# 《操作系统》课第06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: | 10/26/2022 |

## 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》

## 实验题目

多进程拷贝目录，并与单进程拷贝目录做效率比较

## 实验目标

1. Write a c/c++ program
2. To implement copy one directory and it's subdiretories with multi-processes
3. GCC
4. Test directory: 使用最新的Linux Kernel来测试(从[www.kernel.org](file:///Users/zhanghaha/Desktop/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/%E7%AC%AC6%E6%AC%A1/www.kernel.org)下载最新的linux内核)
   1. <https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.19.10.tar.xz>
   2. extract linux-5.19.10.tar.xz to linux-5.19.10 directory,
   3. and copy linux-5.19.10 directory to linux-5.19.10bak directory
5. Verify that the directory copy is correct

## 原理方法

### 3.1 比较目录是否相同

diff -r DirA DirB

### 3.2 get the total time of program execution

$ time pwd

### 3.3 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()

### 3.4 Create a symbol link file

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

#include <unistd.h>

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

### 3.5 create process and execute one program

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

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

**3.5.2 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[]);

### 3.6 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]);

}

}

## 代码分析

4.1 copyfile.cpp 作用是复制文件

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <iostream>

#include <unistd.h>

using namespace std;

const int BUF\_SIZE = 1024;

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

{

int infd, outfd;

char buffer[BUF\_SIZE];

int i;

if ((infd = open(argv[1], O\_RDONLY)) < 0)

{

exit(0);

}

if ((outfd = open(argv[2], O\_WRONLY | O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR)) < 0)

{

exit(1);

}

// cout << "Copyfile from " << argv[1] << " to " << argv[2] << endl;

while (1)

{

i = read(infd, buffer, BUF\_SIZE);

if (i <= 0)

break;

write(outfd, buffer, i);

}

close(outfd);

close(infd);

exit(0);

}

4.2 设计一个结构体，用来存放源地址与目的地址

struct Container

{

char \*srcFile;

char \*dstFile;

};

stack<Container> containerStack; //存放待复制的普通文件的源地址和目的地址的栈

stack<Container> linkStack; //存放软连接连接内容和连接地址的栈

* 1. 遍历目录，存放地址

//遍历目录

void walk\_dir(const char \*srcDir, const 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

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;

linkStack.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

if (ptr->d\_type == 10)

{

char \*linkFile = (char \*)malloc(sizeof(char) \* PATHLEN);

readlink(srcFile, linkFile, 1024);

new\_container.srcFile = linkFile;

linkStack.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);

}

* 1. 实现源地址与目的地址软连接

while (linkStack.empty()==false)

{

Container tmp=linkStack.top();

linkStack.pop();

cout<<"Create link: "<<tmp.srcFile<<"<-----"<<tmp.dstFile<<endl;

symlink(tmp.srcFile,tmp.dstFile);

}

* 1. 创建多线程

//创建多进程复制文件

pid\_t pid[PROCESS\_NUM];

int cnt = 0;

while (containerStack.empty() == false)

{

Container tmp = containerStack.top();

containerStack.pop();

//限制创建的进程总数不超过PROCESS\_NUM

while (cnt == PROCESS\_NUM)

{

wait(NULL);

cnt--;

}

pid[cnt] = fork();

if (pid[cnt] < 0)

{

fprintf(stderr, "InParent: Fork Failed\n");

printf("errno=%d\n", errno);

exit(-1);

}

else if (pid[cnt] == 0)

{

//子进程调用copyfile程序，完成普通文件复制

execlp("//home/parallels/Documents/Program/lab06\_mul/copyfile", "./copyfile", tmp.srcFile, tmp.dstFile, NULL);

}

cnt++;

}

//等待所有子进程完成

while (cnt != 0)

{

printf("In parent: Waiting the child %d\n", cnt);

wait(NULL);

cnt--;

}

* 1. 设置线程数量，控制单进程或者多进程

const int PATHLEN=1024;

const int PROCESS\_NUM=100; //设置100个子进程，完成多进程实验

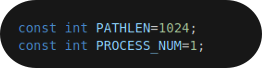
or

const int PROCESS\_NUM=1; //设置1个子进程，完成单进程实验

## 具体步骤

4.1 单进程复制

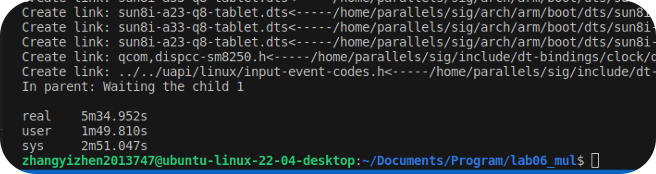
设置最大线程数为1，只创建一个子进程进行文件夹拷贝



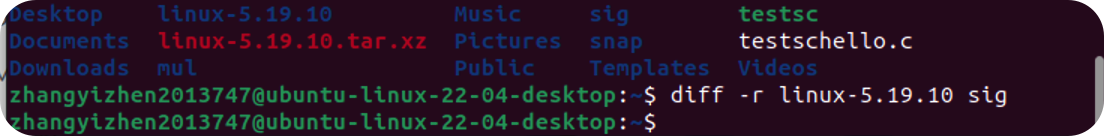
编译代码，用time计算时间，将linux-5.19.10拷贝到同目录的sig文件夹下



仅一个子进程，完成拷贝用时如下：



与原文件进行文件验证



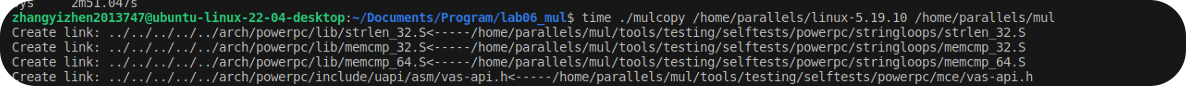
没有问题

4.2 多进程复制

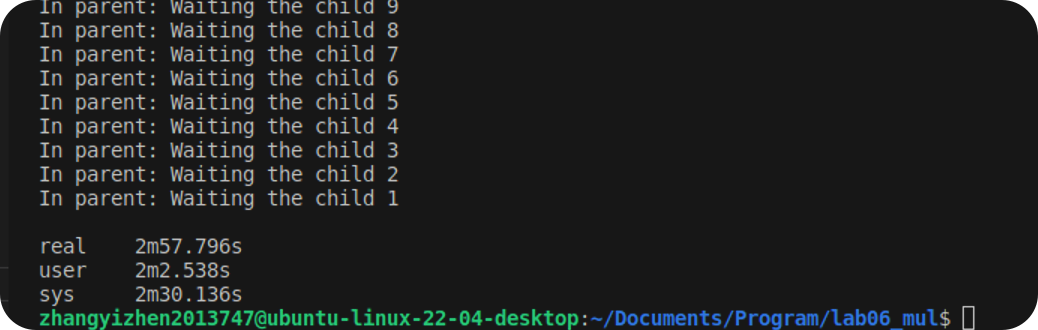
设置最大进程数为100，创建100个子进程进行文件夹拷贝



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

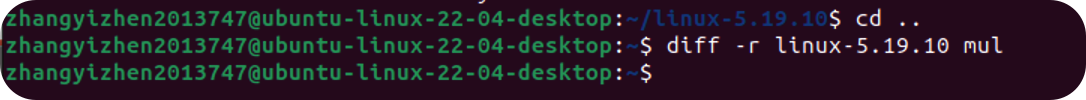


设定100个进程同时处理，得到时间结果如下：

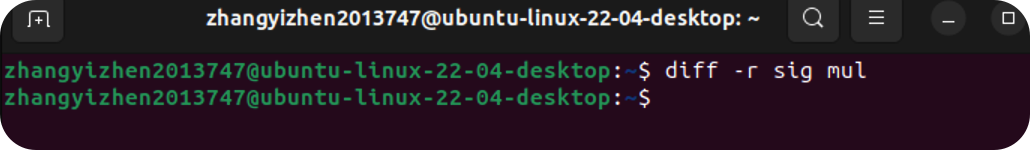


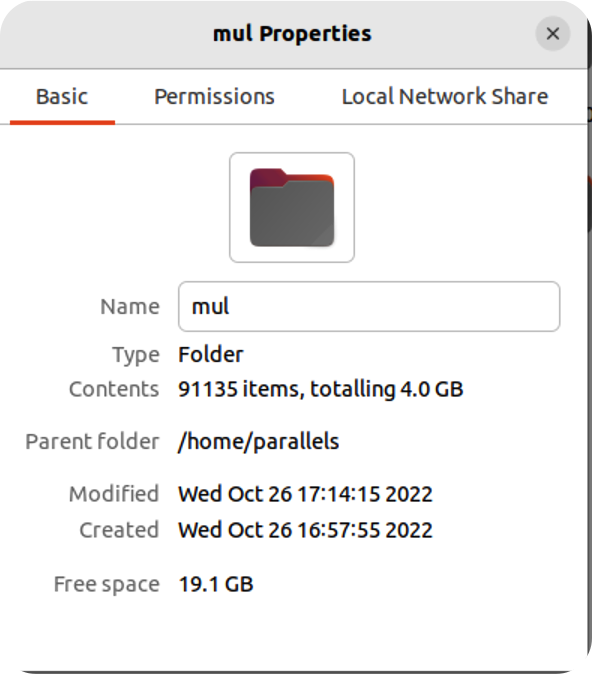
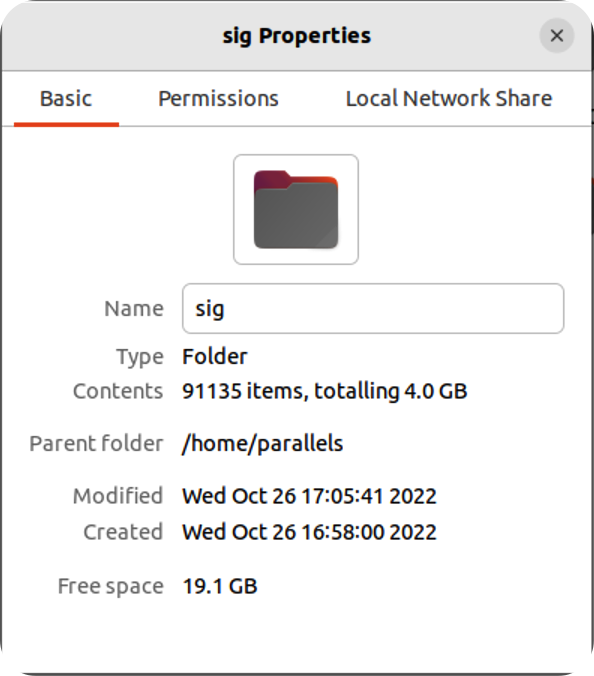
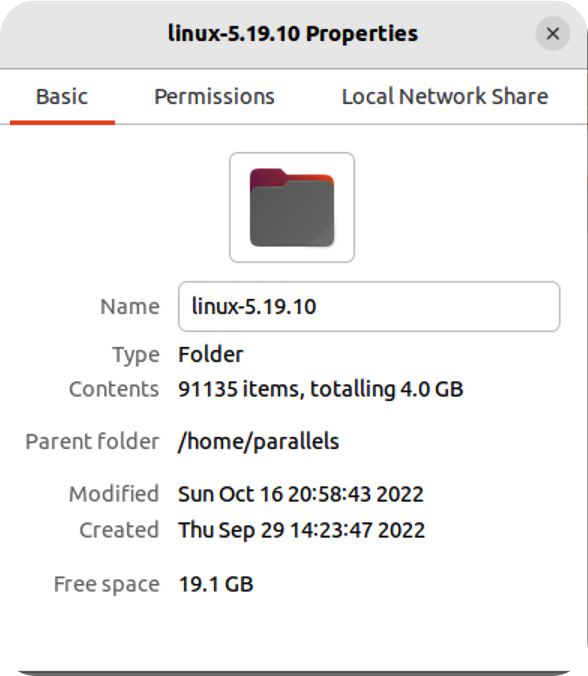
显然，1个进程处理用时比多个进程处理用时更长，100个进程同时处理文件夹的复制可以提高文件夹的复制速度。

与原文件进行文件验证，没有问题



* 1. Differ 进一步比对sig与mul文件夹



对比三个文件夹的属性，验证成功。

## 总结心得

1个进程与100个进程一起运行，由于物理设备的限制，以及进程创建分配的消耗时间并不能提高到100倍，而是根据自己的核数量来进行进程的分配运行，在我的例子中，效率提高约为时间减少一半多一些，提高了效率。同时在复制的过程中，要注意软连接文件的处理。

## 参考资料

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