**《操作系统》课第07次实验报告**

|  |  |
| --- | --- |
| 学院: | 软件学院 |
| 姓名: | 张怡桢 |
| 学号: | 2013747 |
| 邮箱: | 2662765987@qq.com |
| 时间: | 11/3/2022 |

1. **开篇感言**

“你长大后想成为什么人？”

“什么意思？长大后我就不能成为我自己了吗？”

-- 《阿甘正传》

2. **实验题目**

多线程拷贝目录，并与单进程拷贝目录与多进程拷贝进程拷贝做效率比较

3. **实验目标**

1. Write a c/c++ program
2. To implement copy one directory and it's subdiretories with multi-threads
3. GCC
4. Test directory: 使用最新的Linux Kernel来测试(从www.kernel.org下载最新的linux内核)
5. <https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.19.10.tar.xz>
6. extract linux-5.19.10.tar.xz to linux-5.19.10 directory,
7. and copy linux-5.19.10 directory to linux-5.19.10bak directory
8. Verify that the directory copy is correct
9. Compare the advantages and disadvantages of two different solutions: multi-process and multi-threaded

4. **原理方法**

4.1  **比较目录是否相同**

diff -r DirA DirB

4.2  **get the total time of program execution**

$ time pwd

4.3  **structure of directory**

struct dirent

{

ino\_t d\_ino; //d\_ino 此目录进入点的inode

ff\_t d\_off; //d\_off 目录文件开头至此目录进入点的位移

signed short int d\_reclen; //d\_reclen \_name 的长度, 不包含NULL 字符

unsigned char d\_type; //d\_type d\_name 所指的文件类型 d\_name 文件名

har d\_name[256];

};

the value returned in d\_type:

DT\_BLK This is a block device.

DT\_CHR This is a character device.

DT\_DIR This is a directory.

DT\_FIFO This is a named pipe (FIFO).

DT\_LNK This is a symbolic link.

DT\_REG This is a regular file.

DT\_SOCK This is a UNIX domain socket.

DT\_UNKNOWN The file type could not be determined.

opendir()

readdir()

closedir()

4.4  **Create a symbol link file**

#include <fcntl.h> /\* Definition of AT\_\* constants \*/

#include <unistd.h>

int link(const char \*oldpath, const char \*newpath);

4.5  **create process and execute one program**

4.5.1  **fork(): clone a new instance of current process**

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

4.5.2  **exec():**

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg, ..., char \*const envp[]);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

int execve(const char \*path, char \*const argv[], char \*const envp[]);

4.6  **命令行参数**

int main(int argc, char\* argv[]){

int i;

for (i = 0; i < argc; i++)

{

printf ("%3d %s\n", i, argv[i]);

}

}

4.7  **posix thread**

#include <pthread.h>

pthread\_create()

4.8  **posix mutex互斥量**

#include <pthread.h>

pthread\_mutex\_t mutex =PTHREAD\_MUTEX\_INITIALIZER;

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

5. **代码分析**

5.1  **copyfile作用是复制文件**

|  |
| --- |
| C++ //复制文件 void copyfile(string srcFile, string dstFile) {  int infd, outfd;  char buffer[BUF\_SIZE];  int i;  if ((infd = open(srcFile.c\_str(), O\_RDONLY)) < 0)  {  cout << "Open srcFile failed:" << srcFile.c\_str() << endl;  return;  }  if ((outfd = open(dstFile.c\_str(), O\_WRONLY | O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR)) < 0)  {  cout << "Open dstFile failed:" << dstFile.c\_str() << endl;  return;  }  // cout << "Copyfile from " << srcFile << " to " << dstFile << endl;  while (1)  {  i = read(infd, buffer, BUF\_SIZE);  if (i <= 0)  break;  write(outfd, buffer, i);  }  close(outfd);  close(infd); } |

5.2  **设计一个结构体，用来存放源地址与目的地址**

|  |
| --- |
| C++ class Container { public:  string srcFile;  string dstFile;  bool ifLink; }; stack<Container> containerStack; //存放待复制的普通文件的源地址和目的地址的栈 |

5.3 **遍历目录，存放地址**

|  |
| --- |
| C++ //遍历目录 void walk\_dir(char \*srcDir, char \*dstDir) {  DIR \*dir;  struct dirent \*ptr;  //打开目录，并判断打开是否成功  dir = opendir(srcDir);  if (dir == NULL)  {  cout << "打开 srcDir 失败" << endl;  exit(0);  }  //提取文件夹信息  char \*path = new char[PATHLEN];  while ((ptr = readdir(dir)) != NULL)  {  //遇到. ..就跳过  if ((!strcmp(ptr->d\_name, ".")) || (!strcmp(ptr->d\_name, "..")))  {  continue;  }  struct stat s;  snprintf(path, PATHLEN, "%s/%s", srcDir, ptr->d\_name);  lstat(path, &s);  if (stat(path, &s) == 0)  {  //若该地址是一个文件夹  if (S\_ISDIR(s.st\_mode))  {  char \*subSrcPath = new char[PATHLEN];  char \*subDstPath = new char[PATHLEN];  snprintf(subSrcPath, PATHLEN, "%s/%s", srcDir, ptr->d\_name);  snprintf(subDstPath, PATHLEN, "%s/%s", dstDir, ptr->d\_name);  //若该文件夹是一个软连接，将其ifLink设为true，入栈  if (ptr->d\_type == 10)  {  struct Container new\_container;  char \*linkFile = new char[PATHLEN];  readlink(subSrcPath, linkFile, 1024);  new\_container.srcFile = linkFile;  new\_container.dstFile = subDstPath;  new\_container.ifLink = true;  containerStack.push(new\_container);  continue;  }  //在目标地址处创建文件夹  mkdir(subDstPath, S\_IWUSR | S\_IRUSR | S\_IXUSR | S\_IRGRP | S\_IXGRP | S\_IROTH | S\_IXOTH);  //递归遍历子目录  walk\_dir(subSrcPath, subDstPath);  }  //若该地址是一个普通文件  else if (S\_ISREG(s.st\_mode))  {  char \*srcFile = new char[PATHLEN];  char \*dstFile = new char[PATHLEN];  snprintf(srcFile, PATHLEN, "%s/%s", srcDir, ptr->d\_name);  snprintf(dstFile, PATHLEN, "%s/%s", dstDir, ptr->d\_name);  //将待拷贝的文件的源地址和目的地址打包放入栈中  struct Container new\_container;  new\_container.srcFile = srcFile;  new\_container.dstFile = dstFile;  //若该文件是一个软连接，将其ifLink设为true，入栈  if (ptr->d\_type == 10)  {  char \*linkFile = new char[PATHLEN];  readlink(srcFile, linkFile, 1024);  new\_container.srcFile = linkFile;  new\_container.ifLink = true;  containerStack.push(new\_container);  continue;  }  containerStack.push(new\_container);  }  else  {  cout << "not file not directory" << endl;  exit(0);  }  }  else  {  cout << "error, path doesn't exist" << endl;  exit(0);  }  }  closedir(dir); } |

|  |  |
| --- | --- |
| 5.4 **创建多线程** |  |

|  |
| --- |
| C++ int main(int argc, char \*\*argv) {  walk\_dir(argv[1], argv[2]);  //初始化互斥量  pthread\_mutex\_init(&amutex, NULL);  //创建线程  pthread\_t threads[THREAD\_NUM];  for (int i = 0; i < THREAD\_NUM; i++)  {  pthread\_create(&threads[i], NULL, run, NULL);  }  //回收各线程  for (int i = 0; i < THREAD\_NUM; i++)  {  cout << i << endl;  pthread\_join(threads[i], NULL);  }  //销毁互斥量  pthread\_mutex\_destroy(&amutex);  return 0; } |

5.5 **设置线程运行函数**

|  |
| --- |
| C++ //每个线程运行的函数 void \*run(void \*) {  Container tmp;  while (containerStack.empty() == false)  {  //上锁，各线程动态获取拷贝任务  pthread\_mutex\_lock(&amutex);  if (containerStack.empty() == true)  break;  tmp = containerStack.top();  containerStack.pop();  pthread\_mutex\_unlock(&amutex);  //若是软连接，则调用symlink接口创建软连接  if (tmp.ifLink == true)  {  // cout << "Create link:" << tmp.srcFile.c\_str() << " link to " << tmp.dstFile.c\_str() << endl;  symlink(tmp.srcFile.c\_str(), tmp.dstFile.c\_str());  }  //若是普通文件，则调用copyfile函数拷贝文件  else  {  copyfile(tmp.srcFile, tmp.dstFile);  }  }  return NULL; } |

5.6 **设置线程数量，互斥锁等全局变量，以及头文件**

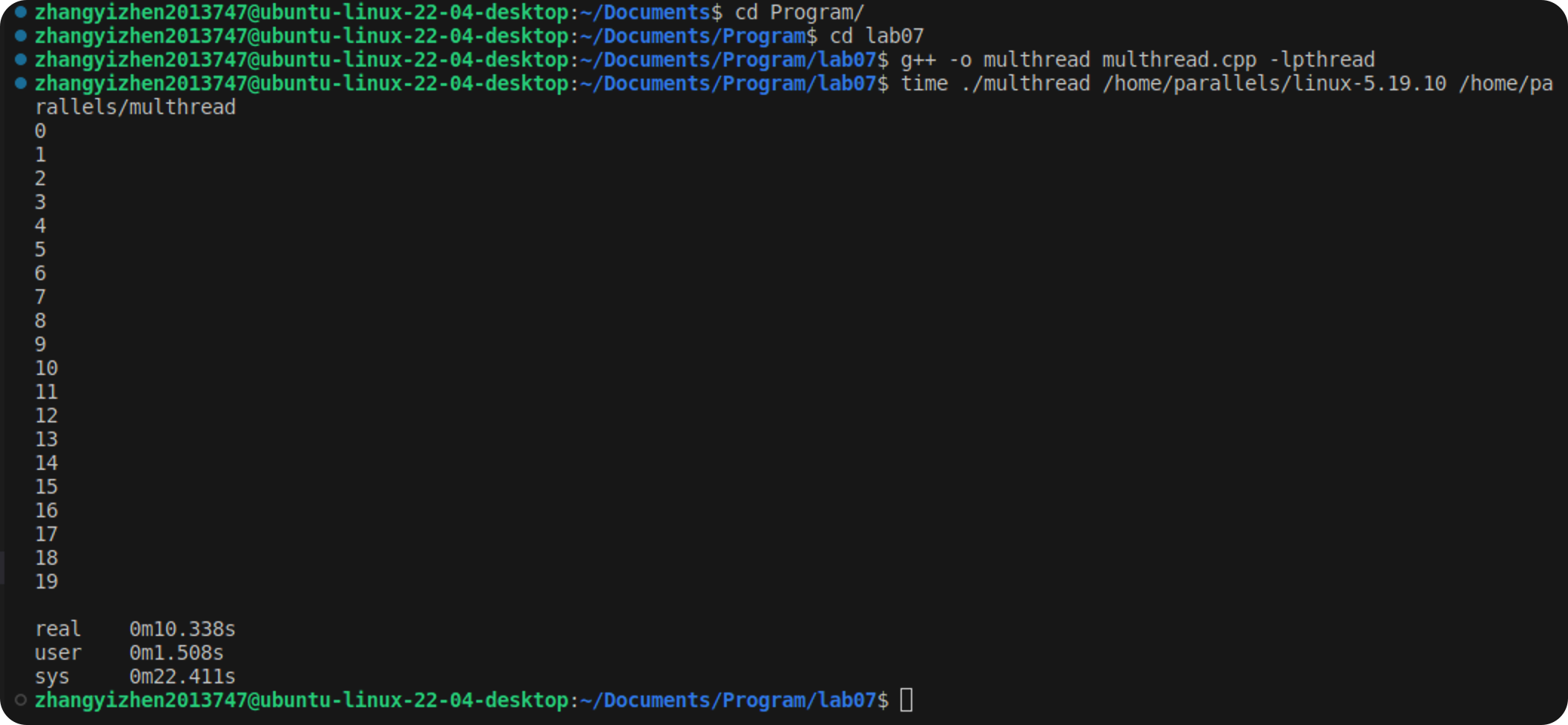
|  |
| --- |
| C++ #include <pthread.h> #include <sys/types.h> #include <sys/stat.h> #include <fcntl.h> #include <stdlib.h> #include <stdio.h> #include <iostream> #include <stack> #include <dirent.h> #include <cstring> #include <csignal> #include <sys/wait.h> #include <unistd.h>  using namespace std; const int PATHLEN=1024; const int THREAD\_NUM=20; const int BUF\_SIZE=1024; pthread\_mutex\_t amutex; |

6. **具体步骤**

6.1 **多线程复制**

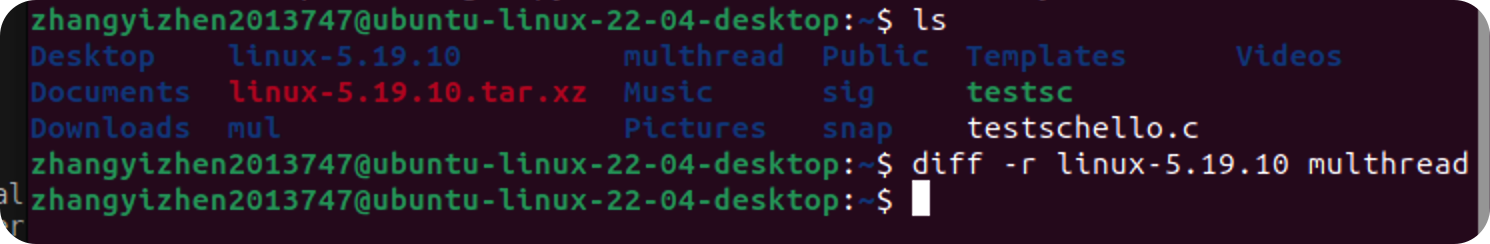
设置最大线程数为20，创建20个线程进行文件夹拷贝

编译代码，用time计算时间，将linux-5.19.10拷贝到同目录的multhread文件夹下



完成拷贝用时如上图，可见多线程拷贝速度非常快。

使用differ与原文件进行文件验证

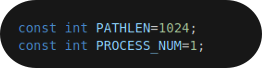


完全一致，没有问题

6.2  **进程复制**

6.2.1 **单进程复制**

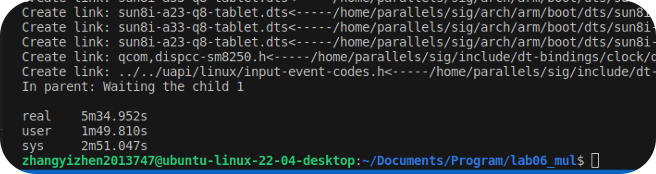
设置最大进程数为1，只创建一个子进程进行文件夹拷贝



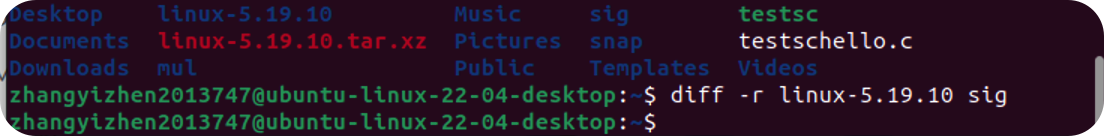
编译代码，用time计算时间，将linux-5.19.10拷贝到同目录的sig文件夹下



仅一个子进程，完成拷贝用时如下：



与原文件进行文件验证



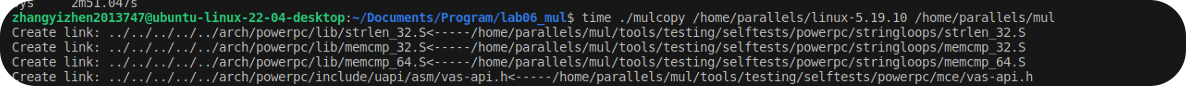
没有问题

6.2.2  **多进程复制**

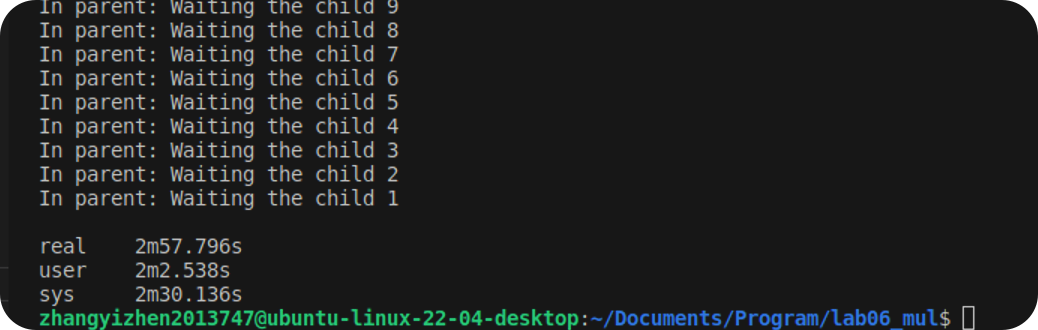
设置最大进程数为100，创建100个子进程进行文件夹拷贝



编译代码，用time计算时间，将linux-5.19.10拷贝到同目录的mul文件夹下

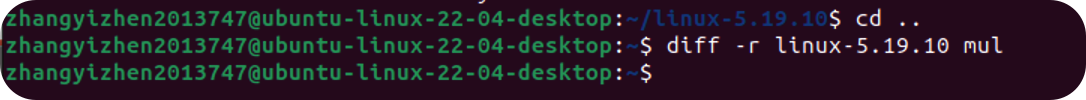


设定100个进程同时处理，得到时间结果如下：



显然，1个进程处理用时比多个进程处理用时更长，100个进程同时处理文件夹的复制可以提高文件夹的复制速度。

与原文件进行文件验证，没有问题



7. **总结对比**

|  |  |  |  |
| --- | --- | --- | --- |
| 复制方式 | 单进程 | 多进程 | 多线程 |
| 进程数/线程数 | 1 | 100 | 20 |
| Real time | 5m34.952s | 2m57.796s | 0m10.338s |
| User time | 1m49.810s | 2m2.538s | 0m1.508s |
| Sys time | 2m51.047s | 2m30.136s | 0m22.411s |

执行耗时：多线程>多进程>单线程

7.1 **Time 的类别分析**

7.1.1 **real time**

概念：程序从开始到结束所经历的时间，也就是用户所感受到的时间。包括当前程序CPU的用时和所有延迟程序执行的因素的耗时总和（比如其他程序耗时，等待I/O完成耗时等）。

来源：real time是由gettimeofday()中结束时间与开始时间相减得来。

7.1.2  **user time**

概念：程序执行过程中在用户空间（user space）中所花费的所有时间，即程序用户模式下的CPU耗时。

仅指当前进程。

其他进程的时间和当前进程I/O阻塞的时间均不计在内。

来源：user time是由wait()或times()系统调用得来。

7.1.3  **sys time**

概念：程序执行过程中内核空间（kernel space）中所花费的时间，即程序在内核调用中的CPU耗时。

仅指当前进程。

程序的库代码调用仍然是在用户空间下。

来源：sys time是由wait()或times()系统调用得来。

7.2 **线程与进程执行优缺点对比**

对比此次多线程拷贝和lab6的单进程，多进程拷贝的执行时间，我们可以看出在本实验任务中，多线程的执行速度明显快于多进程拷贝。

两者均是使用并行机制来提升系统的运行效率。区别在于运行时所占的内存分布不同，多线程是共用一套内存的代码块区间；而多进程是各用一套独立的内存区间。

7.2.1 **稳定性/独立性：多进程>多线程**

多进程的优点是稳定性好，一个子进程崩溃了，不会影响主进程以及其余进程。

而多线程编程的任何一个线程崩溃都可能造成整个进程的崩溃，因为它们共享了进程的内存资源池。

比如在本次实验的过程中，多线程拷贝的代码中将本应return的地方写成了exit，导致该线程退出却未回收，等待回收的程序却一直在等待该已退出的线程执行结束后回收，最终程序崩溃。但多进程拷贝时就不会遇到此情况。

基于这个特性，我们常常会使用多进程来实现守护服务器的功能。

7.2.2 **创建管理代价:多进程>多线程**

多进程编程也有不足，即创建进程的代价非常大，因为操作系统要给每个进程分配固定的资源，并且操作系统对进程的总数会有一定的限制，若进程过多，操作系统调度都会存在问题，会造成假死状态。且多进程在处理I/O密集型任务时（比如本实验中的拷贝文件夹），由于等待时间过长，会使得效率较低。

7.2.3 **批任务处理：多线程>多进程**

多线程编程的优点是效率较高一些，适用于批处理任务等功能。

例如本次拷贝文件夹的任务，多线程拷贝的表现远比多进程编程优秀。

但对于计算密集型的任务，多线程处理速度不如多进程。

7.3  **多线程与多进程适用的场景**

多线程编程和多进程编程各有优缺点，因此它们分别适用于不同的场景。

比如说，对于计算密集型的任务，多进程效率会更高一些。

而对于IO密集型的任务（比如文件操作，网络爬虫），采用多线程编程效率更高。

因为对于 IO 密集型操作，大部分消耗时间其实是等待时间，在等待时间中，会释放资源供新的线程使用，实现了线程间的切换；相反对于计算密集型代码，2个CPU干活肯定比一个CPU快很多，故多进程此时的效率又更高。

在大型的计算机集群系统中，通常都会将多进程程序分布运行在不同的计算机上协同工作，以确保系统的稳定性，而每一台计算机上的进程内部，又会由多个线程来并行工作。

8. **参考资料**

[老师的github实验文档](https://github.com/albertleecn/osplab)