The Case for Threads

- Processes mix two concepts
  - Concurrency
  - Protection through isolation
- Due to protection we need (relatively) heavyweight IPC for communication and we need to copy memory
- Can’t we get only concurrency?

- Some computations can be parallelized and split between different concurrent threads of execution
- Servers are designed to handle multiple clients concurrently
  - Context switch overhead limits scalability
  - “Worker threads”
- Some applications require background activity
  - think word processor spell-check-as-you-type
  - think email client fetching new mail
- Would be difficult and/or expensive to implement with full process isolation

Threads

- Thread of execution = individual, concurrent execution stream
- Every process has at least one thread
- Threads belong to a process
- Threads have their own set of CPU registers (virtually) and their own stack
- Threads all share the heap and data segment of the process
- Context switching is more lightweight
Implementing Threads

- Processes define an address space; threads share the address space.
- Process Control Block (PCB) contains process-specific information:
  - Owner, PID, heap pointer, priority, and pointers to thread information.
- Thread Control Block (TCB) contains thread-specific information:
  - PC, thread state (running, running, ...), register values, a pointer to PCB,
  - Stack for Thread1, Stack for Thread2, Mapped shared libraries, Run-time heap, Read/write segment (.data, .bss).

Disclaimer: Linux Threads

- This is not entirely true in Linux.
- Linux: Thread is more like a special kind of process.
- Thread and Process are both tasks.
- Internally described by a task_struct.
- Tasks can vary in their degree of sharing:
  - (Process, Thread).
- Linux scheduler deals with all kinds of tasks.

Flavors of Threads

- Kernel-Level Threads:
  - The threads that the kernel is aware of and schedules/dispatches, ...
  - Example NPTL.
  - N x N model:
    - one kernel entity = one application thread.
- User-Level Threads (1xN):
  - Process is responsible for creating, handling, dispatching, maintaining, scheduling threads.
  - Usually through a library.
  - 1 x N mode:
    - one kernel entity (process) drives N application threads.
Threads

- **Practical Differences:**
  - OS can deal with blocking threads
  - OS: system calls + context switches = slower

- **Hybrid Threads (NxM)**
  - Kernel passes schedulable entities down to the application
  - User-level threading library schedules application threads to schedulable entities
  - Very fast context switching, avoids system calls
  - Example: Scheduler Activations (Tom Anderson), used in Windows 7+

POSIX Threads

- **pthread_t**
  - (Opaque) handle to a thread
  - Like the PID for a process

- **int pthread_create**
  - Creates a new thread which will start executing the thread_function. Can pass a pointer as an argument.

POSIX Threads

- Returning from a thread
  - return from within the thread function.
  - int pthread_exit (void *status);

- Waiting for a thread to terminate
  - int pthread_join (pthread_t thread, void **status_ptr);
  - similar to wait_for_pid

- Avoiding thread identity crisis
  - pthread_t pthread_self ();

- Using pthreads
  - #include <pthread.h>
  - gcc -lpthread ...