Operating Systems (INFR10079)

2020/2021 Semester 2

Threads

(Basics)

abarbala@inf.ed.ac.uk

Chapter 4.1 (all), 4.2 (all)

Overview

- Concurrency vs Parallelism
- Process vs Thread
- Threads

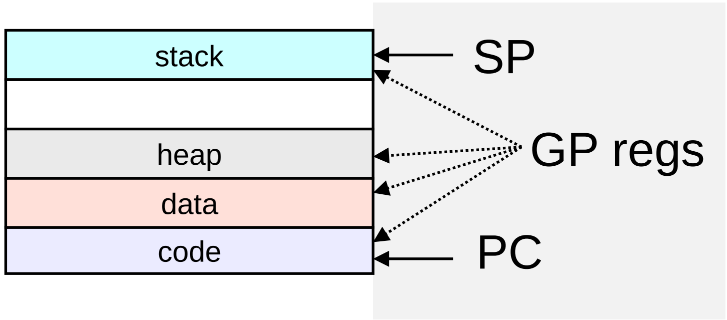
What’s “in” a process?

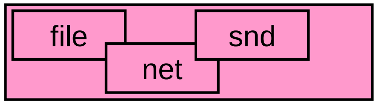
From previous slide-set

- A process consists of (at least)
  - An **address space**, containing
    - Code (instructions) for the running program
    - Data for the running program (static data, heap data, stack)
  - A **CPU state**, consisting of
    - Program counter (PC), indicating the next instruction
    - Stack pointer, current stack position
    - Other general-purpose register values
  - A set of **OS resources**
    - Open files, network connections, sound channels, …

instruction flow

Process address space

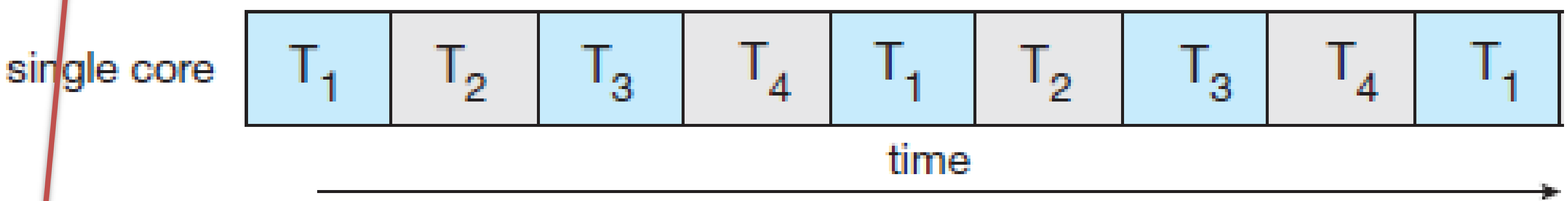




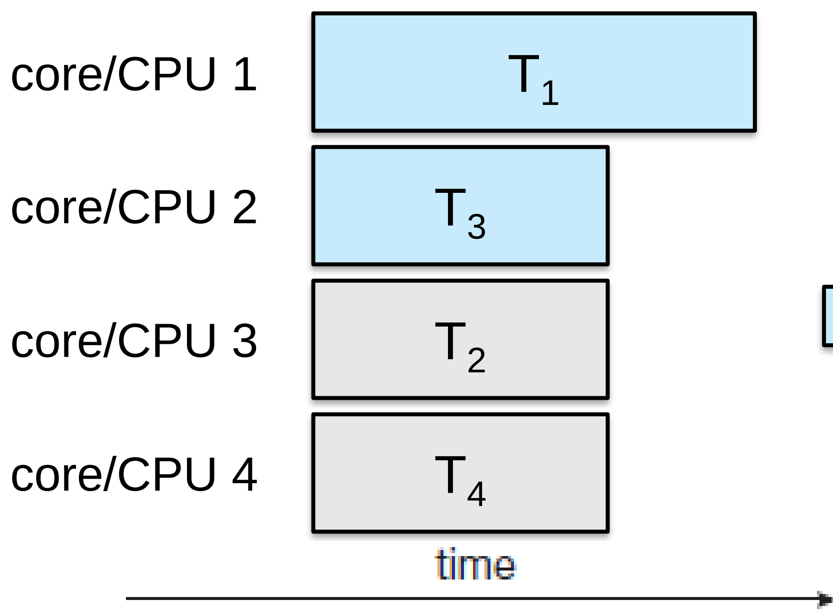
OS kernel address space

Concurrency **vs** Parallelism

- **Multiple tasks:** Task 1 (T_1), Task 2 (T_2), Task 3 (T_3), Task 4 (T_4)
- **Concurrent** execution on a **single-core system**

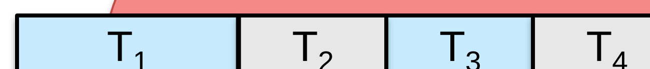


- **Parallel execution** on a **multicore/multiprocessor** system



All tasks make progress at the "same time"

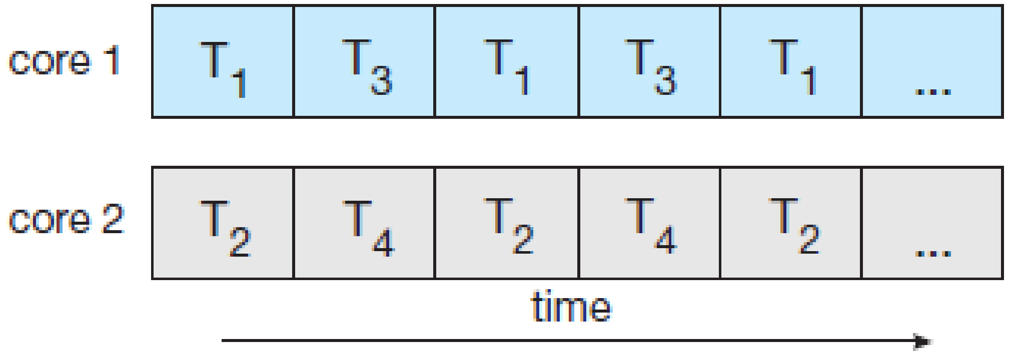
No multi-tasking



(One task at the time makes progress)

Concurrency **and** Parallelism

- **Both Concurrency** and **Parallelism**



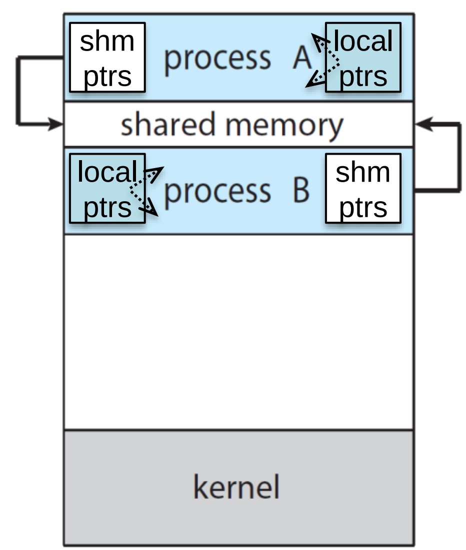
- Multiple **processes** to get *concurrency* and *parallelism*
  - Programs (code) of distinct processes are isolated from each other
- *What if they need to* ***communicate/share data?***
  - Message passing, OS in the middle – **slow**
  - Shared memory, not all pointers work – **limited shareability**
    OS resources not shared by default – **cumbersome**

Concurrent/parallel Communicating Processes

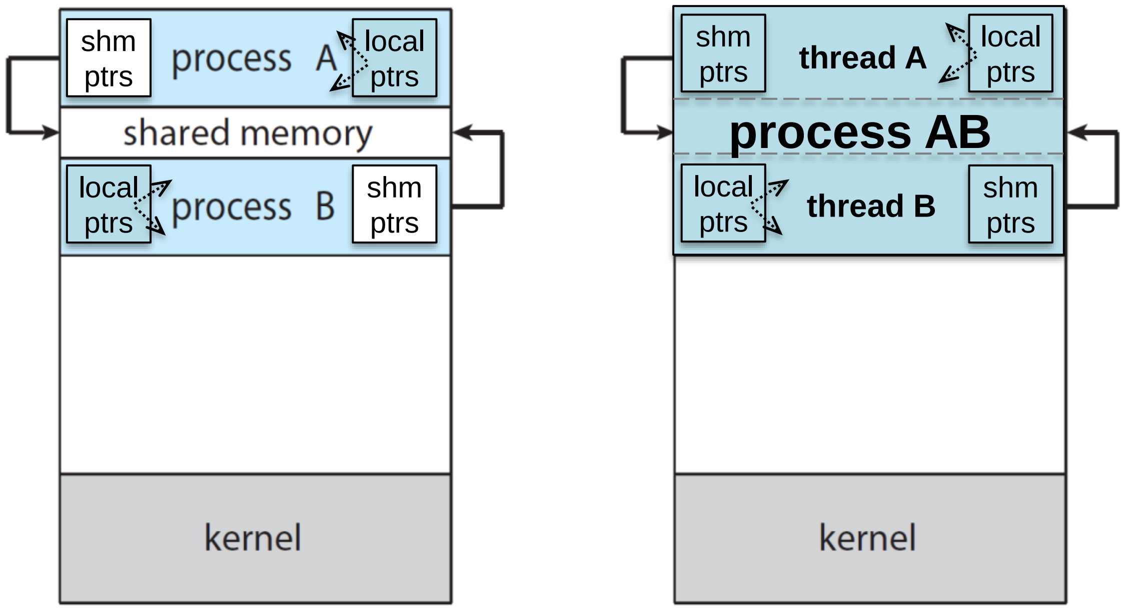
- Given the **process abstraction**
  1. Fork several processes
  2. Cause each of them to *map* to the same memory to share data
     - See **`shmget()`** API for one way to do this
  3. Make them both to *open* the same OS resources

- <u>Cumbersome</u>
- <u>Limited shareability</u>
  – Not all pointers work
- <u>Inefficient</u>
  – **Space**: PCB, page tables, etc.
  – **Time**: creating OS structures,
  fork/copy address space, etc.

Very Complex!



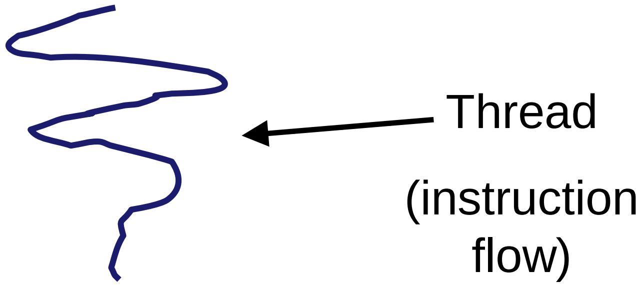
From Processes **to Threads**



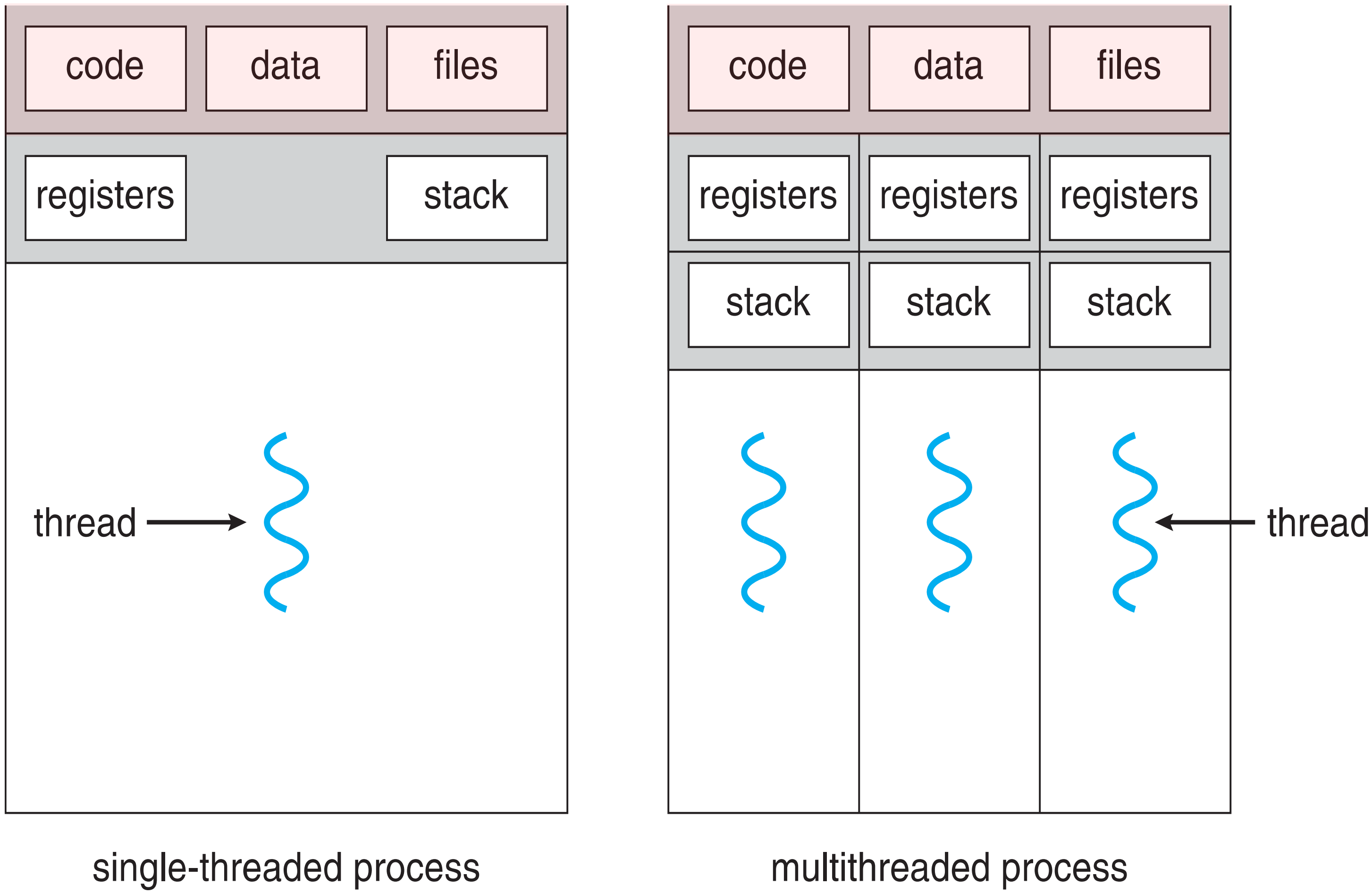
- Multiple **processes** to get *concurrency* and *parallelism*
  - Programs (code) of distinct processes are isolated from each other
- Multiple **threads** to get *concurrency* and *parallelism*
  - Threads **"share a process"** – same address space, OS resources
  - Threads have **different instruction flow** – private stack, CPU state

Threads

- Key **idea**
  - **Separate** the foundational components of a *process* (address space, execution state, OS resources)
  - Into different **abstractions/entities**
    - **PROCESS:** address space, OS resources
    - **THREAD:** CPU state (execution state)
      - program counter, stack pointer, other registers



Single-**threaded** and Multi**threaded** Processes

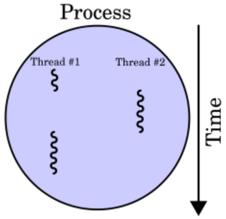


Use Case Scenario

- Various **instruction flows**
  - run the *same or different* **code**
  - access the *same* **data** (or part of it)
  - have the *same* **privileges**
  - use the *same* **OS resources**

- Each **instruction flow** has **hardware execution state**
  - Execution stack and stack pointer (SP)
    - Traces state of procedure calls made
  - Program counter (PC)
    - Next instruction to be executed
  - Set of general-purpose processor registers and their values

Threads and Processes

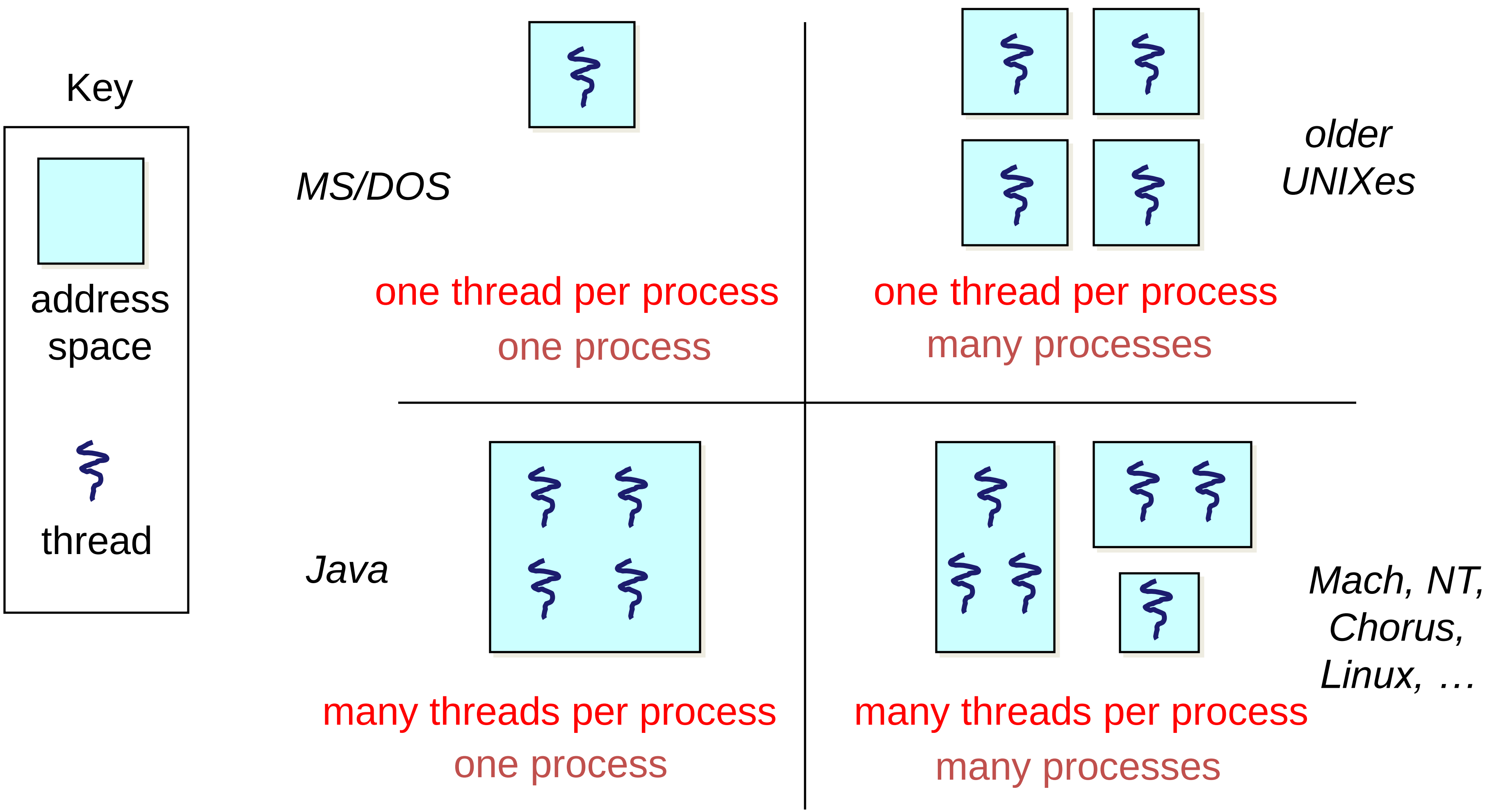


- Most modern OS's (Mach (Mac OS), Chorus, Windows, UNIX) support
  - **Process:** defines the address space and process' OS resources
  - **Thread:** defines a sequential **execution flow** within a process
- A **thread is bound to** a single process (thus address space)
  - However, processes (and address spaces) can have **multiple threads** executing within them
  - Sharing data between threads is **cheap**: all see the same address space
  - Creating threads is **cheap** too!
- **Threads** become the **unit of scheduling**
  - But depends on implementation (see next slides)
  - Processes are just **containers** in which threads execute

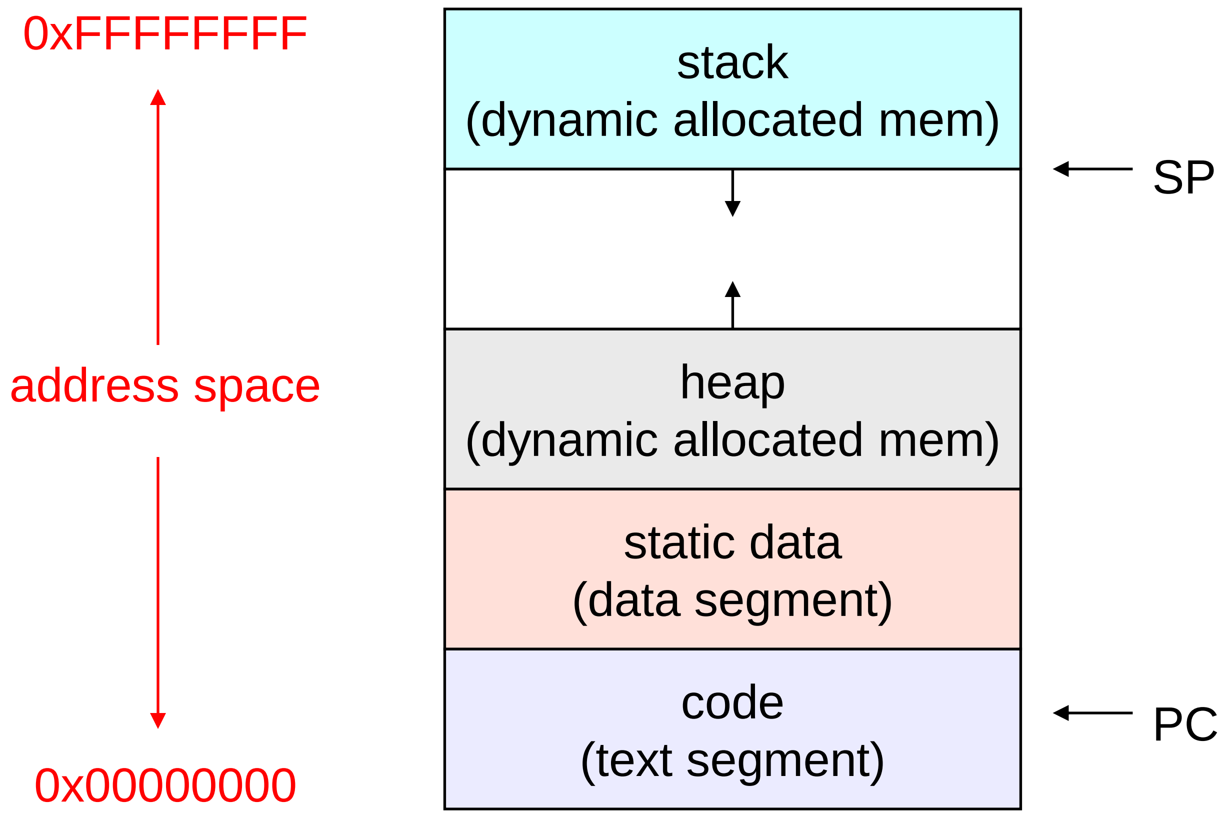
Communication

- **Threads** are diverse execution flows sharing an address space (and OS resources)
- Address spaces provide **isolation**
  - If you can't name it, you can't read nor write it
- **Threads** are in the same address space
  - Same name space (memory addresses)
  - Update a **shared variable**

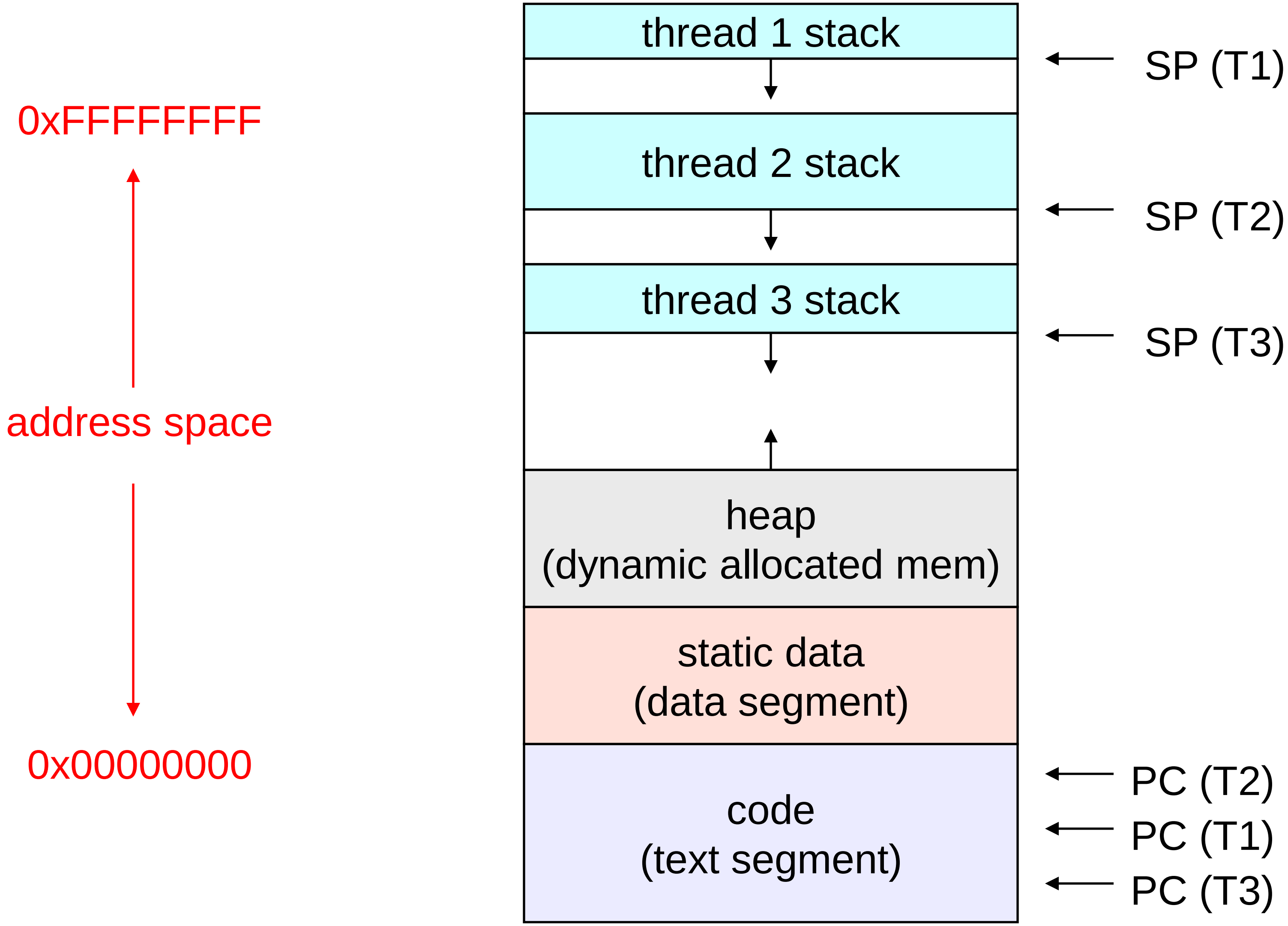
Historical Design Space



(Old) Process Address Space (32bit)



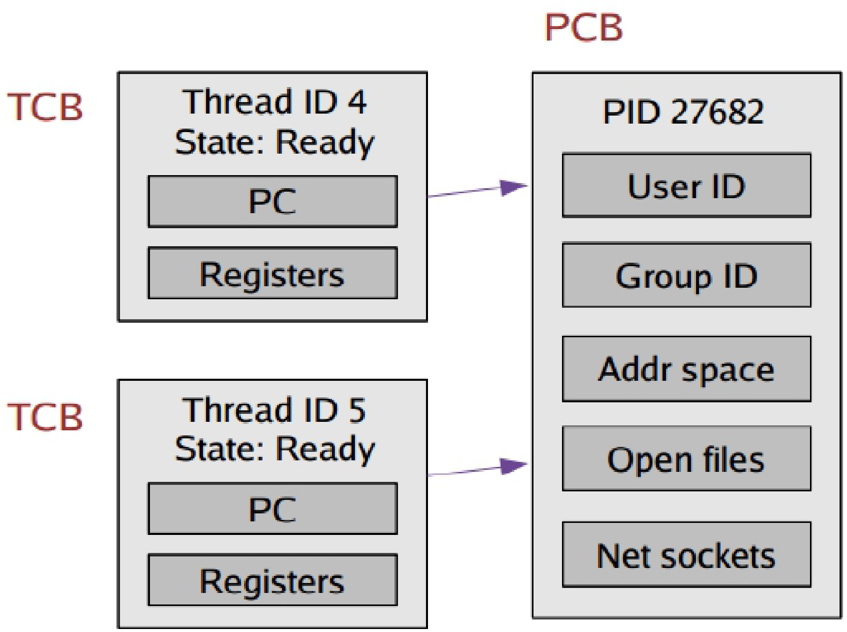
(New) Address Space with Threads (32bit)



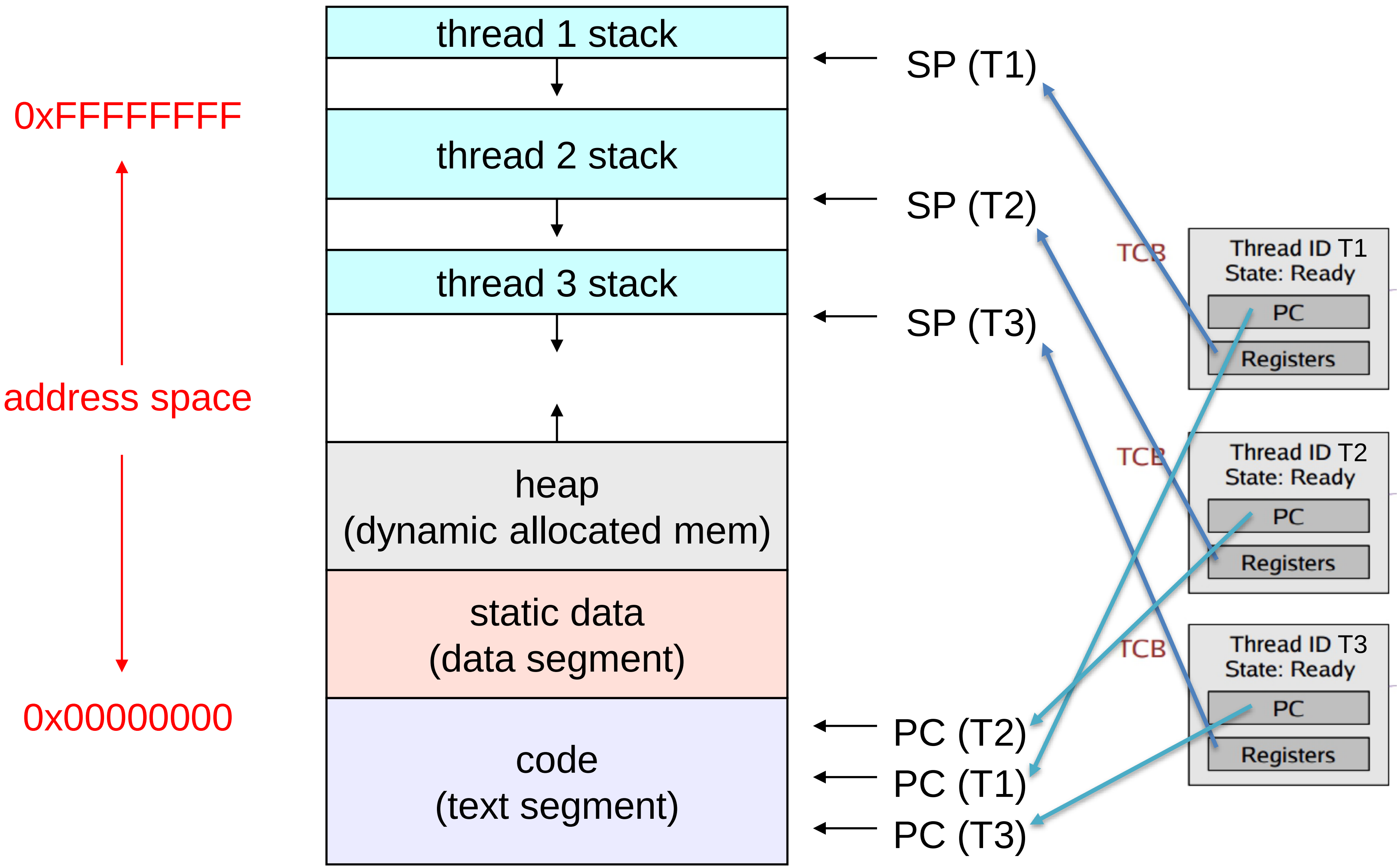
Thread Control Block (TCB) #1

- A PCB for each process

- Break the PCB into two pieces
- Info on program execution stored in Thread Control Block (**TCB**)
  - Program counter
  - CPU registers
  - Scheduling information
  - Pending I/O information
- Other infos stored in Process Control Block (**PCB**)
  - Memory management information
  - Accounting information



Thread Control Block (TCB) #2



Example Applications

- Multithreading is useful for
  - Handling concurrent events (e.g., web servers and clients)
  - Building parallel programs (e.g., matrix multiply, ray tracing)
  - Improving program structure (divide and conqueror)

- Multithreading is useful **on a uniprocessor**
  - Even though only one thread can run at a time

Terminology Note

- There is the potential for some confusion
  - “process” == “address space + OS resources + single execution flow”
  - **OR**
  - “process” == “address space + system resources + multiple execution flows”

- Single-threaded process: 1 thread
- Multi-threaded process: N>1 threads