Threads 线程

线程是进程的一部分

Thread of execution is the **smallest sequence of programmed instructions** that can be managed independently by a scheduler andexecuted by the CPU independently of the parent process.

执行的线程是编程指令的最小串行，可以由调度进程独立管理，并由 CPU 独立于父进程执行。

A thread is a *lightweight process* that can be managed independently bya scheduler.

线程是一个*轻量级进程*，可以由调度进程独立管理。

Executes a series of instructions in order (only one thing happens at a time)

按顺序执行一系列指令（一次只发生一件事）

线程运行的概念和进程类似，一个CPU一个时间片内只能运行一个线程，但是实现多线程的原理是一个线程的程序不一定都要执行完才去执行下一个，CPU会划分时间片，加入线程A和B同时运行一段代码，A处理到断点1的时候停止但是此时仍未结束，此时CPU开始执行线程B也是运行到断点1，然后再返回线程A

When there are multiple threads running for a process, the process provides common memory.

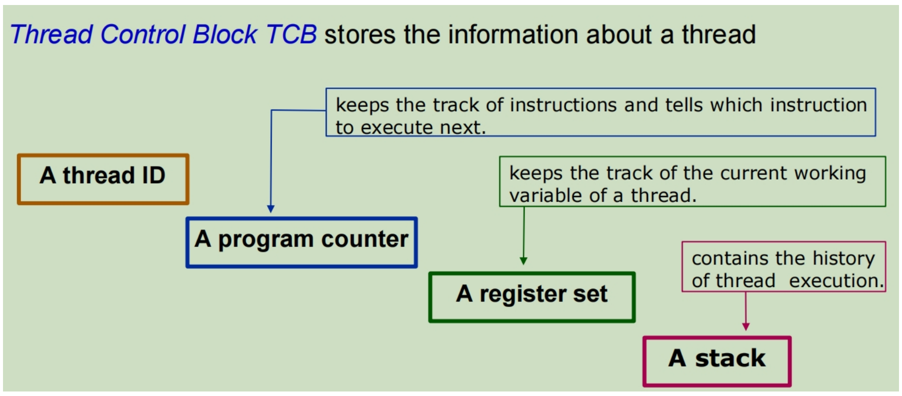
当一个进程有多个线程运行时，该进程将提供公共内存。

Multiple tasks with the application can be implemented by separate threads

应用进程的多个任务可以由单独的线程实现

Threads are scheduled on a processor, and each thread can execute a set of instructions independent of other processes and threads.

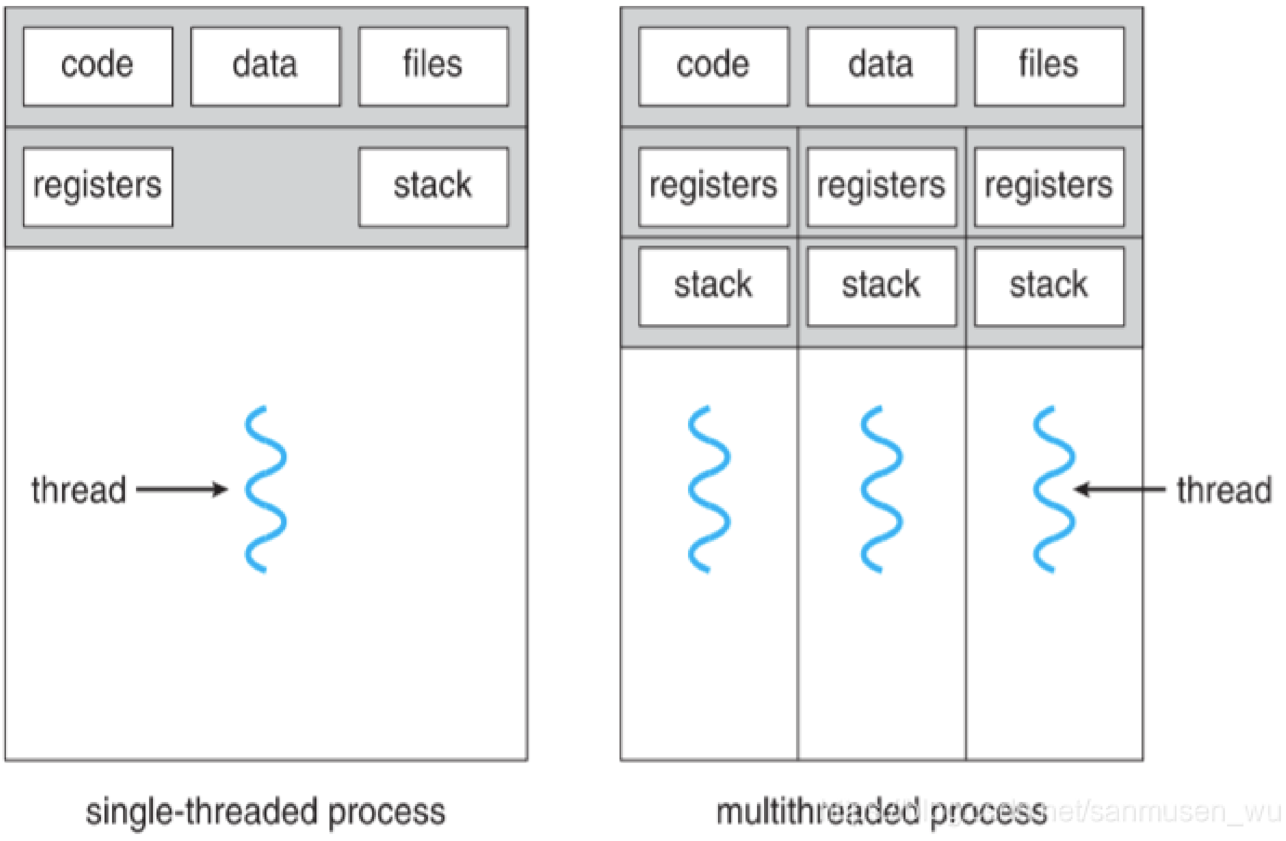
线程在处理器上调度，每个线程可以独立于其他进程和线程执行一组指令。


Thread Control Block TCB stores the information about a thread
keeps the track of instructions and tells which instruction to execute next.
A thread ID
keeps the track of the current working variable of a thread.
A program counter
contains the history of thread execution.
A register set
A stack

O.S view: A thread is the smallest unit of processing that can be performed in an OS.
Software developer view: Thread is a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems

线程是计算机操作系统执行的基本单元，也是CPU使用的基本单元。
线程也可以看成是简化的进程；
属于同一个进程的线程之间可以共享代码，数据，文件；
线程能够帮助程序“同时”执行多个任务

一个线程主要包含：
线程ID，程序指针，寄存器集，栈



Benefit of Threads 线程的好处

- Responsiveness
  - Allows a program continue running if part of it is blocked or its is
  - performing a lengthy operation
- Resource Sharing and Economy
  - Threads share an address space, files, code, and data
  - Avoid resource consumption
  - Perform much a faster context switch
- Utilization of multiprocessor MP Architectures
  - Place code, files and data in the main memory.

    - Distribute threads to each of CPUs, and
    - Let them execute in parallel
- Deadlock avoidance

反应

允许进程在部分进程被阻止或被阻止时继续运行

执行冗长的操作

资源共享与经济

线程共享地址空间、文档、代码和数据
避免资源消耗

执行更快的上下文切换

利用多处理器 MP 架构

将代码、文档和数据放在主存储器中。

将线程分发到每个 CPU，以及

让它们并行执行

避免死锁

因为如果当前运行的thread被锁给卡住了，那么处理器会直接继续运行接下来的线程，这样可能就会在后面的线程中释放掉锁住的资源

Example

User types text in the Word editor -> **threads**:

o **open a file** in Word editor

o **typing the text** (one thread),

o **the text is automatically formatting** (another thread**),**

o **the text is automatically specifying the spelling mistakes** (another thread)

o **the file is automatically saved to disk** (another thread).

Thread States 线程状态

Thread Control Block (TCB) contains:

Execution State: CPU registers, program counter, pointer to stack

Scheduling info: State, priority, CPU time

Various Pointers (for implementing scheduling queues)

螺纹控制块（TCB）包含：

执行状态：CPU 寄存器、进程计数器、指向堆栈的指针

计划信息：状态、优先级、CPU 时间

各种指针（用于实现调度队列）

状态切换:

未创建的线程处于undefined状态；

当线程就绪时，处于ready状态；

同一时刻只能有一个线程处于running状态

当线程被中断运行（suspend），处于suspended状态，随后可调用resume（）回归就绪状态

处于中断terminated状态的进程会被释放，随后处于销毁destroyed状态

Concurrency VS Parallelism 并发和并行

Concurrency: 单核系统并发处理任务，进程一个一个开始，中间互相穿插运行，直到结束完成。（一会吃饭一会看手机）

Parallelism: 多核系统并行处理任务能够同时处理多个任务（一边吃饭一边看手机）

Multithreading Models 多线程模型

1. **1. Kernel Mode** - executing code has **complete** and **unrestricted access** to the underlying hardware.

   内核模式 执行代码对底层硬件具有 **complete** 和 **unrestricted 访问权限**。

   ▪ It can execute any CPU instruction and reference any memory address.

   它可以执行任何 CPU 指令并引用任何内存地址。

   ▪ Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system.

   内核模式通常保留给操作系统的最低级别、最受信任的功能。

   ▪ Crashes in kernel mode are catastrophic; they will halt the entire PC.

   内核模式下的崩溃是灾难性的;他们将停止整个 PC。

由内核安排在CPU上执行的线程。
线程管理通过OS核控制。

2. **User Mode** - executing code has **no** ability to **directly access**hardware or reference memory.

   用户模式执行代码具有**否**直接访问硬件或引用内存的能力。

▪ Code running in user mode must delegate to system APIs (*Application Programming Interface)* to access hardware or memory.

在用户模式下运行的代码必须委托给系统 API（*应用进程编程接口*） 才能访问硬件或内存。

▪ Crashes in user mode are always recoverable.

用户模式下的崩溃始终是可恢复的。

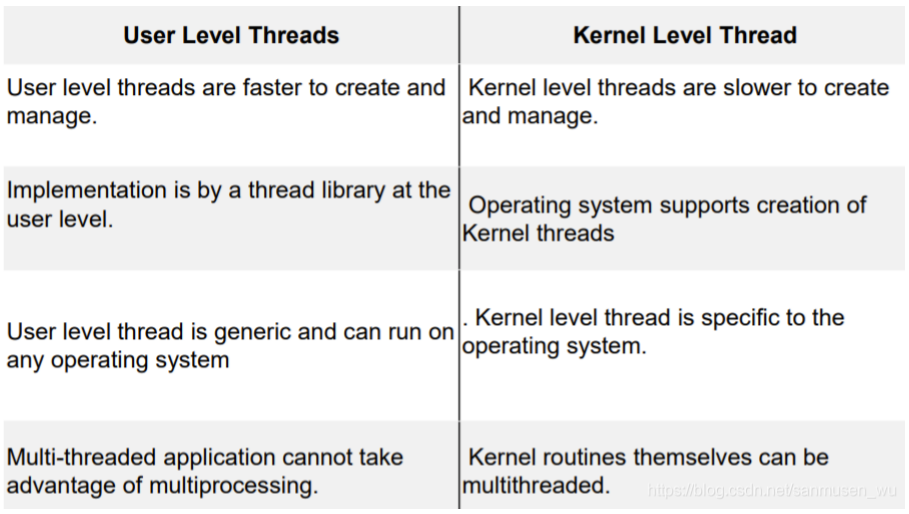
▪ Most of the code running on your computer will execute in user mode.

计算机上运行的大多数代码都将在用户模式下执行。

由用户实现的执行单元，这些线程受内核支持但是内核不管理这些线程。**用户线程比系统线程快**。
通过使用**线程库thread library**来调度这些线程。

The processor switches between the two modes depending on what type of code is running on the processor. *Applications* run in user mode, and *core operating system* components run in kernel mode.

处理器根据处理器上运行的代码类型在两种模式之间切换。***应用进程*在用户模式下运行，*核心操作系统*组件在内核模式下运行**。



User Level Threads	Kernel Level Thread
User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads
User level thread is generic and can run on any operating system	. Kernel level thread is specific to the operating system.
Multi-threaded application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

ULT的最后一条是因为multiprocessing是由processor控制的，但是ULT的级别不够控制System kernel因此也就没办法调节处理器，因此multi-threaded application无法从multiprocessing中获利

Three types

Many-to-One

多个用户级线程映射map到一个核线程。
因为只映射到一个核线程，所以进程只能同时执行一个用户线程，而且当这个用户线程被阻塞，整个进程被阻塞。
Used on systems that do not support kernel threads.
用于不支持内核线程的系统

One-to-one

一个用户级线程映射到一个系统线程。
每次创建一个用户线程就会创建一个系统线程，提供了并发的可能，且当一个线程发生阻塞，其他线程依旧可以运行。
大部分主流操作系统用的这个

Many-to-many

多个用户线程映射到多个系统线程

Allows the operating system to create **a sufficient number of kernel threads**

Number of kernel threads may be specific to an either a particular application or a particular machine.

The user can create any number of threads and corresponding kernel level threads can run in parallel on multiprocessor

允许操作系统创建足够数量的内核线程

内核线程数可能特定于特定应用进程或特定计算机。

用户可以创建任意数量的线程，相应的内核级线程可以在多处理器上并行运行

Thread Libraries 线程库

Threads can be created, used, and terminated via a set of functions that are part of a **Thread API** (a **thread library**).

线程可以通过一组函数创建、使用和终止，这些函数是 **Thread API**（**线程库**）的一部分。

1. 没有内核支持时，在用户空间中提供库（Library entirely in user space）。意味着库的所有代码和数据存放在用户空间，调用时不会使用系统调用。
2. 实现内核级别库（kernel-level library ），库和代码放在内核空间中，调用这些函数会使用系统调用。

三个主要的线程库：POSIX Pthreads，Win32，Java

Managing Threads 管理线程

There are 2 categories: **Explicit** and **Implicit threading**.

Explicit threading – the *programmer* creates and manages threads**.**

Implicit threading – the *compilers and run-time libraries* create and manage threads

显式线程 – 由程序员手动创建和管理线程。

隐式线程 – *编译器和运行时库* 创建和管理线程

Three Approaches for Designing Multithreaded Programs

Thread pool – create a number of threads at process startup and place them into a pool, where they sit and wait for work.

线程池 在进程启动时创建多个线程，并将它们放入池中，它们在那里等待工作。

OpenMP is a set of compiler directives available for C, C++, and Fortran programs that instruct the compiler to automatically generate parallel code where appropriate.

OpenMP 是一组可用于 C、C++ 和 Fortran 进程的编译器指令，用于指示编译器在适当的情况下自动生成并行代码。

Grand Central Dispatch (GCD) – is an extension to C and C++ available on *Apple's MacOS X* and *iOS* operating systems to support

parallelism.

Grand Central Dispatch （GCD） 是 C 和 C++ 的扩展，可在 *Apple 的 MacOS X* 和 *iOS* 操作系统上支持并行。

Thread Issues 线程问题

1. fork（）和exec（）调用
   当fork（）被调用，会产生俩，一个父级，一个子级；他们俩的代码段，数据和堆栈完全相同。
   当exec（）被调用，当前线程执行的程序会被替换成另一个程序，进程ID未变，以别的程序替代了该线程的代码段，数据和堆栈。
2. 信号处理Signal handling
   Unix/Linux系统相应某个条件或操作而生成的中断或时间，由signal handler处理，且每个信号只处理一次，信号大致有以下几种：
   异步信号asynchronous signal：发信号后不等
   同步信号synchronous signal：发送信号后死等

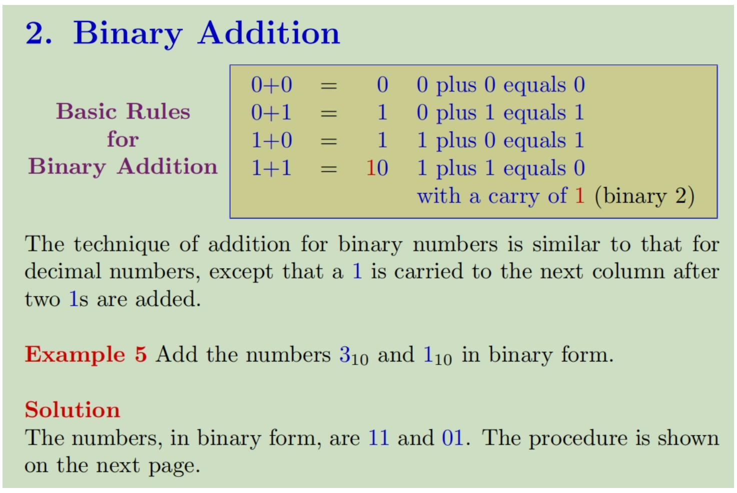
3. 线程关闭Thread cancellation

在线程完成之前关闭线程（如：加载页面时点击取消）

能通过以下方式关闭：

- Asynchronous cancellation 异步取消：立即终止。
- Deferred cancellation 延期取消：允许目标线程检查然后自己终止。

Binary Calculation



2. Binary Addition

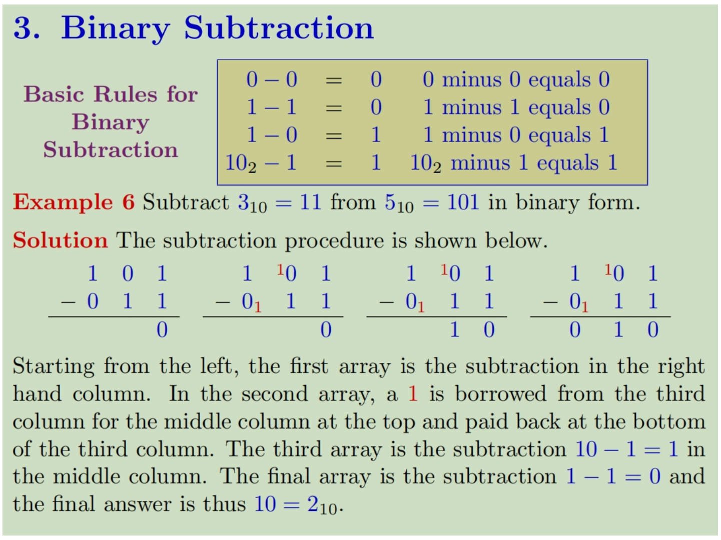
Basic Rules for Binary Addition

0+0	=	0	0 plus 0 equals 0
0+1	=	1	0 plus 1 equals 1
1+0	=	1	1 plus 0 equals 1
1+1	=	10	1 plus 1 equals 0 with a carry of 1 (binary 2)

The technique of addition for binary numbers is similar to that for decimal numbers, except that a 1 is carried to the next column after two 1s are added.

Example 5 Add the numbers 3_{10} and 1_{10} in binary form.

Solution
The numbers, in binary form, are 11 and 01. The procedure is shown on the next page.



3. Binary Subtraction

Basic Rules for Binary Subtraction

$0 - 0$	=	0	0 minus 0 equals 0
$1 - 1$	=	0	1 minus 1 equals 0
$1 - 0$	=	1	1 minus 0 equals 1
$10_2 - 1$	=	1	10_2 minus 1 equals 1

Example 6 Subtract $3_{10} = 11$ from $5_{10} = 101$ in binary form.

Solution The subtraction procedure is shown below.

$$\begin{array}{rccc} & 1 & 0 & 1 \\ - & 0 & 1 & 1 \\ \hline & & & 0 \end{array} \qquad \begin{array}{rccc} & 1 & {}^{1}0 & 1 \\ - & 0_1 & 1 & 1 \\ \hline & & & 0 \end{array} \qquad \begin{array}{rccc} & 1 & {}^{1}0 & 1 \\ - & 0_1 & 1 & 1 \\ \hline & & 1 & 0 \end{array} \qquad \begin{array}{rccc} & 1 & {}^{1}0 & 1 \\ - & 0_1 & 1 & 1 \\ \hline & 0 & 1 & 0 \end{array}$$

Starting from the left, the first array is the subtraction in the right hand column. In the second array, a 1 is borrowed from the third column for the middle column at the top and paid back at the bottom of the third column. The third array is the subtraction $10 - 1 = 1$ in the middle column. The final array is the subtraction $1 - 1 = 0$ and the final answer is thus $10 = 2_{10}$.

1. Binary Numbers (Introduction)

In **Binary Numbers 1**, binary numbers were introduced, as well as the techniques of their addition and subtraction. This package covers the methods of multiplication and division but to begin, here is a reminder of the rules of binary addition and subtraction.

Basic Rules for Binary Addition

0+0	=	0	0 plus 0 equals 0
0+1	=	1	0 plus 1 equals 1
1+0	=	1	1 plus 0 equals 1
1+1	=	10	1 plus 1 equals 0
			with a carry of 1 (binary 2)

$0-0$	=	0	0 minus 0 equals 0
$1-1$	=	0	1 minus 1 equals 0
$1-0$	=	1	1 minus 0 equals 1
10_2-1	=	1	10_2 minus 1 equals 1

Basic Rules for Binary Subtraction

2. Binary Multiplication

Table of Basic Rules for Binary Multiplication

0×0	=	0
0×1	=	0
1×0	=	0
1×1	=	1

The multiplication process for binary numbers is similar to that for decimal numbers. Partial products are formed, with each product shifted one place to the left. This is illustrated below.

Example 7 Multiply $7_{10} = 111$ and $5_{10} = 101$ in binary form.

Solution

$$\begin{array}{cccccc} & & & 1 & 1 & 1\\ & & \times & 1 & 0 & 1\\ \hline & & & 1 & 1 & 1\\ & & 0 & 0 & 0 & 0\\ + & 1 & 1 & 1 & 0 & 0\\ \hline 1 & 0 & 0 & 0 & 1 & 1\end{array}$$

The third row is the multiplication of 111 by 1. In the fourth row, the 0 is the shift left before 111 is multiplied by 0. In the fifth row, the 00 is the shift left before 111 is multiplied by 1. The final row is the binary sum of the preceding three rows.

3. Binary Division

Binary division follows a similar process to that of decimal division.

Example 8 Divide **(a)** 15_{10} by 5_{10} in binary form, and **(b)** 15_{10} by 6_{10} in binary form.

Solution In binary form $15_{10} = 1111$, $5_{10} = 101$ and $6_{10} = 110$. The process for each of these is shown below.

(a)
$$\begin{array}{rcccc} & & & 1 & 1\\ \cline{2-5} 1\;0\;1\,\big) & 1 & 1 & 1 & 1\\ & 1 & 0 & 1 & \\ \cline{2-4} & & 1 & 0 & 1\\ & & 1 & 0 & 1\\ \cline{3-5} & & 0 & 0 & 0\end{array}$$

(b)
$$\begin{array}{rccccc} & & & 1 & 0 & .1\\ \cline{2-6} 1\;1\;0\,\big) & 1 & 1 & 1 & 1 & .0\\ & 1 & 1 & 0 & & \\ \cline{2-4} & & & 1 & 1 & 0\\ & & & 1 & 1 & 0\\ \cline{4-6} & & & 0 & 0 & 0\end{array}$$

In decimal form, $15_{10} \div 5_{10} = 3_{10}$, and 3_{10} is 11 in binary, which is the answer in the left hand array.

In decimal form, $15_{10} \div 6_{10} = 2.5_{10}$, and 2.5_{10} is 10.1 in binary, which is the answer in the right hand array.