Chapter 4

Threads
Process Characteristics

- We’ve mentioned a process as having two characteristics

- Unit of resource ownership:
  - processes have their own dedicated memory address space
  - processes temporarily own I/O devices, files, etc.
  - OS protects the resources owned by the process from unwanted interference by other processes

- Unit of dispatching (scheduling / execution)
  - a process is an entity that is scheduled and dispatched by OS
  - processes each have their own execution state (ready, running, etc.) and priority
  - the execution of one process gets interleaved with other processes (multiprogramming)
Process and Thread

- These two characteristics (i.e., resource ownership and dispatching) are treated independently by modern OS:
  - The unit of dispatching is usually referred to a thread or a lightweight process.
  - The unit of resource ownership is usually referred to as a process.

- That is, by introducing the concept of “threads”:
  - threads are the true unit of dispatchable work.
  - the process is then responsible for resource ownership.
Process and Thread

- **Single threading**: when the OS does not use the concept of thread
- **Multithreading**: when the OS supports multiple threads of execution within a single process

- MS-DOS support a single user process and a single thread
- Early UNIX supports multiple user processes but only supports one thread per process
- All modern operating systems support multiple threads
Process and Thread

- the ability to support multiple, concurrent paths of execution within a single process

Diagram:

- One process, one thread
- One process, multiple threads
- Multiple processes, one thread per process
- Multiple processes, multiple threads per process
Process and Thread

- Associated with a process (resource ownership / protection):
  - memory address space that contains the process image
  - protected access to resources (processors, I/O devices, other resources)

- Associated with a thread (unit of execution):
  - a thread execution state (e.g. running)
  - a saved thread context when not running
  - an execution stack
  - per-thread storage for local variables
  - access to memory and resources of its owning process (shared with all other threads in the process)
Process and Thread

Single-threaded process model
- Process control block
- User address space
- User stack
- Kernel stack

Multithreaded process model
- Thread
  - Thread control block
  - User stack
  - Kernel stack
- Thread
  - Thread control block
  - User stack
  - Kernel stack
- Thread
  - Thread control block
  - User stack
  - Kernel stack
- Thread
  - Thread control block
  - User stack
  - Kernel stack

User address space
General advantages of threads

- faster to create a new thread than a new process
- terminating a thread is faster than terminating a process
- switching between threads within the same process is faster than switching between processes
- inter-thread communication is more efficient than inter-process communication

Why?

Threads have access to shared process data and can interact without invoking the kernel
Process and Thread

- For example: Consider an application that consists of several independent parts that do not need to run in sequence.
- Each part can be implemented as a thread.
- Whenever one thread is blocked waiting for an I/O, execution could possibly switch to another thread of the same application (instead of switching to another process).
Internet of Things
How it Works
Uses of Threads

- Foreground and background work
  - for example, having separate threads of execution allows a GUI to remain responsive even when doing heavy computation in the background

- Asynchronous processing
  - asynchronous elements (event handling) are well-suited to being implemented as threads
  - an event thread might exist solely to respond to user actions, delegating the actual work to another thread
Uses of Threads

- **Speed of execution:**
  - multiple threads within the same process can be executed on multiple processors simultaneously

- **Modular programming structure:**
  - programs involving a variety of activities can be logically broken up into different threads, each of which handles a different type of task
Example: Valve Game Engine
Example: Server Threads

RPC request

Time

RPC request

Process 1

Server

(a) RPC using single thread

RPC request

Server

Thread A (Process 1)

(b) RPC using one thread per server (on a uniprocessor)

Thread B (Process 1)

RPC request

Server

Remote Procedure Call (RPC)
There are different types of thread implementations

- user threads vs kernel threads
- each has different advantages / disadvantages
Types of Threads

- **User-level threads (ULT)**
  - all threads management is done by the application (thread library)
  - the kernel (OS) is not aware of the threads; all it sees is the process

- **Kernel-level threads (KLT)**
  - the OS is responsible for managing, scheduling, and executing kernel-level threads

- **Combined (kernel-supported user-level threads)**
  - user-level threads that map to kernel-level threads
  - one or more user-level threads can be assigned to a single kernel thread
User Level Thread (ULT)

- The kernel is not aware of the existence of threads
- All thread management is done by the application by using a thread library
- Thread switching does not require kernel mode privileges (no mode switch)
- Scheduling is application specific
Pros and Cons of ULT

Advantages
- Thread switching does not involve the kernel: no mode switching
- Scheduling can be application specific: choose the best algorithm
- ULTs can run on any OS, Only needs a thread library

Disadvantage
- Most system calls are blocking and the kernel blocks processes. So all threads within the process will be blocked
- The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors
Kernel-Level Threads (KLT)

- All thread management is done by kernel
- No thread library but an API to the kernel thread facility
- Kernel maintains context information for the process and the threads
- Switching between threads requires the kernel
- Scheduling on a thread basis
Pros and Cons of KLT

Advantages

- the kernel can simultaneously schedule many threads of the same process on many processors
- blocking is done on a thread level

Disadvantages

- thread switching within the same process involves the kernel.
- this results a mode switch, which causes a significant slow down
Combined ULT/KLT Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- May combine the best of both approaches