Lecture 16, Feb 14, 2024

Threading Implementations

- Threading can be implemented either as user threads, or kernel threads
  - User threads run completely in the user-space; the kernel doesn't treat the process any differently
  - Kernel threads are managed by the kernel and gets treated specially
- In both models, threads require a thread table (just like a process' process table); this is either in user or kernel space depending on implementation
- Generally threading libraries can work in one of three ways:
  - Many-to-one: many user threads are mapped to a single kernel thread; threads are completely managed in user space
    - The kernel only sees a single process
    - Since everything is done in user space, thread creation/deletion is fast and no context switching is needed
    - However if one thread blocks, all other threads are blocked since the kernel can't distinguish
    - Does not allow parallelism since the kernel only sees a single process
  - One-to-one: each user thread maps to a single kernel thread
    - The kernel handles everything while the threading library is just a thin wrapper
    - Threads are slower, but can execute in parallel; one thread can't block everything
    - This is what `pthread` is
  - Many-to-many: many user threads map to many kernel threads
    - We have more user threads than kernel threads
    - Can set the number of kernel threads to the same as the number of CPU cores to fully allow parallelism
    - In theory gives the best of both worlds, but in reality can be very complicated and unpredictable
- What happens when we call `fork()` on a multithreaded program?
  - On Linux the new process has only one thread, corresponding to the one that called `fork()`
  - `pthread_atfork()` can be used to register functions to be run on fork, for advanced control of forking behaviour
- If a multithreaded process gets sent a signal, on Linux only one random thread will receive the signal
  - We have no control over the thread that gets interrupted
- Instead of many-to-many thread mappings, a *thread pool* is often used
  - A thread pool maintains a number of threads, usually corresponding to the number of CPUs in the system
  - When no tasks are given, the threads are sleeping; as tasks come in, the threads get waken up and starts executing the tasks
  - These threads are reused after tasks are done
  - This is often done when you have lots of very small tasks, so thread creation can introduce significant overhead