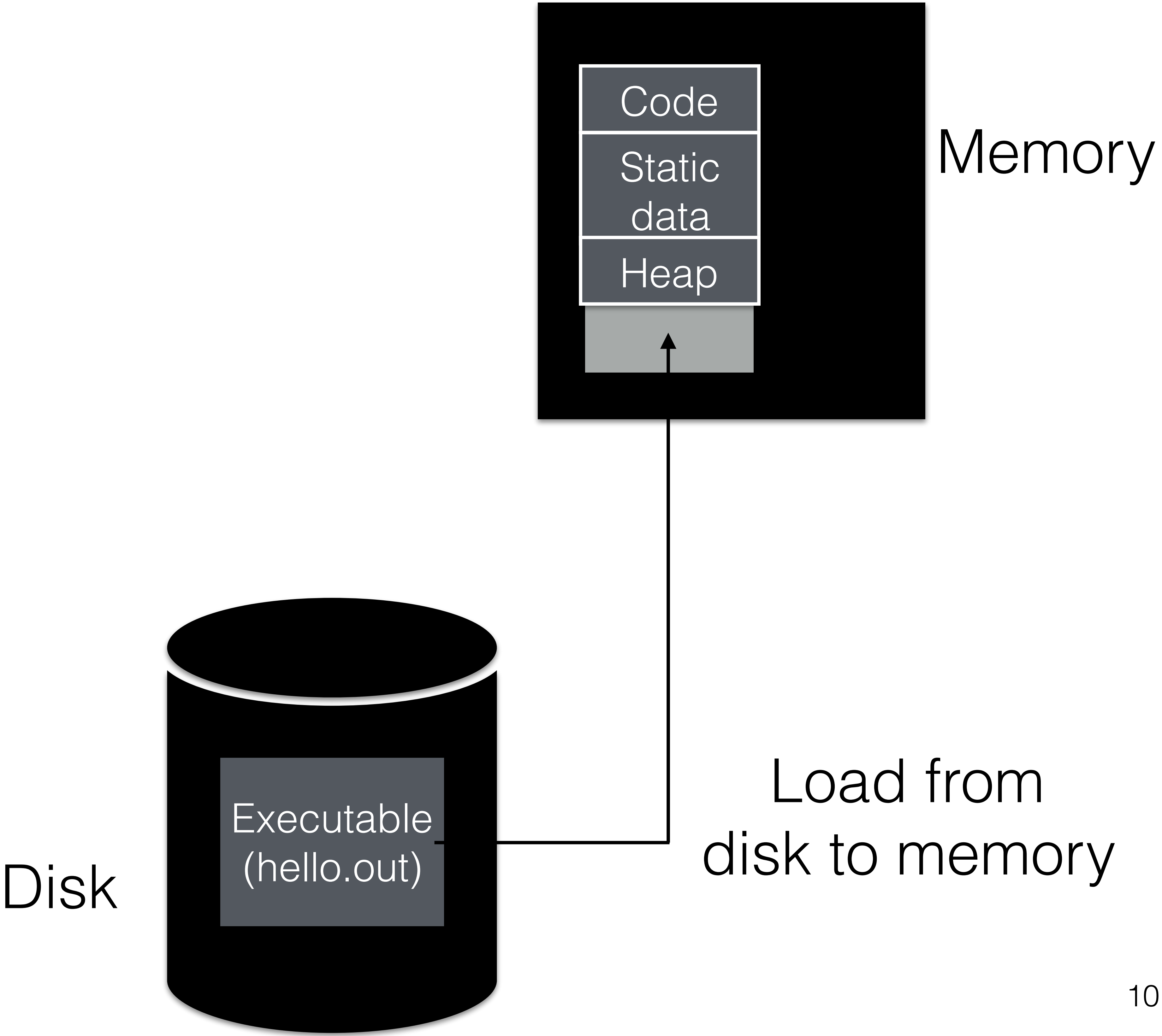


Load from disk to memory

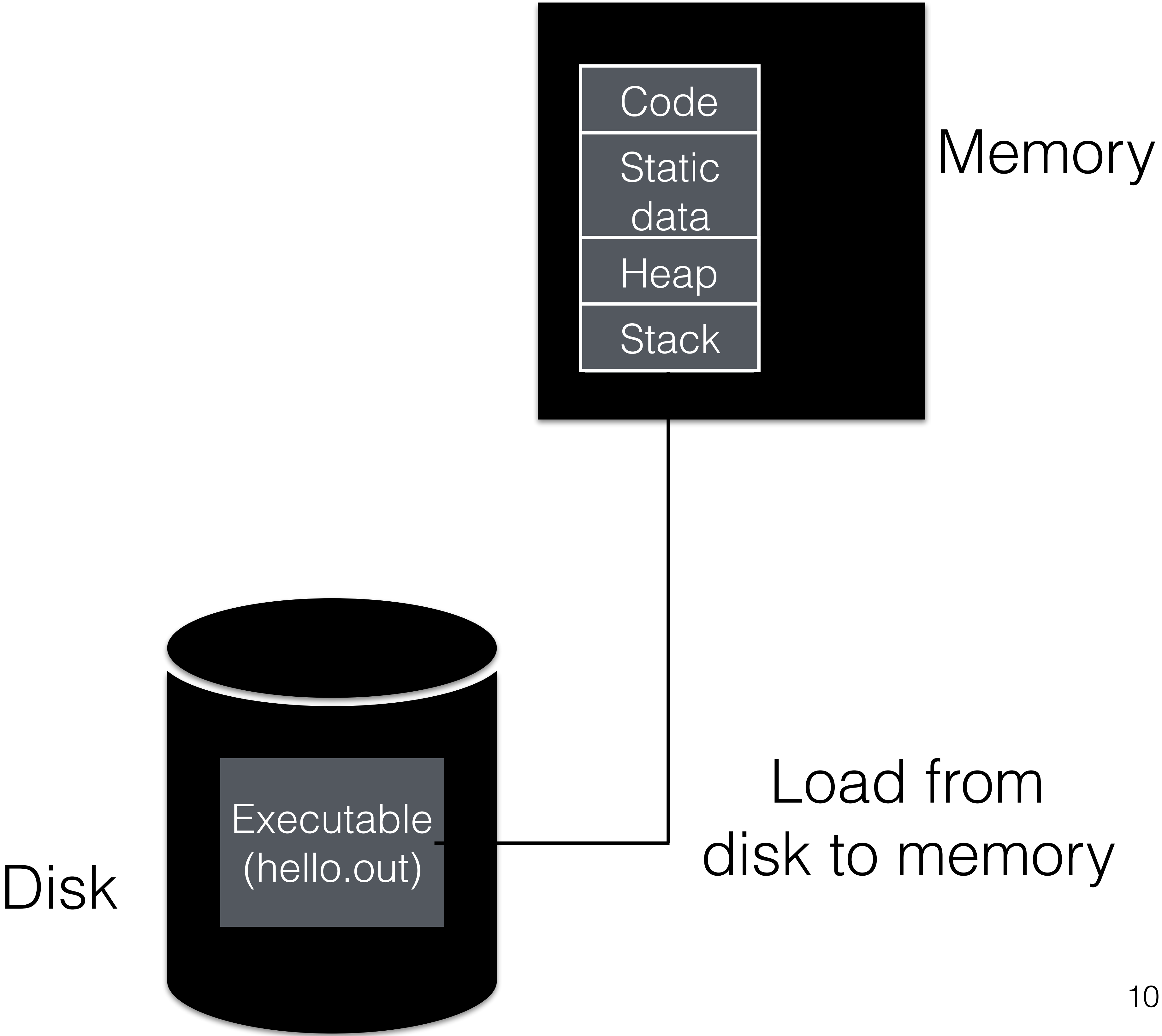
# Process Execution



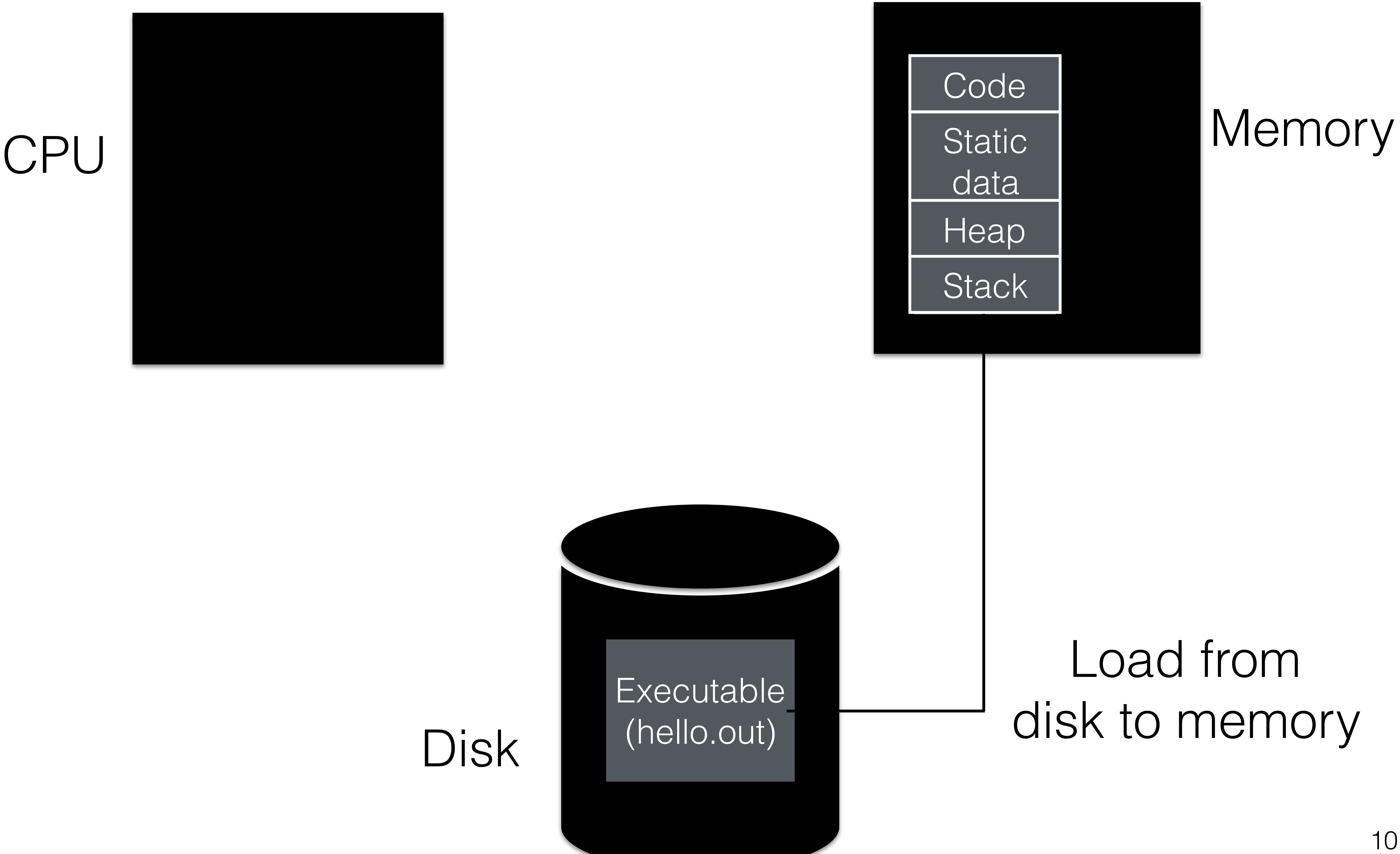
# Process Execution



# Process Execution

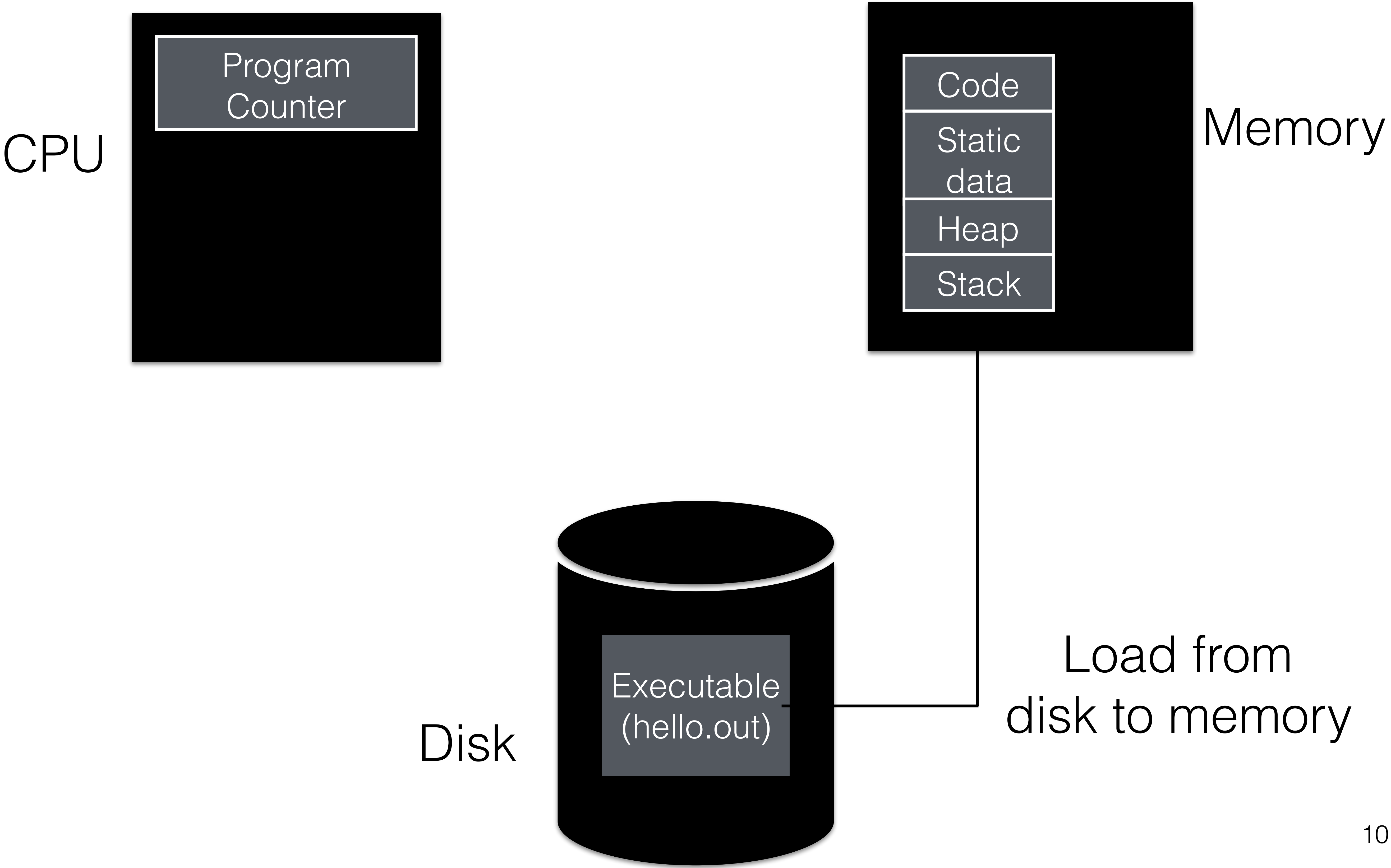


# Process Execution



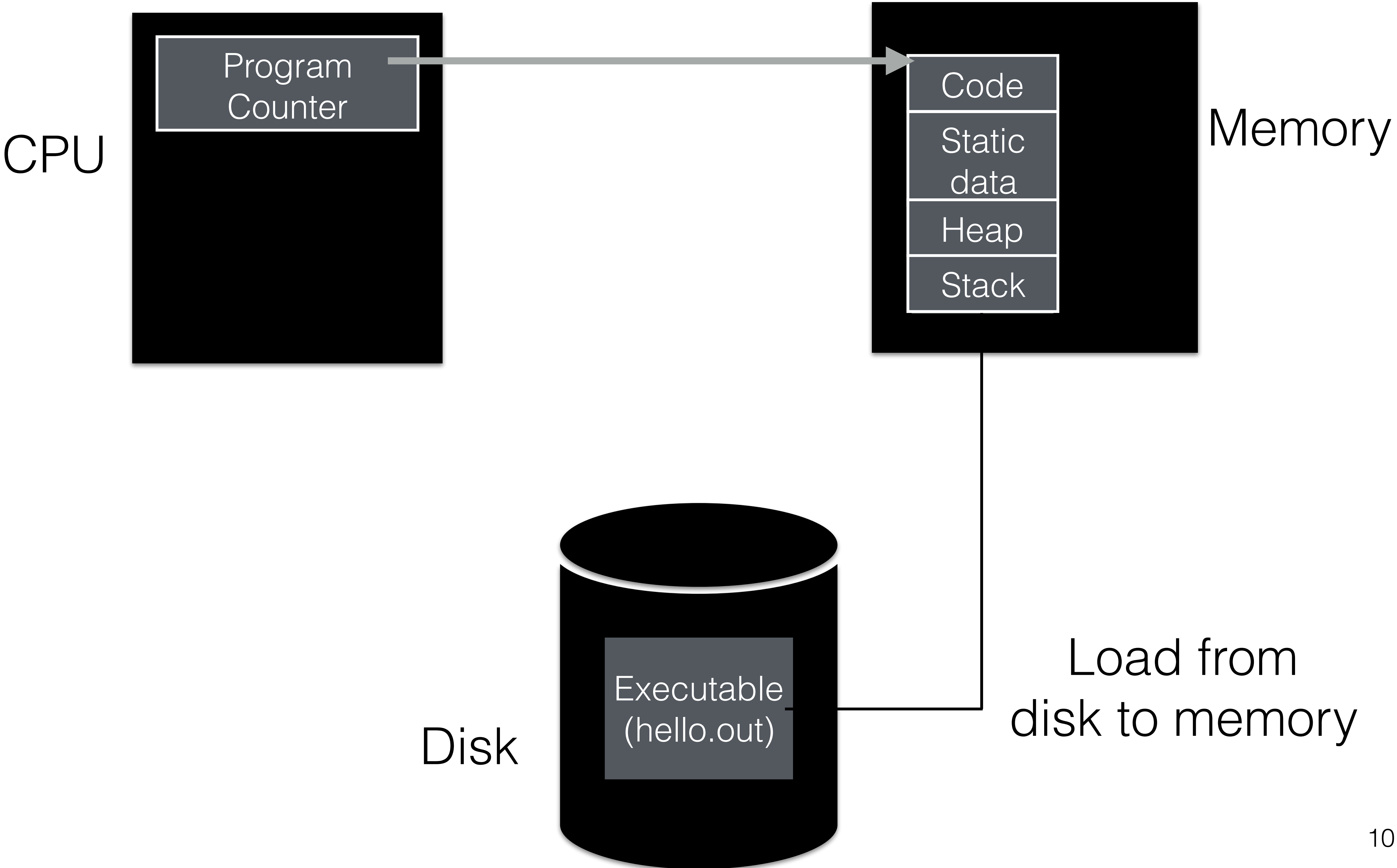
Load from disk to memory

# Process Execution

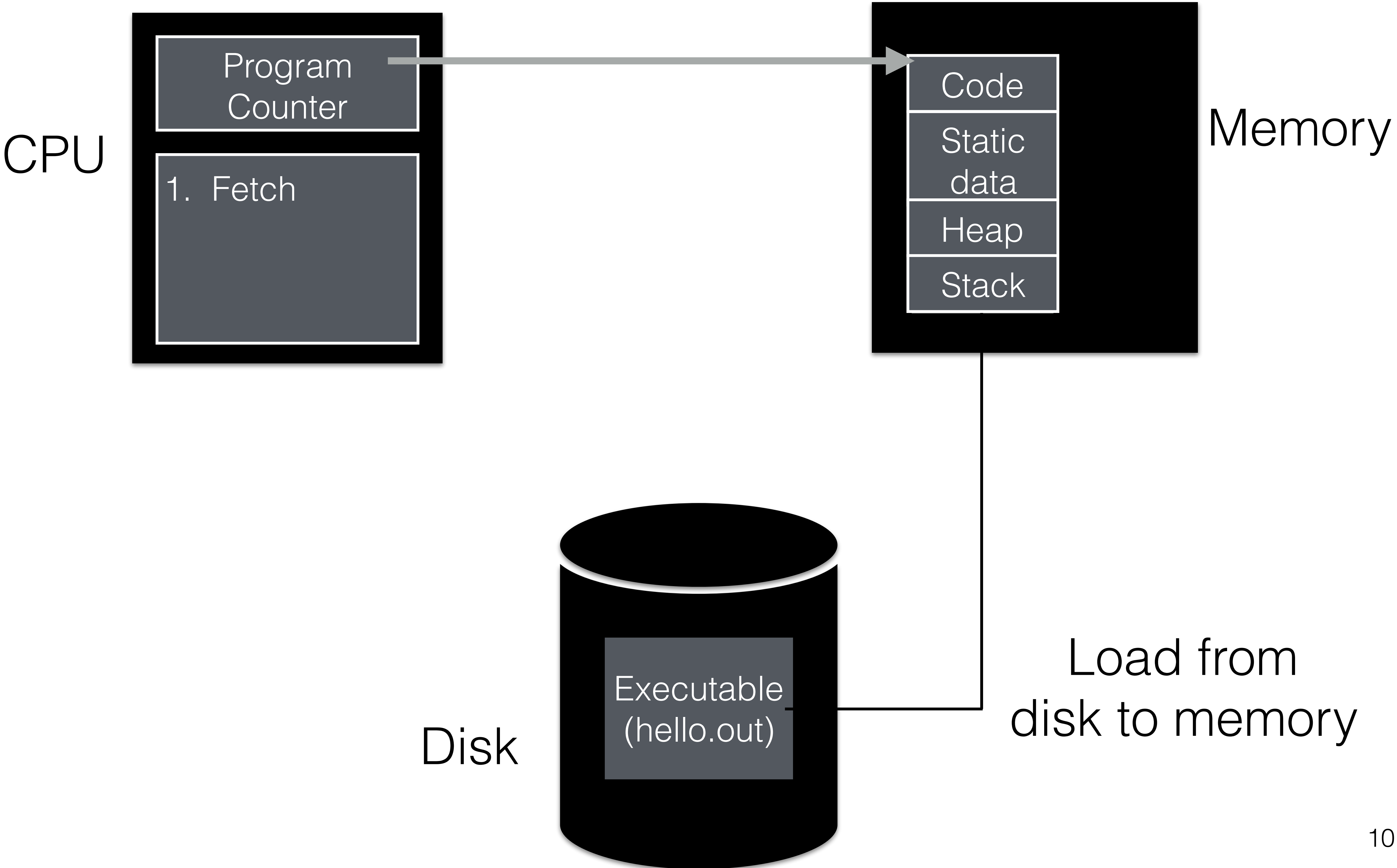


Load from disk to memory

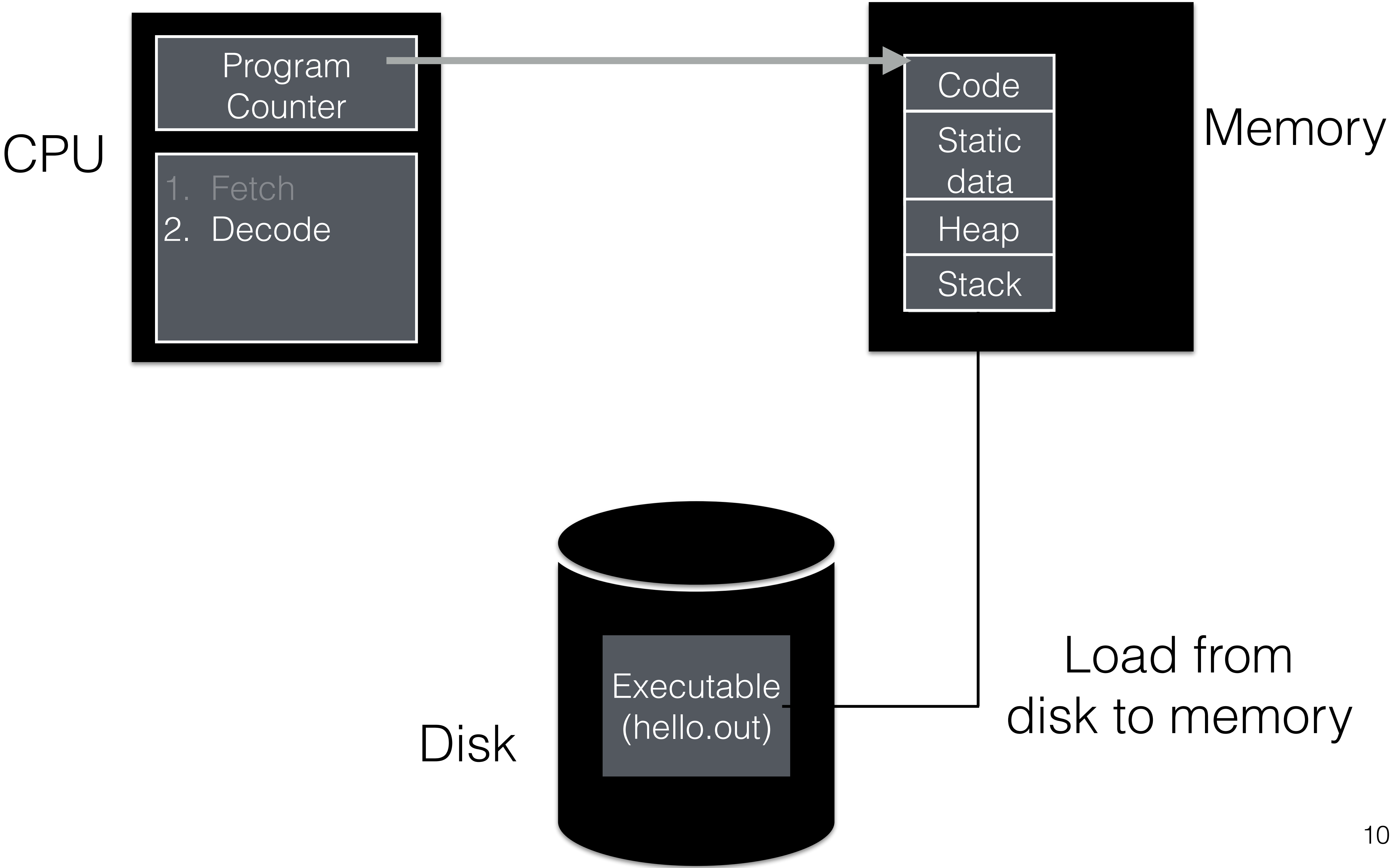
# Process Execution



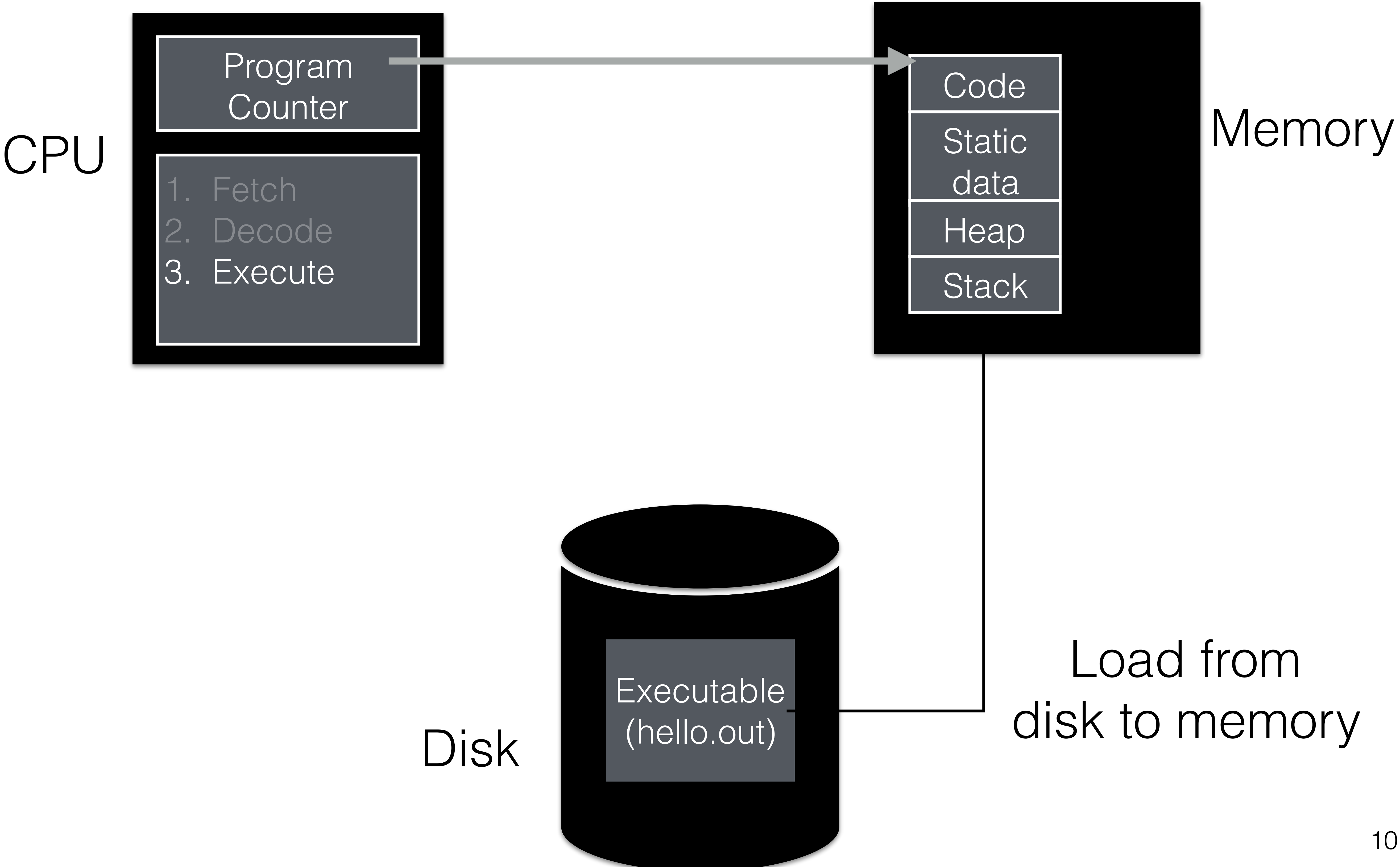
# Process Execution



# Process Execution

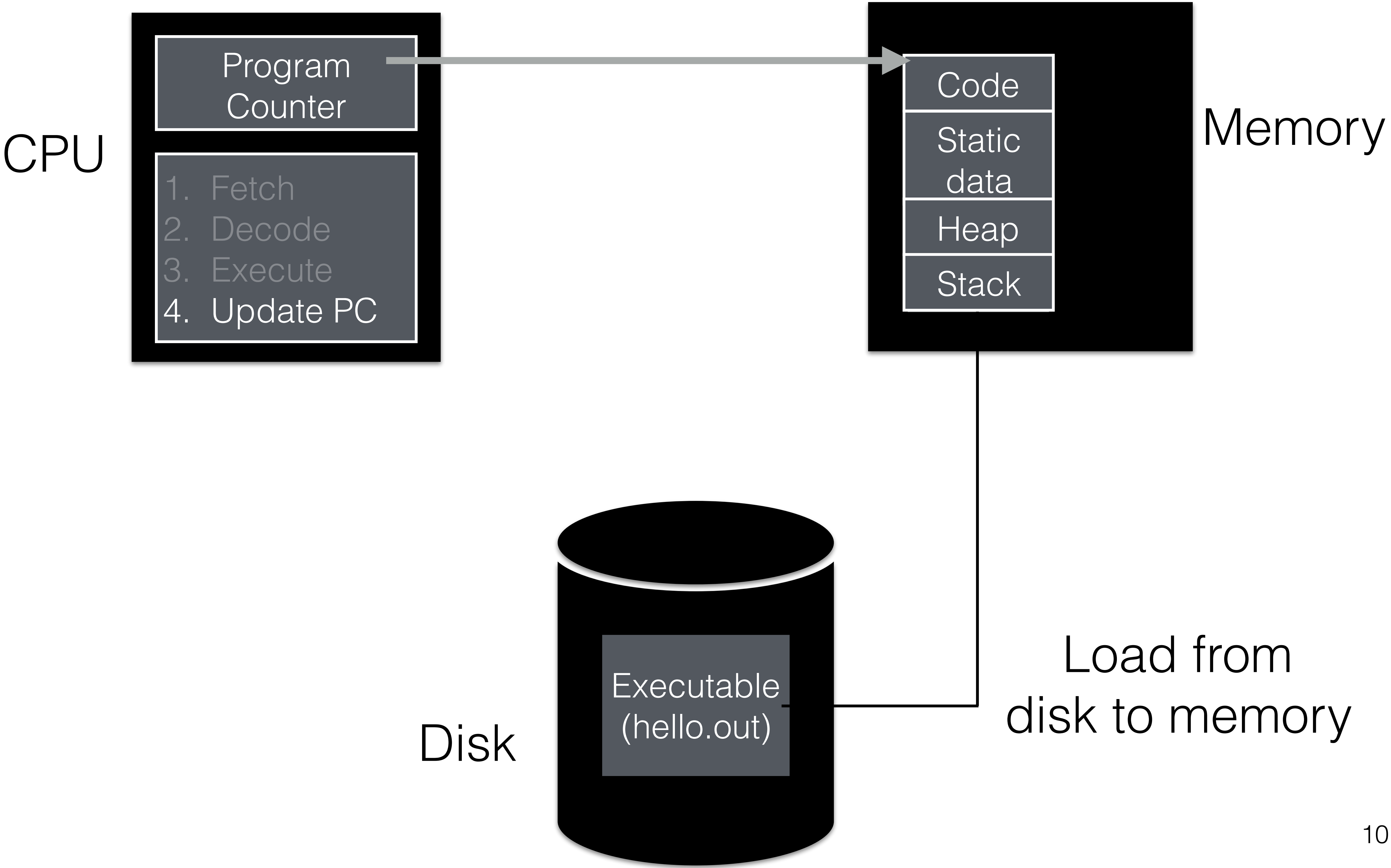


# Process Execution

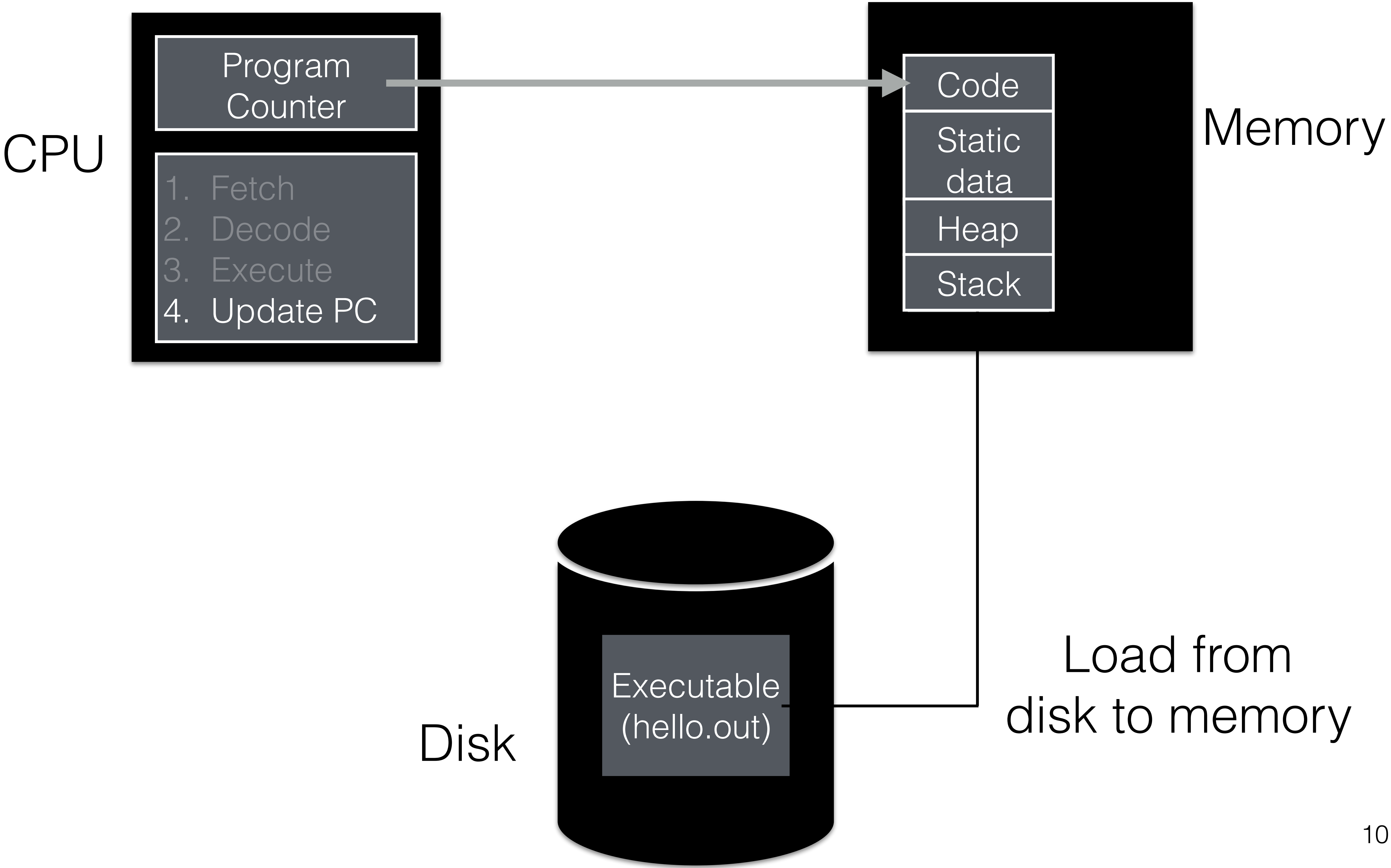


Load from disk to memory

# Process Execution



# Process Execution



Load from disk to memory

# CPU Virtualisation

# CPU Virtualisation

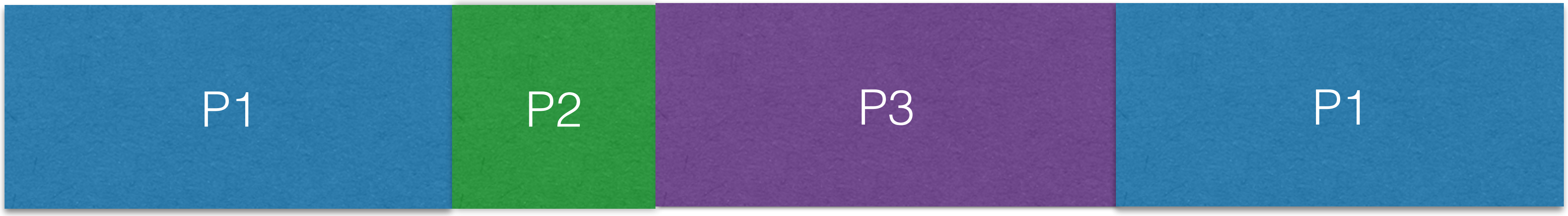
- Goal: Provide an illusion of many CPUs

# CPU Virtualisation

- Goal: Provide an illusion of many CPUs
- How: Time sharing between processes

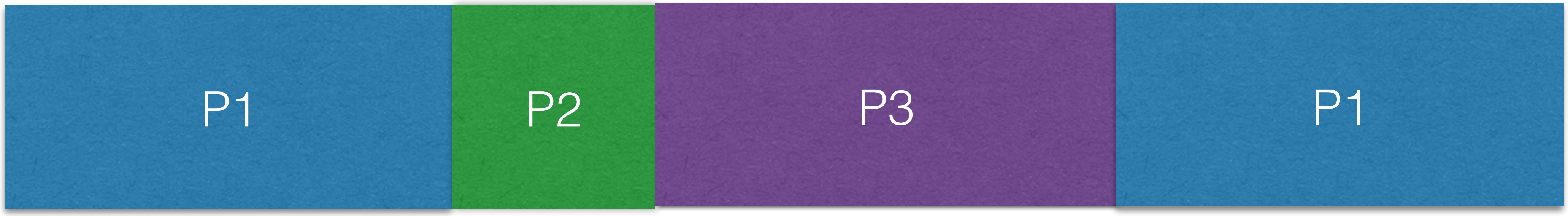
# CPU Virtualisation

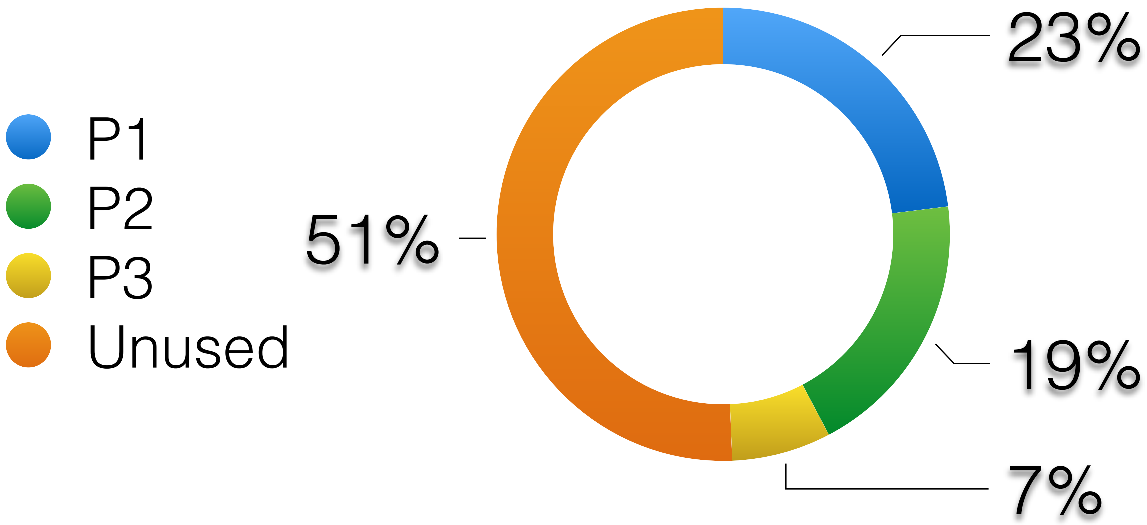
- Goal: Provide an illusion of many CPUs
- How: Time sharing between processes



# CPU Virtualisation

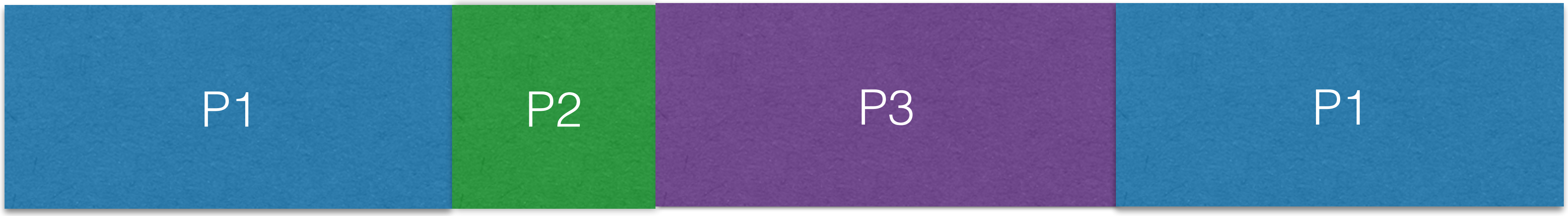
- Goal: Provide an illusion of many CPUs
- How: Time sharing between processes

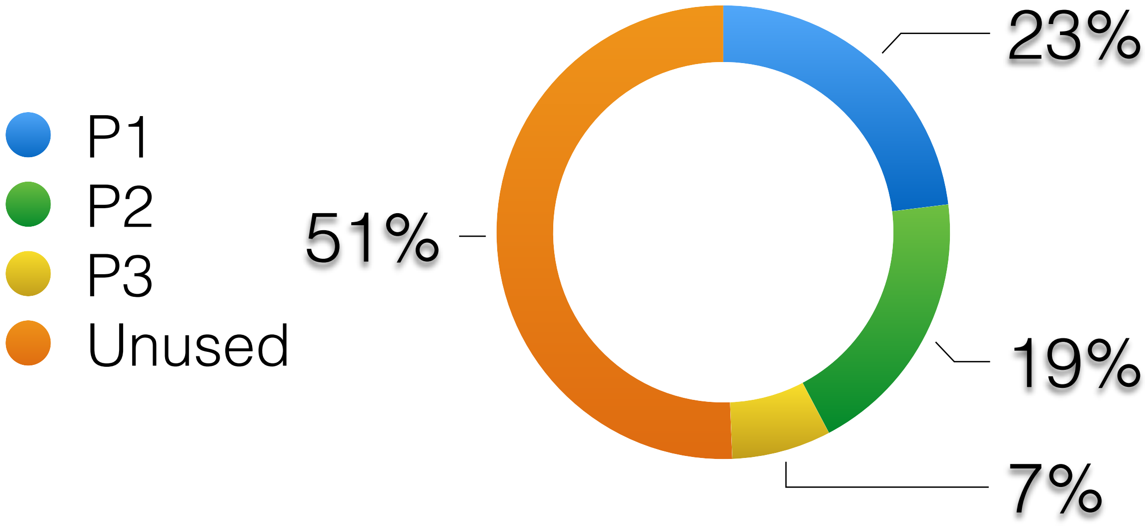




# CPU Virtualisation

- Goal: Provide an illusion of many CPUs
- How: Time sharing between processes





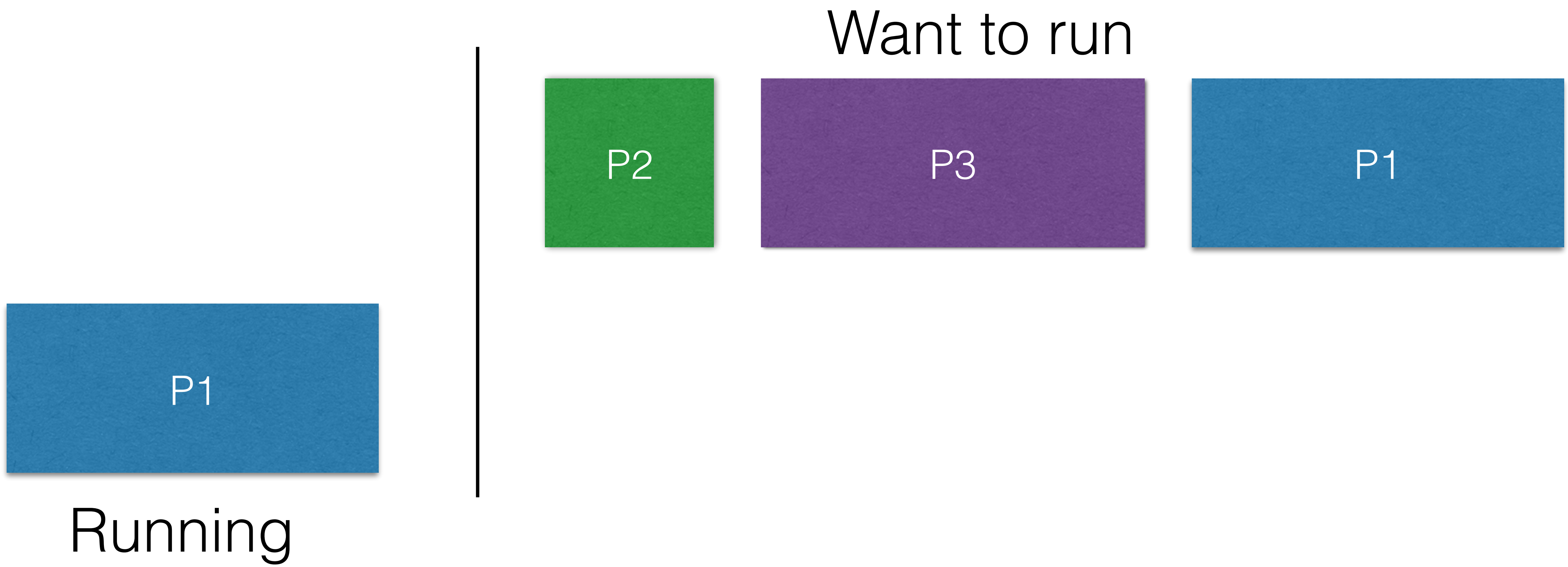
Space sharing for memory

# CPU Virtualisation II

P1

Running

# CPU Virtualisation II



Want to run

P2

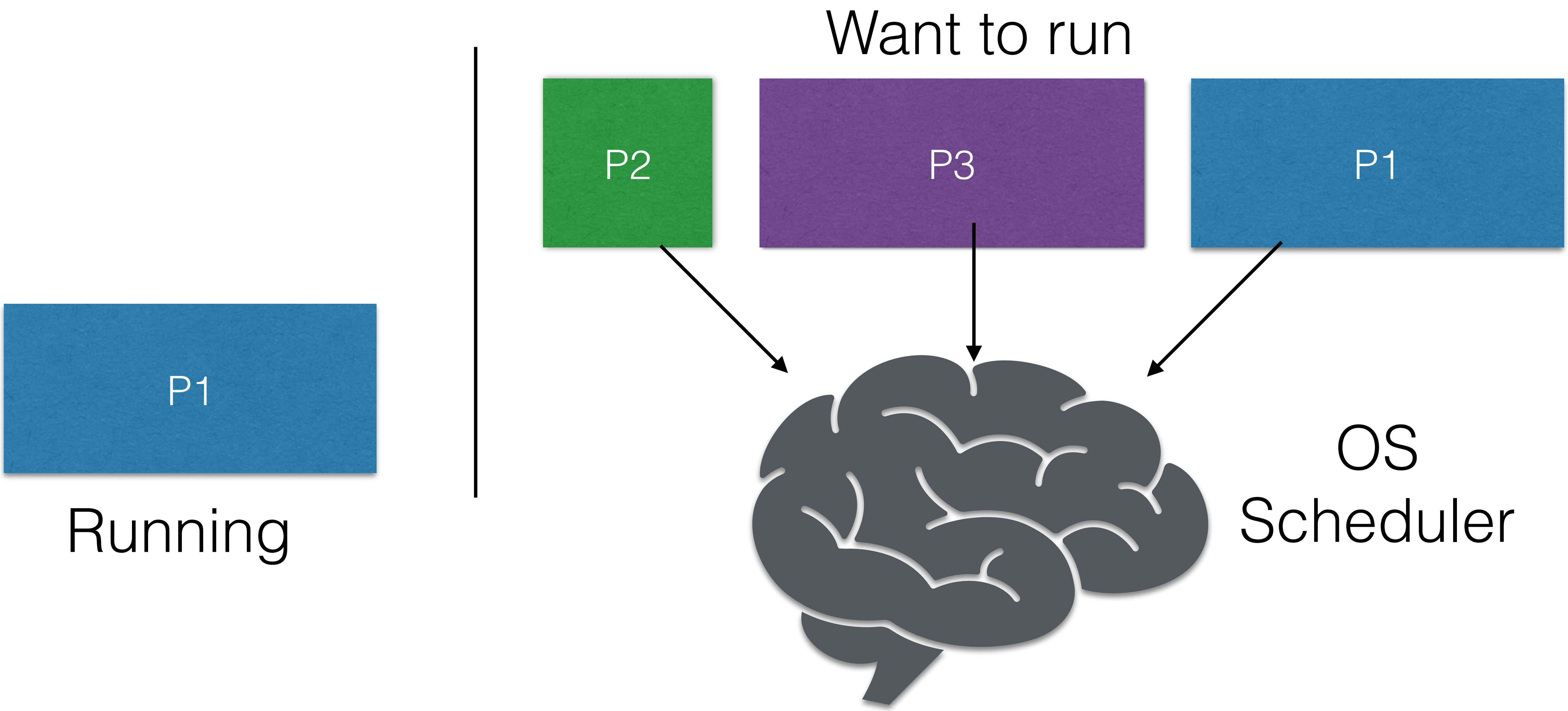
P3

P1

P1

Running

# CPU Virtualisation II



Want to run

P2

P3

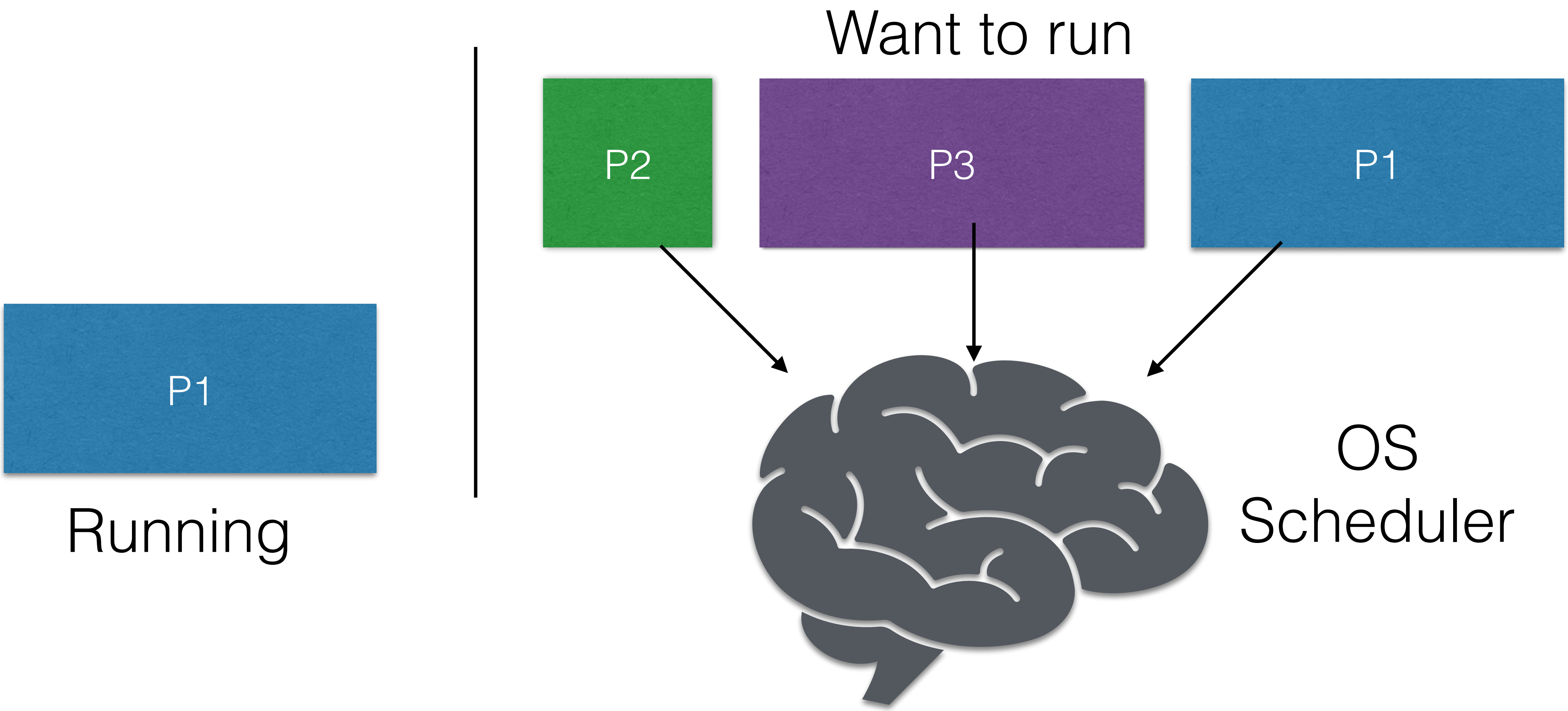
P1

OS Scheduler

P1

Running

# CPU Virtualisation II



Want to run

P2

P3

P1

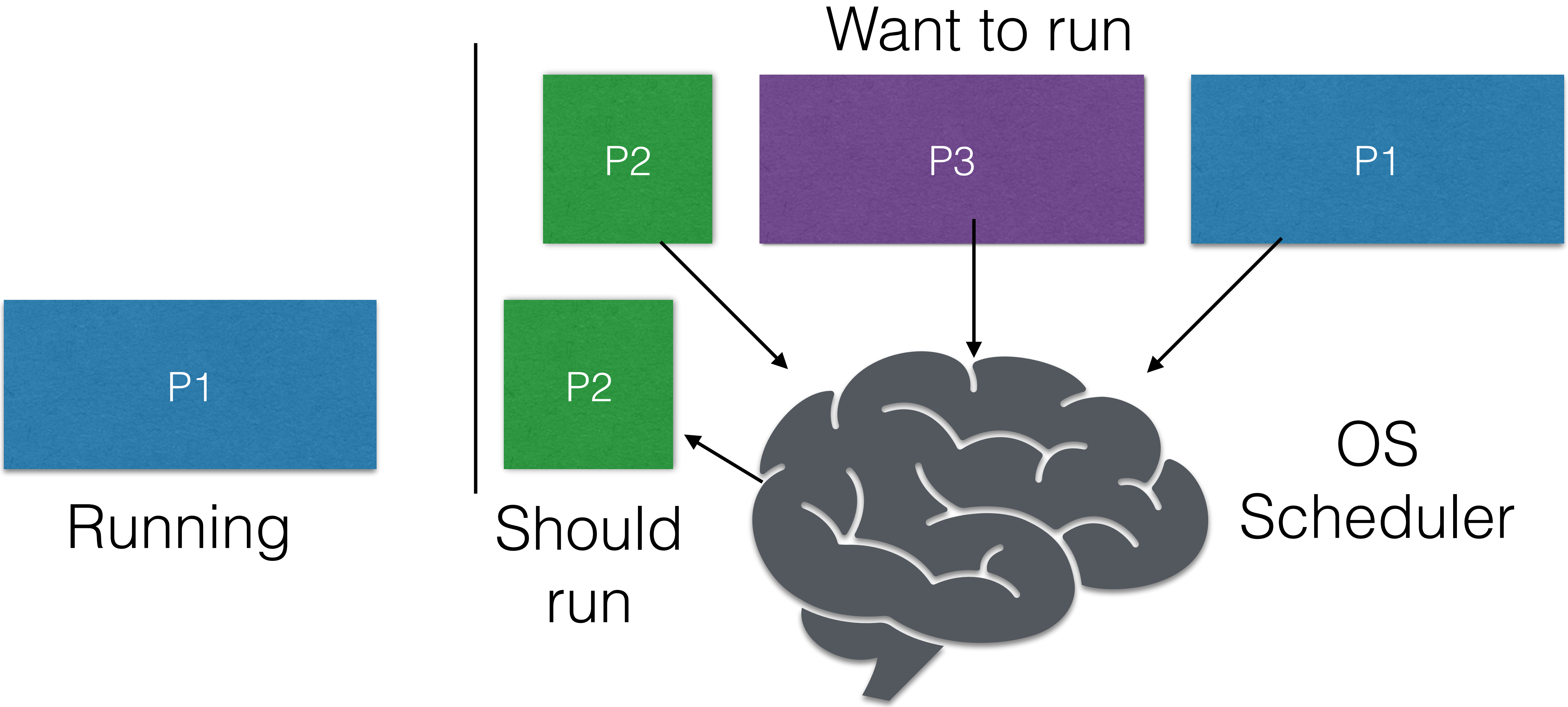
P1

Running

OS Scheduler

Next P = f(run time, metric, type of process, …)

# CPU Virtualisation II



Want to run

P2

P3

P1

P1

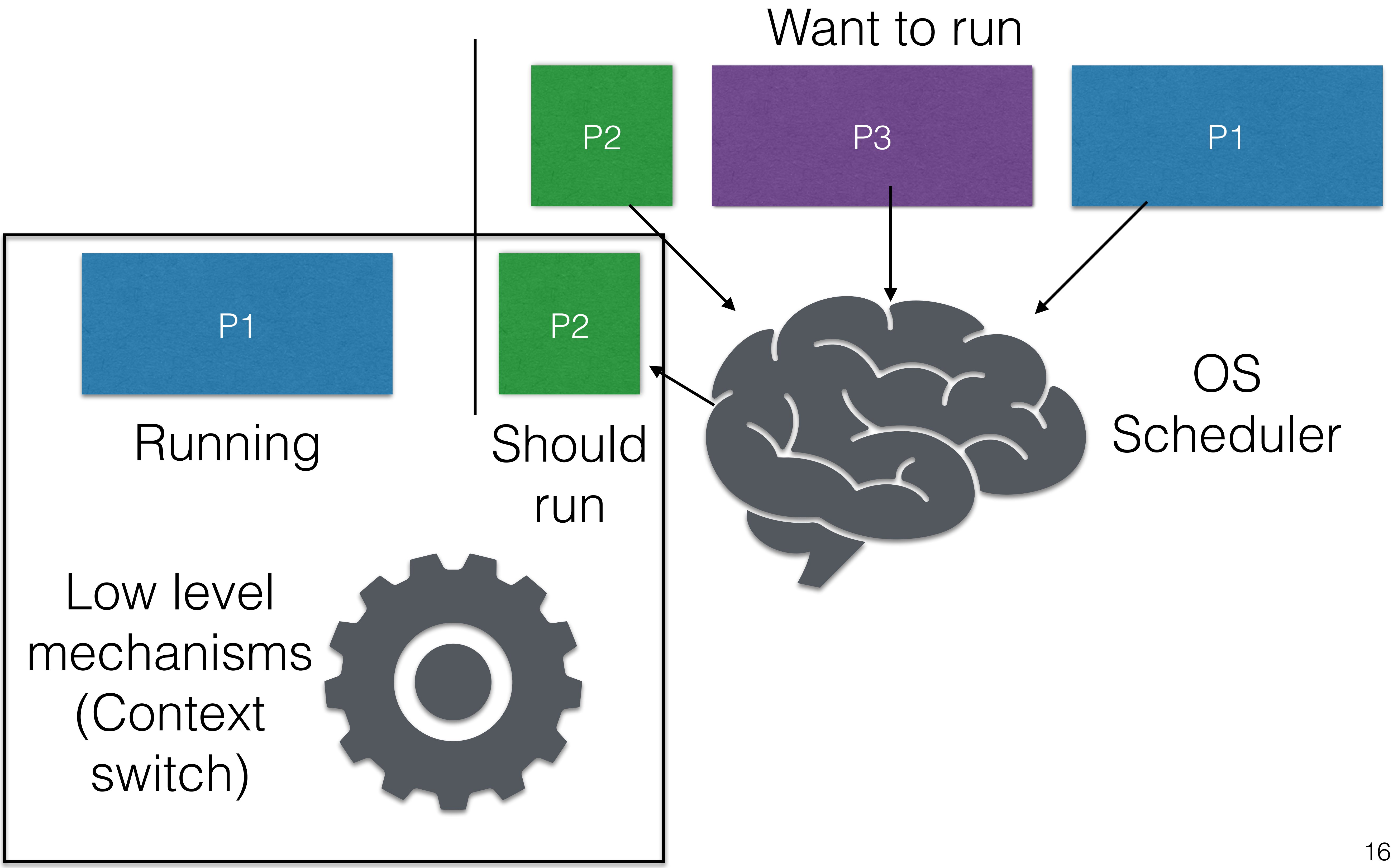
Running

P2

Should run

OS Scheduler

# CPU Virtualisation II



Want to run

P2

P3

P1

P1

Running

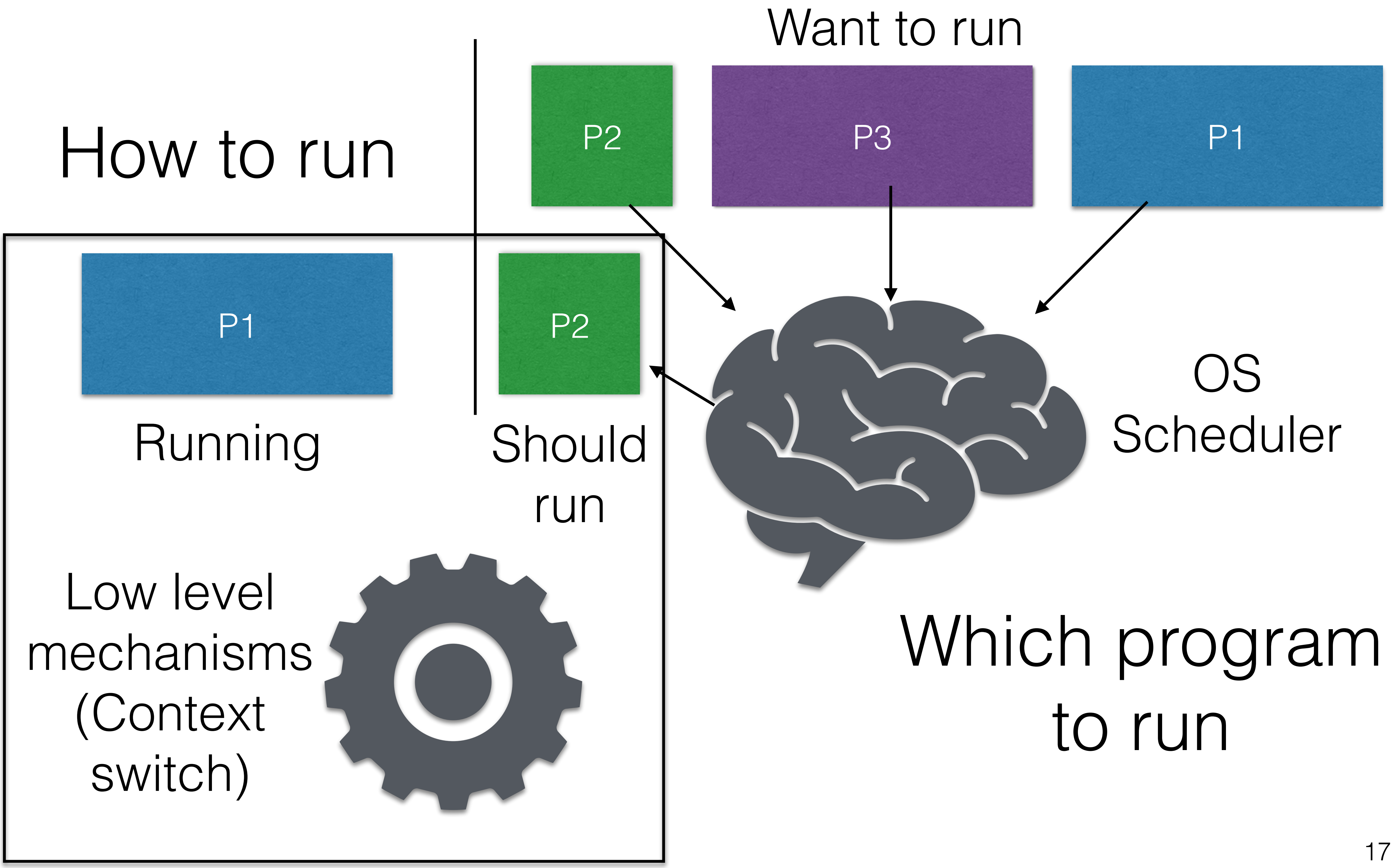
P2

Should run

OS Scheduler

Low level mechanisms (Context switch)

# CPU Virtualisation II



Want to run

P2

P3

P1

How to run

P1

Running

P2

Should run

Low level mechanisms (Context switch)

OS Scheduler

Which program to run

# Process API

# Process API

- Create process:

# Process API

- Create process:
  - Double click

# Process API

- Create process:
  - Double click
  - Run on command line

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line
- Wait:

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line
- Wait:
  - Don’t run process till other process completes

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line
- Wait:
  - Don’t run process till other process completes
- Status:

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line
- Wait:
  - Don’t run process till other process completes
- Status:
  - How long run, what state it is in

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line
- Wait:
  - Don’t run process till other process completes
- Status:
  - How long run, what state it is in
  - Does **top, ps** give us this info?

# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line
- Wait:
  - Don’t run process till other process completes
- Status:
  - How long run, what state it is in
  - Does **top, ps** give us this info?
- Misc.:

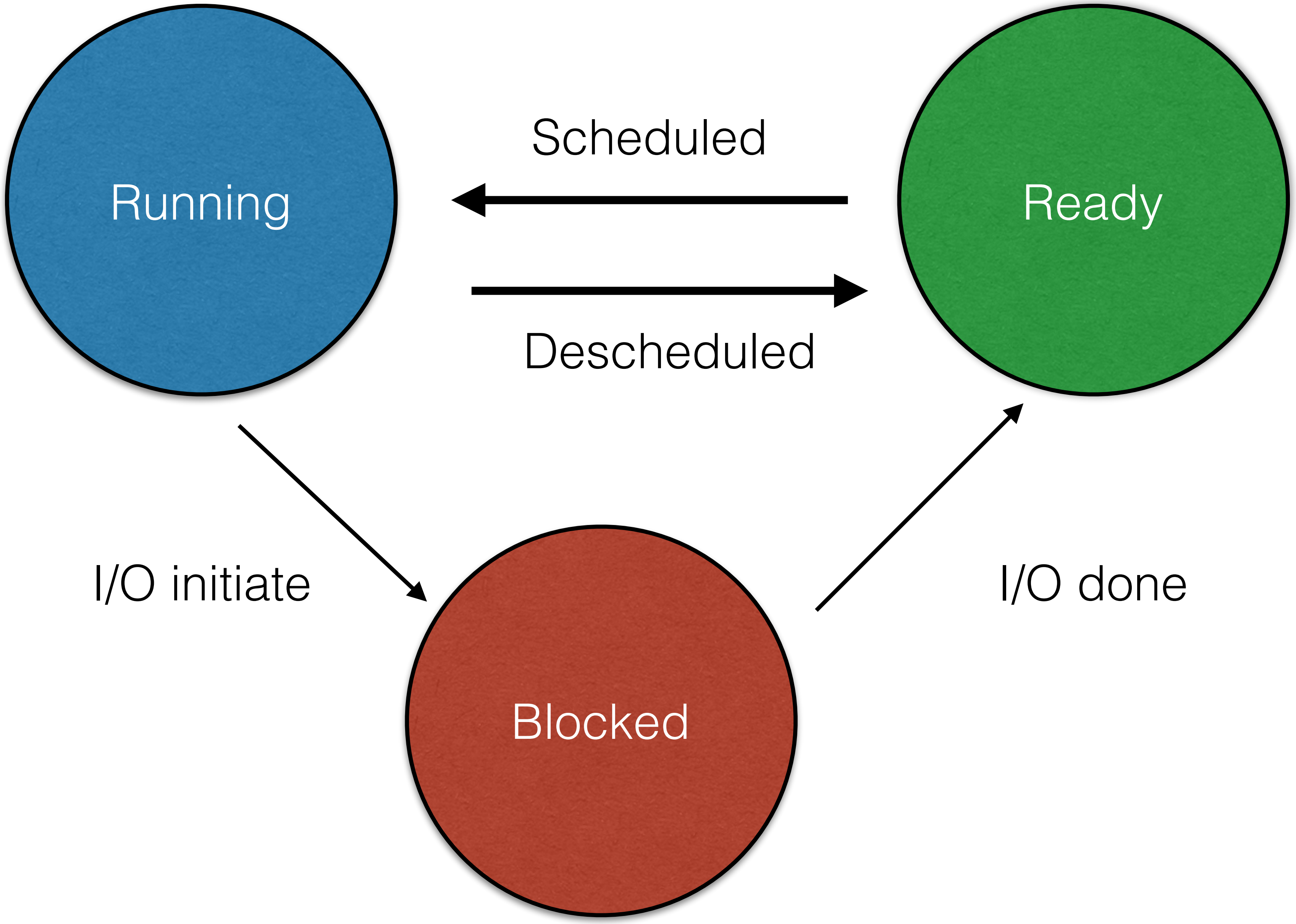
# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line
- Wait:
  - Don't run process till other process completes
- Status:
  - How long run, what state it is in
  - Does **top, ps** give us this info?
- Misc.:
  - Suspend

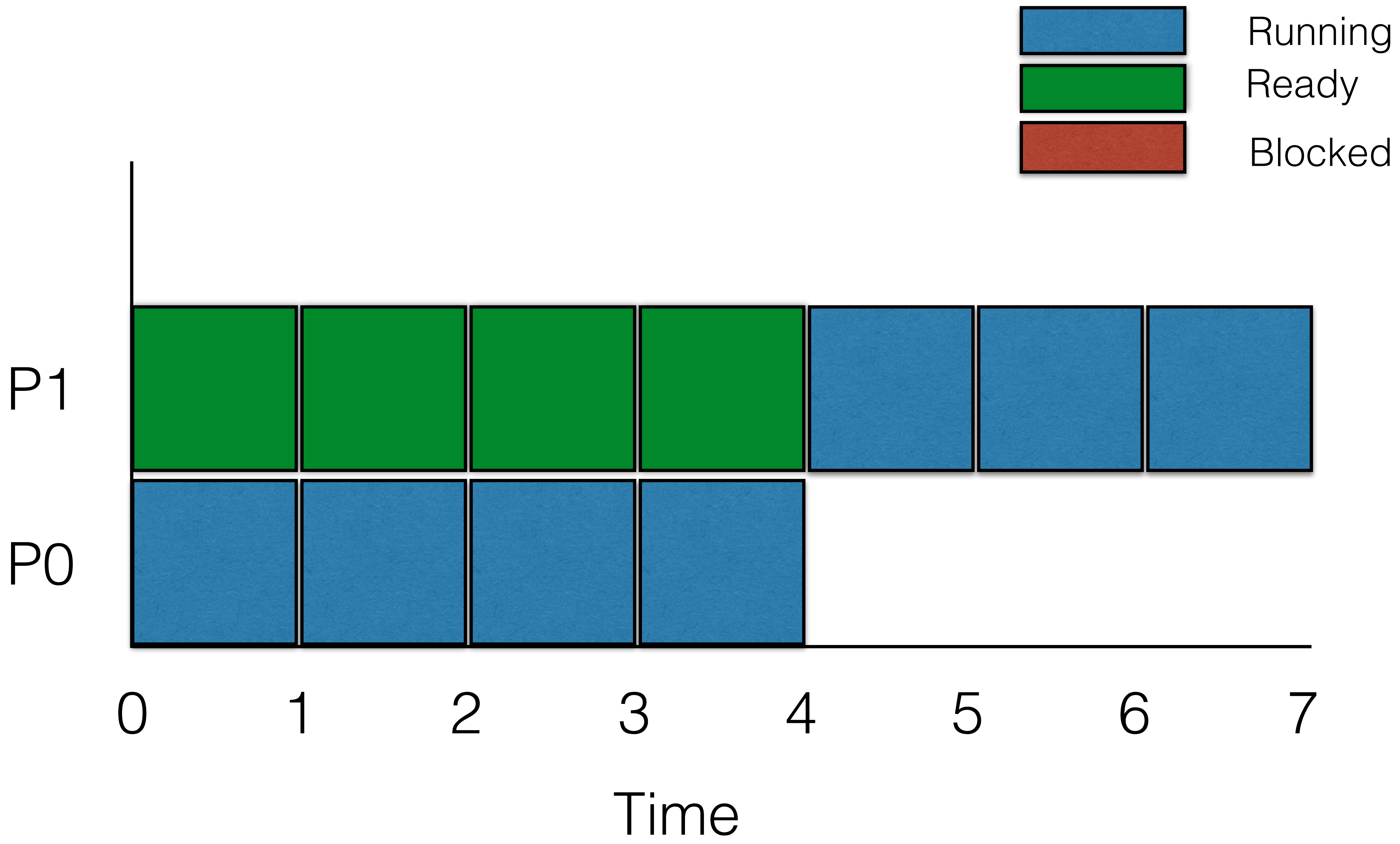
# Process API

- Create process:
  - Double click
  - Run on command line
- Destroy processes:
  - Task manager
  - Command line
- Wait:
  - Don’t run process till other process completes
- Status:
  - How long run, what state it is in
  - Does **top, ps** give us this info?
- Misc.:
  - Suspend

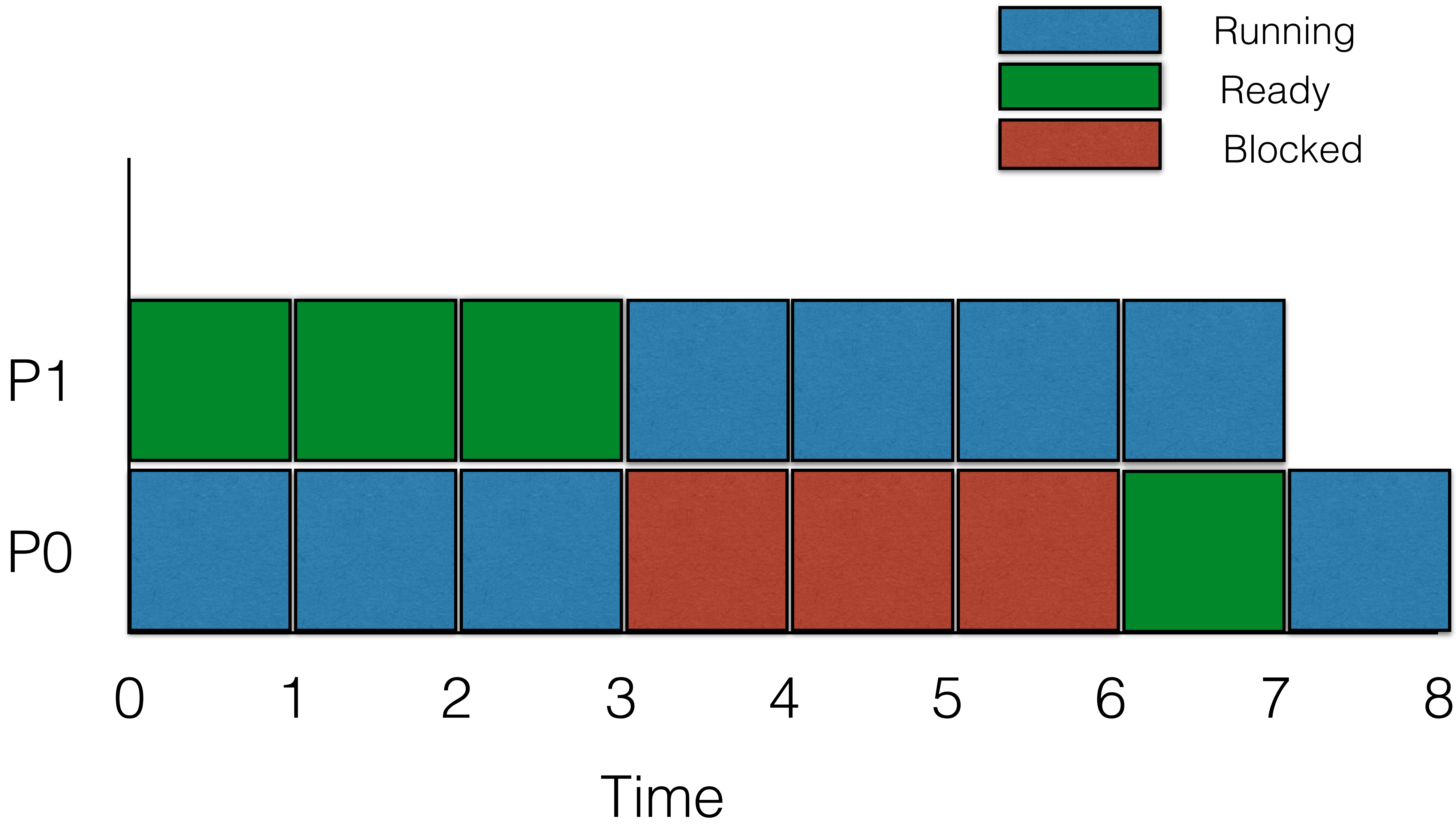
# Process States



# Process States



# Process States


Running
Ready
Blocked
P1
P0
Time