Conventional View of Processes

A process can be viewed two ways:

- A unit of **resource ownership**
  - A process has an address space, containing program code and data
  - A process may have open files, may be using an I/O device, etc.

- A unit of **scheduling**
  - The CPU scheduler dispatches one process at a time onto the CPU
  - Associated with a process are values in the PC, SP, and other registers

Insight (~1988) — these two are usually linked, but they don’t have to be

In many recent operating systems (UNIX, Windows NT), the two are independent:

- Process = unit of resource ownership
- Thread = unit of scheduling

Processes vs. Threads

- **Process** = unit of resource ownership
  - A process (sometimes called a heavyweight process) has:
    - Address space
    - Program code
    - Global variables, heap, stack
    - OS resources (files, I/O devices, etc.)

- **Thread** = unit of scheduling
  - A thread (sometimes called a lightweight process) is a single sequential execution stream within a process
  - A thread shares with other threads:
    - Address space, program code
    - Global variables, heap
    - OS resources (files, I/O devices)
  - A thread has its own:
    - Registers, Program Counter (PC)
    - Stack, Stack Pointer (SP)

Processes vs. Threads

A thread is bound to a particular process

- A process may contain multiple threads of control inside it
- Threads can block, create children, etc.

All of the threads in a process:

- Share address space, program code, global variables, heap, and OS resources
- Execute concurrently (has its own register, PC, SP, etc. values)

Why Threads?

A process with multiple threads makes a great server (e.g., printer server):

- Have one server process, many “worker” threads — if one thread blocks (e.g., on a read), others can still continue executing
- Threads can share common data; don’t need to use inter-process communication
- Can take advantage of multiprocessors

Threads are cheap!

- Cheap to create — only need a stack and storage for registers
- Use very little resources — don’t need new address space, global data, program code, or OS resources
- Context switches are fast — only have to save / restore PC, SP, and registers

But… no protection between threads!
**User-Level Threads**

- User-level threads = provide a library of functions to allow user processes to create and manage their own threads
  - ✔ Doesn’t require modification to the OS
  - ✔ Simple representation — each thread is represented simply by a PC, registers, stack, and a small control block, all stored in the user process’ address space
  - ✔ Simple management — creating a new thread, switching between threads, and synchronization between threads can all be done without intervention of the kernel
  - ✔ Fast — thread switching is not much more expensive than a procedure call
  - ✔ Flexible — CPU scheduling (among those threads) can be customized to suit the needs of the algorithm

**User-Level Threads (cont.)**

- User-level threads = provide a library of functions to allow user processes to create and manage their own threads
  - ❌ Lack of coordination between threads and OS kernel
    - Process as a whole gets one time slice
    - Same time slice, whether process has 1 thread or 1000 threads
    - Also — up to each thread to relinquish control to other threads in that process
  - ❌ Requires non-blocking system calls (i.e., a multithreaded kernel)
    - Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the process
  - ❌ If one thread causes a page fault, the entire process blocks

**Kernel-Level Threads**

- Kernel-level threads = kernel provides system calls to create and manage threads
  - ✔ Kernel has full knowledge of all threads
    - Scheduler may choose to give a process with 10 threads more time than process with only 1 thread
  - ✔ Good for applications that frequently block (e.g., server processes with frequent interprocess communication)
  - ❌ Slow — thread operations are 100s of times slower than for user-level threads
  - ❌ Significant overhead and increased kernel complexity — kernel must manage and schedule threads as well as processes
    - Requires a full thread control block (TCB) for each thread

**Two-Level Thread Model**

(Digital UNIX, Solaris, IRIX, HP-UX)

- User-level threads for user processes
  - “Lightweight process” (LWP) serves as a “virtual CPU” where user threads can run
- Kernel-level threads for use by kernel
  - One for each LWP
  - Others perform tasks not related to LWPs
- OS supports multiprocessor systems