《操作系统》课第七次实验报告

学院:	软件学院
姓名:	郁万祥
学号:	2013852
邮箱:	yuwanxiang0114@163.com
时间:	2022.10.28

0. 开篇感言

这一次实验遇见了几个问题，有的是关于此次具体目标的，有的则是关于虚拟机本身的。

1、在完成，将拷贝目录放入到一个字符串数组的时候，因为收到 C++的影响，再加上待拷贝文件路径是不确定的，就想着动态分配数组，但是这最后并没有很好实现，最后莫不如直接分配一个较大的内存，就像使用一个充足的流水线一样，用来保存待拷贝文件的路径。

2、这是关于虚拟机本身的，由于之前几次拷贝文件都是以较大的 linux 内核作为例子，因此本虚拟机的内存已经消耗的差不多了，导致了一次 reboot 时候，虚拟机出现了开机报错的情况，因此需要进入 recover 模式下的 root，进入命令行后，使用命令清除一些文件，达到正常的 reboot。

3、其次是对于生产者和消费者个数的问题，本次实验 pdf 指导说明一个线程负责创建路径，也就是生产者负责将一个路径放入到缓冲区当中，五个线程负责文件拷贝，也就是五个消费者负责从缓冲区中获取目录，并执行拷贝文

件操作，这其实和本身生产者和消费者所执行功能的复杂程度有关，如果生产者数量过多，而其任务又非常简单，那就会出现缓冲区持续充满的状态，这样会经常性的使生产者线程停止运行，反之，如果消费者数量过多，其任务又非常简单，就会出现缓冲区持续保持空的状态，所以需要根据实际情况，合理定义生产者和消费者的数目。

1. 实验题目

多线程拷贝目录

2. 实验目标

1、编写 C 程序实现目录拷贝。

2、编写 C 程序实现多进程运行拷贝程序。

3、验证拷贝的正确性。

4、比较多线程拷贝，多进程拷贝和单进程拷贝的效率。

5、参考生产者消费者问题。

3. 原理方法

1、目录拷贝：实现过程基于实验二中的拷贝目录的 C 程序，不过此次实验，为了实现后续在多进程运行是对 execlp 函数的使用，我们需要手动输入并传递 main 函数的参数, 因此将之前无参的 main 函数修改为 main(int argc , int *argv[]) 的形式。

2、多线程：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个

进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

3、生产者消费者问题：在同一个进程地址空间内执行两个线程。生产者线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。当生产者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放一个空缓冲区。当消费者线程消费物品时，如果没有满的缓冲区，那么消费者线程将被阻挡，直到新的物品被生产出来。

该问题需要注意的几点：

1）、在缓冲区为空时，消费者不能再进行消费

2）、在缓冲区为满时，生产者不能再进行生产

3）、在一个线程进行生产或消费时，其余线程不能再进行生产或消费等操作，即保持线程间的同步

4）、注意条件变量与互斥锁的顺序

4. 具体步骤

1、编写拷贝目录和多线程并发的 C 语言程序（源码见附录）。

multi_threads.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define PRODUCER_NUM 1 //生产者数目
#define CONSUMER_NUM 5 //消费者数目
#define POOL_SIZE       11  //缓冲池大小
char* pool[POOL_SIZE];  //缓冲区
int head=0;     //缓冲池读取指针
int rear=0; //缓冲池写入指针
sem_t   room_sem;
sem_t   product_sem;
pthread_mutex_t mutex;
int now; // 目前的路径指针
char * alldir[1000];  // 所有的文件目录
int i=0;   // 创建alldir所使用的指针
char* basePath='/home/ywx2013852/linux-5.19.10';   // 原路径
char* targetPath='/home/ywx2013852/Documents/linux-5.19.10bak1';   // 目标路径
void creat_dir(char *basePath)  // 获取所有拷贝文件的路径，放入字符数组当中
{
        DIR *dir;
        struct dirent *ptr;
        char base[1000];
        if ((dir=opendir(basePath)) == NULL)
        {
                perror("Open dir error...");
                exit(1);
```

2、编译 C 文件。

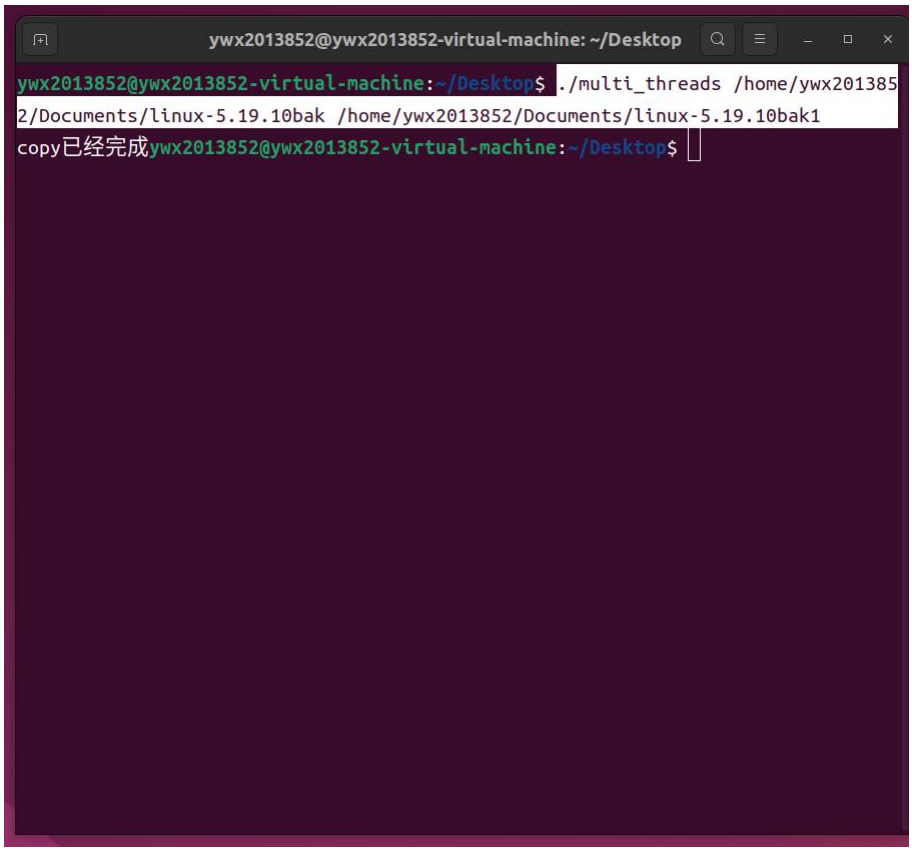
gcc -o multi_threads multi_threads.c

```
ywx2013852@ywx2013852-virtual-machine:~/Desktop$ gcc -o multi_threads multi_thre
ads.c
multi_threads.c:24:16: warning: character constant too long for its type
   24 | char* basePath='/home/ywx2013852/linux-5.19.10';   // 原路径
      |                ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
multi_threads.c:24:16: warning: initialization of ‘char *’ from ‘int’ makes poin
ter from integer without a cast [-Wint-conversion]
multi_threads.c:25:18: warning: character constant too long for its type
   25 | char* targetPath='/home/ywx2013852/Documents/linux-5.19.10bak1';   // 目
标路径
      |                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
multi_threads.c:25:18: warning: initialization of ‘char *’ from ‘int’ makes poin
ter from integer without a cast [-Wint-conversion]
multi_threads.c: In function ‘main’:
multi_threads.c:111:74: warning: cast to pointer from integer of different size
[-Wint-to-pointer-cast]
  111 |     ret =pthread_create(&producer_id[i], NULL, producer_fun, (void*)i);
      |                                                                          ^

multi_threads.c:111:60: warning: passing argument 3 of ‘pthread_create’ from inc
ompatible pointer type [-Wincompatible-pointer-types]
  111 |                ret =pthread_create(&producer_id[i], NULL, producer_fun,
 (void*)i);
      |                                                            ^~~~~~~~~~~~
```

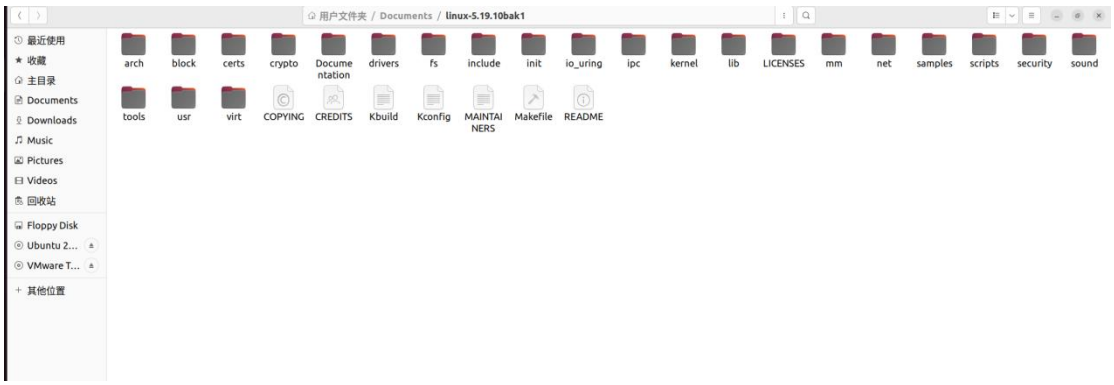
3、运行 C 文件。

./multi_threads /home/ywx2013852/Documents/linux-5.19.10bak /home/ywx2013852/Documents/linux-5.19.10bak1（第一个参数是源文件的地址，第二个参数是目的地址）



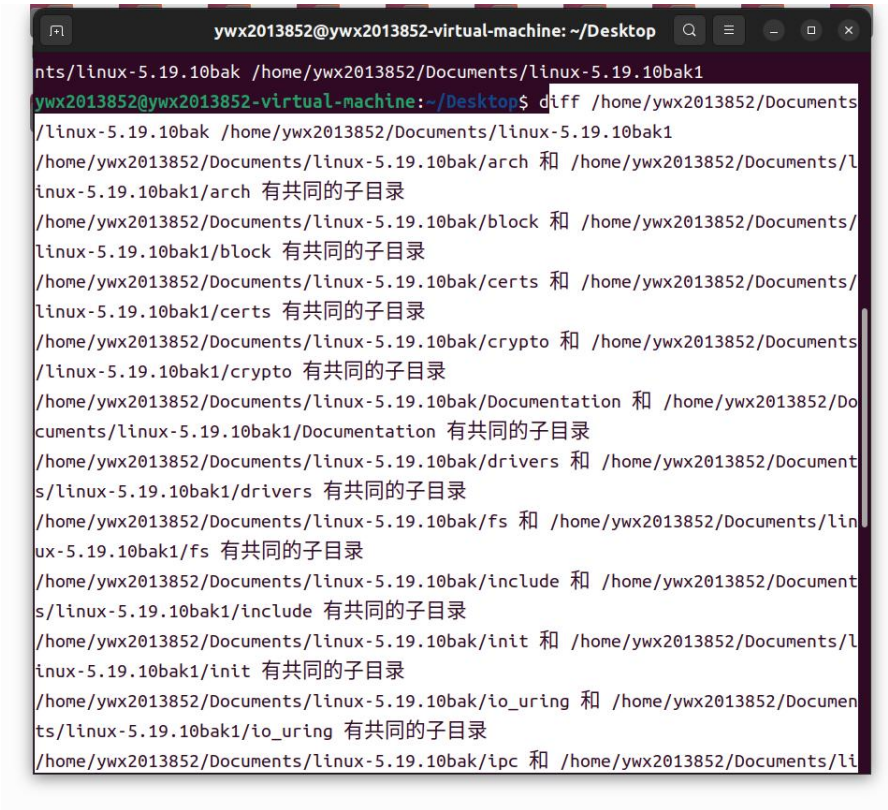
4、检验拷贝正确性。

拷贝情况:



diff 比较:

diff /home/ywx2013852/Documents/linux-5.19.10bak /home/ywx2013852/Documents/linux-5.19.10bak1



5、比较单进程拷贝、多进程拷贝和多线程拷贝的效率(以程序运行时间为衡量标准):

单进程拷贝:

time ./listdir /home/ywx2013852/Documents/linux-5.19.10bak /home/ywx2013852/Documents/linux-5.19.10bak1

多进程拷贝:

time ./multi_processed /home/ywx2013852/Documents/linux-5.19.10bak /home/ywx2013852/Documents/linux-5.19.10bak1

多线程拷贝:

time ./multi_threads /home/ywx2013852/Documents/linux-5.19.10bak /home/ywx2013852/Documents/linux-5.19.10bak1

5. 总结心得

生产者消费者问题对于理解多线程工作原理非常有效，这种模式有许多优点：

1、解耦

假设生产者和消费者分别是两个类。如果让生产者直接调用消费者的某个方法，那 么生产者对于消费者就会产生依赖（也就是耦合）。将来如果消费者的代码发生变化， 可能会影响到生产者。而如果两者都依赖于某个缓冲区，两者之间不直接依赖，耦合也 就相应降低了。

2、支持并发

由于生产者与消费者是两个独立的并发体，他们之间是用缓冲区作为桥梁连接，生产者只需要往缓冲区里丢数据，就可以继续生产下一个数据，而消费者只需要从缓冲区了拿数据即可，这样就不会因为彼此的处理速度而发生阻塞。

3、支持忙闲不均

缓冲区还有另一个好处。如果制造数据的速度时快时慢，缓冲区的好处就体现出来 了。当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。等生产者的制造速度慢下来，消费者再慢慢处理掉。

其次，我们可以看到，此次实验中，多线程比多进程的效率更高一些，那么是否所有问题都是如此呢，什么情况下适用多线程，什么情况下又适用多进程呢？

其实，计算 io 操作密集型的代码多线程效率更高,因为线程创建要比进程创建开销少。

但是计算密集型的代码多 那么进程操作更快,因为多进可以应用多核技术。

6. 参考资料

源码：

multi_threads.c：

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define PRODUCER_NUM 1 //生产者数目
#define CONSUMER_NUM 5 //消费者数目
#define POOL_SIZE       11   //缓冲池大小
char* pool[POOL_SIZE];      //缓冲区
int head=0;   //缓冲池读取指针
int rear=0; //缓冲池写入指针
sem_t     room_sem;
sem_t     product_sem;
pthread_mutex_t mutex;
int now; //  目前的路径指针
char * alldir[1000];   //  所有的文件目录
int i=0;     //  创建 alldir 所使用的指针
char* basePath='/home/ywx2013852/linux-5.19.10';     //  原路径
char* targetPath='/home/ywx2013852/Documents/linux-5.19.10bak1';     //  目标路径
void creat_dir(char *basePath)   //  获取所有拷贝文件的路径，放入字符数组当中
{
      DIR *dir;
            struct dirent *ptr;
            char base[1000];
            if ((dir=opendir(basePath)) == NULL)
            {
                  perror("Open dir error...");
                  exit(1);
```

```
        }

        while ((ptr=readdir(dir)) != NULL)
        {
            if(strcmp(ptr->d_name,".")==0 || strcmp(ptr->d_name,"..")==0)    ///current dir OR parrent dir
                continue;
            memset(base,'\0',sizeof(base));
            strcpy(base,basePath);
            strcat(base,"/");
            strcat(base,ptr->d_name);
            alldir[i]=base;
            i++;
            if(ptr->d_type == 4)    ///dir
            {
                creat_dir(base);
            }
        }
        closedir(dir);
        return;
}
void producer_fun(void *arg)
{
    while (1)
    {
        sleep(1);
        sem_wait(&room_sem);
        pthread_mutex_lock(&mutex);
        //生产者往缓冲池中写入  一个文件目录
        pool[rear] = alldir[now];
        now++;
        rear = (rear + 1) % POOL_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&product_sem);
    }
}


void consumer_fun(void *arg)
{
    while (1)
    {
        char* data;
        sleep(10);
```

```
            sem_wait(&product_sem);
            pthread_mutex_lock(&mutex);
            //消费者从缓冲池读取数据
            data = pool[head];
            head = (head + 1) % POOL_SIZE;
            // 进行拷贝        data:目前拷贝的源路径
            link(data,targetPath);
            pthread_mutex_unlock(&mutex);
            sem_post(&room_sem);
        }
}

int main()
{
        creat_dir(basePath);
        pthread_t producer_id[PRODUCER_NUM];
        pthread_t consumer_id[CONSUMER_NUM];
        pthread_mutex_init(&mutex, NULL);  //初始化互斥量
        int ret = sem_init(&room_sem, 0, POOL_SIZE-1);    //初始化信号量 room_sem 为缓冲池
大小
        if (ret != 0)
        {
            printf("sem_init error");
            exit(0);
        }
        ret = sem_init(&product_sem, 0, 0);  //初始化信号量 product_sem 为 0，开始时缓冲池
中没有数据
        if (ret != 0)
        {
            printf("sem_init error");
            exit(0);
        }
        for (int i = 0; i < PRODUCER_NUM; i++)
        {
            //创建生产者线程
            ret =pthread_create(&producer_id[i], NULL, producer_fun, (void*)i);
            if (ret != 0)
            {
                printf("producer_id error");
                exit(0);
            }
            //创建消费者线程
            ret = pthread_create(&consumer_id[i], NULL, consumer_fun, (void*)i);
            if (ret != 0)
```

```
            {
                printf("consumer_id error");
                exit(0);
            }
        }
        for(int i=0;i<PRODUCER_NUM;i++)
        {
            pthread_join(producer_id[i],NULL);
            pthread_join(consumer_id[i],NULL);
        }

        exit(0);
}
```