多线程与异步

无03 唐昌礼

基础知识

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

控制单元和算术逻辑单元组成了中央处理单元，简称CPU（Central Processing Unit）。

CPU

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。

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。

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。

操作系统简介

操作系统是计算机系统中一个功能强大的软件。它能够有效、合理地管理和分配计算机的软硬件资源，使它们正常运转，使整个计算机系统能够高效运行。

在设计一个操作系统时，需要硬件提供多方面的支持，包括

- 特权指令
- 系统调用
- 内存保护
- 中断机制
- 输入输出

特权指令

不是所有指令都允许用户调用，否则计算机系统将非常不安全。部分指令只允许操作系统来调用。那么在硬件上就必须提供支持。一般来说，是通过控制处理器状态实现的。

系统调用

系统调用，就是请求操作系统为其调用某种功能。例如要从用户态陷入内核。

内存保护

对于每一个程序，为其设定程序的活动范围，程序只能访问这个范围的代码或数据。

中断机制

当中断发生时，CPU将暂停当前的程序，转而执行中断处理程序。

进程 Process

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，仿佛同时有很多个程序都在运行。实际上程序的并发执行。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。

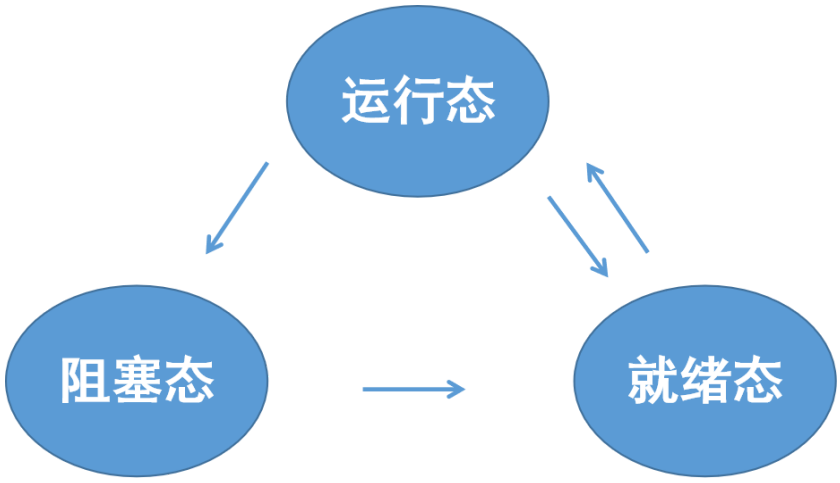
一个程序往往包括：代码段、数据段、堆、栈等。在多程序并发运行时，不同的程序之间将被操作系统隔离开。它们的程序内容互相独立，各程序间也“看不到”对方。正在运行这样一个被隔离的单独的程序就是一个”进程“。我们可以看到，进程之间是相互独立的，互不干扰。

进程的状态

进程有三种状态：运行态、就绪态、阻塞态。

- 运行态：进程正在CPU上运行。
- 就绪态：其他进程正在执行，当前进程只能等待被调度到CPU上执行。
- 阻塞态：进程被阻塞，例如等待输入。

进程间状态的切换关系如下



虚拟内存

虚拟内存也是一个重要工作，需要 CPU 内的内存管理单元（MMU）与操作系统的紧密协作。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：

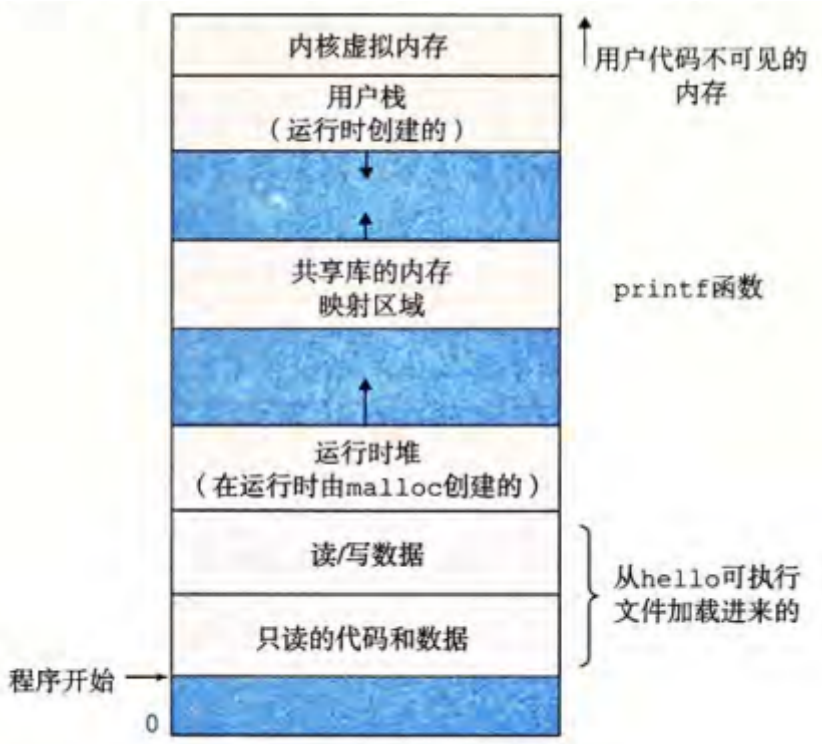


图 1-13　进程的虚拟地址空间

线程 Thread

为什么要引入线程

在编程的实际应用中，我们不仅希望程序能并发执行，同时也希望并发执行的程序能共享地址空间，这将更方便程序员编程。进程之间是相互独立的，在一个进程中分配的空间，在另一个进程中是不能直接访问的，多进程的编程因此就非常困难。所以，人们又提出了更小的能独立运行的程序结构——线程。

线程比进程更加轻量，一般是指进程中的一条执行流程。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。

线程的资源结构

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（因为这些线程共属一个进程，这部分信息可以共享），如进程标识符、优先级、代码段、数据段等等。

独享资源为进程独有，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# 线程

线程使用方法

使用C#开发可以很方便的使用多线程的编程。

线程类`Thread`位于`System.Threading`命名空间中，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。

`Thread`的常用方法如下

```
Thread t = new Thread(Func); //创建一个线程
t.Start();  //启动一个线程
t.Join(); //t线程join主线程，即：主线程等t线程结束后，主线程才运行。无返回类型。
t.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。
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
//当Sleep或Join的时候，线程处于阻塞状态。
```

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();
```

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

创建线程与销毁线程的开销都不算小，如果能提前开辟好多个线程，等到需要使用新线程时，就直接使用已经创建好的线程，这就能减少开销。线程池就是通过预先创建一系列可循环使用的线程来减少开销的。

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`System.Threading.ThreadPool`静态类中。

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。

线程池中的线程都是后台线程。

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。本节课最后会详细介绍 `Task`。

进程间通信（IPC）

进程间通信讨论三方面的问题：

- 进程间如何传递信息
- 多进程访问共享资源时，该如何保证有序运行（进程间互斥问题）
- 如何控制进程间的运行次序（进程间同步问题）

虽然这里讨论的是进程问题，但实际上下面讲的基本都是线程。

进程间通信方式

低级通信

低级通信，指的就是进程间只需传递很少量的信息，比如一个字节或者一个整数等。例如信号量（后面讲）。

高级通信

进程间需要传输大量信息。主要实现方式为

- 共享内存
- 建立通信链路
- 管道

互斥

互斥问题的原因

现代操作系统普遍采用多道程序技术，同时有多个进程在内存中运行。不过并发运行只是宏观上的，微观上再任意时刻，一个CPU上值运行着一个进程。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
int temp1 = a;
temp1 += 2;
a = temp1;

// Process 2
int temp2 = a;
temp2 += 3;
a = temp2;
```

假设`Process1`和`Process2`同时启动，运行在单核CPU上，最后`a`的结果会是什么?

临界区

上面的代码中，`Process1`和`Process2`对共享资源`a`进行了访问，`a`就是临界区内的资源。所谓临界区，实际上就是多进程（或线程）要访问共享资源的那段程序。对临界区的访问需要严格控制“互斥”，否则程序往往不能满足设计的要求（如上面的1+2+3）。

人们提出了实现互斥访问的4个条件：

- 任何两个进程不能同时进入临界区
- 不能事先假定CPU的数量和速度
- 临界区外的进程不能妨碍其他进程进入临界区
- 任何一个进程不能无限期等待进入临界区

基于忙等待的互斥

锁标志

很容易想到，我们可以使用一个锁标志，来实现进程间互斥

```
int lock;

//...

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

这样回到那个1+2+3的例子中，

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

// Process 1
while (lock == 1);
lock = 1;
int temp1 = a;
temp1 += 2;
a = temp1;
lock = 0;
```

```
11
12  // Process 2
13  while (lock == 1);
14  lock = 1;
15  int temp2 = a;
16  temp2 += 3;
17  a = temp2;
18  lock = 0;
```

这样的写法，能够保证计算结果正确吗?

算法问题:

- 假如进程1运行到第6行时，时间片用完了，换到进程2执行。此时lock的值仍为0，进程1、2同时进入临界区，不满足互斥条件。
- 忙等待：一直在进行`while`判断，占着CPU，但没有进行任何其他有意义的事，只是在死循环，这种现象我们称为忙等待。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。

强制轮流法

```
1   int turn = 0;
2
3   void Process1()
4   {
5       while (1)
6       {
7           while (turn == 1);
8           // 临界区
9           turn = 1;
10          // 非临界区
11      }
12  }
13
14  void Process2()
15  {
16      while (1)
17      {
18          while (turn == 0);
19          // 临界区
20          turn = 0;
21          // 非临界区
22      }
23  }
```

以上方法确实能够实现互斥，但是一方面，它可能违背条件三（临界区外的进程不能妨碍其他进程进入临界区），另一方面它还可能存在优先级反转问题。

此外还有Peterson算法等方法来实现进程互斥，但是都存在忙等待问题。

信号量 Semaphore

1965年，荷兰著名计算机科学家Dijkstra提出了信号量的概念，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操作与V操作

> **原子操作**

> 不可再分的操作，即原子操作不会被打断，时钟中断不会在原子操作中发生。

- P 操作：让计数变量的值减1，若结果小于 0，则进入阻塞状态。
- V 操作：让计数变量的值加1，如果结果不大于0，则唤醒一个阻塞的线程。

信号量的最大值代表着可使用临界区的最大进程数，或者可以理解为资源的数量。当设置信号量的最大值为1时，就相当于解决了我们上面说的进程互斥的问题。

信号量的使用方式一般为：

```
int semaphore = 1; //初始化信号量

// 非临界区
P(semaphore);
// 临界区
V(semaphore);
```

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);
int temp1 = a;
temp1 += 2;
a = temp1;
V(sem);

// Process 2
P(sem);
int temp2 = a;
temp2 += 3;
a = temp2;
V(sem);
```

这样的写法，能够保证计算正确吗？

互斥量

我们上面的例子中，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。

同步

进程间同步，指的是在多个进程中发生的事件之间存在某种时序关系，需要严格依赖这种时序关系执行各个进程。基于信号量，我们可以很方便地实现保证进程间的同步。这里不作过多介绍，留一个小例子作为作业。

C# 的库支持

原子操作

C# 提供了原子操作库，位于 `System.Threading` 命名空间的 `Interlocked` 类，其中提供了多种原子操作（作为静态方法），例如：

- `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`

信号量

C# 的信号量位于`System.Threading`命名空间下，常用的类为`Semaphore`和`SemaphoreSlim`，`SemaphoreSlim`是`Semaphore`的一个轻量替代。

我们主要介绍`Semaphore`类的使用，`SemaphoreSlim`用法类似。

`Semaphore`的最常用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 Semaphore(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。

`Semaphore`类的`WaitOne`方法实现P操作，`Release`方法实现`V`操作

锁

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，我们需要创建一个对象作为锁，例如：

```
var lockObj = new object();
```

然后就可以使用锁访问临界区

```
lock(lockObj)
{
    // code
}
```

它完全等同于

```
object __lockObj = lockObj;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

我们在编程时，应尽可能缩短持有锁的时间，以减少锁争用。

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。后者的更加轻量。

以下代码创建一个读写锁。

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

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

什么是死锁

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。这组进程称为死锁进程。

例如

```
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);
}
```

上述程序可能造成死锁。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，就会造成死锁。

死锁产生的条件

1971年，Coffman提出了死锁产生的4个条件：

- 互斥条件：资源要么分配给了某个进程，要么就是可用的
- 占有和等待条件：已经得到了某个资源的进程可以再请求新的资源（即一层锁不会导致死锁）
- 不可抢占条件：已经分配给一个进程的资源不能强制抢占，只能由占有它的进程自己释放
- 环路等待条件：死锁发生时，进程一定可以形成一个环路，环路中每一个进程都等待着下一个进程所占有的资源

以上4个条件缺一不可。但是这4个条件仅是死锁的必要不充分条件。

处理死锁的方法

~~比较常用的~~方法是

- 人工观察法
- 鸵鸟算法
- ...

死锁的检测、解除，往往围绕死锁产生的四个条件进行，感兴趣的同学可以查阅资料自行学习，这里不作过多介绍。

C# 异步

同步 与 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

在`System.Threading.Tasks`命名空间中，含有`Task`类，可以用于创建各种任务，其底层是基于线程池（注：线程池中的都是后台线程）。

```
Task.Run(Method1);           // 无返回值的任务
Task<TResult>.Run(Method2); // 返回值类型为TResult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;
});
```

这里，`var`会自动推导出`Task<int>`。

也可以这样创建：

```
var t = new Task<int>(() =>
{
    Thread.Sleep(1000);
    return 1;
});
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。 例如

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

将会输出：

```
1
End!
```

`await` & `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。一般只用记住前三个即可。

`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 `async` 方法中，`async` 方法中一般都要含有 `await`（否则没太大意义声明为 `async`）。

`await` 可以看成是 `Task` 的语法糖。

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

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);
}
```

异步 `Main` 方法

从 C# 7.1 开始，`Main` 方法可以声明为异步的。异步 `Main` 方法有以下四种形式：

```
public static async Task Main();
public static async Task<int> Main();
public static async Task Main(string[] args);
public static async Task<int> Main(string[] args);
```

这样在 `Main` 方法内也可以使用 `await` 了。在 `Main` 外部会调用 `Main` 所返回的 `Task` 的 `Wait` 方法或 `Result` 属性。

作业说明

题目及要求

现在有一个消费者，一个生产者。消费者生产一个蛋糕，消费者消费一个蛋糕。初始时没有蛋糕。两个线程开始的时间不确定。

要求：保证消费者消费蛋糕时，生产者已经把蛋糕生产好了。

修改 `MultiThreading/hw1/hw1/Program.cs` 指定部分的代码，允许添加字段或者属性或者方法，使程序满足要求。

使用工具

VS 2022

作业的 C# 程序运行在.NET 6平台上，建议使用Visual Studio 2022进行编写。

使用 Visual Studio 2022直接打开 `MultiThreading/hw1/hw1.sln` 进行编写。

VScode

也可以使用 VScode，使用 VScode 请保证你的计算机上有.NET 6 环境。

使用 VScode 直接修改 `MultiThreading/hw1/hw1/Program.cs` 即可。若要检验结果是否正确，可以使用命令启动 `Program.cs` 程序

```
dotnet run --project MultiThreading/hw1/hw1
```

或直接用我们写好的测试程序进行测试

```
dotnet test MultiThreading/hw1/hw1.sln
```

提交方式

提供两种提交方式：

1. GitHub 提交
   - fork 仓库： TCL606/MultiThreading: 2022 THU EESAST 软件部暑期培训 (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信息写为 `多线程_姓名_班级`（如：`多线程_小明_无19`）。
2. 邮箱提交
   - 只允许修改 `MultiThreading/hw1/hw1/Program.cs` 代码，修改好后，将 `MultiThreading` 整个文件夹打包成常见压缩格式（如 `.rar`、`.zip` 等），并命名为：`多线程_姓名_班级`（如：`多线程_小明_无19`）发送到邮箱 **tcl606_thu@163.com**。

截止日期

2022.7.21

参考文献

1. 《操作系统》 谌卫军，王浩娟 编著，清华大学出版社
2. 多线程与异步 | EESΛST Docs (eesast.com)，2022年7月2日