Threads

Chapter 4 of [OS3e],
Chapter 5 of [OSC]:
- Thread Concept
- Kernel-level Threads
- User-level Threads
- Terminology
- Threads in Java

Thread Concept

Threads (of control) come in two flavours
- user-level threads (application threads)
  - view programmer has of how control passes through program
  - sequential programs have single thread
  - parallel programs have multiple threads, with threads dynamically created and destroyed
- kernel-level threads (light-weight processes)
  - unit of dispatching by OS

How are processes and kernel-level threads related?
Each process within system associated with:
- one or more kernel-level threads
  - only one thread assumed last lecture
- resources (address space, open files, I/O channels,...)

Each of these threads:
- shares access to all resources held by process
- is in running, ready or blocked state
  - threads are queued individually for short-term scheduling
  - process is simply in admitted or suspended state
- can switch between user and supervisor mode
  - runtime stack for procedure/method calls made in user mode
  - system stack for calls made in supervisor mode (if OS executes within user process)
Modern operating systems e.g., Windows XP, Solaris
e.g., Linux (using clone system call)
( unlike older versions of UNIX)
support multiple kernel-level threads per process.

Why?

_Efficiency:_
- creation/termination is faster for threads than for processes (creation by factor of 30 in Solaris)
- switching between threads of same process is faster than switching between processes (by 5x in Solaris) because there is less work involved

**User-level Threads**
- Within a given process, each kernel-level thread operates on same program
- It engages in execution of (at least) one user-level thread

This implies
- applications and system software must be expressed as parallel programs (multithreaded) in order to take advantage of multiple kernel-level threads

Another reason for writing parallel programs (even on top of single-threaded OS) is _convenience_
e.g. it is natural to assign user-level thread to each client of (Web, ftp, file,...) server
Multithreading an interactive application on a multithreaded OS generally improves responsiveness of the application. This is because:

- User-level threads mapped on to ready kernel-level threads can make progress even though others may be blocked or are long-running.
- Threads can execute simultaneously if there are multiple processors (multiprocessing).

More threads available for scheduling also implies better utilisation of processors.

What do you need to write multithreaded applications?

*Threads library*
- Package of routines for managing user-level threads
  - E.g., pthreads on POSIX-compliant systems
  - E.g., Windows supports application-scheduled fibers, as well as kernel-scheduled threads

*Or programming-language support*
- E.g., Java’s Runnable interface
How is threads library or JVM implemented on OS?

- Pure user-level: library/JVM maps (dynamically schedules) all threads within process on to single kernel-thread

  This isn’t ideal because
  - it cannot exploit multiprocessing
  - OS schedules processes, being unaware of threads, so if one user-level thread blocks, all are blocked

  Note that blocking can be avoided by “jacketing” requests for I/O by first checking (with non-blocking call) availability of device

- Pure kernel-level: each user-level thread simply maps on to its own kernel-level thread
  - e.g., Windows XP, Solaris 1:1 scheduling model
  - overcomes problems of pure user-level
  - switching threads may be slower because it involves switching to and returning from supervisor mode

- Combined approach: user-level threads are mapped on to kernel-level threads
  - e.g., Solaris MxN scheduling model
  - synchronization at user-level can be much cheaper (1 instruction) than at kernel-level (100s or 1000s)

Note that MxN model was retired in Solaris 9 because switching between threads at user-level turned out not to be significantly faster than at kernel-level
Terminology

Try not to get confused by the fact that
- “kernel-level threads” switch between “user mode” and “supervisor mode”, whereas
- “user-level threads” are mode-less

Watch out that
- in [OSC] “-level” is missed out, and
- in most books “supervisor” and “kernel” are used interchangeably
Threads in Java

When a JVM is invoked on a Java class
- **main** method of that class executes on a thread
- further threads can be explicitly constructed and started by a running thread
- other internal threads also start running in the background upon initialization of the JVM
  e.g., to handle garbage collection

N.B. The way these user-level threads are mapped to kernel-level threads varies from one implementation of the JVM to another

- Code that is to be executed on a separate thread is best isolated in the **run** method of a class that implements Runnable
- A thread is an object that is created when an instance of the class is supplied as an argument of the constructor **Thread**
- A thread starts running when its **start** method is invoked

```java
public class MultithreadedJavaApplication {

    public static void main(String argv[]) {
        class Code implements Runnable {
            public void run() {
                // code to be executed on child thread
            }
        }
        Code c = new Code();
        Thread t = new Thread(c);
        t.start();
        // code to be executed on main thread after spawning child
        try{
            t.join();
        }
        catch(InterruptedException e) {
        }
        // code to be executed on main thread after termination of child
    }
}
```