Threads

• A thread or “lightweight process” consists of
  - Thread id
  - Program counter
  - Register set
  - Stack

• A process can have many threads, all sharing the same
  - Code section
  - Data section (globals + heap)
  - O.S. Resources (open files, signals)
Threads

- single-threaded app
  - code
  - data
  - stack

- multi-threaded app
  - code
  - data
  - stack

registers
resources
Benefits of using threads

- Responsiveness: some threads can block while others respond to user
- Resource sharing: all threads in a process share memory and resources
- Economy: threads are faster to create than processes
- Multiprocessor deployment: a multithreaded app will automatically use multiple cpus (e.g. One per thread).
User threads v. kernel threads

• **user threads** are implemented in a library. The OS doesn't know about them. Problem: one thread can be blocked in a system call, and OS will suspend entire process (all threads). Usually “cooperative”: no preemption, threads “yield”.

• **kernel threads** are known to the OS, so it can block one thread in a process while keeping the others running (ready). Problem: operations on threads may require system calls, which are costlier than library procedure calls.
Multithreading Models

- How are user threads mapped to kernel threads?
  - Many-to-one: many user threads in a single kernel thread. (blocking calls block all user threads)
  - One-to-one: each user thread runs in a kernel thread. Limits on number of kernel threads....
  - Many-to-many: many user threads on many kernel threads. Best of both worlds.

- However its done, not usually visible to users unless many-to-one. (blocking issue)