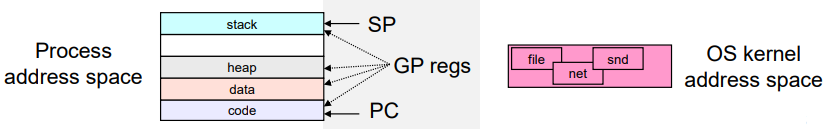
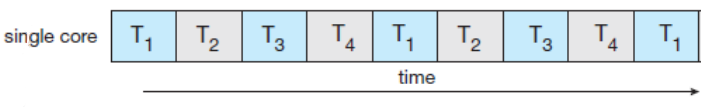
Operating Systems

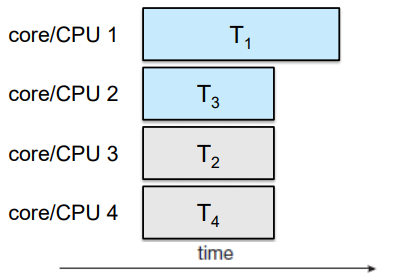
Threads

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…

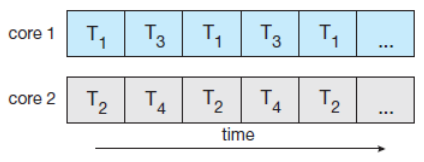
Concurrency Vs Parallelism

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

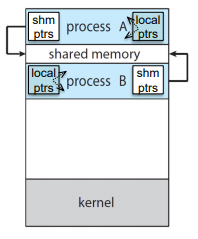
Concurrent execution on a single-core system involves switching out which task the CPU is working on.

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

Concurrency And Parallelism

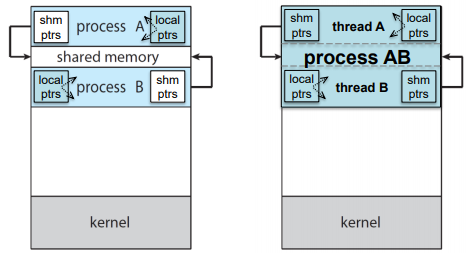
We can mix both concurrency and parallelism, in this case we can use concurrency for each core/CPU.

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

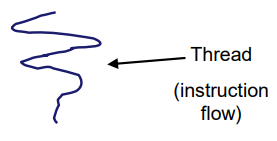
1. Fork several processes
2. Cause each of them to map to the same memory to share data
   1. See shmget() API for one way to do this
3. Make them both open the same OS resources

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

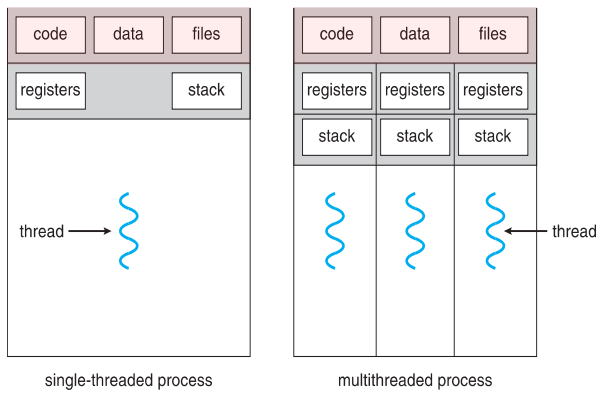
Instead of multiple processes we can have multiple threads all belonging to the same process, this means they will share the same address space and OS resources while still having different instruction flow (private stacks, CPU state).

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

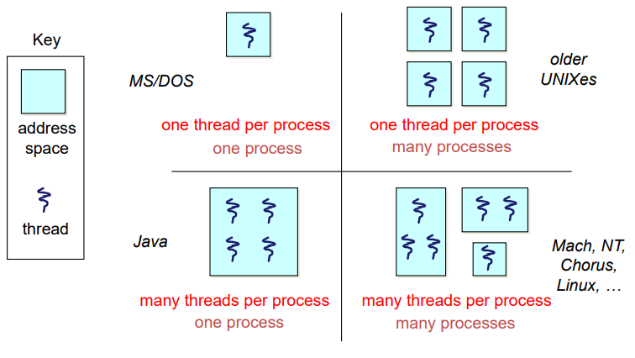
Various instruction flows allow threads to:

* 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

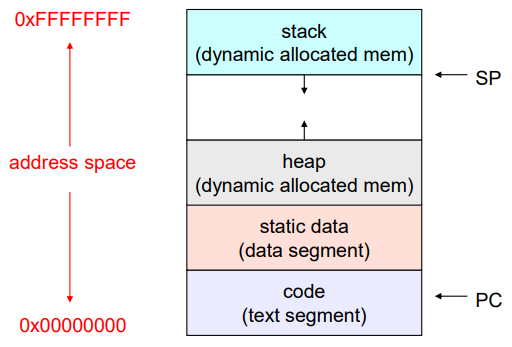
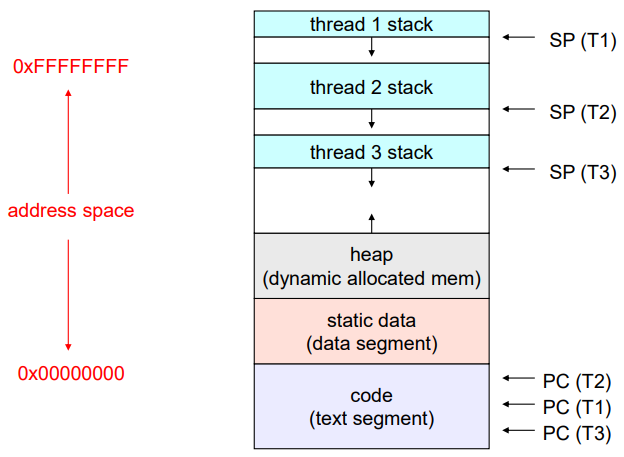
Threads and Processes

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.

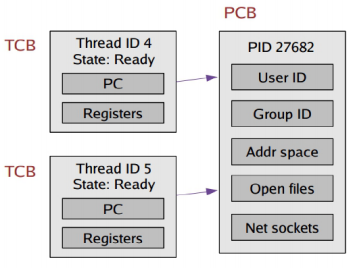
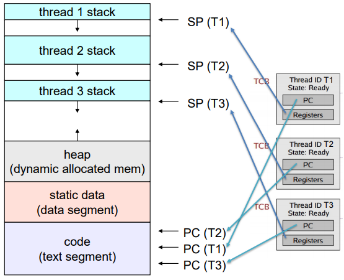
Communication

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.

Thread Control Block

We still have a PCB for each process, but we break the PCB into two pieces:

* info on the program execution stored in a Thread Control Block (TCB)
  + Program counter
  + CPU registers
  + Scheduling information
  + Pending I/O information
* Other info stored in the PCB
  + Memory management information
  + Accounting information

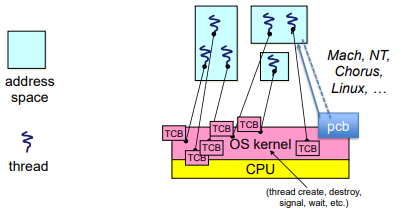
Who is Creating/Managing Threads?

The OS kernel is responsible for creating and managing threads, the kernel call to create a new thread would:

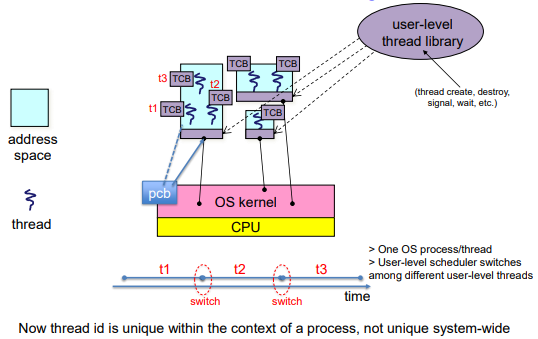
1. Allocate an execution stack within the process address space
2. Create and initialise a Thread Control Block (TCB)
   * Stack pointer, program counter register values…
3. Stick the thread on the ready queue

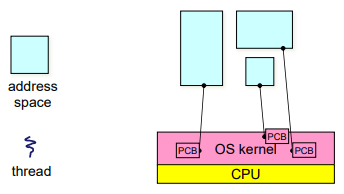
This is known as kernel-level threading or 1:1 threading. There is a ‘thread name space’ using thread identifiers (TID) which are integers similar to PIDs.

Kernel-Level Threading

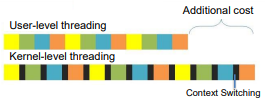
The OS manages threads and processes, all thread operations are implemented in the kernel. The OS schedules all threads in a system, if one thread in a process blocks the OS knows about it and can run other threads from that process, this makes it possible to overlap I/O and computation within a process.

Kernel-managed threads are cheaper than processes as they have less state to allocate and initialise but they’re pretty expensive for fine-grained use. They are orders of magnitude more expensive than a procedure call as thread operations are system calls which require contexts switches and argument checks, we must also maintain a kernel state for each thread.

User-Level Threading

This is an alternative to kernel-level threading. All the threads are managed at the user level, within the process, a library in the program manages the threads, the thread manager doesn’t need to manipulate address spaces (only the kernel can do that), threads differe (roughly) onlin in hardware contexts (PC, SP, registers) which can be manipulated by user-level code and the thread package multiplexes user-level threads in a process.

User-level threading is also known as 1:N threading, the kernel is unaware of the threads existences and TCBs are at the user level, this is all the kernel sees:

User-level threading is lightweight and fasts manged entirely by the user-level library. Each thread is represented simply by PC, registers, a stack and a small thread control block. Creating a thread, switching between threads and synchronising threads are all done via procedure calls with no kernel involvement necessary meaning no slow context switches.

User-level threading operations can be 10-100 times faster than kernel threads.

User-Level Threading Implementation

1. The OS schedules a process
2. The process executes user code (at the user-level) including the thread support library and its thread scheduler
3. The thread scheduler determines when a user-level thread runs using queus to keep track of what threads do (run, ready, wait, …) like to OS but in user-space.
4. Context switch at the user-level
   1. Save the context of the currently running thread (push CPU state onto the thread stack)
   2. Restore the context of the next thread (pop the CPU state from the next thread’s stack)
   3. Return as the new thread (resume execution at the PC of the next thread)

This works at the level of the procedure calling convention, no changes to memory mapping are required

How to Prevent User-Level Thread CPU Hogging

There are two main strategies:

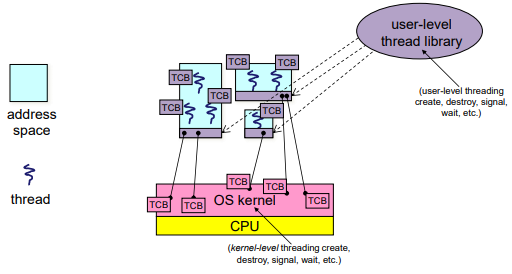
1. Forces everyone to cooperate
   1. A thread willingly gives up the CPU by calling yield() which calls into the scheduler which then context switches to another ready thread, but what if a thread never calls yield()?
2. Use pre-emption
   1. The scheduler requrests that a timer interrupt be delivered by the OS periodically, usually delivered as a UNIX signal (signals are like software interrupts but are delivered to the user-level by the OS instead of delivered to the OS by hardware). At each timer interrupt, the scheduler gains control and context switches as appropriate.

Thread I/O

If a thread tries to do I/O the process “powering” it “is lost” for the duration of the I/O operation:

* The process blocks in the OS
* The OS is not aware of the threads only seeing one thread-process
* None of the process’ other threads can make progress
* Other process can progress though

This is not the case with kernel-level threading, how can we merge kernel and user level threading to fix this?

THE N:M Threading Model

This model mixes both level of threading by allowing a many to many mapping of user-level threads to kernel level threads. The user-level threads essentially ‘attach’ to a kernel level thread, this means when a user-level thread attempts I/O, it’s associated kernel thread can have a different user-level thread ‘attached’ to continue working.

Explicit Vs Implicit Thread Interfaces

There are two ways to use threads, explicitly where you create wait for exit and join threads manually when writing the code (e.g. with the POSIX pthread API) this can be user or kernel or user level-level; or implicitly (user-level).

There are may types of implicit thread interfaces. These assign thread management to the thread library/ the runtime and identifies an application’s tasks, not threads. This uses compiler-level support (in most cases) with Code annotation, pragmas and templates.

Examples of implicit thread interfaces include:

* Thread pools
* Fork-join
* OpenMP
* Grand Central Dispatch
* Intel Thread building blocks
* Etc…