Single and multithreaded processes

![Diagram showing single-threaded and multithreaded processes.](image)
Why threads?

• Express **concurrency**
  – Web server (multiple requests), Browser (GUI + network I/O + rendering), most GUI programs ...
    ```
    for(;;) {
      struct request *req = get_request();
      create_thread(process_request, req);
    }
    ```

• **Efficient** communication
  – Using a separate process for each task can be heavyweight

• Leverage **multiple cores** (depends)
  – Unthreaded process can only run on a single CPU
Threads vs. Processes

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls `main()` & has the process’s stack
- Inexpensive creation
- Inexpensive context switching
- Efficient communication
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- A process has at least one thread
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- Interprocess communication can be expressive
- If a process dies, its resources are reclaimed & all threads die
Multithreading models

• Where to support threads?

• **User threads**: thread management done by user-level threads library; kernel knows nothing

• **Kernel threads**: threads directly supported by the kernel
  – Virtually all modern OS support kernel threads
User vs. Kernel Threads

E.g., GreenThreads, any OS (event ancient ones like DOS)

E.g., LinuxThreads, Solaris

Example from Tanenbaum, Modern Operating Systems 3 e,
(c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
Scheduling User Threads

• Non-preemptive Scheduling
  – No timer to make a thread yield the CPU
  – Threads must voluntarily yield control to let another thread run, e.g., `pthread_yield()`
  – Thread history isn’t taken into account by scheduler
  – Threads are *co-operative*, not competitive

• Preemptive Scheduling
  – Can use signals to simulate interrupts, e.g., alarm
  – But then user code can’t use alarm directly
User Thread Blocking

• What happens when a process does a read()?
  – Data needs to be fetched from disk
  – Kernel **blocks** the process (i.e., doesn’t return) until disk read is done
  – Kernel unaware of thread structure: all user level threads will block as well!

• One solution: wrapper functions
  – Thread library contains alternate versions of syscalls
  – Check for blocking **before** calling the kernel
  – E.g., select() before read()
  – If the call will block, then schedule another thread
  – Complex – need to handle **all** blocking calls!
User vs. Kernel Threads (cont.)

**User**
- **Pros:** fast, no system call for creation, context switch
- **Cons:** kernel doesn’t know ➔ one thread blocks, all threads in the process blocks
- **Cons:** can’t benefit from multicore or multiple CPUS

**Kernel**
- **Cons:** slow, kernel does creation, scheduling, etc
- **Pros:** kernel knows ➔ one thread blocks, schedule another
- **Pros:** can fully utilize multiple cores/CPUs

No free lunch!
Scheduler Activations

• Hybrid approach (Tru UNIX, NetBSD, some Mach, implementations for Linux)
  – Benefits of both user and kernel threads
  – Relies on upcalls (like signals)

• Scheduling done at user level
  – When a syscall is going to block, kernel informs user level thread manager via upcall
  – Thread manager can run another thread
  – When blocking call is done, kernel informs thread manager again

Multiplexing User-Level Threads

• A thread library must map user threads to kernel threads

• Big picture:
  – kernel thread: physical concurrency, how many cores?
  – User thread: application concurrency, how many tasks?

• Different mappings exist, representing different tradeoffs
  – Many-to-One: many user threads map to one kernel thread, i.e. kernel sees a single process
  – One-to-One: one user thread maps to one kernel thread
  – Many-to-Many: many user threads map to many kernel threads
Many-to-One

- Many user-level threads map to one kernel thread

- Pros
  - Fast: no system calls required
  - Portable: few system dependencies

- Cons
  - No parallel execution of threads
    - All thread block when one waits for I/O
One-to-One

• One user-level thread maps to one kernel thread

• Pros: more concurrency
  – When one blocks, others can run
  – Better multicore or multiprocessor performance

• Cons: expensive
  – Thread operations involve kernel
  – Thread need kernel resources
Many-to-Many

- Many user-level threads map to many kernel threads ($U \geq K$)
  - Supported in some versions of BSD and Windows

- Pros: **flexible**
  - OS creates kernel threads for physical concurrency
  - Applications creates user threads for application concurrency

- Cons: **complex**
  - Most programs use 1:1 mapping anyway
Thread pool

• Problem:
  – Creating a thread for each request: costly
    • And, the created thread exits after serving a request
  – More user request ➔ More threads, server overload

• Solution: thread pool
  – Pre-create a number of threads waiting for work
  – Wake up thread to serve user request --- faster than thread creation
  – When request done, don’t exit --- go back to pool
  – Limits the max number of threads
Other thread design issues

• Semantics of fork() system calls
  – Does fork() duplicate only the calling thread or all threads?
    – Running threads? Threads trapped in system call?
    – Linux fork() copies only the calling thread

• Signal handling
  – Which thread to deliver signals to?
  – Segmentation fault kills process or thread?

• When using threads
  – Make sure to use re-entrant functions
  – Only stack variables for per-call data (no globals)
  – If you want globals? Use thread-local storage (pthread_key_create), or an array with one entry per-thread