This chapter introduces concepts associated with multithreaded computer systems and covers how to create and manipulate threads.

Motivations and Benefits of Threads

- A process contains two major components, *thread* (execution unit) and *address space*. A thread is a flow of control within a process, represented by *program counter*, *stack* and *registers* (including *execution flags*). A multithreaded process contains several different flows of control within the same address space.
- Examples of multi-thread processes: A web browser (the client) might have one thread displaying images or text while another retrieves data from networks. A word processor can have a thread for displaying text, one handling input from keyboard, and another thread performing spelling checking in the background.
- The distinctive feature in a multithreaded process is that there is **concurrency** in execution of different threads within one process. If these are realized or implemented as separate processes (which is feasible), it would create a great deal of redundancies (e.g., both data and program are duplicated) and add significant complexities, e.g., frequent context switching between different processes wastes lots of times and resources.
- The main motivation for introducing **the concept of a thread** is that a process may involve performing multiple tasks; creating several processes to handle each task is cumbersome, both time-consuming and resource-consuming. Such tasks are often tightly related in that they share codes, data or other resources (e.g., opened files). Creating multiple threads within a single process eases this job. Otherwise, multiple processes need to be created with redundant codes and data.
- A thread is an "**active**" entity, each having its thread identifier (ID), program counter, registers and stacks. A thread shares resources with other threads belonging to the same process such as code, data, files and heap. This creates the concurrency within the execution of a single process with the existence of multiple threads.
- Each thread is represented in OS by a **Thread Control Block** or **TCB** that specifies the state of a thread, scheduling and accounting information. If a process has more than one thread, there is no definition on the state of the process, only the state for each individual thread within that process.
- There are four primary benefits to multithreaded applications: (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- **Concurrency** vs**. Parallelism**: Concurrency exists when multiple threads are making progress, whereas parallelism exists when multiple threads are making progress simultaneously. On a system with a single CPU, only concurrency is possible; parallelism requires a multicore system that provides multiple CPUs.
- **Multicore systems**: multiple computing cores reside on a single chip, which are more efficient than multiple physical chips each with single core, because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for mobile devices as well as laptops relying on battery.

- There are many challenges in designing multithreaded applications such as dividing and balancing the work, dividing the data between the different threads, and identifying any data dependencies. Finally, multithreaded programs are especially challenging in test and debug.
- **Data parallelism** distributes subsets of the same data across different computing cores and performs the same operation on each core. **Task parallelism** distributes not data but tasks across multiple cores. Each task is running a unique operation.

Multi-threading Models

- The support for multithreads can be provided either at the user level, referred to as *user threads*, or by the kernel, called *kernel threads*. Only kernel threads are supported and managed by the OS. All modern OSes support kernel threads.
- User-level threads are visible only to programmers and unknown to the kernel. Threads libraries provide APIs for creating and managing user threads. Three commonly used thread libraries are: POSIX Pthreads (for Unix and Linux), Window threads, and Java threads. This creates modularity in programs.
- Ultimately, there must be a mapping between user threads and kernel threads, so that the OS can schedule and manage threads for execution. There are three common ways of mapping.
- **Many-to-One** model: many user threads are mapped into one kernel thread. Thus, thread management is done by the thread library in user space; this only provides efficiency or convenience for user programming with modularity. The entire process will be blocked if one thread makes a blocking system call since only one thread can access the kernel at a time. Multiple threads are unable to run in parallel in multi-processor systems.
- **One-to-One model**: each user thread is mapped into one kernel thread. It provides the maximum concurrency by allowing each thread to run when another thread is blocked; It also allows multiple threads (within a process) to run in parallel on multiprocessors. The drawback is that the creation of each user thread requires the creation of a corresponding kernel thread. There is usually more overhead involved in creating a kernel thread than creating a user thread. Each kernel thread consumes certain resources, so most implementation of this model restricts the total number of threads that can be supported in a system.
- **Many-to-Many model**: this typically allows many user-level threads to be mapped to a smaller or equal number of kernel threads. It provides better concurrency than many-to-one model, less concurrency than one-to-one model. This is flexible in that the number of user-threads created is not restricted by the number of kernel threads available in a system.
- Please notice that a kernel thread is what the operating system manages, so a TCB is only associated with a kernel thread, not a user thread. In another word, a kernel thread is the only thread that CPU executes, and it can run either a kernel program or a user program. It is the kernel thread that CPU scheduler or the short-term scheduler is scheduling to execute (to be discussed in Chapter 5).

Threading Issues

- In a multi-threaded program, fork() called by a thread creates a new process that either duplicates all threads of the parent process or only the thread that invokes the fork(). exec() works in the same way that replace the entire process including all threads. Apparently, if exec() is called right after fork(), there is no need for fork() to duplicate all threads in the process, but just one thread.
- Creating a new thread within a process is much simpler than creating a new process, since the process-specific data structure is unchanged; only thread-specific data structure needs to be handled including a new TCB and stack.
- A **signal** is used in UNIX to notify a process that a particular event has occurred. A signal can be either *synchronous* or *asynchronous*, handled by a default handler or a user-defined handler (which overrides the default handler). Synchronous signals are usually delivered to the same process that trigger the signal (for example, illegal memory access and division by zero). When a signal is generated by an event external to a running process, that process receives the signal asynchronously (for example, an external timer expires, or "kill").
- Threads may be terminated using either **asynchronous** or **deferred cancellation**. Asynchronous cancellation stops a thread immediately, even if it is in the middle of performing an update. In this case, the operating system may not be able to reclaim all resources. Deferred cancellation informs a thread that it should terminate but allows the thread to terminate in an orderly fashion. In most circumstances, deferred cancellation is preferred to asynchronous termination.
- The ***user-thread library*** provides programmers with API for creating and managing threads – user threads (visible only to programmers). The OS uses one of the three mapping models to map user thread to kernel thread(s).
- Under the many-to-many model, the OS provides coordination between the kernel and the thread library, which allows the number of kernel threads mapped to be dynamically adjusted to help ensure the best performance.
- An intermediate data structure called a ***lightweight process***, or ***LWP*** between the user and kernel threads is typically used. To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run (thread scheduling will be discussed in Chapter 5). Each LWP is attached to a kernel thread, and it is kernel threads that OS schedules to run on CPU(s). If a kernel thread blocks (e.g., waiting for I/O), the LWP blocks as well.
- Unlike many other operating systems, Linux does not distinguish between processes and threads; instead, it refers to each as a task. The Linux clone() system call can be used to create tasks that behave either more like processes or more like threads depending on the resources shared with the parent.
- The clone() system call in Linux, instead of creating a copy of the parent process like in fork(), it creates a separate process that can share the address space of the calling process. The set of flags is used to determine the extent of sharing. The clone() requires loading a new function (specified by *fn) to execute.