Topic 11: Threads vs. Processes

Reading: None
Next reading: 7.1

- The process abstraction is a convenient way of allowing multiple computations to share a single computer. A process consists of a sequential execution stream (processor state), memory, and the state of the resources the computation is using (open files, current working directory, etc.).
    - Processes are useful because they provide each computation with its own virtual machine, allowing computations to be independent.
    - Processes are relatively heavy-weight, however, since there is a lot of work involved in creating and destroying processes, and switching between them.
    - The process abstraction may also not be suitable for all applications, since cooperating processes may not need nor want separate memory.
- A lighter-weight abstraction is that of the *thread*, which is simply a sequential execution stream (thread of control).

Multiple threads can share memory and other resources -- a collection of such threads is called a *process* or a *task*.

- Each thread has its own stack and processor state.
- Unlike processes, the threads of a process are not independent. They share memory, and thus there is no protection between them. This is ok because the threads are part of the same process.
- Synchronization mechanisms such as semaphores require shared memory.
- In USLOSS, what we've been calling "processes" are actually threads within the single UNIX process that runs USLOSS.

- Why write a multi-threaded application?
  - Multiple threads allow a single task to overlap computation and I/O.
    - One thread computes, while another waits for I/O to complete. Consider a web server that handles many clients.
    - Multiprogramming only overlaps computation and I/O of different processes. This improves system throughput, but not the running time of individual tasks.

- Simplified programming (e.g., producer/consumer). Some problems are easier solved by cooperating threads, rather than by a single process that does everything.
- Parallel computation on multiple processors. The computation is done by many threads, allowing many processors to be used.

- Where should threads be implemented?
  - User-level (library):
    - OS implements processes -- processor state and memory.
    - Library (thread package) implements threads, thread locking/synchronization, and thread scheduling.
    - Advantages:
      - Fast context switch between threads (no system calls).
      - Application-specific thread scheduling.
      - Application-specific locking and synchronization.
    - Disadvantages:

        - OS schedules processes, which may interfere with thread scheduling. In particular, if one thread blocks for I/O the whole process blocks (and all of the threads it contains), undermining one of the reasons for having threads.
        - Threads share OS resources such as open files, etc. May be too inflexible.
        - It is difficult for a user-level scheduler to implement a time quantum. Threads must voluntarily give up control of the processor by invoking the scheduler. USLOSS does this using a UNIX signal, which is very heavy-weight.
        - One process per application means that a single application cannot take advantage of a multiprocessor.
    - Kernel-level (in the OS):
        - OS separates processor state from address spaces.
        - Advantages and disadvantages are opposites of user-level threads.
- On a multiprocessor, it is difficult to know how many threads to create.