Processes and Threads

DS 5110: Big Data Systems

Spring 2025

Lecture 3

Yue Cheng

University of Virginia

Some material taken/derived from:
- Wisconsin CS 544 by Tyler Caraza-Harter.

Learning objectives

- Describe the interaction between schedulers, CPUs, processes vs. threads, and address spaces
- Understand various basic CPU scheduling policies: FIFO, SJF (STCF), RR
  - And their pros and cons
- Use Linux commands to track running programs and manipulate the scheduling behaviors of them

Motivation

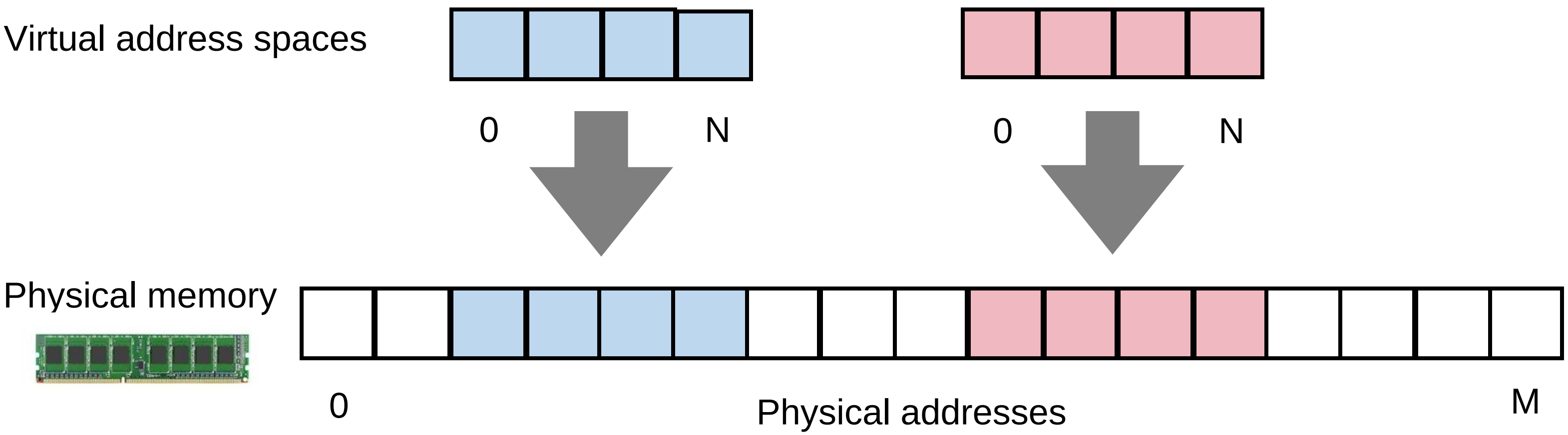
- Modern CPUs have many cores (maybe dozens)
- **Trend:** more cores rather than faster cores
- **Problem:** a simple Python program can use at most ONE core
  - Less if it accesses files or the Internet
- Understanding processes and threads will:
  - Let us write programs that fully utilize CPU resources
  - Decide the structure of our concurrent program (processes or threads) depending on the situation

Outline

- Virtual address spaces
- Processes vs. Threads
- CPU scheduling policies
- Demos

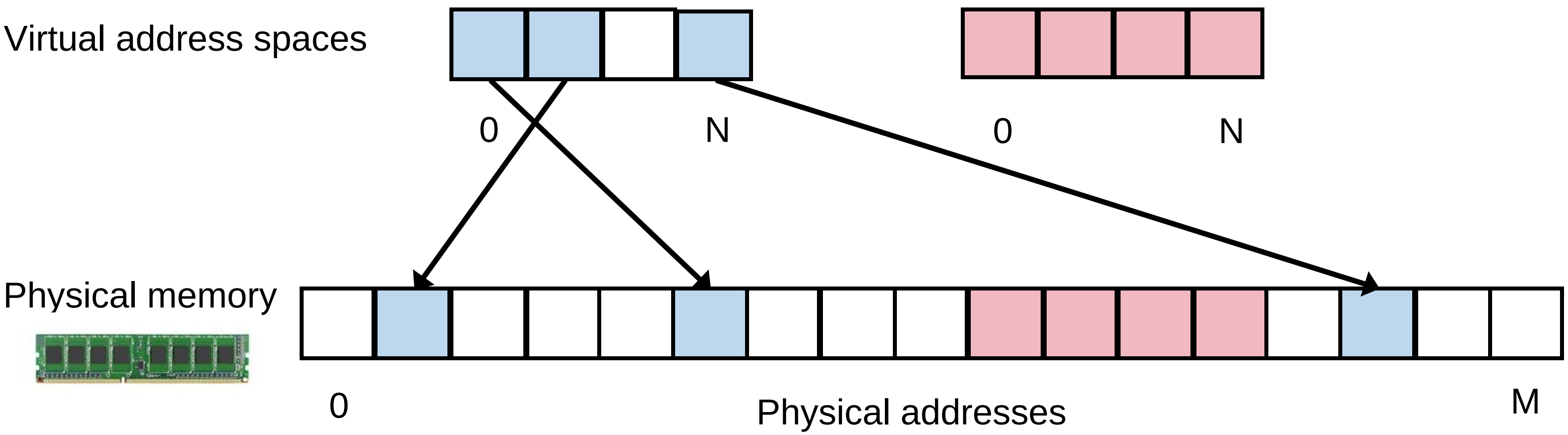
Processes and address spaces

- Address spaces
  - A process is a running program
  - Each process has its own virtual address space
  - The same virtual address generally refers to different memory in different processes
  - Regular processes cannot directly access physical memory or other addr spaces



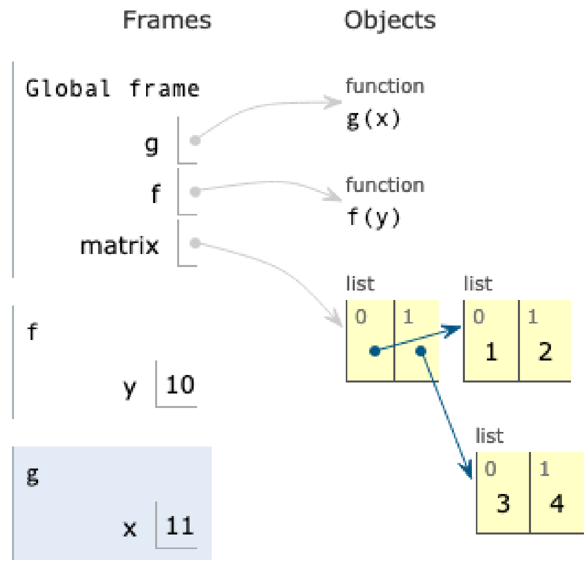
Processes and address spaces

- Address spaces
  - A process is a running program
  - Each process has its own virtual address space
  - The same virtual address generally refers to different memory in different processes
  - Regular processes cannot directly access physical memory or other addr spaces
  - Address spaces can have holes (N is typically much bigger than M)
  - Physical memory for a process need not be contiguous



What goes in an address space

```
def g(x):
    return x * 2

def f(y):
    return g(y+1)

matrix = [[1,2], [3,4]]
f(10)
```



https://pythontutor.com

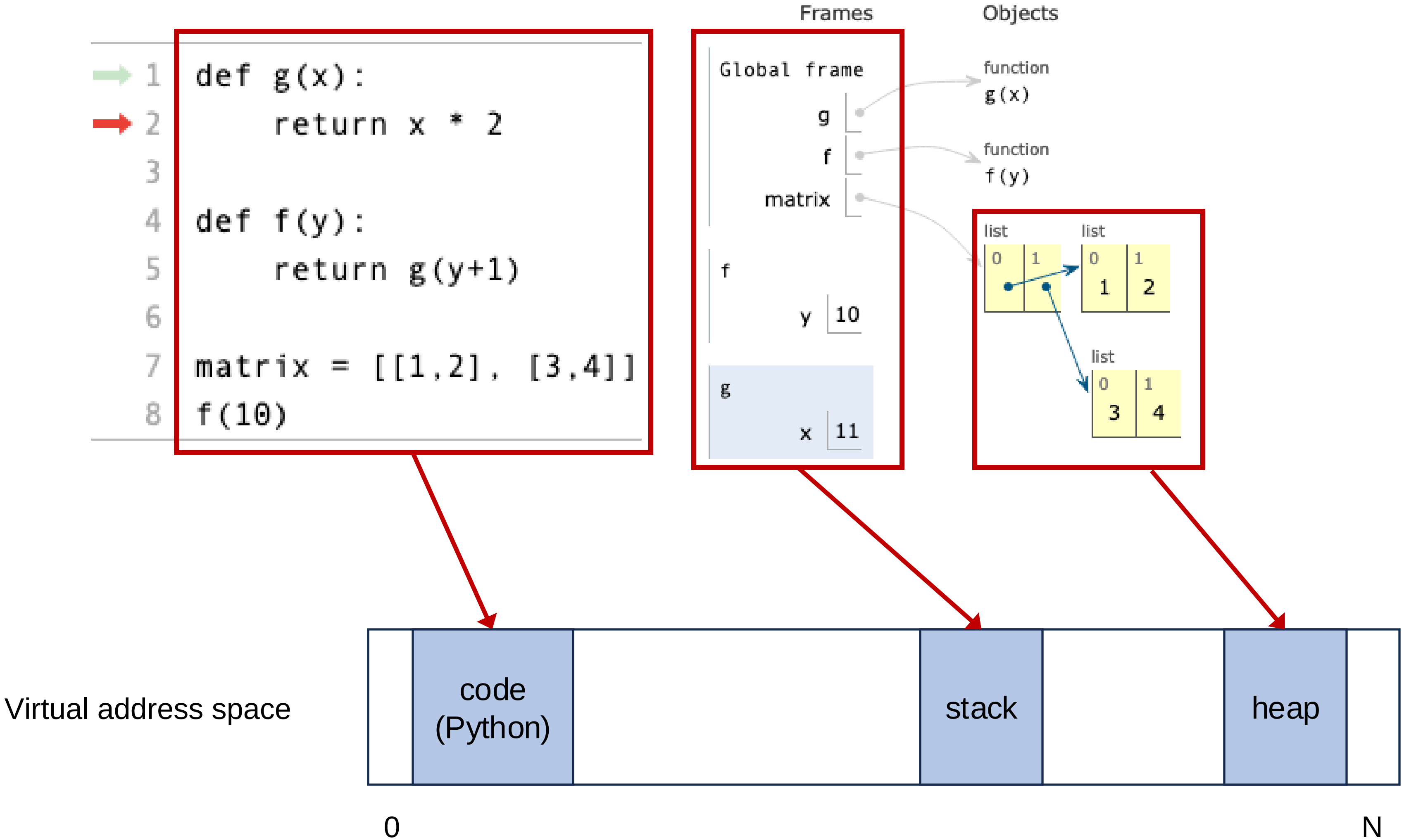
Virtual address space

What goes here?

0

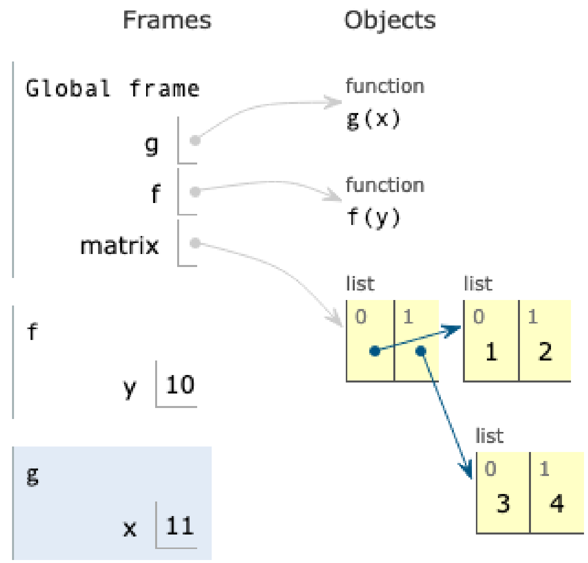
N

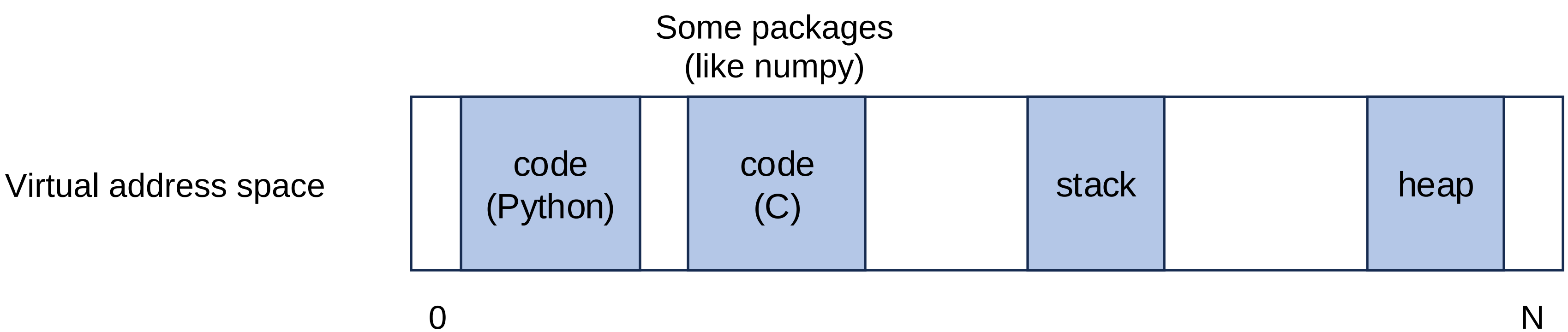
What goes in an address space



What goes in an address space

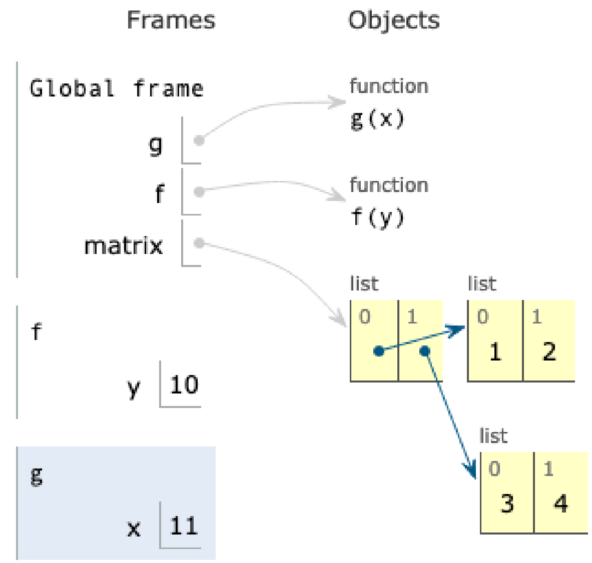
```python
def g(x):
    return x * 2

def f(y):
    return g(y+1)

matrix = [[1,2], [3,4]]
f(10)
```

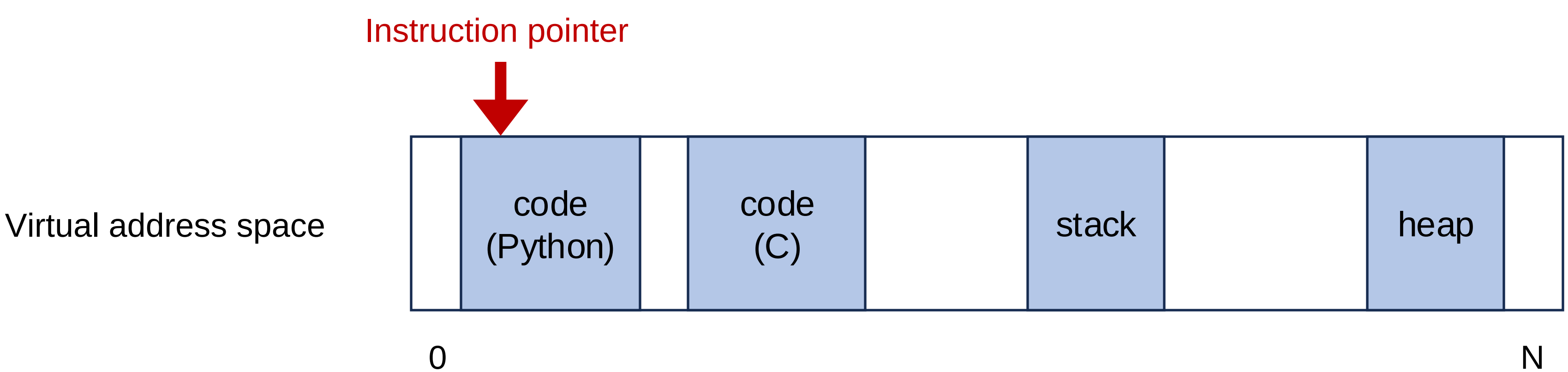




What goes in an address space

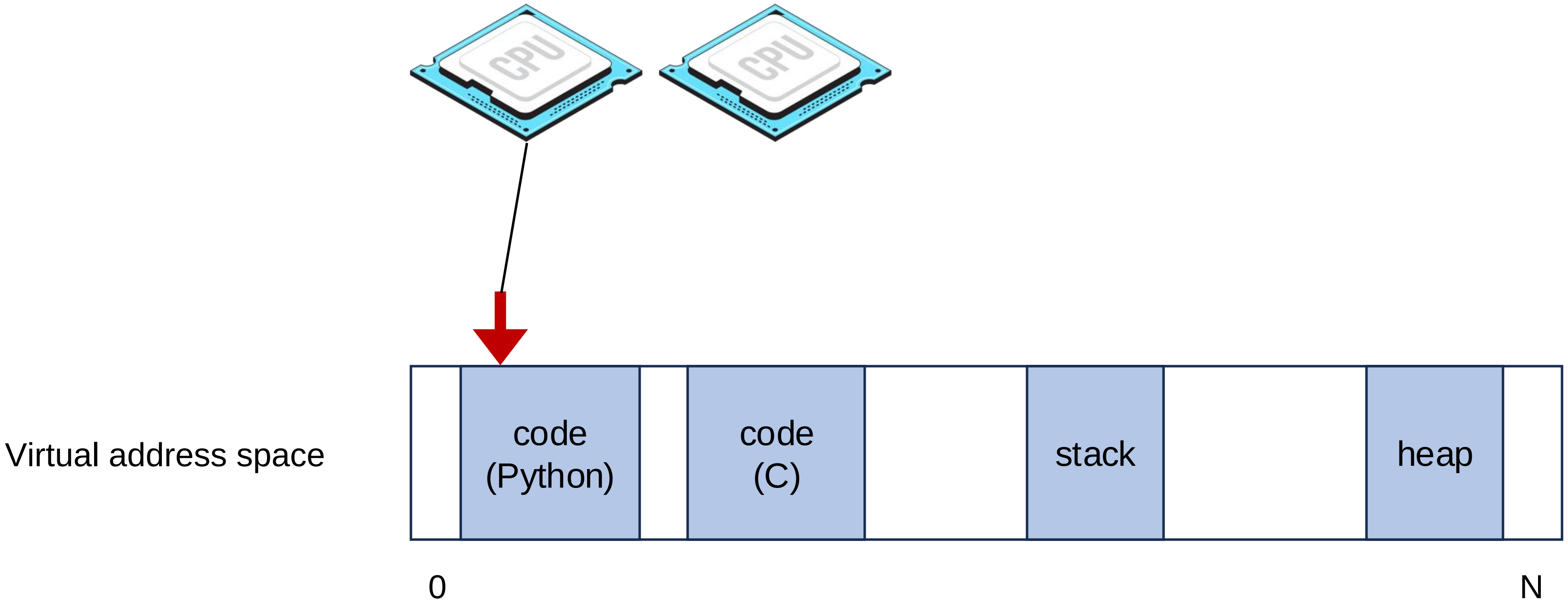
```
def g(x):
    return x * 2

def f(y):
    return g(y+1)

matrix = [[1,2], [3,4]]
f(10)
```





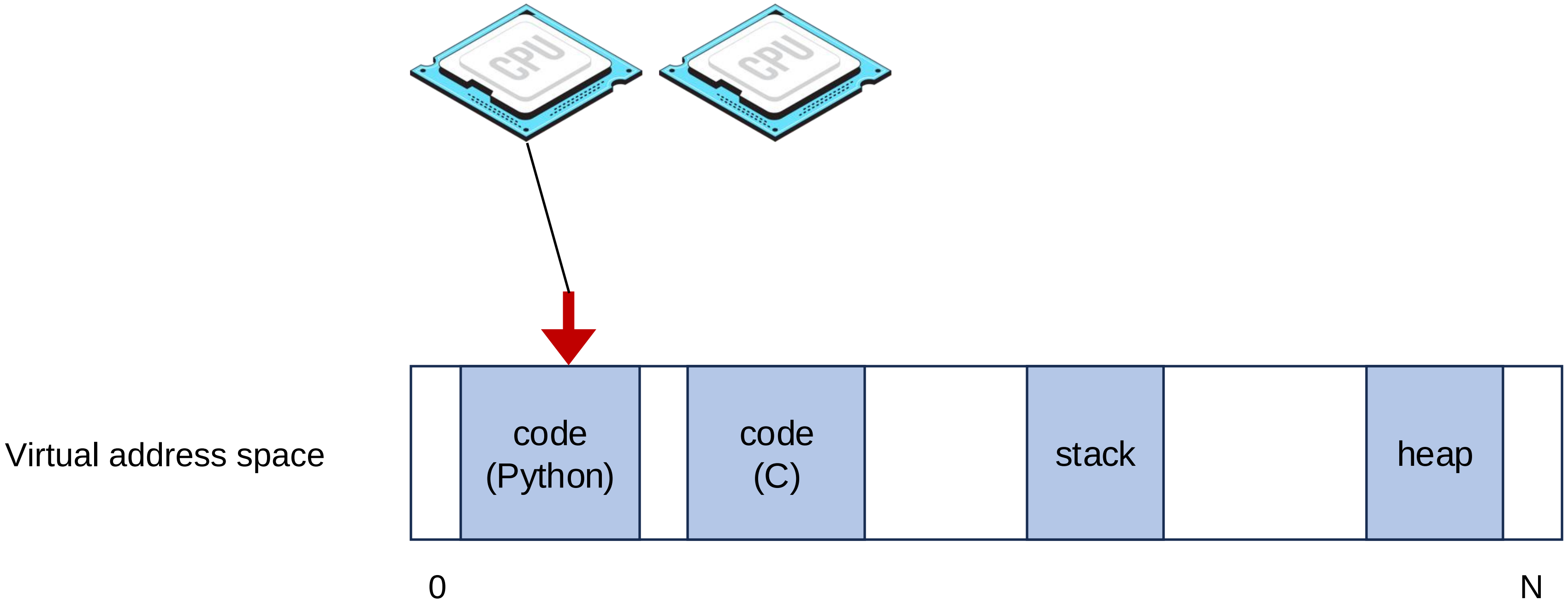
What goes in an address space

- CPUs
  - CPUs are attached to at most one instruction pointer at any given time
  - They run code by executing instructions and advancing the instruction pointer
  - **Note:** interpreter left out for simplicity (CPU points to interpreter code, which points to Python bytecode)



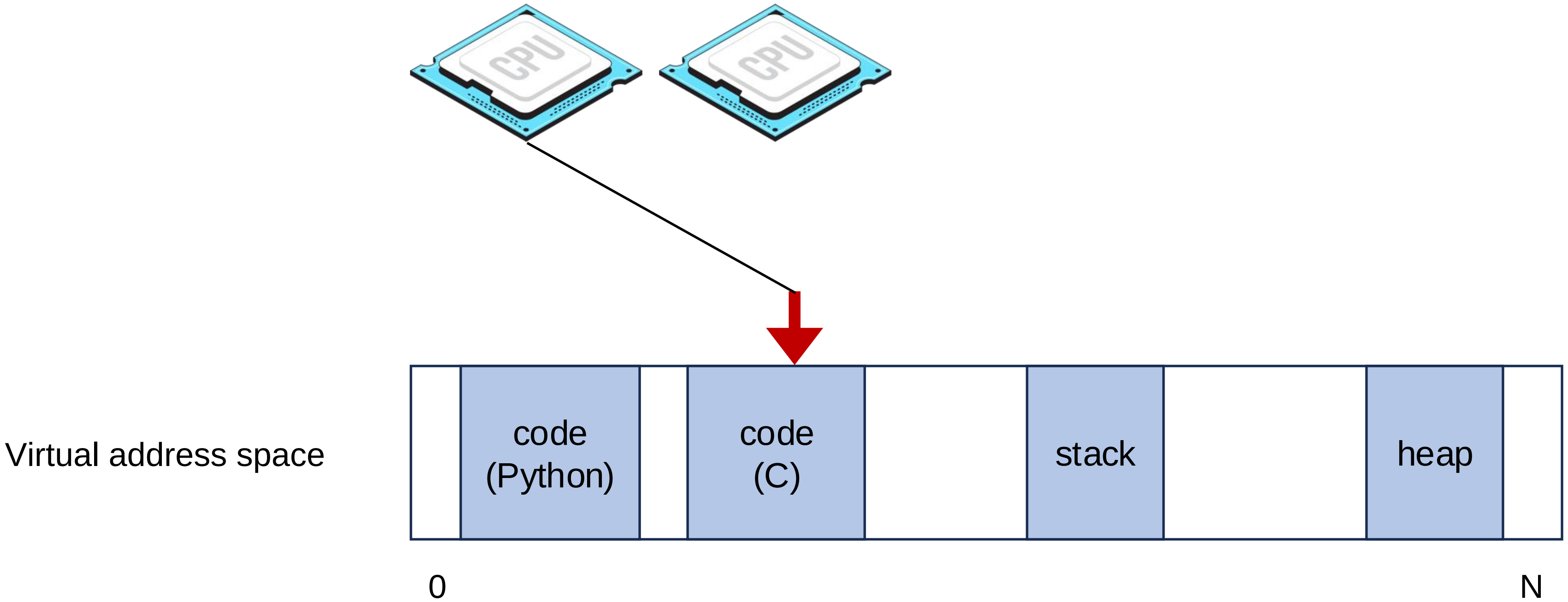
What goes in an address space

- CPUs
  - CPUs are attached to at most one instruction pointer at any given time
  - They run code by executing instructions and advancing the instruction pointer
  - **Note:** interpreter left out for simplicity (CPU points to interpreter code, which points to Python bytecode)



What goes in an address space

- CPUs
  - CPUs are attached to at most one instruction pointer at any given time
  - They run code by executing instructions and advancing the instruction pointer
  - **Note:** interpreter left out for simplicity (CPU points to interpreter code, which points to Python bytecode)

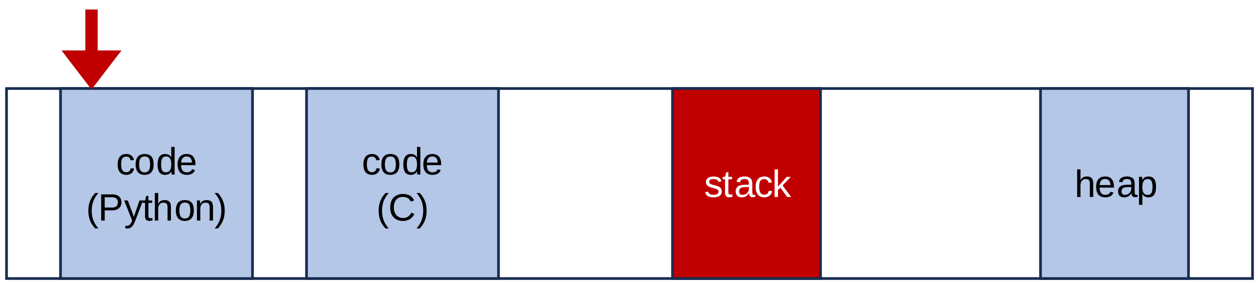


Threads

Threads have their own instruction pointers and stacks, but share heap

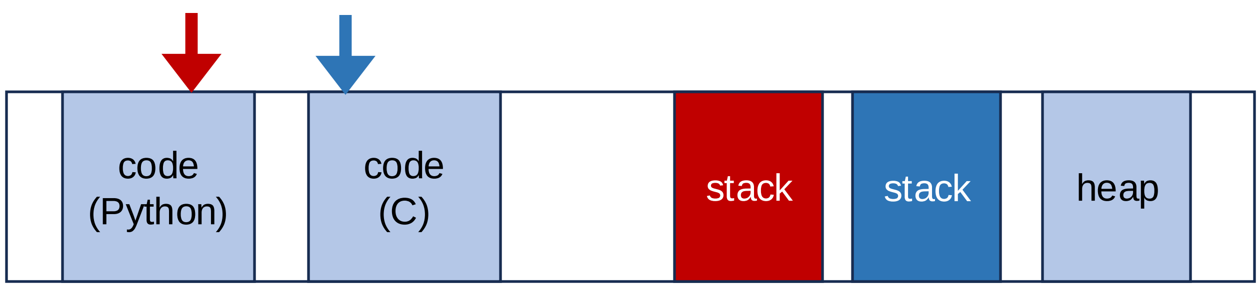
Single-threaded process:

Virtual address space



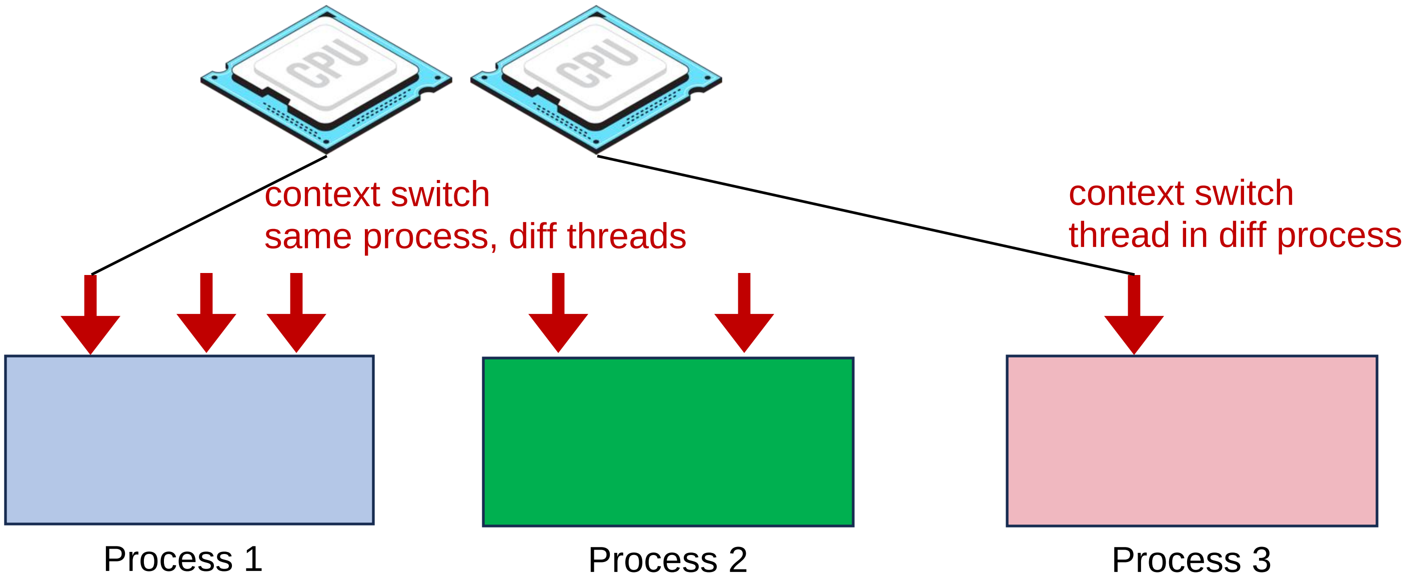
Multi-threaded process:

Virtual address space



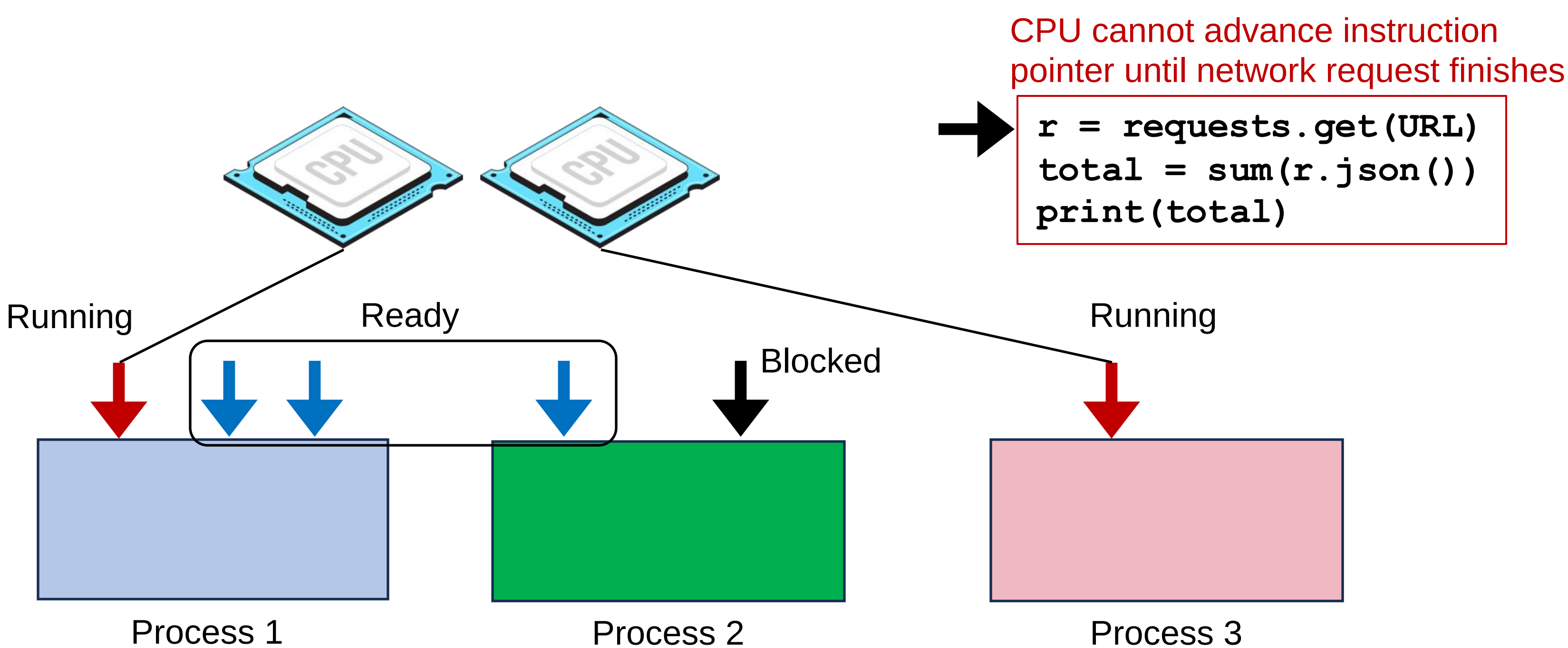
CPU scheduling

- CPU scheduling
  - CPU scheduler is an important sub system in an operating system
  - A scheduler decides when to run which threads
  - Context switch: change which thread a CPU is running



Scheduling restrictions: blocked threads

- Threads can be in one of three states
  - **Running:** CPU is executing it
  - **Blocked:** waiting on something other than CPU (network, input, disk, etc.)
  - **Ready:** scheduler can choose to run it



CPU scheduling policies

- Threads get queued up and the CPU scheduler will select one from the ready queue for execution
- The scheduling policies may have tremendous effects on the system efficiency

First-In, First-Out

Workload assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. The runtime of each job is known

FIFO

- First-In, First-Out: Run jobs in arrival order

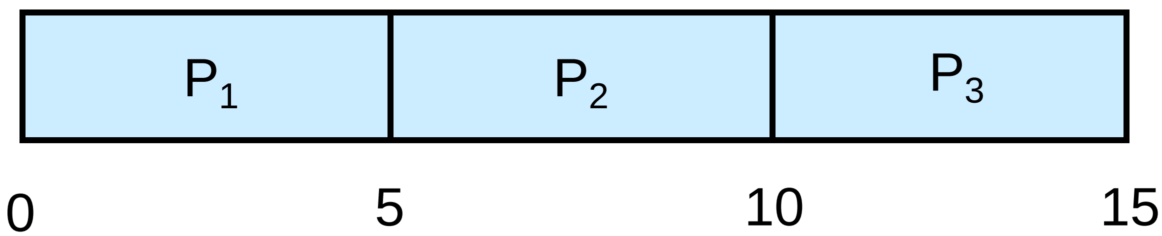
Proc	Arrival time	Runtime
P1	~0	5
P2	~0	5
P3	~0	5

FIFO

- First-In, First-Out: Run jobs in arrival order

Proc	Arrival time	Runtime
P1	~0	5
P2	~0	5
P3	~0	5

Gantt chart

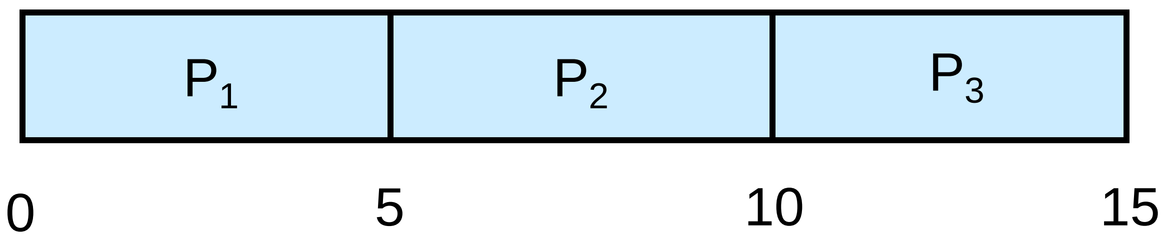


FIFO

- First-In, First-Out: Run jobs in arrival order

Proc	Arrival time	Runtime
P1	~0	5
P2	~0	5
P3	~0	5

Gantt chart



What is the average turnaround time?

Def: turnaround_time = completion_time – arrival_time

Workload assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. The runtime of each job is known

Workload assumptions

1. ~~Each job runs for the same amount of time~~

2. All jobs arrive at the same time

3. All jobs only use the CPU (no I/O)

4. The runtime of each job is known

Example: big first job

Proc	Arrival time	Runtime
P1	~0	80
P2	~0	5
P3	~0	5

What is the average turnaround time?

Example: big first job

Proc	Arrival time	Runtime
P1	~0	80
P2	~0	5
P3	~0	5

What is the average turnaround time?

Example: big first job

Proc	Arrival time	Runtime
P1	~0	80
P2	~0	5
P3	~0	5

What is the average turnaround time?

Average turnaround time: (80+85+90) / 3 = 85

Convoy effect!!

Better schedule?

Shortest Job First (SJF)

Passing the tractor

- New scheduler: SJF (Shortest Job First)

- Policy: When deciding which job to run, choose the one with the smallest runtime

Example: SJF

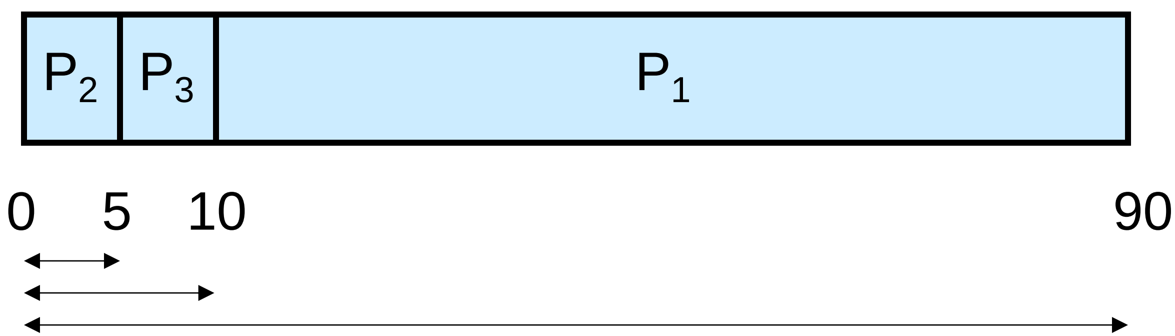
Proc	Arrival time	Runtime
P1	~0	80
P2	~0	5
P3	~0	5

What is the average turnaround time with SJF?

Example: SJF

Proc	Arrival time	Runtime
P1	~0	80
P2	~0	5
P3	~0	5

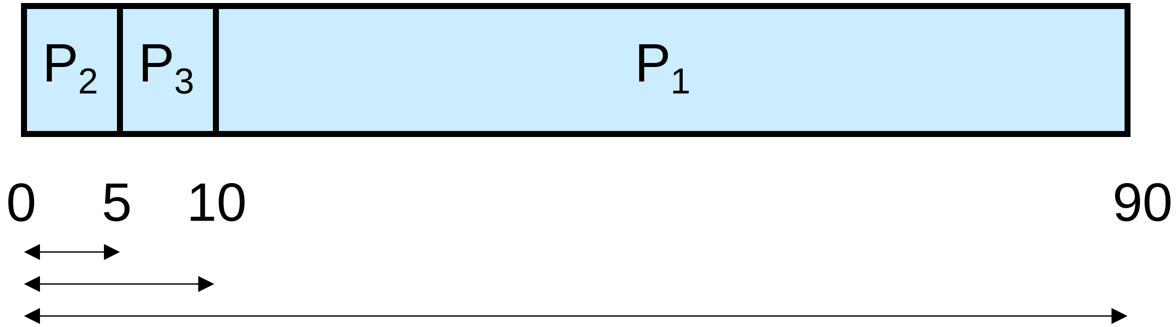
What is the average turnaround time with SJF?



Example: SJF

Proc	Arrival time	Runtime
P1	~0	80
P2	~0	5
P3	~0	5

What is the average turnaround time with SJF?



Average turnaround time: (5+10+90) / 3 = 35

Workload assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. The runtime of each job is known

Workload assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. The runtime of each job is known

What if jobs arrive at different time?

Shortest Job First (arrival time)

Proc	Arrival time	Runtime
P1	~0	80
P2	~15	20
P3	~15	10

What is the average turnaround time with SJF?

Shortest Job First (arrival time)

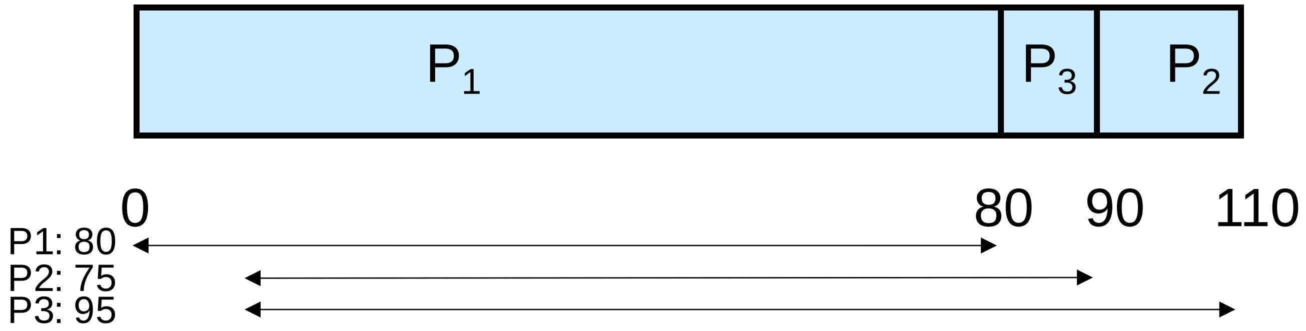
Proc	Arrival time	Runtime
P1	~0	80
P2	~15	20
P3	~15	10

What is the average turnaround time with SJF?

Shortest Job First (arrival time)

Proc	Arrival time	Runtime
P1	~0	80
P2	~15	20
P3	~15	10

What is the average turnaround time with SJF?



Average turnaround time: (80+75+95) / 3 = ~83.3

A preemptive scheduler

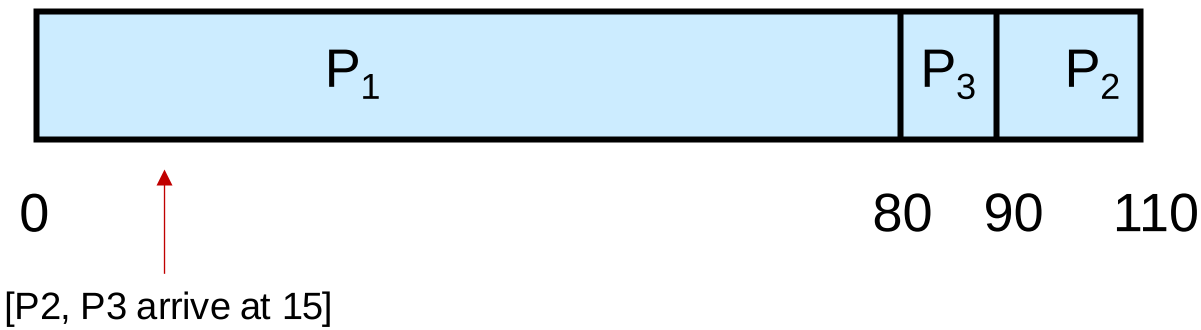
- Previous schedulers: FIFO and SJF are non-preemptive

- New scheduler:
  STCF (Shortest Time-to-Completion First)

- Policy: Switch jobs so we always run the one that will complete the quickest

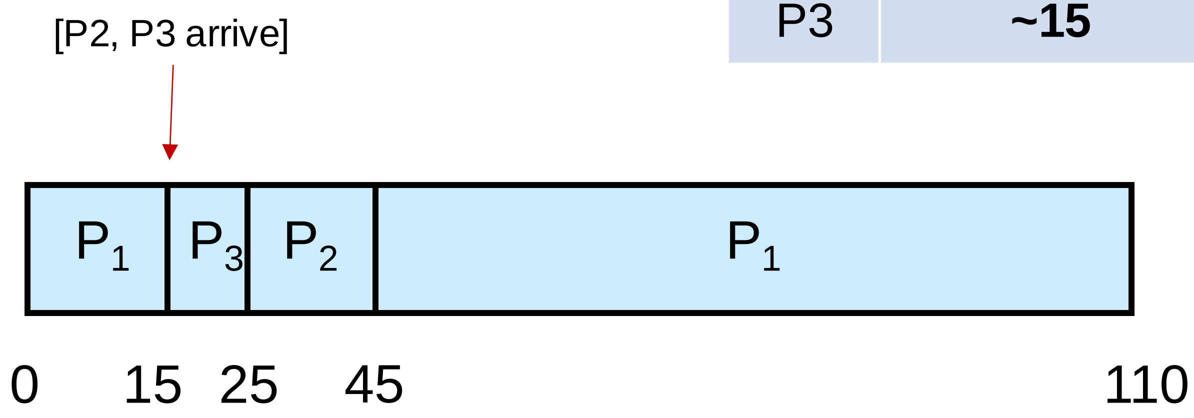
SJF

Proc	Arrival time	Runtime
P1	~0	80
P2	**~15**	20
P3	**~15**	10



STCF

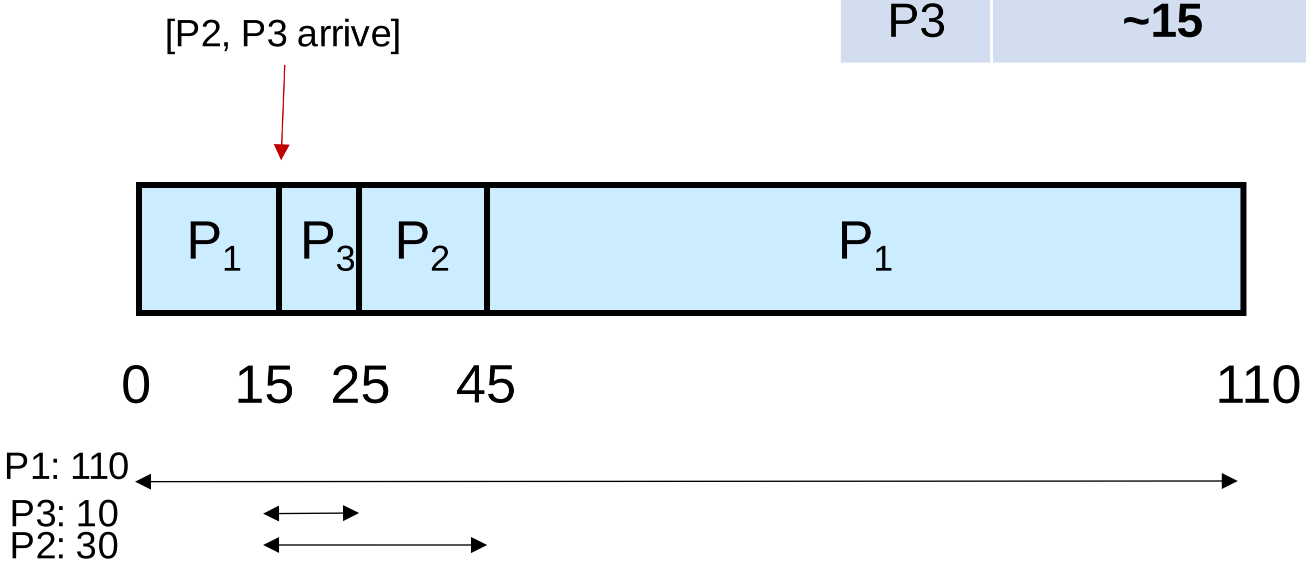
Proc	Arrival time	Runtime
P1	~0	80
P2	**~15**	20
P3	**~15**	10



What is the average turnaround time with STCF?

STCF

Proc	Arrival time	Runtime
P1	~0	80
P2	**~15**	20
P3	**~15**	10



Average turnaround time: (110+30+10) / 3 = 50

Workload assumptions

1. Each job runs for the same amount of time

2. All jobs arrive at the same time

3. All jobs only use the CPU (no I/O)

4. The runtime of each job is known

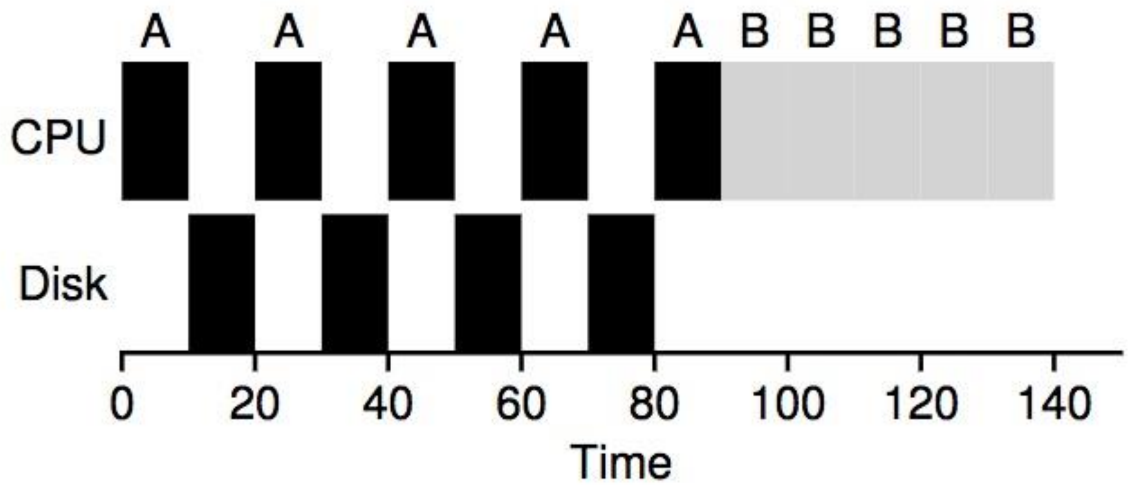
Workload assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. The runtime of each job is known

What if jobs do I/Os as well?

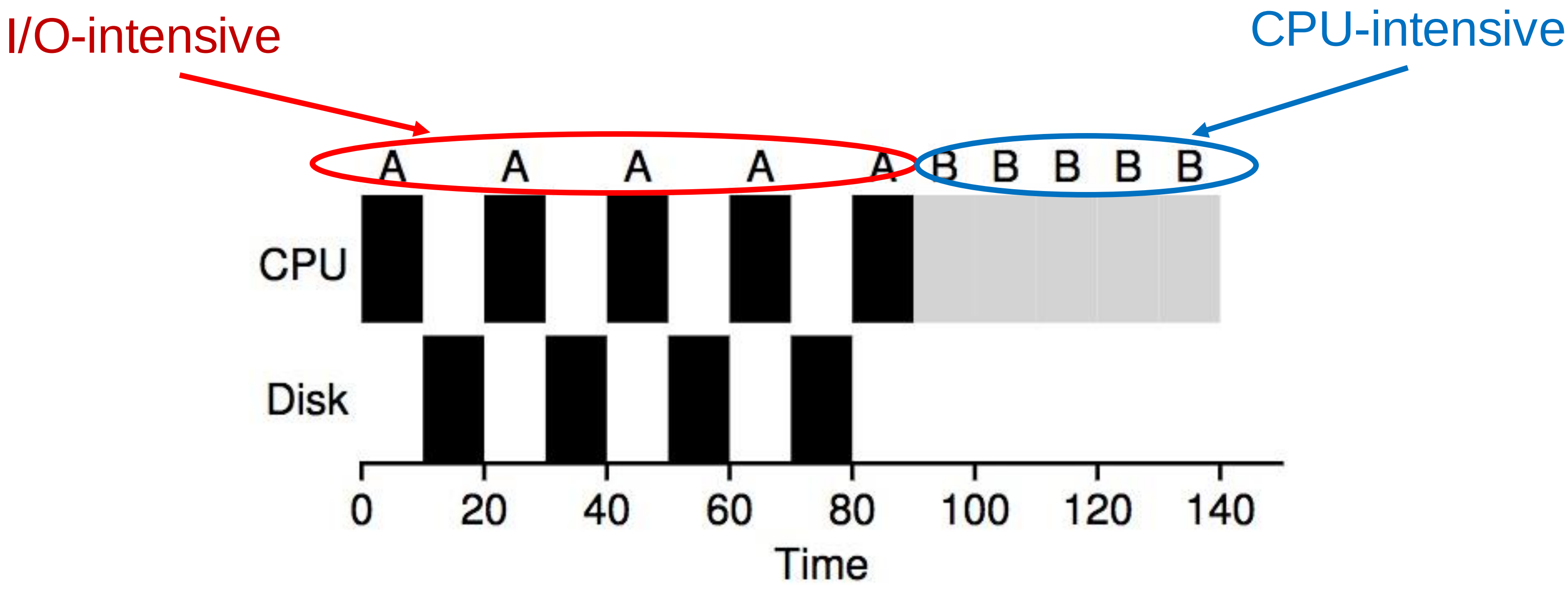
- No good if a program can only do pure CPU-intensive compute
- A common execution pattern of the typical big data applications (**Hadoop, Spark, Dask**)
  - Completes the CPU burst, performs I/O (e.g., read more CSV files from disk into DRAM), rejoins the ready queue and completes the second CPU bursts…

Not I/O Aware



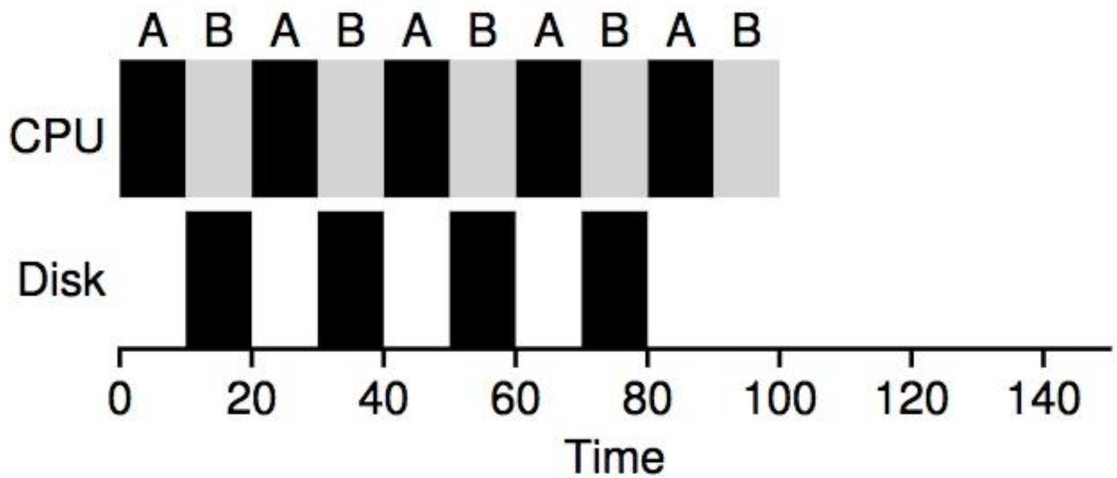
Poor use of resources

Not I/O Aware



Poor use of resources

I/O Aware (Overlap)



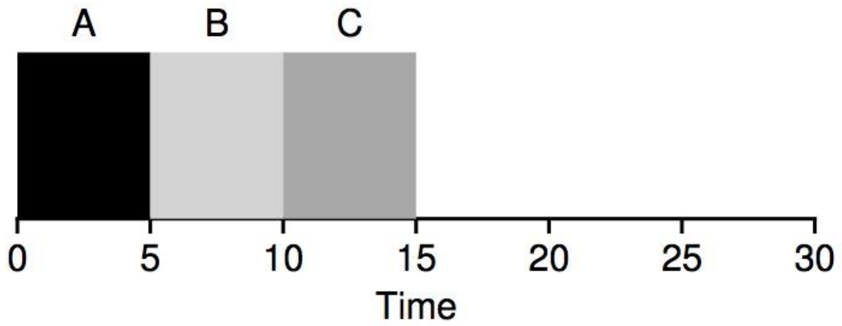
Overlap allows better use of resources!

Round Robin (RR)

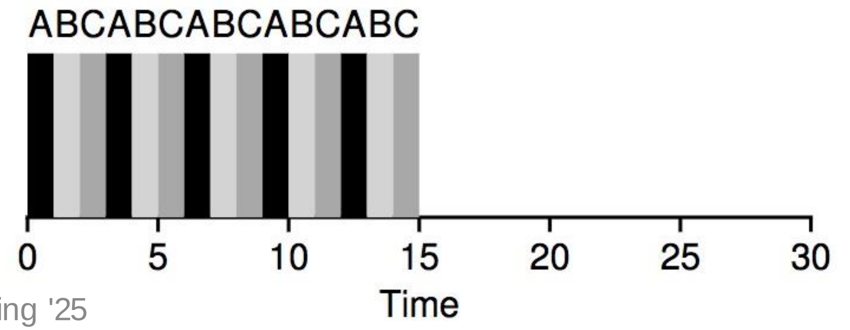
Process	Burst time
A	~5
B	~5
C	~5

- Each process gets a small unit of CPU time (time slice). After this time has elapsed, the process is preempted and added to the end of the ready queue

- SJF



- **RR** (time slice = 1)



Workload assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. The runtime of each job is known

Workload assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. The runtime of each job is known ?

Demos ...