

教育部与高通公司产学合作协同育人项目

燕山大学  
机上作业报告

课程名称： 计算机操作系统

主题： 感受、体验和欣赏 OS 中的创新

题目： 多线程程序设计

姓名	学号	班级	自评成绩
乔翱	201811040809	软件六班	A+

指导教师： 申利民

2020 年 10 月 26 日

[www.wtosonline.com](http://www.wtosonline.com)

# 目录

一、实验目的和意义.....	3
二、实验基本要求.....	3
三、实验设计和实现.....	3
四、比较单道程序和多道程序的运行时间.....	4
1、实验环境.....	4
2、实验设计思路.....	4
3、实验结果.....	4
4、结果分析.....	5
五、改进实验.....	6
1、实验设计思路.....	6
2、实验结果.....	6
3、结果分析.....	6
六、设计生产者-消费者实验体会多线程.....	7
1、实验设计思路.....	7
2、实验结果.....	7
3、结果分析.....	8
七、创新实验-多线程爬虫.....	9
1、设计思路.....	9
2、实验结果.....	9
3、结果分析.....	10
八、多线程的应用场景.....	10
1、应用服务器软件.....	10
2、后台离线任务.....	10
3、异步处理任务.....	10
4、爬虫程序.....	11
九、实验总结与体会.....	11

# 一、实验目的和意义

多线程（multithreading），是指从软件或者硬件上实现多个线程并发执行的技术。具有多线程能力的计算机因有硬件支持而能够在同一时间执行多于一个线程，进而提升整体处理性能。具有这种能力的系统包括对称多处理机、多核心处理器以及芯片级多处理或同时多线程处理器。在一个程序中，这些独立运行的程序片段叫作“线程”（Thread），利用它编程的概念就叫作“多线程处理”。

通过本次实验，编写单线程程序和多线程程序，比较多道程序和单道程序的运行时间，体会多线程的优点。掌握多线程编程的特点和工作原理，掌握如何编写多线程程序，了解多线程的工作原理，了解线程调度和执行过程，了解并且掌握线程同步原理。体会多线程的优点，体验多线程编程的魅力。

## 二、实验基本要求

编写一个程序，满足下列要求：

输入一个 1-10 的整数  $x$ ，计算  $y=f_1(x)+f_2(x)+f_3(x)$ ，并输出  $y$ ；

(2) 其中  $f_1$ ， $f_2$ ， $f_3$  三个函数分别由不同的线程同时执行；

(3)  $f_1$ ：计算  $x$  到 50 的和并返回；

(4)  $f_2$ ：计算  $x$  到 500 的和并返回；

(5)  $f_3$ ：计算 1 到  $x$  的阶乘并返回；

(6) 不限制语言。

## 三、实验设计和实现

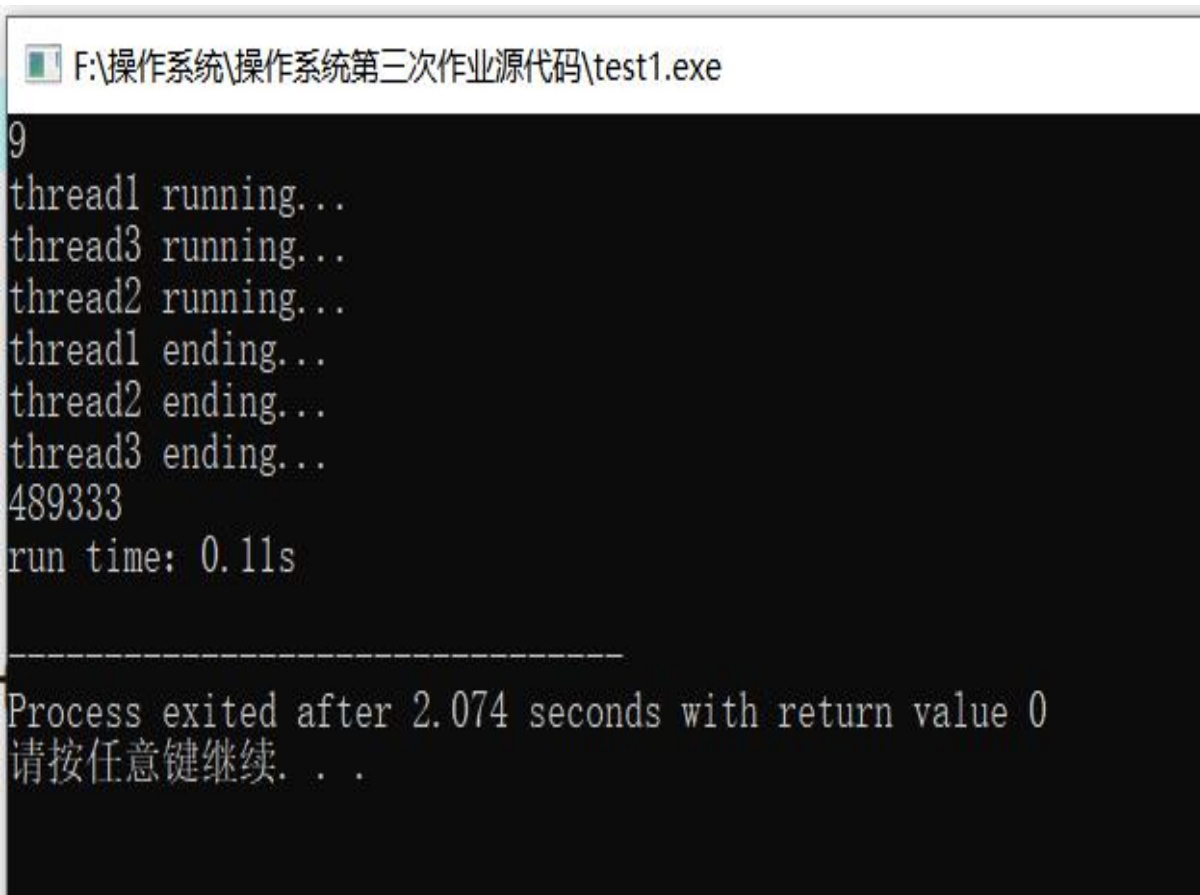
C++ 11 新标准中，正式的为该语言引入了多线程概念。新标准提供了一个线程库 `thread`，通过创建一个 `thread` 对象来管理 C++ 程序中的多线程。C++11 之前，Windows 和 Linux 平台分别有各自的多线程标准。使用 C++ 编写的多线程往往是依赖于特定平台的。Window 平台提供用于多线程创建和管理的 `win32 api`；Linux 下则有 POSIX 多线程标准，`Threads` 或 `Pthreads` 库提供的 API 可以在类 Unix 上运行。

在 C++11 新标准中，可以简单通过使用 `thread` 库，来管理多线程。`thread` 库可以看做对不同平台多线程 API 的一层包装；因此使用新标准提供的线程库编写的程序是跨平台的。

对于本次基础实验，采用 C++ 语言进行程序的编写，首先编写  $f_1$ 、 $f_2$ 、 $f_3$  三个函数，三个函数都较为简单，容易编写。接着利用 C++ 中的线程库中的 `thread` 声明三个线程， $f_1$ 、 $f_2$ 、 $f_3$  三个函数分别利用不同的线程同时进行，使得三个函数并行执行。

以  $x=9$  为例，运行该多线程程序，输出程序运行结果，并且输出程序的执行时间。记录实验结果。

实验运行结果如下图所示：



```
F:\操作系统\操作系统第三次作业源代码\test1.exe
9
thread1 running...
thread3 running...
thread2 running...
thread1 ending...
thread2 ending...
thread3 ending...
489333
run time: 0.11s

-----
Process exited after 2.074 seconds with return value 0
请按任意键继续. . .
```

图 3.1 基础实验程序运行结果

## 四、比较单道程序和多道程序的运行时间

### 1、实验环境

操作系统：Windows10；

编译环境：Dev c++；

处理器核数：10。

### 2、实验设计思路

设计实验比较单道程序和多道程序的运行时间，分析实验结果，体会单线程和多线程各自的特点。

首先编写一个单线程的程序，即顺序执行 f1、f2、f3 三个函数。在单线程程序和多线程程序中分别多次循环执行 f1、f2、f3 三个函数，记录程序的执行时间，进行比较。

### 3、实验结果

改变程序中对于 f1、f2、f3 的循环次数，比较单道程序和多道程序的运行时间，记录实验程序执行时间，实验结果以折线图形式展示，横轴代表的是 f1、f2、f3 三个函数循环的次数，纵轴代表的是程序执行的时间。

实验结果折线图如下图所示：

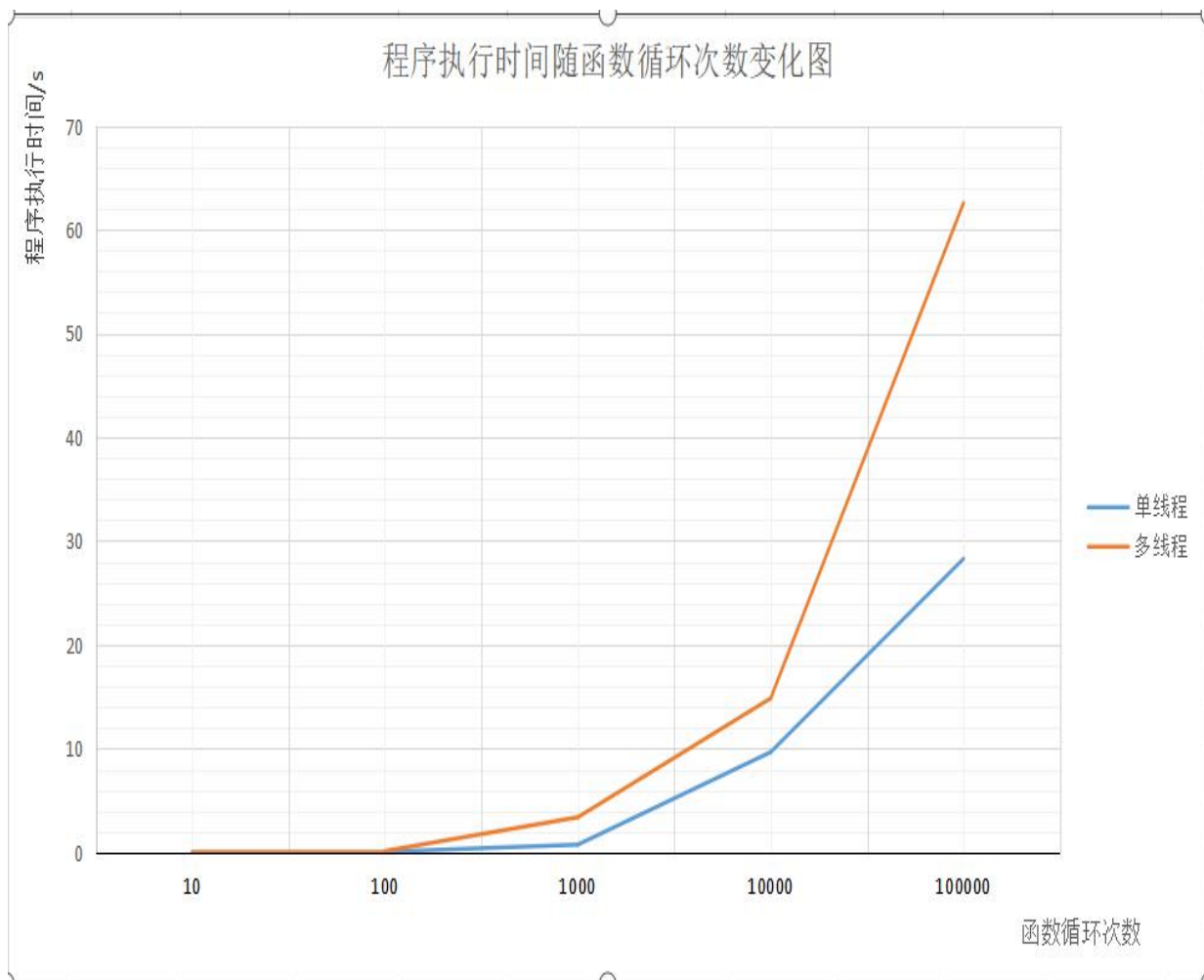


图 4.2 基础实验实验结果

#### 4、结果分析

实验之前，根据理论分析，多线程程序在多核下运行时间应该小于单线程在多核下运行，但是经过实验后发现，多线程在多核下并没有提升速度，反而时间比单线程运行时间更长。分析结果可以得出，可能是因为多线程程序较为简单，导致线程之间的切换时间收益比线程并行的收益大。多线程执行时线程调度所花费的时间比程序执行时间高，导致最终多线程程序执行时间比单线程多。

分析结果可以得出，并不是任何时候使用多线程都更好，对于处理时间短的服务或者启动频率高的用单线程更好，

## 五、改进实验

### 1、实验设计思路

根据前面的基础实验可以看出，对于简单的程序来说，多线程的运行时间反而要高于单线程，这是由于多线程中，CPU 调度要花的时间多于程序本身执行的时间。

由于基础实验中的程序 f1、f2、f3 三个函数较为简单，处理时间较短，在多线程中调度的时间大于处理的时间，所以会导致多线程的程序比单线程的程序运行时间多。所以在改进的程序中，在 f1、f2、f3 中都加入延时函数 Sleep（100），在每个函数中延时 0.1s，使得每个函数的处理时间变长，再次进行对多线程程序和单线程程序的执行时间的比较。

### 2、实验结果

改变程序中对于 f1、f2、f3 的循环次数，比较单道程序和多道程序的运行时间，记录实验程序执行时间，实验结果以折线图形式展示，横轴代表的是 f1、f2、f3 三个函数循环的次数，纵轴代表的是程序执行的时间。

实验结果折线图如下图所示：

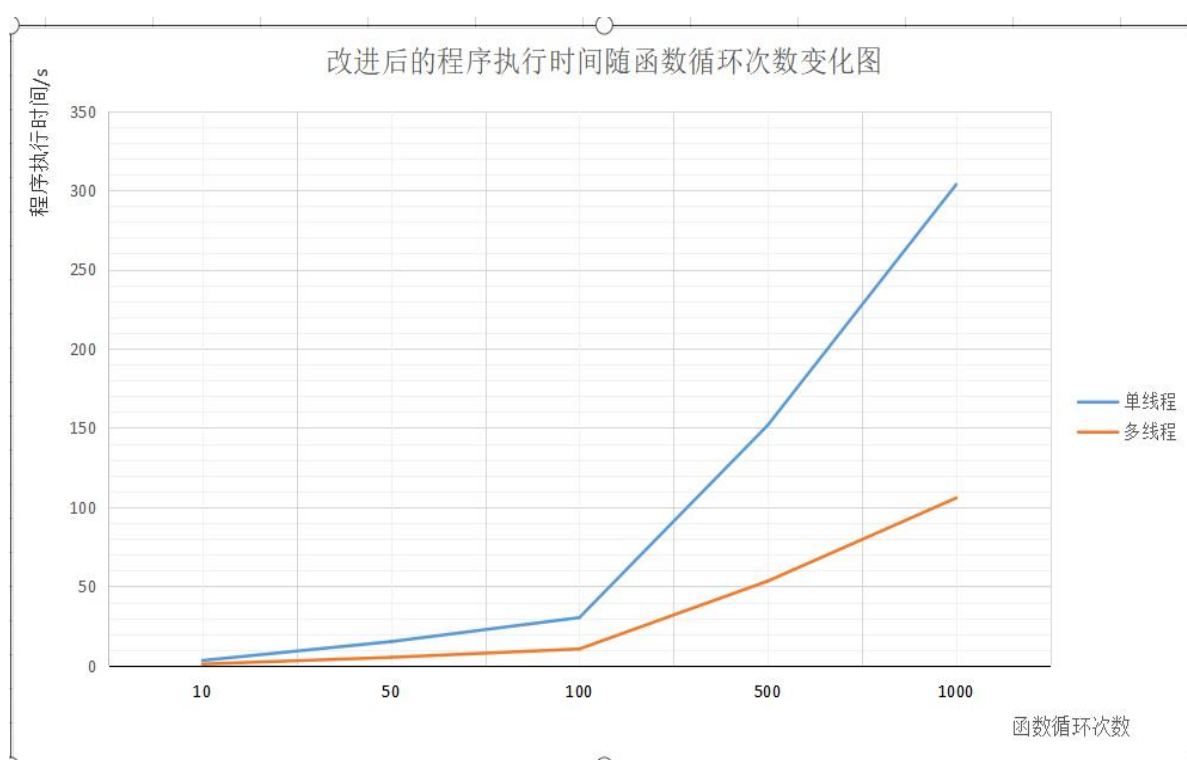


图 5.2 改进实验实验结果

### 3、结果分析

根据实验结果可以发现，多线程程序在多核下运行时间小于单线程在多核下运行，因为多线程在运行时由于多个线程并行运行，所以节省了时间。

多线程程序的并行执行加快了程序的执行速度，提高了效率，多道程序的并行执行相对于顺序执行的单道程序，大大缩短了程序的执行时间。

## 六、设计生产者-消费者实验体会多线程

### 1、实验设计思路

生产者-消费者模式是一个十分经典的多线程并发协作的模式，通过生产者-消费者问题可以加深对并发编程的理解。所谓的生产者-消费者问题，实际上主要是包含了两类进程，一种是生产者线程用于生产，另一种是消费者线程用于消费数据，为了解耦生产者和消费者的关系，通常会采用共享的数据区域，就像是一个仓库，生产者生产数据之后直接放置在共享数据区中，并不需要关心消费者的行为；而消费者只需要从共享区中获取数据，就不再关心生产者的行为。

设计单线程和多线程的程序进行比较，单线程的程序即只有一个线程，只能先生产完所有的产品后，消费者再进行消费所有的产品。而多线程则有两个线程，一个是生产者线程，一个是消费者线程，两个线程并发执行，但是对于缓冲区的操作同一时刻只能有一个线程对其操作。

对于多线程的程序需要使用互斥锁，缓冲区相对于一个互斥的资源，同一时刻，不能有多个进程对于缓冲区的数据进行操作，这样其中的数据就会发生混乱，也就会造成程序的混乱。

在多线程中，有多个线程竞争同一个公共资源，就很容易引发线程安全的问题，因此就需要引入锁的机制，来保证任意时刻只能有一个进程在访问公共资源。直接利用 C++ 中的 `mutex`，`mutex` 互斥锁是一个可锁的对象，它被设计成在关键的代码段需要独占访问时发出信号，从而阻止具有相同保护的其他线程并发执行并访问相同的内存位置。

C++ 中的 `unique_lock` 对象以独占所有权的方式(官方叫做 `unique ownership`)管理 `mutex` 对象的上锁和解锁操作；独占所有权，就是没有其他的 `unique_lock` 对象同时拥有某个 `mutex` 对象的所有权。

`unique_lock` 对象在析构的时候一定保证互斥量为解锁状态；因此它作为具有自动持续时间的对象特别有用，因为它保证在抛出异常时，互斥对象被正确地解锁。

`mutex` 以及 `unique_lock` 的使用示例如下：

```
#include <mutex>
mutex mtx; //声明互斥锁
unique_lock<mutex> lock(b->mtx); //设置互斥锁
...
...
...
lock.unlock(); //操作完毕，接触互斥锁
```

### 2、实验结果

多次运行单线程程序和多线程程序，记录运行时间，实验结果以折线图的形式展现，横轴是程序运行次数，纵轴是程序执行时间。

实验结果如下图所示：

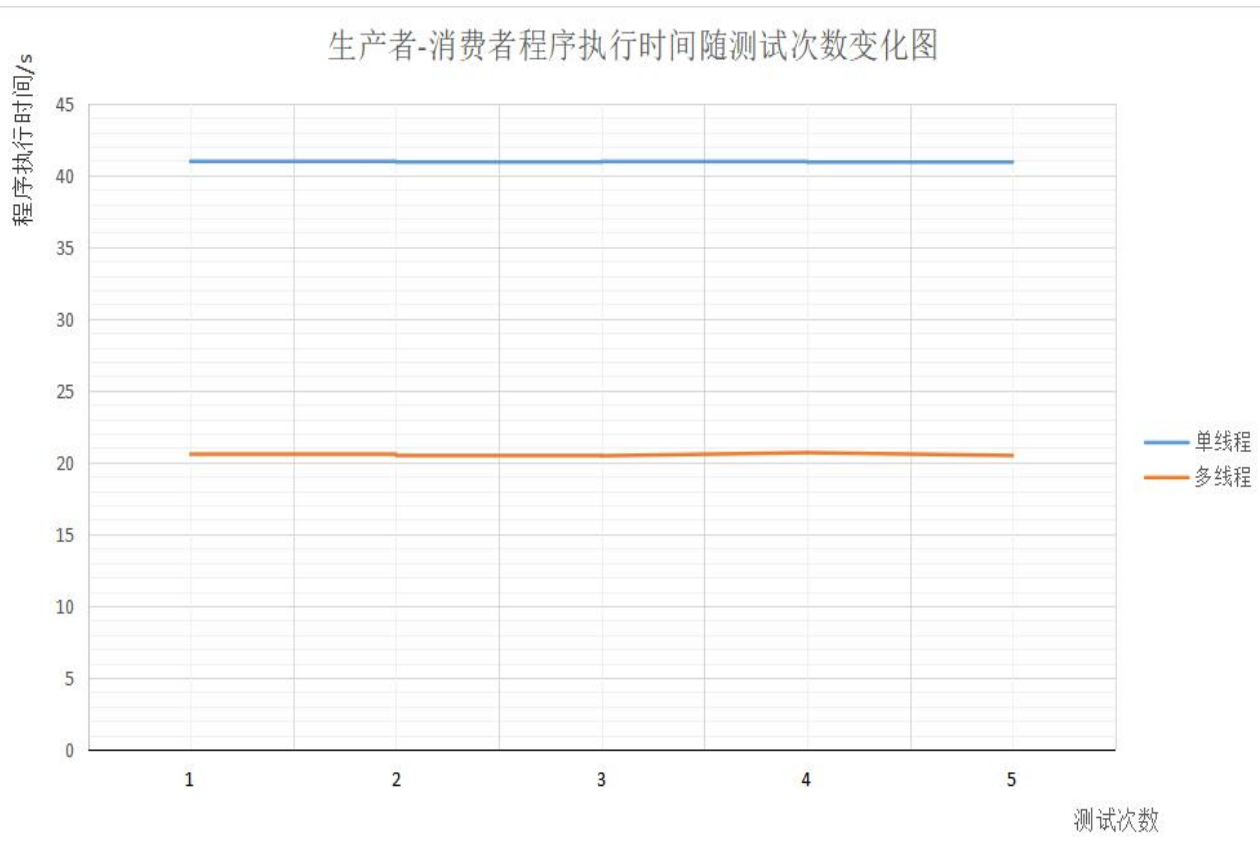


图 6.2 生产者消费者实验结果

### 3、结果分析

根据实验结果，可以看出多线程的生产者消费者程序执行的速度远远快于单线程的程序。通过学习生产者消费者模型，更好的理解了多线程程序的编写，更深刻理解了进程的互斥和同步。

同步也叫作直接制约关系，指为完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调它们的工作次序而等待、传递信息所产生的制约关系。进程间的直接制约关系就是源于它们之间的相互合作。

生产者进程向缓冲区放数据，消费者进程从缓冲区取出数据。当缓冲区为空时，消费者进程不能获得所需数据而阻塞，一旦生产者进程将数据送入缓冲区，消费者进程被唤醒。当缓冲区满时，生产者进程就被阻塞，仅当消费者进程取走缓冲数据时，才能唤醒生产者进程。互斥亦称间接制约关系。当一个进程进入临界区使用临界资源时，另一个进程必须等待，当占用临界资源的进程退出临界区后，另一进程才允许去访问此临界资源。

对于缓冲区来说，同一时刻只能由一个进程对其进行访问，当生产者进程访问缓冲区往缓冲区放数据的时候，消费者进程不能对缓冲区操作，取出数据。

在编写多线程程序的时候要注意对于资源的访问，判断该资源是否是互斥资源，注意根据各个进程的互斥和同步关系来编写程序。



## 七、创新实验-多线程爬虫

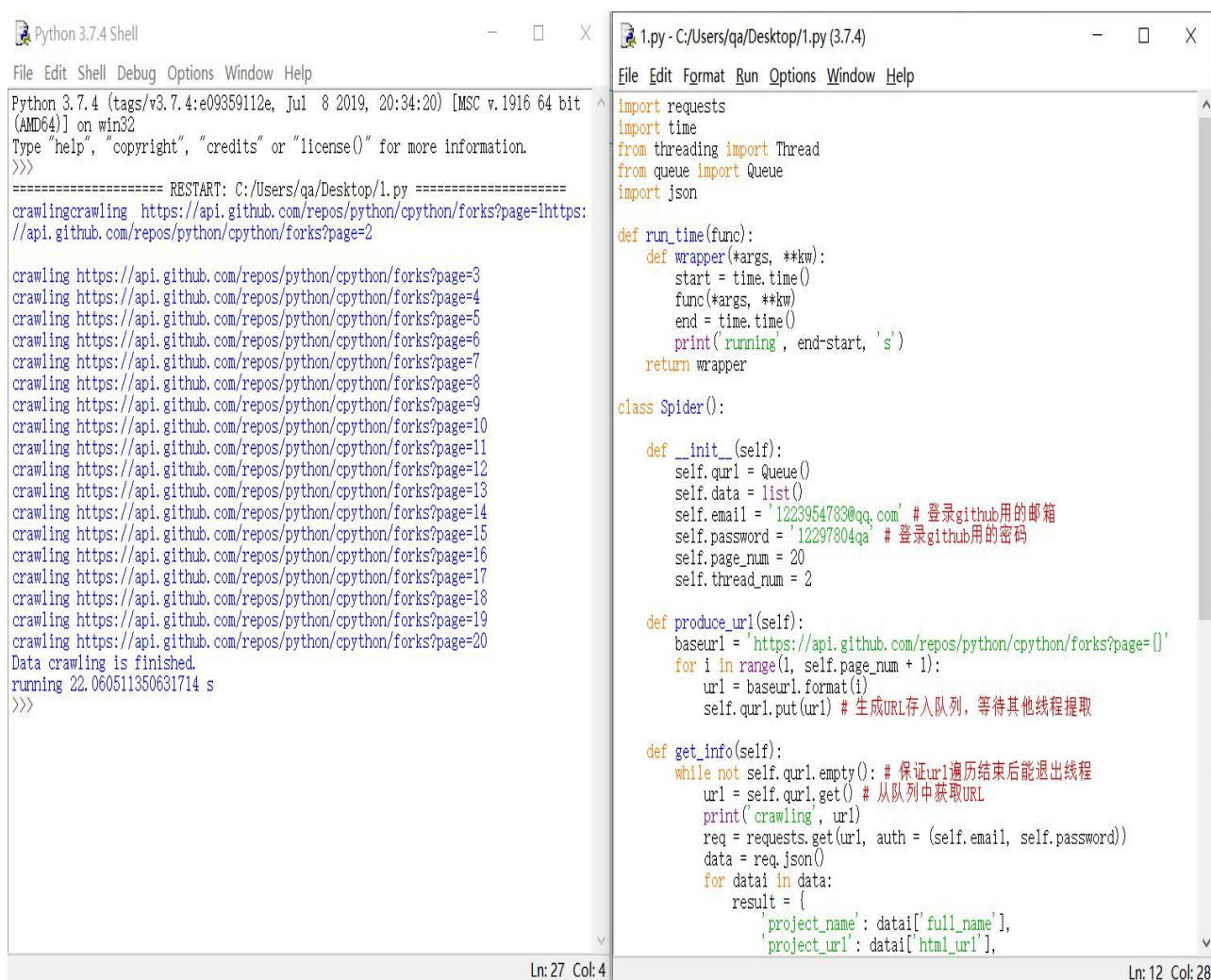
### 1、设计思路

爬虫主要运行时间消耗是请求网页时的 IO 阻塞，开启多线程让不同请求的等待同时进行，可以大大提高爬虫运行效率。在日常生活中，利用爬虫爬取大量的数据往往需要花费大量的时间，通常利用多线程爬虫则会加快爬取速度，提高爬取数据的效率。

本创新实验，基于多线程，使用 Github 的 API，抓取 fork cpython 项目的所有五千个项目的信息，将数据存储到 json 文件中。Github 上对爬取的速度有限制，所以从理论上来看，利用多线程进行爬取可以缩短爬取时间。

### 2、实验结果

本实验分别开启不同的线程数爬取了 20 个页面，记录下程序运行时间，进行比较，实验结果以折线图的形式进行显示，横轴是使用的线程数，纵轴是程序执行时间。



```
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/qa/Desktop/1.py =====
crawling crawling https://api.github.com/repos/python/cpython/forks?page=1https://api.github.com/repos/python/cpython/forks?page=2
crawling https://api.github.com/repos/python/cpython/forks?page=3
crawling https://api.github.com/repos/python/cpython/forks?page=4
crawling https://api.github.com/repos/python/cpython/forks?page=5
crawling https://api.github.com/repos/python/cpython/forks?page=6
crawling https://api.github.com/repos/python/cpython/forks?page=7
crawling https://api.github.com/repos/python/cpython/forks?page=8
crawling https://api.github.com/repos/python/cpython/forks?page=9
crawling https://api.github.com/repos/python/cpython/forks?page=10
crawling https://api.github.com/repos/python/cpython/forks?page=11
crawling https://api.github.com/repos/python/cpython/forks?page=12
crawling https://api.github.com/repos/python/cpython/forks?page=13
crawling https://api.github.com/repos/python/cpython/forks?page=14
crawling https://api.github.com/repos/python/cpython/forks?page=15
crawling https://api.github.com/repos/python/cpython/forks?page=16
crawling https://api.github.com/repos/python/cpython/forks?page=17
crawling https://api.github.com/repos/python/cpython/forks?page=18
crawling https://api.github.com/repos/python/cpython/forks?page=19
crawling https://api.github.com/repos/python/cpython/forks?page=20
Data crawling is finished.
running 22.060511350631714 s
>>>

1.py - C:/Users/qa/Desktop/1.py (3.7.4)
File Edit Format Run Options Window Help
import requests
import time
from threading import Thread
from queue import Queue
import json

def run_time(func):
    def wrapper(*args, **kw):
        start = time.time()
        func(*args, **kw)
        end = time.time()
        print('running', end-start, 's')
    return wrapper

class Spider():

    def __init__(self):
        self.qurl = Queue()
        self.data = list()
        self.email = '1223954783@qq.com' # 登录github用的邮箱
        self.password = '12297804qa' # 登录github用的密码
        self.page_num = 20
        self.thread_num = 2

    def produce_url(self):
        baseurl = 'https://api.github.com/repos/python/cpython/forks?page='
        for i in range(1, self.page_num + 1):
            url = baseurl.format(i)
            self.qurl.put(url) # 生成URL存入队列，等待其他线程提取

    def get_info(self):
        while not self.qurl.empty(): # 保证url遍历结束后能退出线程
            url = self.qurl.get() # 从队列中获取URL
            print('crawling', url)
            req = requests.get(url, auth = (self.email, self.password))
            data = req.json()
            for data1 in data:
                result = {
                    'project_name': data1['full_name'],
                    'project_url': data1['html_url'],
```

图 7.2.1 爬虫程序运行结果

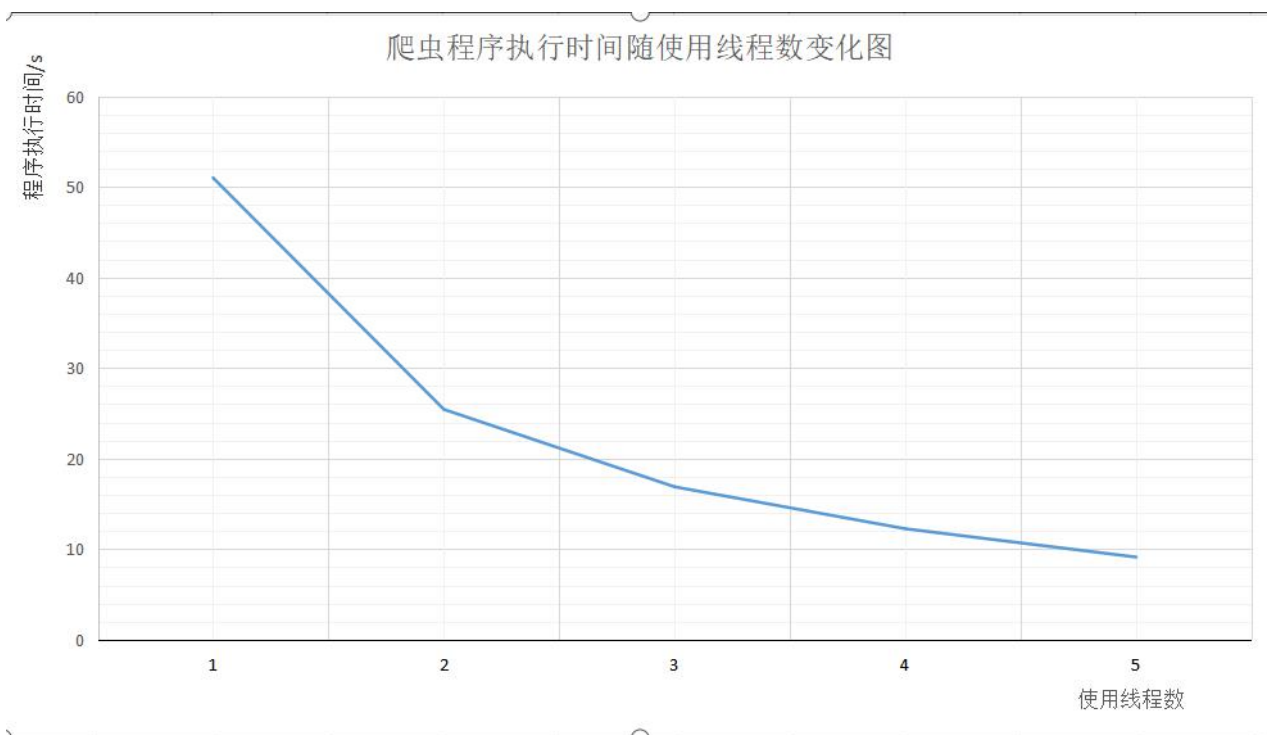


图 7.2.2 爬虫实验结果

### 3、结果分析

根据实验结果可以看出，多线程爬虫比单线程爬虫的速度快很多，而且在一定范围内，随着线程的增多，爬虫程序的爬取速度也会更快。

对于某些网站对访问速度有限制，这样的网站可以开启多个线程，每个线程使用一个代理，去提取页面的一部分内容，这样可以大大加快爬虫的速度，提高爬取数据的效率。

## 八、多线程的应用场景

### 1、应用服务器软件

对于常用的 Tomcat、Nginx、Jetty 等各种服务器软件都是支持多线程的。因为线上服务肯定都不是一个人使用的，只要有多人访问，就可能出现并发的情况，其次，服务端软件对处理速度也是有一定的要求的，所以服务端软件必须是支持多线程的。

### 2、后台离线任务

后台离线任务通常都需要通过多线程的方式加快处理速度，比如对于 mysql 数据表进行统计，对多个表进行汇总，每个表作为一个汇总指标存在，这样就可以针对每个表创建一个线程处理。或者有分库分表的情况，可以针对每个分表开启一个线程处理。多线程的方式大大加速了后台离线任务的处理速度。

### 3、异步处理任务

异步处理任务类似于离线任务，同样是对实时性要求不是很高的情况下。比如电商系统中经常用到的短信、邮件通知的情况，用户在某电商网站下单付款后，通常会收到短信或者

邮件的通知，而通知信息对于整个购买环节并不是最重要的，商家最关心的就是减掉库存和收到付款，所以对于通知的发送一般都采用异步方式，允许一定的延时甚至发送失败的情况。

当用户购买商品成功后，系统会向消息队列中写入订单相关的信息，发送通知的异步任务去消息队列拉取消息，拉到一个订单就向对应的手机或者邮箱发送通知消息。而这里的发送通知任务一般都采用多线程的方式，用来提高并发度，减小用户下单成功到收到通知之间的延时。

## 4、爬虫程序

一般用爬虫就是需要大批量数据的场景，比如说抓取某个或某些微博大 V 的微博内容，比如抓取知乎回答，豆瓣电影排行信息。对于大量的数据的爬取如果采用单线程通常会消耗大量的时间，为了加快速度、提高效率，一般都会采用多线程的方式爬取数据。

爬虫也是生产者消费者模式的经典应用场景，多个线程（这里的线程也就是生产者模式中的生产者线程）到目标网站上抓数据；然后将数据放到一个中间队列，最后，多个消费者线程订阅中间队列，将数据加工处理后存入数据库或者写入文件。

## 九、实验总结与体会

通过本次实验体会到了多线程程序的优势，但是多线程程序也不是在任何情况下都比单线程程序运行速度快，多线程可以提高 CPU 的利用率，更好地利用系统资源

采用多线程技术的应用程序可以更好地利用系统资源。充分利用了 CPU，用尽可能少的时间来对用户的要求做出响应，使得进程的整体运行效率得到较大提高，同时增强了应用程序的灵活性。

同一进程的所有线程是共享同一内存，不需要特殊的数据传送机制，不需要建立共享存储区或共享文件，从而使得不同任务之间的协调操作与运行、数据的交互、资源的分配等问题更加易于解决。

在多线程应用中，考虑不同线程之间的数据同步和防止死锁。当两个或多个线程之间同时等待对方释放资源的时候就会形成线程之间的死锁。为了防止死锁的发生，需要通过同步来实现线程安全。

总结来说，多线程程序资源利用率更好，设计更简单，程序响应更快。