多线程

现代计算机架构

- 冯·诺伊曼结构（是图灵机的一种等效描述）
  - 存储单元
  - 控制单元
  - 算术逻辑单元
  - 输入设备

    - 输出设备
- 哈佛结构
    - 数据存储单元
    - 指令存储单元
    - 控制单元
    - 算术逻辑单元
    - 输入设备
    - 输出设备

处理器

现代 CPU 可以包含算术逻辑单元（ALU）、控制单元（CU）、寄存器堆、内存管理单元（MMU），高速缓存（cache）等等。

寄存器（register）是 CPU 用来暂存指令、数据和地址的存储器。其特点是读写速度非常快。

处理器一般含有"程序计数器（PC）"，用于储存指令的存储地址，每次执行时，处理器会从按照程序计数器中的地址从指令存储器中取出指令来执行。

进程

进程的概念

对于单核CPU，理论上同时只能运行一个程序。但是，在我们用户看来，仿佛同时有多个程序在运行。实际上操作系统不断地调度这些程序，将一个个程序装入CPU，又将它们从CPU中移出，在宏观上就像很多个程序同时运行一样。

在多程序并发运行时，不同的程序之间将被操作系统隔离开。它们的程序内容互相独立，各程序间也"看不到"对方。正在运行这样一个被隔离的单独的程序就是一个"进程"。但是仍然可以通过进程间通信的一些方式达到多个进程共享资源的目的。

虚拟内存

每个进程在运行时都被分配了一块虚拟的内存，让每个进程看起来都是自己独占了整个主存。每个进程看到的内存都相同，这个内存空间就叫做"虚拟地址空间"。下面是 Linux 系统的虚拟地址空间：

进程的三个状态

进程一般有三个状态：

- 运行态
- 就绪态
- 阻塞态

如果一个进程正在 CPU 上运行，那么这个程序就处在运行态；如果一个程序由于自身的原因，例如等待输入等，而不能向下执行，它就出于阻塞态；如果由于有其他进程抢占了 CPU 等一些原因导致进程无法继续运行而处在等待运行的状态，则这时进程出于就绪态。

不过以上只是理论模型，实际上的操作系统实现可能更加复杂。

线程

线程的概念

对于我们之前所讲的进程来说，一个进程时从头到尾依次运行的，这种情况我们称之为单线程执行。但是对于我们要并行地做多件事情的时候，单线程执行并不能满足我们的要求。这时候，与之前所说的在CPU 上同时运行多个进程一样，我们就需要在一个进程内同时运行多个"线程"。与多个进程不同，同一个进程之间的不同线程是共享这个进程的资源的，例如共享同一个虚拟地址空间，等等。

可以看到，线程比进程更加轻量级，因此线程之间的切换所需的事件更短。

Windows中，允许多进程并发执行，且每个进程中允许有多个进程；Linux中通过进程组的方式，让一组相关的进程共享资源空间，起到类似线程的作用。

线程的资源，一般来说分为共享资源和独享资源。

共享资源包括进程方面的大多数信息，如进程标识符、优先级、代码段、数据段等等。

独享资源为线程独有，例如线程运行的栈资源。栈用于保存程序运行的上下文信息，需要由每个线程独享一份。

C# 中的线程库

创建一个线程

C# 创建多线程的方式是使用 `Thread` 类，`Thread` 类位于 `System.Threading` 命名空间内，构造方法可以接收一个返回值为 `void`，无参数或只有一个 `object?` 作为参数的方法（或委托）作为参数，然后用 `Start` 开始线程的执行，并使用 `Join` 等待线程完成：

```
using System.Threading;
var thr = new Thread
(
    n =>
    {
        for (int i = 0; i < 10000; i++)
        {
            Console.WriteLine($"In child thread: {n}");
        }
    }
    Thread.Sleep(1000); //静态方法，暂停当前线程1000ms。注：Thread.Sleep(0)会导致线程立即放弃当前的时间片，自动将CPU移交给其他线程。
    Thread.Yield(); //与Sleep(0)类似，但是它只会把执行交给同一处理器上的其他线程。即Yield是主动让出CPU，如果有其他就绪线程就执行其他线程，如果没有则继续当前线程
);
thr.Start(999);
for (int i = 0; i < 10000; i++)
{
    Console.WriteLine("In main thread: 888");
}
thr.Join();
thr.Join(1000); //设置超时，参数用毫秒或TimeSpan。返回bool类型：等待的线程结束了，就返回true，超时了返回false。
```

前台线程与后台线程

C# 的线程模型中，线程分为前台线程和后台线程。区别是，前台线程在整个程序退出时会阻塞程序的退出，当前台线程退出时，整个程序才会退出；而后台线程不同，程序退出时，后台线程自动终止。默认情况下，线程是前台线程。前台或后台进程由 `Thread` 对象的 `IsBackGround` 属性来指定：

```
var thr = new Thread(() => { Console.WriteLine("thread"); });
thr.IsBackground = true;    // 指定为后台线程
thr.Start();
```

或者简写为：

```
new Thread
(
    () =>
    {
        Console.WriteLine("threads");
    }
)
{ IsBackground = true }.Start();
```

注意上述花括号是 C# 的一种语法，可以在 `new` 的同时设置该对象的属性。注意该种语法下对象属性的设置时间在构造方法执行之后。

此外，`Thread` 类还有 `IsAlive` 属性用来标志该线程是否正在运行，`IsThreadPool` 属性用来标识该线程是否是托管线程池中的线程（线程池将在 稍后讲述）。

此外，可以调用 `Thread` 类的静态属性 `CurrentThread` 来得到当前线程的句柄。

线程池

创建线程和销毁线程带来的开销其实是不算小的，一个想法就是预先开辟多个线程，然后等到需要开辟线程的时候直接使用已经创建好的线程以减小开销。

当向线程池申请线程执行时，如果线程池中的线程已经用完了，那么线程池会停顿一小段时间，观察是否有执行完毕的线程，如果有则使用该线程，如果没有则将线程池扩容。

线程池的相关操作都在 `ThreadPool` 静态类中（静态类意味着不能实例化，其内都是静态的字段、属性、方法）。

在对线程池进行操作之前，我们可以设置线程池线程的最大线程数和最小线程数。线程池扩容不会超过设置的最大线程数。

我们可以通过 `SetMaxThreads`、`SetMinThreads` 静态方法来设置，通过 `GetMaxThreads` 和 `GetMinThreads` 静态方法来获取。这些方法都有两个参数，我们现在暂时只用关心第一个参数，设置的时候第二个参数使用原来的值即可。

使用 `QueueUserWorkItem` 静态方法方法可以向线程池申请一个线程。该方法接收一个 `WaitCallBack` 对象作为参数。我们可以将要执行的方法作为 `WaitCallBack` 的构造方法的参数，然后将新创建的对象作为参数调用 `QueueUserWorkItem`。

但是一般情况下我们可以使用封装好的 `Task`、`Timer` 等间接使用线程池，而不是直接使用，因此在此处不举例子。这些将在后文进行介绍。

进程/线程间通信

进程间通信的难题

进程间通信的难题有三：

- 进程间互相传递信息
  - linux
    - 套接字（即 Socket，利用网络）
    - 信号（例如 Ctrl + C）
    - 管道
    - 消息队列
    - System V 信号量
    - 共享内存（`mmap`）
  - Windows
    - 套接字
    - 管道
    - 共享内存（`GetProcAddress`）
    - 邮件槽
- 资源竞争（互斥问题）
- 执行顺序的正确性

后两个问题和解决方法大多同样适用于线程通信，因此我们在介绍进程通信问题时，也可以将很多方法适用于线程通信，处理多线程问题。

互斥

互斥问题

多进程实际上是轮流使用CPU的。每个进程会有自己的时间片，当一个进程时间片消耗完时，会引发时钟中断，从而切换到另一个进程运行。中断的时机顶层程序的编写者是难以控制的。如果在多进程中去访问共享资源，将会引起互斥问题。

例如我们想要在两个进程中计算从1+2+3（只是个例子而已）

```
// Global variable
a=1;
// Process 1
a+=2;
// Process 2
a+=3;
```

假设Process1和Process2同时启动，运行在单核CPU上，结果是?

我们把访问共享数据的这部分程序片段称作“临界区”。如果我们能设计一个办法，使得两个进程不能同时进入临界区，那么我们就可以避免资源竞争，即，我们需要的是“互斥”。一般，一个好的办法设计需要满足下列条件：

- 任何两个进程不能同时处于临界区

- 对 CPU 的速度和数量没有要求
- 临界区外运行的进程不能阻塞到其他进程
- 不能让一个进程无限期等待进入临界区

原子操作

所谓原子操作，就是不可再分的操作，原子操作不会被打断，因此也就不会出现互斥问题。

C# 提供了原子操作库，位于 `System.Threading` 命名空间的 `Interlocked` 类中，其提供一下几种操作（作为静态方法）：

- `Add(ref a, b)`：`a += b`
- `And(ref a, b)`：`a &= b`
- `CompareExchange(ref a, b, c)`：`a = a == c ? b : a`
- `Decrement(ref a)`：`--a`
- `Exchange(ref a, b)`：`a = b`
- `Increment(ref a)`：`++a`
- `Or(ref a, b)`：`a |= b`
- `Read(ref a)`：`a`

基于忙等待的互斥

锁变量

这里有一种非常容易想到的防止互斥的做法，即使用锁变量：

```
int lock;

while (lock == 1);
lock = 1;
// 临界区
lock = 0;
// 非临界区
```

回到那个1+2+3的例子中，

```
int a = 1;
int lock = 0;

// Process 1
while (lock == 1);
lock = 1;
a+=2;
lock = 0;

// Process 2
while (lock == 1);
lock = 1;
a+=3;
lock = 0;
```

这样的写法，能够保证计算结果正确吗？

如果有一个进程，由于检测到 `lock` 为零而从 `while` 循环中跳出，但是还没来得及把 `lock` 设置为 `0`，这时候发生了进程调度，另一个进程也判断 `lock` 为零，也跳出了 `while` 循环，这样这两个进程便都进入了临界区，因此算法是错误的。

此外，这个算法还存在着忙等待的问题。

忙等待：

- 上面的 `while` 循环一直在对 `lock` 进行判断，一直有占用 CPU 的需求。这种现象我们称为忙等待。此外，忙等待还可能存在其他的一些问题。
- 我们在日常的编程当中，要尽量避免忙等待。只有在能确认忙等待的时间非常短的情况下，才可勉强使用忙等待。
- 用于忙等待的锁，称为自旋锁。

严格轮换法

下面是一种自旋锁的方法：

```
int turn = 0;

void Process1()
{
        while (turn == 1);
        // 临界区
        turn = 1;
        // 非临界区
        while (turn == 1);
        // 临界区
        turn = 1;
        // 非临界区
        ...
}

void Process2()
{
        while (turn == 0);
        // 临界区
        turn = 0;
        // 非临界区
        while (turn == 0);
        // 临界区
        turn = 0;
        // 非临界区
        ...
}
```

这段代码确实能实现互斥。但是一方面，它可能违反条件 `3`，即一个进程可能被另一个进程的临界区外的代码阻塞；另一方面，它只允许最多两个进程。

此外还有 TSL 指令、XCHG 指令、Peterson 算法等用来实现这一功能。但是都存在忙等待的问题。

信号量

针对上述问题，Dijkstra 提出了“信号量（Semaphore）”的概念。无需使用忙等待就能实现进程间互斥， 同时还能控制允许进入临界区的进程数。

一个信号量包含一个整数变量用来计数，以及两个原子操作：

- P 操作：让计数变量的值减一，若结果小于 0，则进入休眠状态
- V 操作：让计数变量的值加一，若结果不大于 0，则唤醒一个休眠的进程

```
int sem = n;//初始化

P(sem);
// 临界区
V(sem);
```

结合1+2+3的例子，我们来理解一下信号量的作用

```
int a = 1;
int sem = 1;
// 最多允许一个进程进入临界区

// Process 1
P(sem);
a+= 2;
V(sem);
// Process 2
P(sem);
a += 3;
V(sem);
```

计数变量<0时，其绝对值代表线程阻塞的数目；计数变量>0时，其绝对值代表能允许继续进入临界区的线程数目。

C# 中使用 `System.Threading` 命名空间的 `Semaphore` 类和 `SemaphoreSlim` 类来支持信号量的操作。`SemaphoreSlim` 是 `Semaphore` 的一个轻量替代。

> `Semaphore`用于控制对资源池的访问,可以跨进程
>
> `SemaphoreSlim` 在等待时间预计很短的情况下，用于在单个进程内的等待，效率更高

`SemaphoreSlim` 的最常用的构造方法接收两个参数，第一个是计数变量的初始值，第二个是计数变量的最大值，例如：

```
var sem = new SemaphoreSlim(0, 5);
```

创建了一个计数变量最大值为 5 的信号量，计数变量初始值为 0。

`sem` 的 `WaitOne` 方法实现 P 操作，而 `Release` 方法实现 V 操作。

互斥量

注意我们以上两个例子，对临界区的互斥控制有一个特殊性：信号量的初始值是 1，且 P、V 操作在一个线程中成对出现，先有 P，后有 V，两个操作之间是互斥的临界区。这意味着同时只能有一个进程进入两个 P、V 操作之间。因此我们对这个特殊的情况进行单独处理，对信号量进行简化，得到“互斥量（mutex）”。

一个互斥量包含两个操作：加锁（lock，对应于 P 操作）和解锁（unlock，对应于 V 操作）。

使用互斥量需要注意的点是，必须在同一个进程内为其加锁与解锁，且解锁在加锁之后。

C# 提供了 `System.Threading.Mutex` 类支持互斥量，但是更常用的是 `System.Threading.Monitor` 类来实现的互斥量。而这个类不经常显式使用，而是使用 `lock` 关键字隐式使用。使用时，我们先要创建一个对象作为锁：

```
object lockObject = new object();
```

然后我们便可以使用锁了：

```
lock (lockObject)
{
    /*Some code*/
}
```

当程序进入 `lock` 语句块时，便进行加锁，然后执行语句块中的代码；当程序跳出 `lock` 语句块后，便进行解锁。

等价于

```
object lockObject = new object();
bool acquiredLock = false;

try
{
    Monitor.Enter(lockObject, ref acquiredLock);

    // Code that accesses resources that are protected by the lock.

}
finally
{
    if (acquiredLock)
    {
        Monitor.Exit(lockObject);
    }
}
```

我们在编程时，应尽可能缩短持有锁的时间，以减少锁争用。

条件变量

有锁线程的三种状态：持有锁的线程，就绪队列，等待队列

运行优先顺序为：

1.先运行 持有锁的线程 。

2.然后是 就绪队列 中排在最前的线程。

3.只有 等待队列 的线程进入 就绪队列 才有排队资格。

C# 可以用 `Monitor` 类实现条件变量。

在 `lockObject` 加锁的情况下，调用 `Monitor.Wait(lockObject)`，将 持有锁的线程 的本线程状态改为 等待队列，就绪队列 中排在最前的线程获得锁

详见Monitor.Wait 方法 (System.Threading) | Microsoft Learn

`Monitor.Pulse(lockObject)` 将 等待队列 中的一个线程移到 就绪队列

`Monitor.PulseAll(lockObject)` 将排在 等待队列 最前的线程移到 就绪队列

读写锁

在编程当中我们经常回遇到这样的问题：一个资源，我们可以需要对它进行读写。读操作是不对资源进行更改的，而写操作对资源进行更改。这对我们提出了要求：可以多个进程同时读，但是读的过程中不允许写；只能有一个进程对它写入，写的过程不允许任何其他进程同时写或读。

C# 为我们提供了 `System.Threading.ReaderWriterLock` 和 `System.Threading.ReaderWriterLockSlim` 实现读写锁。后者性能更高。

以下代码创建一个读写锁：

```
var rwLock = new ReaderWriterLockSlim();
```

使用 `EnterReadLock` 和 `ExitReadLock` 可以锁定和退出读者锁；使用 `EnterWriteLock` 和 `ExitWriteLock` 可以锁定和退出写者锁。

由于加锁之后就一定要解锁。为了防止由于异常的抛出而导致没有解锁，因此读写锁一般采用 `try-finally` 语句：

```
rwLock.EnterWriteLock();
try
{
    // 写者代码
}
finally
{
    rwLock.ExitWriteLock();
}

rwLock.EnterReadLock();
try
{
    // 读者代码
}
```

```
finally
{
    rwLock.ExitReadLock();
}
```

死锁

在一组进程中，每一个进程都占用着若干资源，但又在同时等待被其他进程占用的其他资源，从而造成所有进程都进行不下去的现象，称为死锁。

```
semaphore sem1 = 1;
semaphore sem2 = 1;
void Process1(void)
{
    P(sem1);
    // 使用资源 1
    P(sem2);
    // 使用资源 1、2
    V(sem2);
    V(sem1);
}
void Process2(void)
{
    P(sem2);
    // 使用资源 2
    P(sem1);
    // 使用资源 1、2
    V(sem1);
    V(sem2);
}
```

那么这就可能造成死锁。因为如果第一个进程持有资源 1，而第二个进程持有资源 2，这时它们都不放弃持有的资源而想要获取对方的资源，于是造成死锁。

死锁预防

- 避免死锁的最好方法就是不用锁：
  - 原子操作
- 不允许多层锁
- 给资源进行编号，人为规定锁的优先级
  - 多数锁为类的private字段

异步

同步操作会在返回调用者前完成所有工作，而异步操作会在返回调用者后去完成它的大部分工作。

启用异步方法时，异步方法会与调用者并行执行。一般来说，异步方法会很快返回到调用者手中。

C# 支持异步编程，这是基于 Task 实现的。

Task

创建一个 Task

`System.Threading.Tasks` 命名空间内含有 `Task` 类，可以用于创建各种任务，底层是使用线程池。

```
Task.Run(Method1);   // 无返回值的任务
Task<TResult>.Run(Method2); // 有返回值的任务
```

可以使用`Task.Run`方法创建一个`Task`，该方法会启动一个`Task`，并返回一个`Task`引用对象。

```
var t = Task.Run(() =>
                    {
                        Thread.Sleep(1000);
                        return 1;
                    });
```

`Run` 方法返回一个 `Task` 对象引用。

还可以这样创建：

```
var t = new Task(Method1);
t.Start();
```

可以调用 `Wait` 方法等待任务执行完毕。

对于有返回值的任务，可以用 `Result` 属性获取返回值。若任务尚未完成，会等待其完成再返回。

```
var t = Task.Run(() =>{
                        Thread.Sleep(1000);
                        return 1;
                    });
Console.WriteLine("Start!");
Console.WriteLine(t.Result);
```

Task 的生命周期

`Task` 具有 `Status` 属性，用于获取它的生命周期的状态（均为 `TaskStatus` 枚举的成员，例如 `TaskStatus.Created`）：

- `Created`：已经被创建但尚未开始
- `Running`：正在运行
- `RanToCompletion`：已经执行完毕
- ……

此外还有其他状态，再次不做过多展开。

`await` 与 `async`

`async` 创建一个异步方法，一个异步方法的返回值必须是 `void`、`Task`、`Task<T>`、`IAsyncEnumerable<T>`、`IAsyncEnumerator<T>`，或者是一个与 `Task` 相似的对象类型。记住前三个即可。

```
async void TestAsync();
```

`await` 顾名思义就是等待一个异步方法或一个任务的完成。例如下面的：

```
await Task.Delay(1000);
```

上面的代码是等待 1000 毫秒。

```
static async Task<int> TestAsync()
{
    Task<int> t = new Task<int>(() => 5);
    t.Start();
    return await t;
}
```

上面的代码执行到 `await` 时异步方法先返回任务 `t`，且该任务最终返回 5。

需要注意的是，`await` 只能出现在异步方法中；`async` 方法内也通常有 `await` 关键字，否则它和同步方法没有任何区别。而且执行到 `await` 时，异步方法会先行返回所要最终等待的任务，下面的代码均会在任务中执行。

实际上，`await` 就是 `Task` 的语法糖。运行下面的代码：

```
static async Task<int> TestAsync()
{
    Task<int> t = new Task<int>(() => 0);
    t.Start();
    Console.WriteLine("Before await.");
    var x = await t;
    await Task.Delay(1000);
    Console.WriteLine("After await.");
    return x;
}

static void Main(string[] args)
{
    var t = TestAsync();
    Console.WriteLine("Main");
    Console.WriteLine(t.Result);
}
```

得到输出：

```
Before await.
Main
After await.
0
```

异步 lambda 表达式

lambda 表达式也可声明为异步的，方法是在 lambda 表达式前加上 “async” 关键字：

```
async () =>
{
    await Task.Delay(1000);
}
```

其他

限于课时和篇幅的限制，很多有趣的内容我们没有涉及，例如：

- 时间片与进程优先级；用户级线程与内核级线程；银行家算法
- Windows 与 Linux 的进程与线程模型、Linux 守护进程、Linux 信号编程
- 回调函数的概念与 C# 定时器 `System.Threading.Timer` 与 `System.Timers.Timer`

略去上述内容不会对我们后续的学习产生太大影响，感兴趣的同学可以阅读《Computer Systems: A Programmer's Perspective》、《Modern Operating Systems》等。

作业

- 根据Program.cs中ILongProgressByTime的要求修改LongProgressByTime类中的代码
- 作业地址https://github.com/shangfengh/EESAST-hw2023-MultiThreading

提交方式

GitHub 提交

- fork 仓库：shangfengh/EESAST-hw2023-MultiThreading(github.com)到个人仓库，按要求修改好后，从个人仓库提pr到原本的仓库，pr信息填写为：`CSharp_姓名_班级` （如：`CSharp_大佬_无29`）。

截止日期

由yxgg决定

参考文献

1. https://en.wikipedia.org/wiki/Central_processing_unit，2021 年 7 月 14 日
2. Computer Organization Design, David A. Patterson, John L. Hennessy, Fifth Edition
3. https://zh.wikipedia.org/zh-cn/%E5%AF%84%E5%AD%98%E5%99%A8
4. 《深入理解计算机系统》Randal E. Bryant、David R. O'Hallaron 著作，龚奕利、贺莲 译，机械工业出版社，2021 年 3 月第一版

5. 《现代操作系统》Andrew S. Tanenbaum、Herbert Bos 著，陈向群、马洪兵 译，机械工业出版社，2019 年 7 月第一版
6. https://docs.eesast.com/docs/tools/os，2021 年 7 月 14 日
7. https://leetcode-cn.com/problemset/all/，2021 年 7 月 15 日
8. https://docs.eesast.com/docs/languages/CSharp_2_multithread，2021 年 7 月 15 日

##