《操作系统》课第七次实验报告

学院:	软件学院
姓名:	杨万里
学号:	2013774
邮箱:	2013774@mail.nankai.edu.cn
时间:	2022/10/30

0. 开篇感言

这两次实验让我感受到 debug 的痛苦与快乐。

在两次实验中，我建立思路、写完整体代码所用时间并不多，但是对于其中的奇怪的 bug 的发现与解决用了我大量的时间。

其实多线程和多进程编程难的不是并行本身，而是因为并行导致思路复杂之后，容易导致非并行的错误，并且定位 bug 的难度大大提高。

同时我还学习到，在调式代码的时候，最好不要直接用 Linux 内核目录调试，因为那样很难定位问题，可以先用简单的目录进行调试，达成目标之后，最后再用 Linux 内核测试程序的正确性。

本次实验因为指针和内存分配导致的 bug 困扰了我几个小时，在解决之后对于内存管理有了更多的理解，可谓因祸得福。

1. 实验题目

编写一个 C/C++程序实现使用多线程拷贝目录

2. 实验目标

（1） 编写 C/C++程序；

（2） 实现利用多线程拷贝一个目录及其所有子目录；

（3） 使用最新的 Linux 内核 linux-5.19.10 进行测试；

（4） 验证拷贝结果的正确性；

（5） 对比多线程和多进程各自的优缺点；

3. 原理方法以及源代码

项目完整源代码如下（具体原理方法和代码分析在此之后，助教老师可以直接看后面的分析）：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <semaphore.h>
#define  BUFSIZE 1024
#define maxNum 5000

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
sem_t sem_empty;
sem_t sem_full;

struct product{
    char* sourceFile;
    char* targetFile;
    int flag;  //标记是链接文件 0 还是普通文件 1
};


int thread_nums = 4;  //线程数目
char* sourcePath;
char* targetPath;
//一个可以存放需要拷贝文件的目录
//类似于生产者消费者问题中的产品存放区
//属于临界资源
```

```
33. struct product files_to_copy[maxNum];
34. int Index = 0;  //表示生成者生产的文件数目
35. int left_num = 0; //表示还未复制的文件数目
36. 
37. //遍历目录，记录总文件数
38. void file_counter(char* sourcePath){
39.    DIR* dir;
40.    struct dirent * ptr;
41.    char source[1000];
42.    char target[1000];
43.    // wrong path
44.    if((dir=opendir(sourcePath)) == NULL){
45.       perror("Open dir error");
46.       exit(1);
47.    }
48.    while((ptr=readdir(dir)) != NULL){
49.       if(strcmp(ptr->d_name,".")==0 || strcmp(ptr->d_name,"..")==0)
     ///current dir OR parrent dir
50.           continue;
51.        //char arr set 0
52.        memset(source,'\0',sizeof(source));
53.        strcpy(source,sourcePath);
54.        strcat(source,"/");
55.        strcat(source,ptr->d_name);
56.        if(ptr->d_type == DT_DIR)    //dir
57.        {
58.            file_counter(source);
59.        }
60.        else if(ptr->d_type == DT_LNK || ptr->d_type == DT_REG)    //
   link
61.        {
62.            left_num++;
63.        }
64.        else{
65.
66.        }
67.    }
68.    closedir(dir);
69. }
70. 
71. //拷贝文件
72. int copy_file(const char* src, const char* dest) {
73.     FILE* fp1=NULL, * fp2=NULL;
74.     fp1 = fopen(src, "r");
```

```
75.     if (fp1 == NULL) {
76.         return -1;
77.     }
78.     fp2 = fopen(dest, "w");
79.     if (fp2 == NULL) {
80.         return -2;
81.     }
82.     char buffer[BUFSIZE];
83.     int readlen, writelen;
84.     while ((readlen = fread(buffer, 1, BUFSIZE, fp1))>0)
85.     {
86.         writelen = fwrite(buffer, 1, readlen, fp2);
87.         if (readlen != writelen) {
88.             return -3;
89.         }
90.     }
91.     fclose(fp1);
92.     fclose(fp2);
93.     return 0;
94. }
95.
96. //生产者辅助函数
97. void produce_helper(char* sourcePath, char* targetPath){
98.    //char source[1000];
99.    //char target[1000];
100.   //错误之缘，地址阿！！！
101.      DIR* dir;
102.      struct dirent * ptr;
103.      if((dir=opendir(sourcePath)) == NULL){
104.         perror("Open dir error");
105.         exit(1);
106.      }
107.      while((ptr=readdir(dir)) != NULL){
108.         if(strcmp(ptr->d_name,".")==0 || strcmp(ptr->d_name,"..")==0
    )
109.             continue;
110.           //char arr set 0
111.           struct product pro;
112.           pro.sourceFile = malloc(sizeof(char) * 1000);
113.           pro.targetFile = malloc(sizeof(char) * 1000);
114.           memset(pro.sourceFile,'\0',sizeof(char)*1000);
115.           memset(pro.targetFile,'\0',sizeof(char)*1000);
116.           strcpy(pro.targetFile,targetPath);
117.           strcpy(pro.sourceFile,sourcePath);
```

```
118.            strcat(pro.targetFile,"/");
119.            strcat(pro.sourceFile,"/");
120.            strcat(pro.targetFile,ptr->d_name);
121.            strcat(pro.sourceFile,ptr->d_name);
122.            if(ptr->d_type == DT_DIR)    //dir
123.            {
124.                mkdir(pro.targetFile,S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S
    _IXOTH);
125.                produce_helper(pro.sourceFile, pro.targetFile);
126.            }
127.            else if(ptr->d_type == DT_LNK)    //link
128.            {
129.                sem_wait(&sem_empty);
130.
131.                pthread_mutex_lock(&mutex);
132.
133.                pro.flag = 0;
134.                files_to_copy[Index] = pro;
135.                Index++;
136.
137.               pthread_mutex_unlock(&mutex);
138.                sem_post(&sem_full);
139.            }
140.            else if(ptr->d_type == DT_REG){
141.                sem_wait(&sem_empty);
142.
143.                pthread_mutex_lock(&mutex);
144.
145.                pro.flag = 1;
146.                files_to_copy[Index] = pro;
147.                Index++;
148.
149.               pthread_mutex_unlock(&mutex);
150.                sem_post(&sem_full);
151.
152.
153.            }
154.            else{
155.
156.            }
157.        }
158.        closedir(dir);
159.    }
160.
```

```
161.  //生产者线程函数
162.  void* thread_produce(){
163.     printf("我是生产者！\n");
164.     produce_helper(sourcePath, targetPath);
165.     printf("生产者 finish！\n");
166.  }
167.
168.  //消费者线程函数
169.  void* thread_consumer(){
170.     printf("我是消费者！\n");
171.     char* source;
172.     char* target;
173.     while(1){
174.        pthread_mutex_lock(&mutex2);
175.        if(left_num <= 0){
176.           //结束了
177.           pthread_mutex_unlock(&mutex2);
178.           break;
179.        }
180.        else{
181.           sem_wait(&sem_full);
182.           pthread_mutex_lock(&mutex);
183.
184.           Index--;
185.           struct product mypro;
186.           mypro = files_to_copy[Index];
187.
188.           left_num--;
189.           pthread_mutex_unlock(&mutex2);
190.           pthread_mutex_unlock(&mutex);
191.           sem_post(&sem_empty);
192.           if(mypro.flag==1){
193.              copy_file(mypro.sourceFile, mypro.targetFile);
194.           }else{
195.              link(mypro.sourceFile,mypro.targetFile);
196.           }
197.           free(mypro.sourceFile);
198.           free(mypro.targetFile);
199.        }
200.     }
201.     printf("finish!\n");
202.  }
203.
204.  int main(int argc, char *argv[]){
```

```
205.      //接收源目录和目标目录
206.      sourcePath = argv[1];
207.      targetPath = argv[2];
208.
209.      //sem 初始化
210.      sem_init(&sem_empty, 0, maxNum);
211.      sem_init(&sem_full, 0, 0);
212.
213.      //mutex 初始化
214.      pthread_mutex_init(&mutex, 0);
215.      pthread_mutex_init(&mutex2, 0);
216.
217.      //统计文件数目
218.      file_counter(sourcePath);
219.      printf("文件总数: %d", left_num);
220.
221.      //线程初始化,1 个生产者线程，4 个消费者线程
222.      pthread_t threads[5];
223.      pthread_create(&threads[0], NULL, thread_produce, 0);
224.
225.      for(int i=1;i<=4;i++){
226.        pthread_create(&threads[i], NULL, thread_consumer, 0);
227.      }
228.
229.      //barrier
230.      for(int i=0;i<5;i++){
231.        pthread_join(threads[i], 0);
232.      }
233.
234.
235.      pthread_mutex_destroy(&mutex);
236.      pthread_mutex_destroy(&mutex2);
237.
238.      sem_destroy(&sem_full);
239.      sem_destroy(&sem_empty);
240.
241.      left_num = 0;
242.      file_counter(targetPath);
243.      printf("%d", left_num);
244.
245.      return 0;
246.  }
```

具体原理方法分析：

（1）单进程拷贝目录的原理方法在此前的实验中已经介绍：获取指向源目录的指针，逐个遍历其下的文件/子目录，判断其类型，如果是目录则递归调用该拷贝函数、如果是普通文件则正常拷贝（实现一个拷贝普通文件的函数）、如果是链接文件则采用 link 函数进行拷贝，源代码此前实验报告已经展示。

（2）多进程拷贝目录的原理方法在上一个实验介绍：先在主进程当中遍历源目录 linux-5.19.10，直接拷贝该级（不是所有）的普通文件和链接文件，对于一级子目录，只需要记录它们的路径。启动四个子进程（可以设定数目），每个子进程负责拷贝一定数量的（四等分）一级子目录及其下所有子目录和文件，从而实现多进程拷贝。

（3）多线程拷贝目录采用**生产者消费者模式**的基本思想：

1）设定一个线程作为生产者，四个线程作为消费者

2）设置一个结构体表示产品，其包含要拷贝的文件的源路径和目标路径，以及记录文件类型（链接文件还是普通文件）

```
struct product{
    char* sourceFile;
    char* targetFile;
    int flag;   //标记是链接文件0还是普通文件1
};
```

3）临界区资源包括：产品数组、产品索引、还没有拷贝的文件数目（用于消费者线程退出判断）

```
//一个可以存放需要拷贝文件的目录
//类似于生产者消费者问题中的产品存放区
//属于临界资源
struct product files_to_copy[maxNum];
int Index = 0;  //表示生成者生产的文件数目
int left_num = 0; //表示还未复制的文件数目
```

4）需要两个信号量 sem 和两个互斥锁 mutex

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
sem_t sem_empty;
sem_t sem_full;
```

5）遍历一边源目录，记录下需要拷贝的总文件数，记录为 left_num

6）启动生产者和消费者线程，对应两个线程函数 thread_produce 和 thread_consumer

```
//线程初始化,1个生产者线程，4个消费者线程
pthread_t threads[5];
pthread_create(&threads[0], NULL, thread_produce, 0);

for(int i=1;i<=4;i++){
  pthread_create(&threads[i], NULL, thread_consumer, 0);
}
```

7）对于生产者线程函数，其主要功能是获取要拷贝的文件名，将其放入产品数组。遍历源目录 linux-5.19.10 时，遇到子目录就递归，遇到文件则说明手中有一个产品（要拷贝的文件）。

先试图减少信号量 sem_empty，表示剩余空间，如果没有剩余空间生产者线程就会阻塞，如果有剩余空间就会对临界资源（产品数组、产品索引）上锁，接着把产品（待拷贝文件的源路径、目标路径和文件类型）放入产品数组，最后释放锁并且将信号量 sem_full 加 1，表示产品线上产品多了一个。

当源目录遍历完成，生产者线程自动结束。

```
else if(ptr->d_type == DT_LNK)    //link
{
    sem_wait(&sem_empty);

    pthread_mutex_lock(&mutex);

    pro.flag = 0;
    files_to_copy[Index] = pro;
    Index++;

   pthread_mutex_unlock(&mutex);
    sem_post(&sem_full);
}
else if(ptr->d_type == DT_REG){
    sem_wait(&sem_empty);

    pthread_mutex_lock(&mutex);

    pro.flag = 1;
    files_to_copy[Index] = pro;
    Index++;

   pthread_mutex_unlock(&mutex);
    sem_post(&sem_full);
}
```

8）对于消费者线程函数，其主要功能是从产品数组中取下产品，利用产品中的信息执行拷贝文件的操作。

消费者线程间共享一个资源 left_num，一旦 left_num==0，表示没有文件需要拷贝，消费者线程就会退出。（需要单独的锁 mutex2）

如果还有文件需要拷贝，消费者线程会先试图将 sem_full 信号量减 1，如果失败表示目前还没有产品，消费者线程阻塞；否则消费者线程会对临界资源加锁，从而取下产品，再解锁。最后开始自己的拷贝文件操作。

（见下一页图）

```
sem_wait(&sem_full);
pthread_mutex_lock(&mutex);

Index--;
struct product mypro;
mypro = files_to_copy[Index];
left_num--;
pthread_mutex_unlock(&mutex2);
pthread_mutex_unlock(&mutex);
sem_post(&sem_empty);
if(mypro.flag==1){
   copy_file(mypro.sourceFile, mypro.targetFile);
}else{
   link(mypro.sourceFile,mypro.targetFile);
}
free(mypro.sourceFile);
free(mypro.targetFile);
```

9）设置 barrier，等待子线程结束，最后销毁锁和信号量

```
//barrier
for(int i=0;i<5;i++){
  pthread_join(threads[i], 0);
}


pthread_mutex_destroy(&mutex);
pthread_mutex_destroy(&mutex2);

sem_destroy(&sem_full);
sem_destroy(&sem_empty);
```

4. 具体步骤

（1）按照上述原理方法编写代码（mulThread.c 文件）

（2）编译生成可执行文件 multhr

```
wanliyang2013774@wanliyang2013774-virtual-machine:~$ gcc mulThread.c -o multhr -lpthread
```

（3）命令行输入源路径和目标路径，执行程序并记录时间

```
wanliyang2013774@wanliyang2013774-virtual-machine:~$ time ./multhr ./linux-5.19.10 ./LinuxMulThr
文件总数: 97535我是生产者！
我是消费者！
我是消费者！
我是消费者！
我是消费者！
生产者finish！
finish!
finish!
finish!
finish!
97535
real    1m26.568s
user    0m5.976s
sys     2m27.545s
```

可以看到，遍历记录源目录待拷贝文件数目为 97535；接着启动生产者线程和消费者线程，当所有子线程结束时，遍历目标目录，发现其下也有 97535 个文件，大致可以判断出应该拷贝成功了

（4）使用 diff -r 命令验证，发现拷贝结果完全正确。实验成功！

```
real    1m26.568s
user    0m5.976s
sys     2m27.545s
wanliyang2013774@wanliyang2013774-virtual-machine:~$ diff -r ./linux-5.19.10 ./LinuxMulThr
wanliyang2013774@wanliyang2013774-virtual-machine:~$
```

（5）单进程、多进程、多线程效率对比：

（均采用时间指标中的 real 进行比较）

在上一次实验中测量了单进程与多进程拷贝目录的时间

单进程拷贝目录用时：3m10s，也就是 190s。

多进程（四进程）拷贝目录用时：2m9s，也就是 129s，加速比达到 1.47。

多线程（四线程）拷贝目录用时：1m26s，也就是 86s，加速比达到 2.21。

关于加速比与进程/线程数目（4）差距较大的分析：

由于启动、管理、维护多进程/线程的用时、为多进程/线程的前期准备工作、线程间通信（临界资源访问）等原因，会比单进程有额外的时间开销，因此加速比距离理想的 4 倍有差距，这是合理的。

关于四进程与四线程拷贝目录用时差距的分析：

首先进程与线程的时间开销并不一致是正常的，但是在本实验中似乎差距比较大，这主要是因为多进程和多线程采用的方法不一致。在**多进程**实验中，我采用的思路是**提前规划**，每个进程负责几个 linux 内核的一级子目录的拷贝，这样完全**避免了进程间通信**的问题，访问**资源彼此独立**，不会有冲突，但是这样做的代价就是**负载不均衡**，有可能某个进程负责的几个子目录内容比另一个进程多很多。负载均衡会导致加速比不理想。

而多线程采用生产者消费者思想，消费者之间根据实时的情况获取资源进行拷贝，因此需要更多的**线程通信**，但是这样**负载基本均衡**，每个消费者线程的工作量大致相同。同时加上生产者线程的话，可以算是五个线程在工作。在多进程当中生产者线程的工作由主进程在准备阶段完成。

综上所述，多线程和多进程采用的方法各有利弊，再加上线程和进程本身也不相同，因此时间上有一定差距是正常的。

就实验结果而言，对于一个程序，也许启用多线程会比多进程更简洁、快速一些。

5. 总结心得

（1）通过本次实验，亲自创建、维护线程，进行线程间的通信，让我对线程的概念有了更深的理解，线程作为更小的 CPU 执行单位，相比进程更容易维护和使用。

（2）由于本次实验采用了生产者消费者的思路，遇到了很多次线程间通信的 bug，在多次 debug 最终完成实验之后，让我对线程通信有了更深的理解。

（3）本次实验加深了我对信号量和互斥锁的理解，特别是感受到 sem_empty 和 sem_full 和 mutex 的先后顺序的重要性。

（4）通过改进实验方法，采用生产者消费者思路，使得并行程序负载更加均衡，时间也减少了不少，让我感受到优秀的算法的巨大优势。

6. 参考资料

实验指导书《CopyDirWithMultiThreads_README.pdf》