Object Oriented Methodology

Week – *7*, Lecture – *2*

Multithreading Basics

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

About the ITP Lectures …

There were some concepts that I covered during *Digression* lectures in previous semester

- Since they were not a part of the syllabus, I don't think too many people watched them :P

Some content of this lecture is common with the lecture **D2** from ITP

- There is no harm in watching that lecture now :-D

In addition, some of the content may seem a repeat of Lecture **0.2** from ITP as well

But there is more content here that follows …

Processes (1/3)

saurabhsrivastava — top — 108×31

```
Processes: 461 total, 2 running, 1 stuck, 458 sleeping, 1903 threads                 22:38:17
Load Avg: 2.25, 2.78, 2.90  CPU usage: 2.60% user, 9.45% sys, 87.94% idle
SharedLibs: 189M resident, 48M data, 19M linkedit.
MemRegions: 104891 total, 1612M resident, 82M private, 2086M shared.
PhysMem: 8143M used (2874M wired), 48M unused.
VM: 2085G vsize, 1991M framework vsize, 10656080(0) swapins, 11626592(0) swapouts.
Networks: packets: 19531890/15G in, 19297002/9525M out. Disks: 18190442/327G read, 11737886/199G written.

PID    COMMAND      %CPU  TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE
318    mds_stores   18.6  25:08.81 8     6    111   102M+  1832K  21M-   318   1     stuck
0      kernel_task  8.4   08:48:29 185/4 0    0     87M-   0B     0B     0     0     running
74644  top          4.5   00:04.36 1/1   0    32    4872K  0B     0B     74644 71465 running
73997  VirtualBoxVM 3.7   01:49.13 37    1    370   2669M  7892K  80M    73997 73996 sleeping
243    WindowServer 2.8   13:43:27 10    4    2828  283M-  4216K  79M    243   1     sleeping
183    hidd         2.6   01:49:15 5     3    312-  3080K- 0B     1028K  183   1     sleeping
71463  Terminal     1.2   00:02.68 10    3    296   26M    1264K- 6292K  71463 1     sleeping
73996  VBoxSVC      0.8   00:14.82 21    2    178   5928K  0B     4136K  73996 1     sleeping
529    com.apple.ge 0.4   02:06.55 6     4    96+   10M+   64K    7160K- 529   1     sleeping
73994  VBoxXPCOMIPC 0.3   00:06.85 1     0    21    2636K  0B     2220K  73994 1     sleeping
73371  Google Chrom 0.3   10:39.90 16    1    367   390M-  0B     87M    73282 73282 sleeping
74656  screencaptur 0.3   00:00.23 5     3    175   6608K  8192B  0B     74656 1     sleeping
73282  Google Chrom 0.2   06:45.28 28    1    691   196M   0B     84M    73282 1     sleeping
73296  Google Chrom 0.1   01:28.96 9     1    119   34M    0B     12M    73282 73282 sleeping
73992  VirtualBox   0.1   00:07.76 9     1    222   89M    0B     80M    73992 1     sleeping
511    Spotlight    0.1   05:30.38 11    5    1228  69M    920K-  18M    511   1     sleeping
508    USBOverdrive 0.1   07:28.72 2     1    169   12M    0B     5196K  508   1     sleeping
74606  Google Chrom 0.1   00:05.20 13    1    144   79M+   4096B  18M    73282 73282 sleeping
132    fseventsd    0.0   10:30.38 13    1    335-  7576K- 0B     4624K  132   1     sleeping
139    powerd       0.0   06:29.80 3     2    125   2352K  0B     1000K  139   1     sleeping
73295  Google Chrom 0.0   05:48.52 10    1    299   179M-  20M+   60M-   73282 73282 sleeping
1      launchd      0.0   31:50.46 2     1    4797  29M    0B     14M    1     0     sleeping
```

If you use a UNIX-based operating system, such as Linux or macOS, there is a good chance that you can issue `top` command on your terminal

The top command shows you a list of the *currently running* processes on your system, along with some other information

`%CPU` and `MEM` are usually of interest, representing the average CPU usage and the amount of RAM your process is using

Processes (2/3)

As you may already know, *processes* are units of execution that your operating system executes

For the operating system, even a “Hello World” program is a distinct process

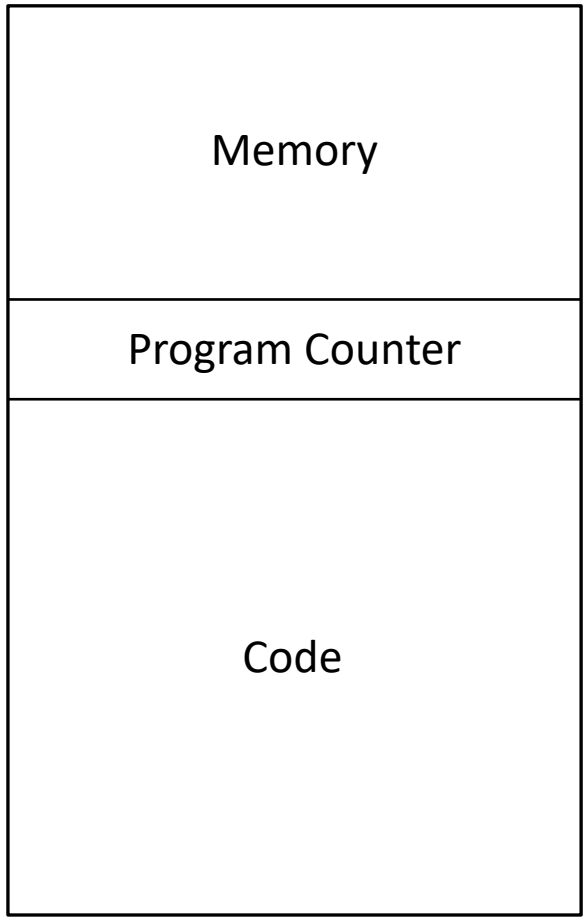
An operating system has a fixed template for creating, executing and ending processes

- We call it the *Process Lifecycle*

At any point of time, a process has a particular *state*

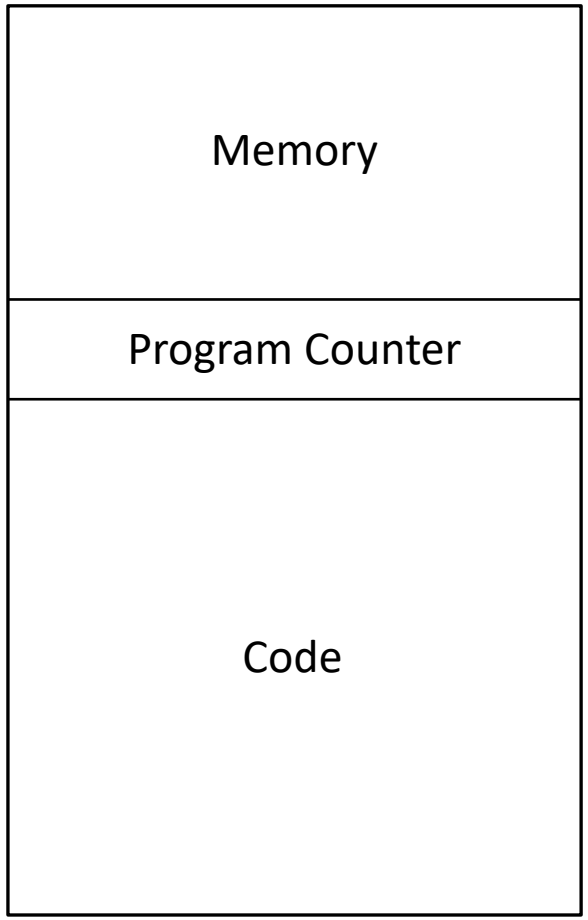
- A process is in the state *ready*, if it wishes to use the CPU, but doesn’t have it currently
- A process is in the state *running*, if it currently has the CPU for use
- A process is in the state *blocked*, if it does not require CPU due to some reason (e.g., a request for I/O)
- In addition, some categorisations also consider *new* and *terminated* as process states

Processes (3/3)



This is an oversimplified view of a process

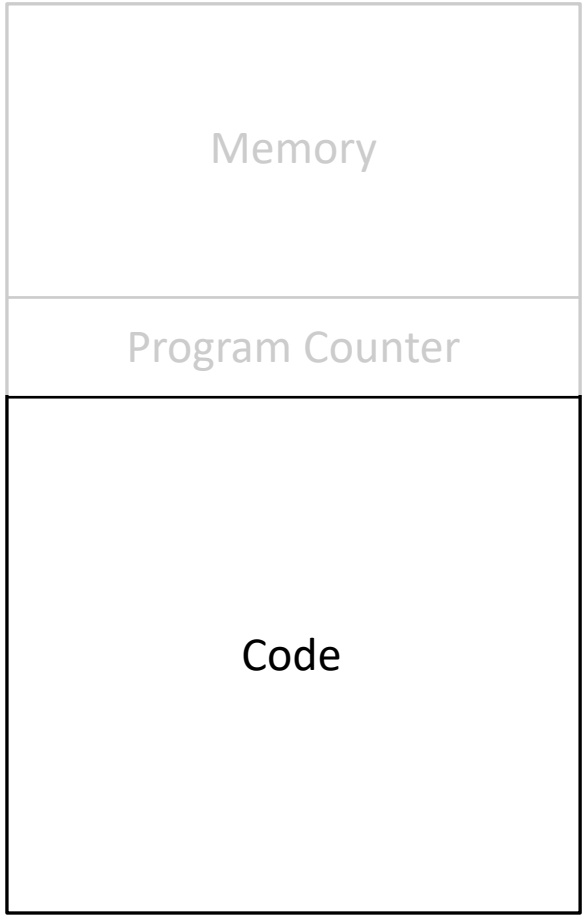
Processes (3/3)



This is an oversimplified view of a process

Still, this is all you need to know right now !!

Processes (3/3)

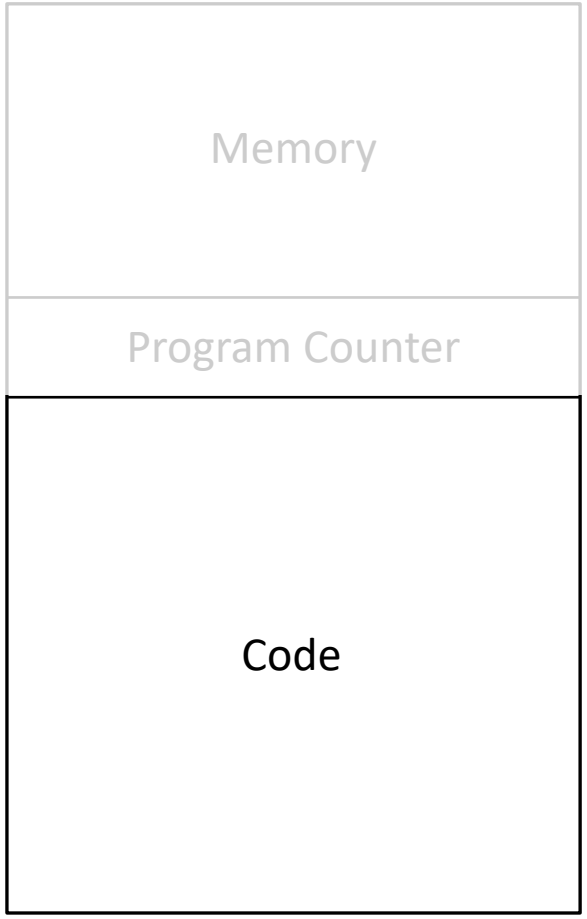


This is an oversimplified view of a process

Still, this is all you need to know right now !!

This is the code the operating system executes as part of running the process

Processes (3/3)



This is an oversimplified view of a process

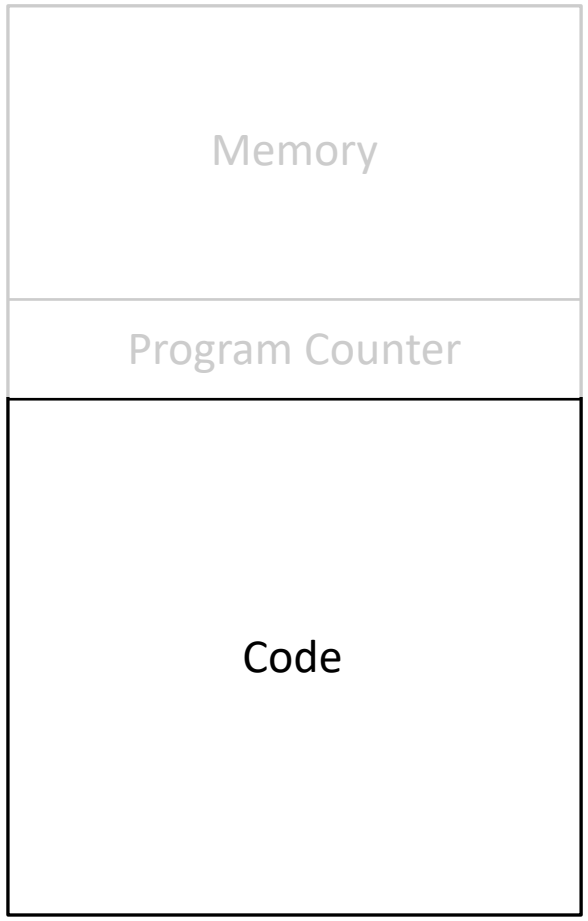
Still, this is all you need to know right now !!

This is the code the operating system executes as part of running the process

This is not C++ or C code, but something far more fundamental

Processes (3/3)

Similar to what we saw in Week 0 of ITP



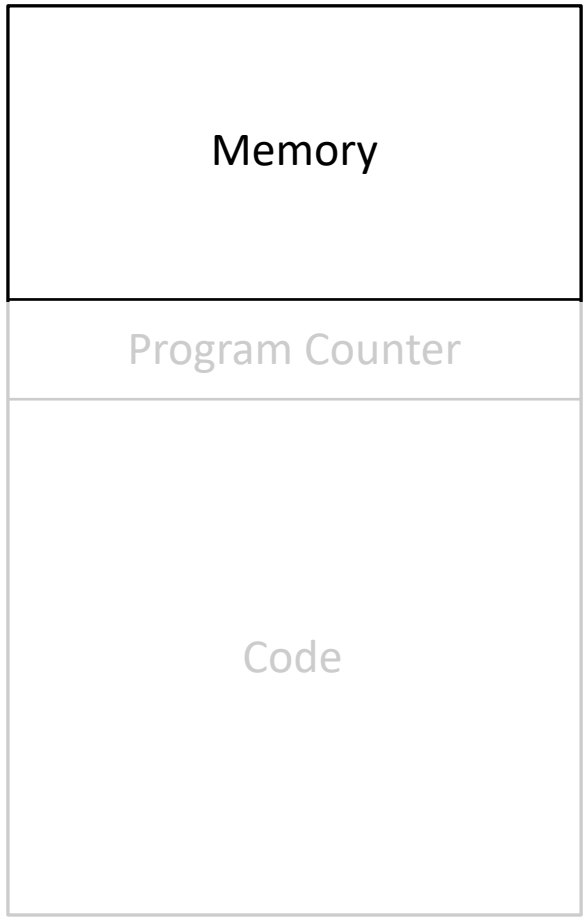
This is an oversimplified view of a process

Still, this is all you need to know right now !!

This is the code the operating system executes as part of running the process

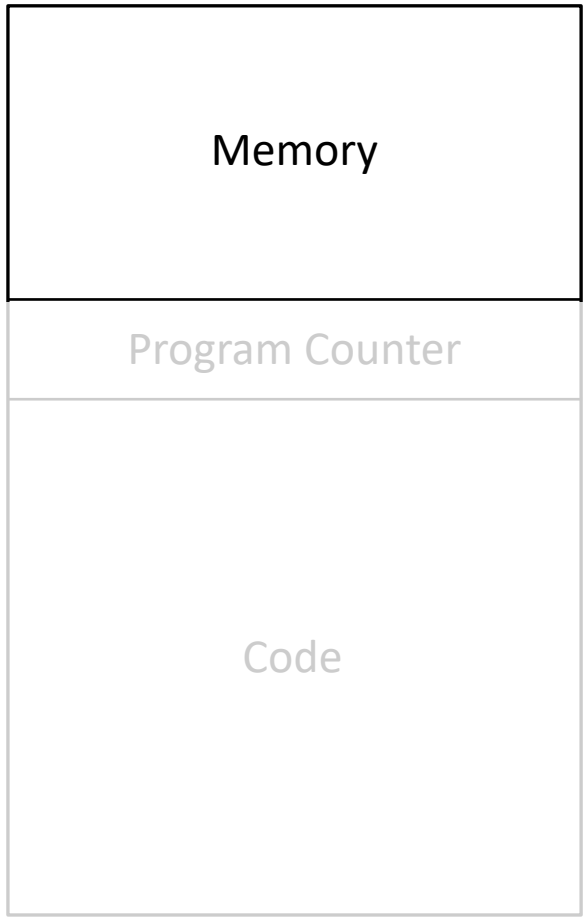
This is not C++ or C code, but something far more fundamental

Processes (3/3)



Memory hosts all the variables and constants that the code requires to run

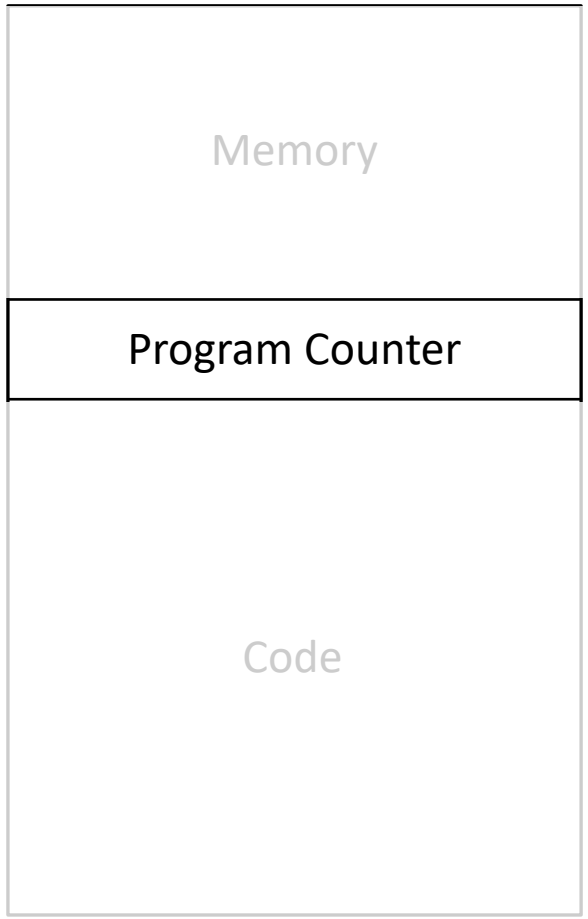
Processes (3/3)



Memory hosts all the variables and constants that the code requires to run

There are different types of memory that a process has, but we don't need to know these details as of now

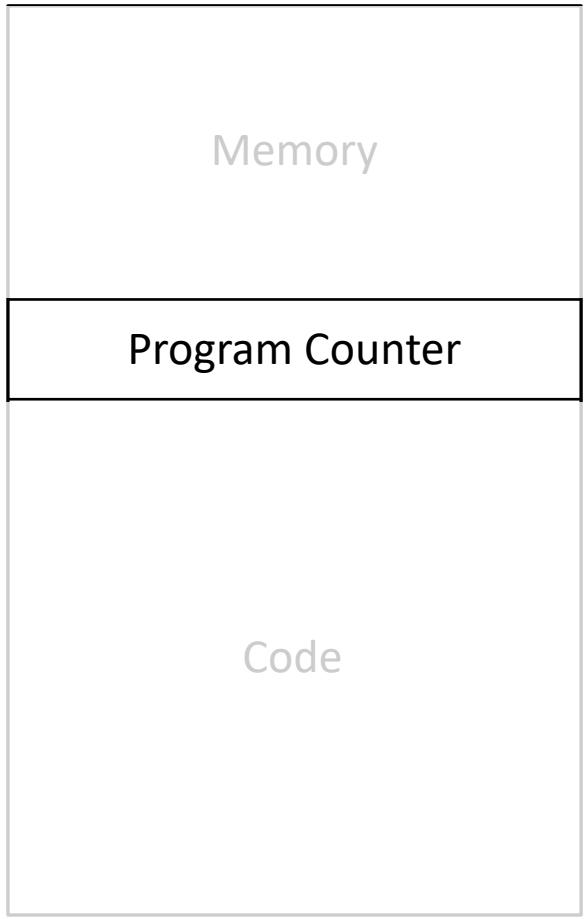
Processes (3/3)



The Program Counter tells the CPU the location of the next line of code (known as *instruction* at that level) that must be executed

Processes (3/3)

We did discuss the Program Counter in Week 0 of ITP !!



The Program Counter tells the CPU the location of the next line of code (known as *instruction* at that level) that must be executed

Multi-tasking

Almost all operating systems today perform *multi-tasking*

It means that they keep a set of processes ”alive” simultaneously

The term ”alive” here means that there are multiple processes in the main memory (or RAM)

The operating system picks one of them, executes it for some time, and puts it in the *ready* state again

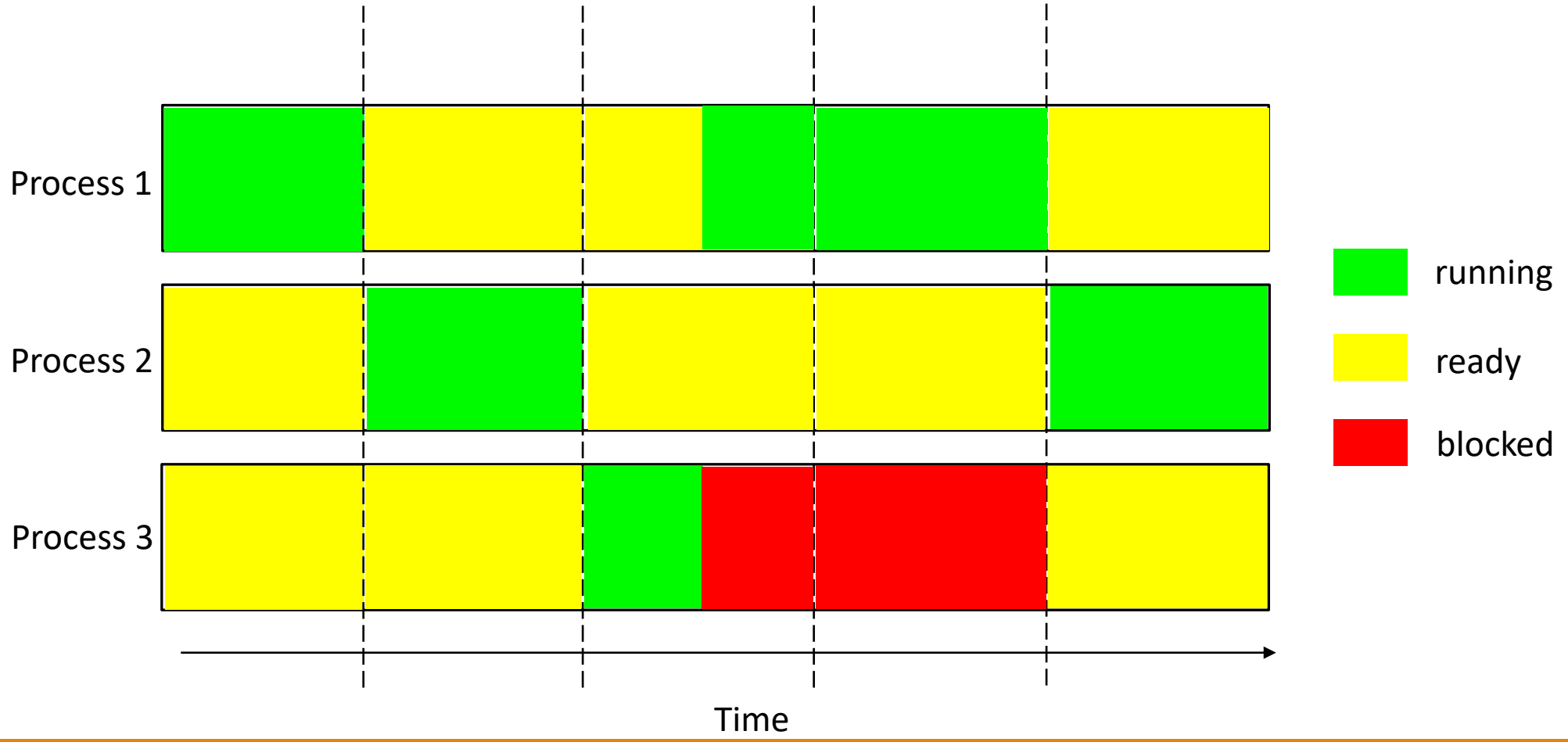
- Usually, the execution happens in multiples of small time *quantums*

The switch is so fast that for a naked eye, it seems that *multiple processes are running simultaneously*

In reality, only one process is running (if we have just one CPU)

The value of Program Counter is crucial, since it allows execution from where it was left last time

Multi-tasking



Threads

Threads are *lightweight* processes

- Alternatively, you can consider processes as *heavyweight threads*

Imagine a process that has *multiple Program Counters*

When such a process gets the CPU, it can choose to execute code pointed by any of these

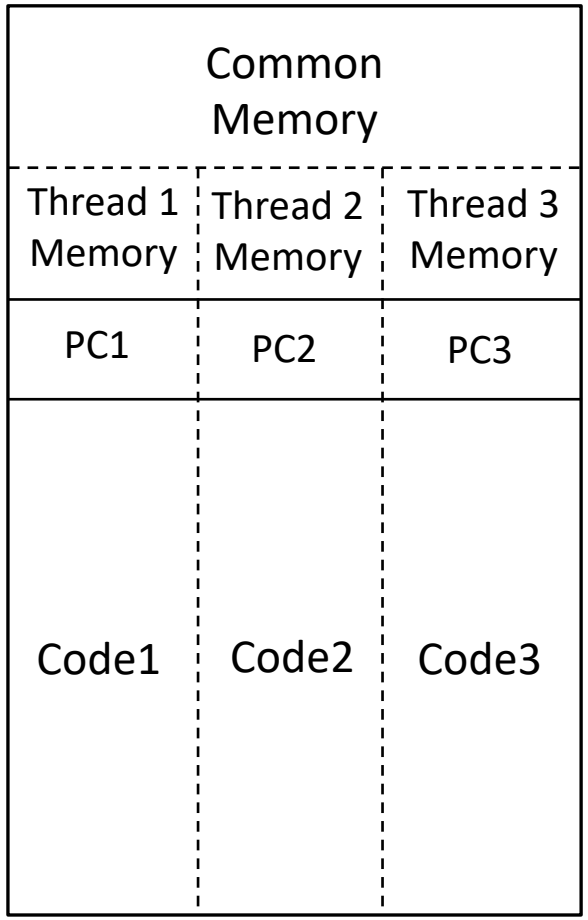
We say that such a process has *multiple threads of execution*

- Out of which, it may pick anyone based on some policy, to execute next

The operating system deals with processes, the processes in turn, can manage multiple threads

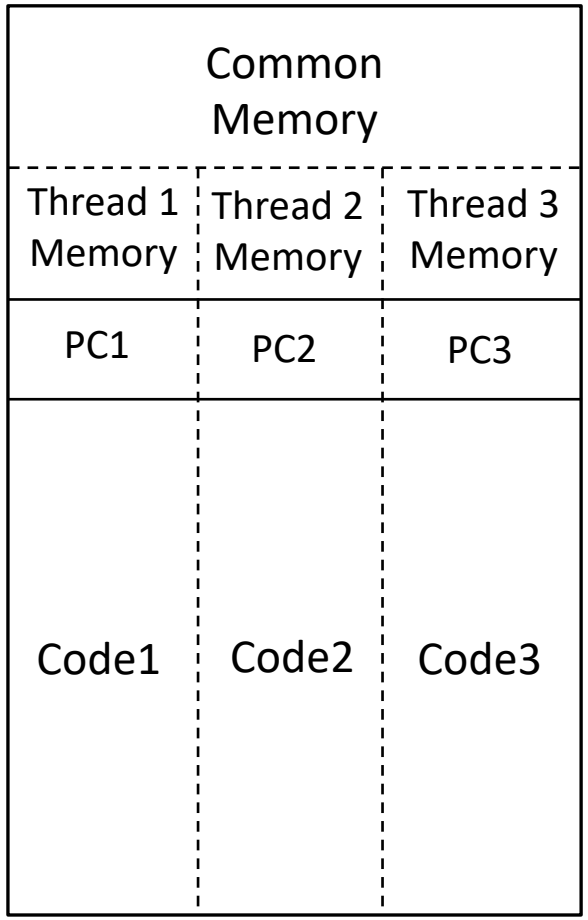
- This is not true always, i.e., in some cases, operating system may also manage underlying threads
- However, for our discussions, assume that managing threads, is a responsibility of the process

Threads



This is an oversimplified view of a multi-threaded process

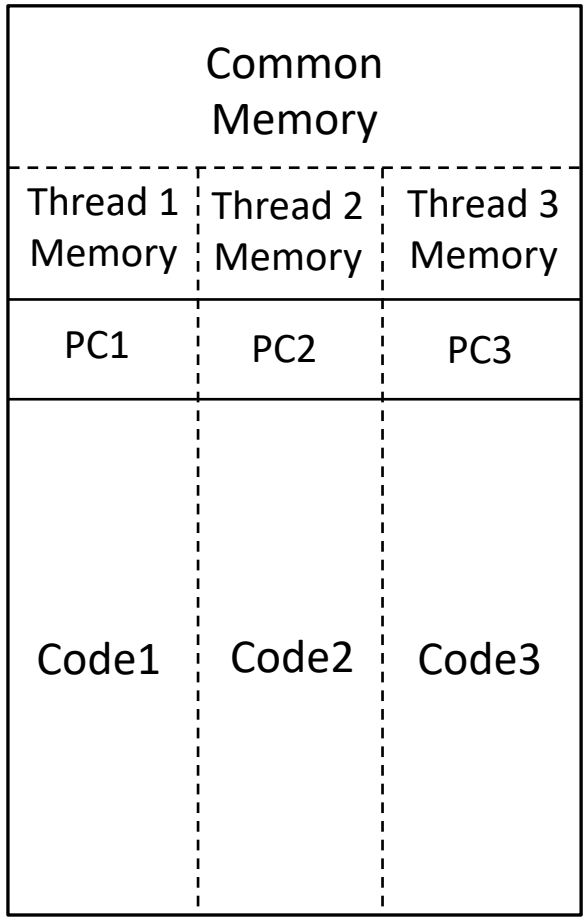
Threads



This is an oversimplified view of a multi-threaded process

Each thread has its own Program Counter

Threads

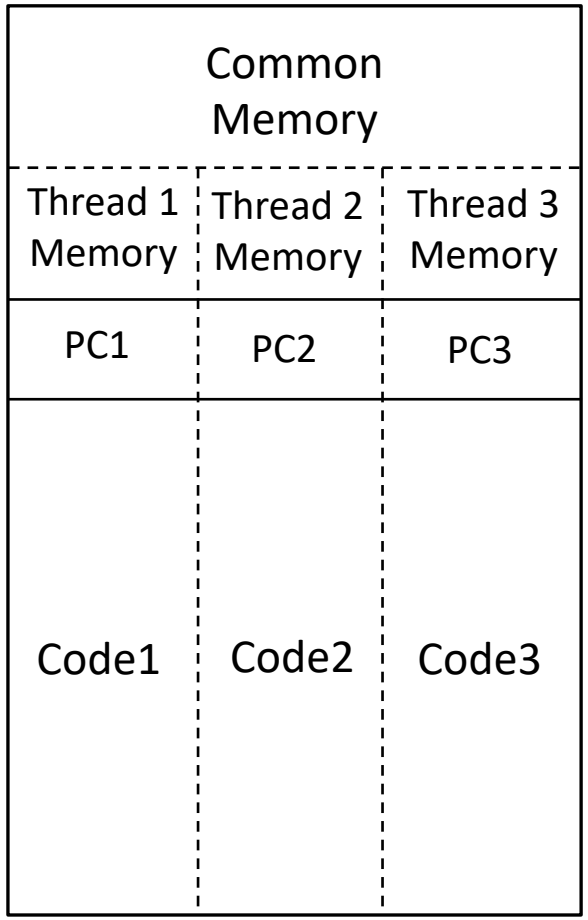


This is an oversimplified view of a multi-threaded process

Each thread has its own Program Counter

Each thread has its own code to execute as well

Threads



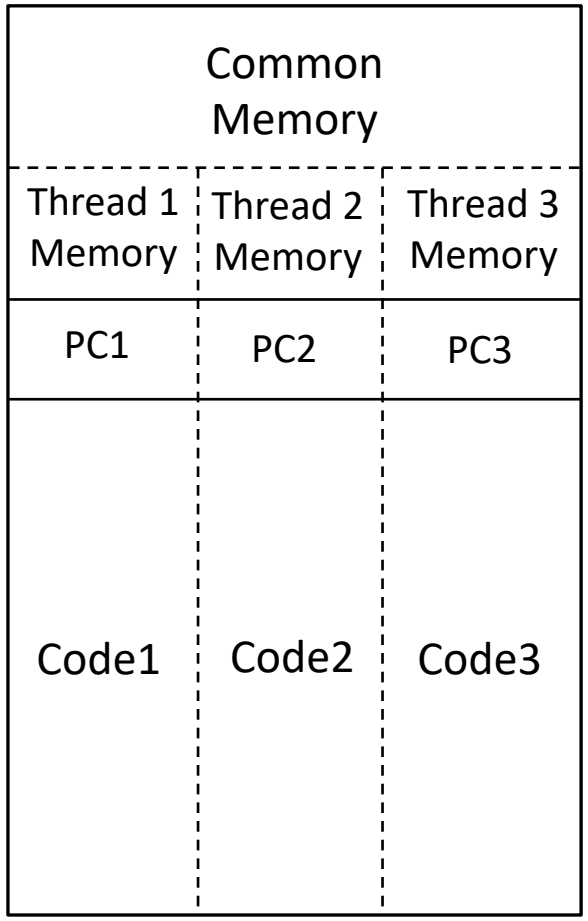
This is an oversimplified view of a multi-threaded process

Each thread has its own Program Counter

Each thread has its own code to execute as well

Each thread has some memory which is dedicated towards it and not accessible to other threads

Threads



This is an oversimplified view of a multi-threaded process

Each thread has its own Program Counter

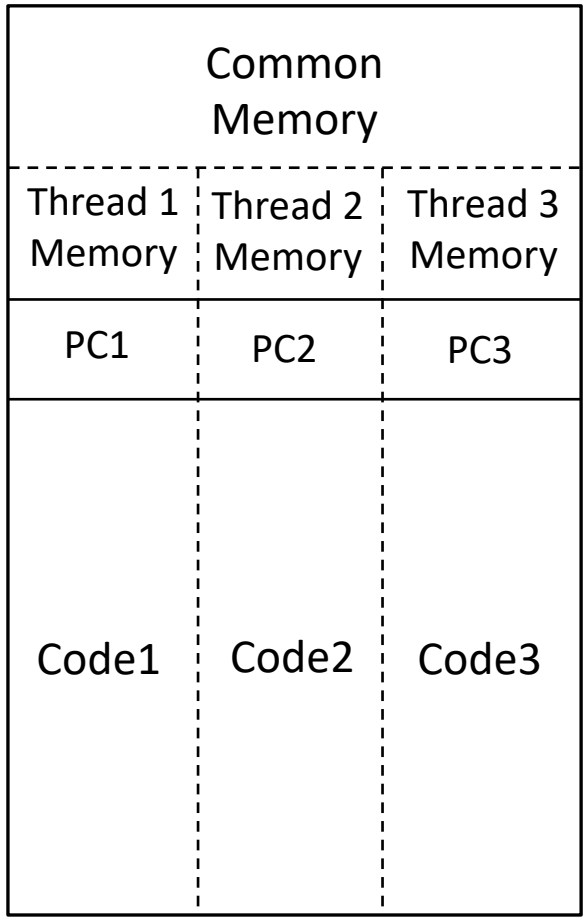
Each thread has its own code to execute as well

Each thread has some memory which is dedicated towards it and not accessible to other threads

There is some common memory too, accessible to all the threads

Threads

For our discussion, we will assume that the decision to execute code for a particular thread, is internal to a process (i.e., the operating system has no say in it)



This is an oversimplified view of a multi-threaded process

Each thread has its own Program Counter

Each thread has its own code to execute as well

Each thread has some memory which is dedicated towards it and not accessible to other threads

There is some common memory too, accessible to all the threads

Implementing Threads in C/C++

Portable Operating System Interface (POSIX) is a set of standards

- These standards provide Application Programming Interface (API) for many crucial tasks
- This includes managing processes, performing networking, **creating threads** etc.

Although they are meant to be portable across platforms, they are more popular on *nix systems

There are different POSIX modules for different tasks

- There is one for implementing threads over *nix systems called the **POSIX Threads** or **pthreads**

These modules are accessible through a programming interface for C/C++

- You need to enter the header file called `pthread.h` for the same …
- … and link the pthread library at the linking phase (via the switch `-lpthread`)

We will have a look at some common usage examples in the next lecture

Homework !!

Watch the recorded Digression Lecture D2 and 0.2 from ITP

- Although we have covered the contents today as well, it is better to have a look at it again