Motivations and Benefits of Threads

- Examples of multi-thread processes: A web browser might have one thread displaying images or text while another thread retrieves the data from the network. The word processor application such as Microsoft Word might have one thread for displaying text, another thread handling the input from the keyboard, and a third thread performing spelling checking in the background. The distinctive feature in such a process is that there are concurrency in the execution of different entities in the process. If these are implemented as separate processes (which is doable), these would create great deal of redundancies (data and programfor instance) and add significant complexities (consistent switching between different processes waste lots of times).
- The main motivation for introducing **the thread** is that a process may require or 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 either sharing codes, data or other resources (e.g., opened files). Creating multiple threads within a single process eases this job. Otherwise, multiple processes have to be created with redundant codes, data and etc.
- Each thread is an active execution unit (entity), each having its thread identifier (ID), program counter, registers and stacks (for saving the context of the thread). Thus, a thread shares many other resources with other threads belonging to the same process such as code, data, files and heap (dynamic memory allocated to a process). This creates concurrency within the execution of a single process with the existence of multiple threads.
- Each thread has 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 process state, only the state for each individual thread within that process.
- This can bring a number of advantages in terms of responsiveness, resource sharing, economy and taking better utilization of multi-processor architecture.
- **Concurrency vs. Parallelism**: a parallel system can perform more than one task simultaneously. A concurrent system supports more than one task by allowing multiple tasks to make progress interleaved with one another. In another word, multiple tasks are multiplexed onto one CPU over the time.

Multi-threading Models

- The support for multithreads can be provided either at the user level, called *user threads*, or by the kernel, called *kernel threads*. Kernel threads are supported and managed directly by the OS. All modern OS support kernel threads.
- User-level threads are threads that are visible only to programmers and unknown to the kernel. Threads libraries provide application programmers with 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 kernel can handle such threads for execution. There are three common ways of mapping.
- **Many-to-One** model: many user threads are mapped into one kernel thread. Thread management is done by the thread library in use space; this only provides efficiency 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. Also multiple threads are unable to run in parallel on multi-processors. In another word, there is no concurrency in process execution.
- **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 (currently running) 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, which is a hybrid model of the first two models. 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.
- **Two-level model**: similar to the many-to-many model but restricts a user-level thread to be bounded to certain kernel thread. For instance, user threads A and B are mapped to a kernel thread X, and a user thread C can only be mapped to a kernel thread Y. In many-to-many model, each user thread can be mapped to any kernel thread available. Apparently, many-to-many model is more flexible.

Threading Issues

- In a multi-threaded program, fork() called by a thread creates a new process that either duplicates all threads of the 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 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 the new TCB and stack.
- A **signal** is used in UNIX to notify a process that a particular event has occurred (for instance, illegal memory access, divide by 0, timer expires). A signal is either synchronous or asynchronous, which can be handled by a default handler or a user-defined handler (that overrides the default handler).
- Thread cancellation involves terminating a thread (called *target thread*) before it

has completed. This can be either **asynchronous** (terminate immediately) or **deferred**, which periodically checks whether or when to terminate. The later is safer for resource reclaim.

- The clone() system call in Linux behaves similarly as fork(), but instead of creating a copy of the process like in fork(), it creates a separate process that shares the address space of the calling process. The flags are used to determine the extent of sharing.