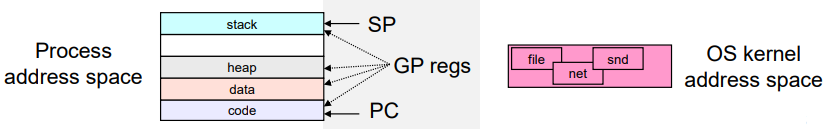
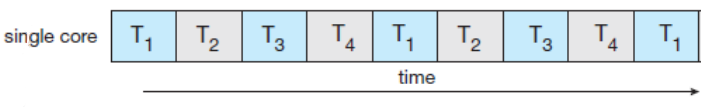
Operating Systems

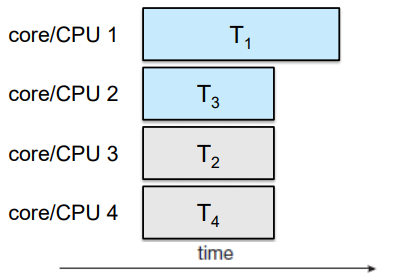
Threads

What’s ‘In’ a Process

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, etc…

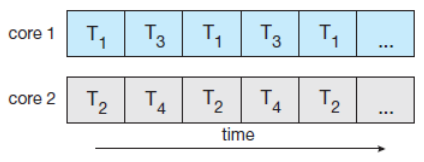
Concurrency Vs Parallelism

Say we have multiple tasks: Task 1 (T1), …, Task 4 (T4)

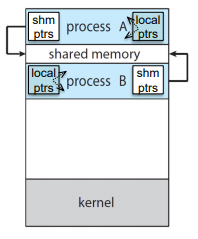
Concurrent execution on a single-core system involves switching out which task the CPU is working on.

Parallel execution on a multicore/multiprocessor system invovles running each task on a separate core/CPU at the same time.

Concurrency And Parallelism

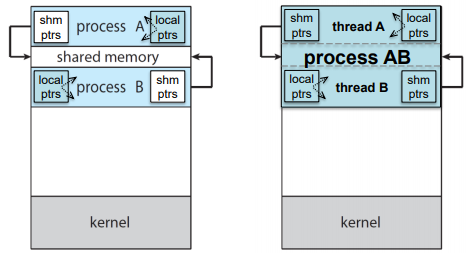
We can mix both concurrency and parallelism, in this case we can use concurrency for each core/CPU.

What if these tasks need to communicate or share data? Could use message passing (slow) or shared memory, but not all the pointers will work, it’s limited by shareability and the OS resources aren’t shared by default making it cumbersome.

Concurrent/Parallel Communicating Processes

Given the process abstraction

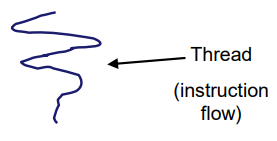
1. Fork several processes
2. Cause each of them to map to the same memory to share data
   1. See shmget() API for one way to do this
3. Make them both open the same OS resources

This is cumbersome, has limited shareability and inefficient (space, PCB, page tables etc…, and time, creating OS structures, fork/copy address space, etc..)

From Processes to Threads

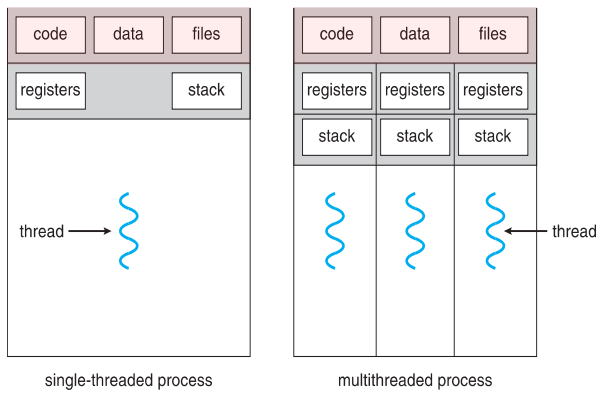
Instead of multiple processes we can have multiple threads all belonging to the same process, this means they will share the same address space and OS resources while still having different instruction flow (private stacks, CPU state).

Threads

The key idea is to 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 Multithreaded Processes



Use Case Scenario

Various instruction flows allow threads to:

* 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 a hardware execution state consisting of:

* an execution stack and stack pointer which traces the state of the procedure calls made
* a program counter (the next instruction to be executed)
* a set of general-purpose processor registers and their values

Threads and Processes

Most modern OS’s support both processes and threads which define the address space + OS resources and execution flow respectively. A thread is bound to a single process (and thus address space) however, processes (and address spaces) can have multiple threads executing within them. Sharing data between such threads is cheap as they all see the same address space and creating threads is cheap too.

Thread have become the unit of scheduling, but this depends on the implementation, processes are just containers in which threads execute.

Communication

Threads are diverse execution flows sharing an address space (and OS resources). The address spaces provide isolation for threads within a process as ‘if you can’t name it, you can’t read or write it’. Threads in the same address space have the same name space and as such can update shared variables to send data between themselves.