《操作系统原理》实验报告

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 | MTX | 学号 | U2021 | 专业班级 | 信安 | 时间 | 2023.11.28 |

**一、实验目的**

（1）理解进程/线程的概念和应用编程过程；（2）理解进程/线程的同步机制和应用编程；

（3）掌握和推广国产操作系统（推荐银河麒麟或优麒麟，建议）

**二、实验内容**

（1）在Linux/Windows下创建2个线程A和B，循环输出数据或字符串。

（2）在Liunx下创建（fork）一个子进程，实验wait/exit函数

（3）在Windows/Linux下，利用线程实现并发画圆画方。

（4）在Windows或Linux下利用线程实现“生产者-消费者”同步控制

（5）在Linux下利用信号机制(signal)实现进程通信

（6）在Windows或Linux下模拟哲学家就餐，提供死锁和非死锁解法。

（7）研读Linux内核并用printk调试进程创建和调度策略的相关信息。

**三、实验环境和核心代码**

**3.1 两线程循环输出数据和字符串**

Linux下创建2个线程A和B，循环输出数据或字符串。

实验环境：Ubuntu 20.04.6 LTS，GCC 9.4.0

核心代码：

#include <iostream>#include <pthread.h>#include <unistd.h>pthread\_mutex\_t mutex;int countA = 1;int countB = 100;void\* threadFunctionA(void\*) { while (countA <= 100) { pthread\_mutex\_lock(&mutex); std::cout << "Thread A: " << countA << std::endl; countA++; pthread\_mutex\_unlock(&mutex); usleep(200000); // 延迟 200 毫秒 } pthread\_exit(NULL);}void\* threadFunctionB(void\*) { while (countB >= 1) { pthread\_mutex\_lock(&mutex); std::cout << "Thread B: " << countB << std::endl; countB--; pthread\_mutex\_unlock(&mutex); usleep(200000); // 延迟 200 毫秒 } pthread\_exit(NULL);}int main() { pthread\_t threadA, threadB; pthread\_mutex\_init(&mutex, NULL); pthread\_create(&threadA, NULL, threadFunctionA, NULL); pthread\_create(&threadB, NULL, threadFunctionB, NULL); pthread\_join(threadA, NULL); pthread\_join(threadB, NULL); pthread\_mutex\_destroy(&mutex); return 0;}

编译命令：g++ -o lab2-task1 lab2-task1.cpp -lpthread

**3.2子进程建立并实现wait和exit函数**

在 Linux下，可以使用fork函数创建一个子进程。子进程是父进程的副本，它会从fork函数返回的地方开始执行。

实验环境：Ubuntu 20.04.6 LTS，GCC 9.4.0

核心代码：

#include <stdio.h>#include <stdlib.h>#include <sys/types.h>#include <sys/wait.h>#include <unistd.h>int main(int argc, char \*argv[]) { if (argc != 2) { fprintf(stderr, "Usage: %s <effect\_number>\n", argv[0]); return 1; } int effect = atoi(argv[1]); pid\_t pid = fork(); if (pid < 0) { fprintf(stderr, "Fork failed\n"); return 1; } else if (pid == 0) { // 子进程 printf("子进程：进程ID：%d，父进程ID：%d\n", getpid(), getppid()); if (effect == 1) { // 效果1：子进程进入死循环 while (1) { } } else if (effect == 2) { // 效果2：子进程休眠5秒后使用exit函数返回参数 sleep(5); exit(42); } else { fprintf(stderr, "Invalid effect number\n"); return 1; } } else { // 父进程 if (effect == 1) { // 效果1：父进程不用wait函数 printf("父进程：进程ID：%d，子进程ID：%d\n", getpid(), pid); } else if (effect == 2) { // 效果2：父进程用wait函数 int status; wait(&status); // 等待子进程退出 if (WIFEXITED(status)) { printf("父进程：进程ID：%d，子进程ID：%d，子进程返回：%d\n", getpid(), pid, WEXITSTATUS(status)); } } else { fprintf(stderr, "Invalid effect number\n"); return 1; } } return 0;}

编译命令：g++ -o lab2-task2 lab2-task2.cpp

**3.4 Linux实现“生产者-消费者”同步控制**

生产者函数producer和消费者函数consumer分别在不同的线程中运行。生产者函数随机生成一定范围内的数值，并将其放入缓冲区中，同时更新索引和计数器。消费者函数从缓冲区中取出数据，并进行相应的处理。在生产者和消费者之间使用了互斥锁pthread\_mutex\_t和信号量sem\_t来实现线程间的同步和互斥操作。主函数main中初始化了互斥锁和信号量，并创建了多个生产者线程和消费者线程。然后使用pthread\_join函数等待线程的结束，并在最后销毁互斥锁和信号量。

实验环境：Ubuntu 20.04.6 LTS，GCC 9.4.0

核心代码：

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <unistd.h>

#include <semaphore.h>

const int BUFFER\_SIZE = 10;

int buffer[BUFFER\_SIZE];

int itemCount = 0;

int inIndex = 0;

int outIndex = 0;

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

sem\_t emptySlots;

sem\_t fullSlots;

void\* producer(void\* arg)

{

int id = \*(static\_cast<int\*>(arg));

int start = id == 1 ? 1000 : 2000;

int end = id == 1 ? 1999 : 2999;

for (int i = start; i <= end; i++)

{

usleep((rand() % 901 + 100) \* 1000); // 随机间隔100ms-1s

sem\_wait(&emptySlots);

pthread\_mutex\_lock(&mtx);

buffer[inIndex] = i;

std::cout << "Producer " << id << " produced: " << buffer[inIndex] <<" at "<<inIndex<< std::endl;

inIndex = (inIndex + 1) % BUFFER\_SIZE;

itemCount++;

pthread\_mutex\_unlock(&mtx);

sem\_post(&fullSlots);

}

return nullptr;

}

void\* consumer(void\* arg)

{

int id = \*(static\_cast<int\*>(arg));

while (true)

{

usleep((rand() % 901 + 100) \* 1000); // 随机间隔100ms-1s

sem\_wait(&fullSlots);

pthread\_mutex\_lock(&mtx);

int data = buffer[outIndex];

std::cout << "Consumer " << id << " consumed: " << data <<" at "<<outIndex<< std::endl;

outIndex = (outIndex + 1) % BUFFER\_SIZE;

itemCount--;

pthread\_mutex\_unlock(&mtx);

sem\_post(&emptySlots);

}

return nullptr;

}

int main()

{

srand(static\_cast<unsigned int>(time(nullptr)));

sem\_init(&emptySlots, 0, BUFFER\_SIZE);

sem\_init(&fullSlots, 0, 0);

int producerId1 = 1;

int producerId2 = 2;

int consumerId1 = 1;

int consumerId2 = 2;

int consumerId3 = 3;

pthread\_t producerThread1, producerThread2, consumerThread1, consumerThread2, consumerThread3;

pthread\_create(&producerThread1, nullptr, producer, &producerId1);

pthread\_create(&producerThread2, nullptr, producer, &producerId2);

pthread\_create(&consumerThread1, nullptr, consumer, &consumerId1);

pthread\_create(&consumerThread2, nullptr, consumer, &consumerId2);

pthread\_create(&consumerThread3, nullptr, consumer, &consumerId3);

pthread\_join(producerThread1, nullptr);

pthread\_join(producerThread2, nullptr);

pthread\_join(consumerThread1, nullptr);

pthread\_join(consumerThread2, nullptr);

pthread\_join(consumerThread3, nullptr);

sem\_destroy(&emptySlots);

sem\_destroy(&fullSlots);

return 0;

}

编译命令：g++ -o lab2-task4 lab2-task4.cpp -lpthread

**3.6 Linux下模拟哲学家就餐**

哲学家就餐问题（Dining Philosophers Problem）是一个经典的并发算法问题，用来说明在并发环境中可能出现的资源竞争和死锁情况。

实验环境：Ubuntu 20.04.6 LTS，GCC 9.4.0

核心代码：

死锁解法：以下为哲学家的线程函数，其中，将思考抽象为sleep(rand)的过程，将吃饭也抽象成挂起随机时间，为了使结果更明显，在拿起第一根筷子后，即为左边筷子加锁后，每个哲学家都sleep(2)，增加产生死锁的可能性。

void \*philosophy(void \*t) //第i位哲学家

{

float think;

int i = \*((int\*) t);

printf("%d\n",i);

int ret;

while(flag){

think = (rand()%400 + 100)/1000;

sleep(think); //思考

pthread\_mutex\_lock(&mutex[i]); //加锁

sleep(2);

pthread\_mutex\_lock(&mutex[(i+1)%5]);

//等待左右筷子可用

//吃饭

ate[i]=1;

printf("Philosophor %d is eating\n",i+1);

think = (rand()%400 + 100)/1000;

sleep(think);

//放下筷子

pthread\_mutex\_unlock(&mutex[i]);

pthread\_mutex\_unlock(&mutex[(i+1)%5]);

}

pthread\_exit(0);

}

非死锁解法：限制拿到筷子的人数最多为4个人

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

pthread\_mutex\_t mutex[NUM\_PHILOSOPHERS];

pthread\_mutex\_t chop;

int chops = 0;

int ate[NUM\_PHILOSOPHERS] = {0};

int flag = 1;

void \*philosopher(void \*t) {

float think;

int i = \*((int\*) t);

printf("%d\n", i);

while (flag) {

think = (rand() % 400 + 100) / 1000;

sleep(think); //思考a

if (chops == 4) continue;

pthread\_mutex\_lock(&mutex[i]); //加锁

pthread\_mutex\_lock(&chop);

chops++;

pthread\_mutex\_unlock(&chop);

sleep(2);

pthread\_mutex\_lock(&mutex[(i+1) % NUM\_PHILOSOPHERS]);

//等待左右筷子可用

//吃饭

ate[i] = 1;

printf("Philosopher %d is eating\n", i+1);

think = (rand() % 400 + 100) / 1000;

sleep(think);

//放下筷子

pthread\_mutex\_unlock(&mutex[i]);

pthread\_mutex\_unlock(&mutex[(i+1) % NUM\_PHILOSOPHERS]);

pthread\_mutex\_lock(&chop);

chops--;

pthread\_mutex\_unlock(&chop);

}

pthread\_exit(0);

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int i;

int philosophersIndex[NUM\_PHILOSOPHERS];

pthread\_mutex\_init(&chop, NULL);

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_mutex\_init(&mutex[i], NULL);

}

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

philosophersIndex[i] = i;

pthread\_create(&philosophers[i], NULL, philosopher, (void\*)&philosophersIndex[i]);

}

sleep(10); // 让哲学家就餐一段时间

flag = 0; // 设置标志位结束循环

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_join(philosophers[i], NULL);

}

pthread\_mutex\_destroy(&chop);

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_mutex\_destroy(&mutex[i]);

}

return 0;

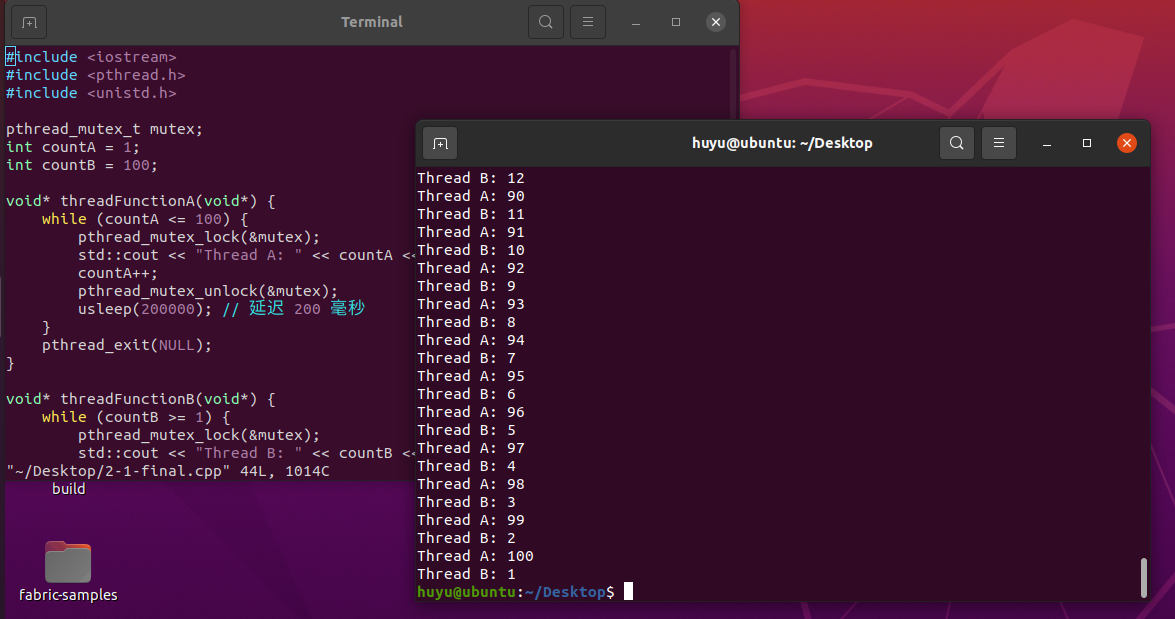
}

编译命令：g++ -o 2-6 2-6.cpp -lpthread

**四、实验结果**

**4.1 两线程循环输出数据和字符串**

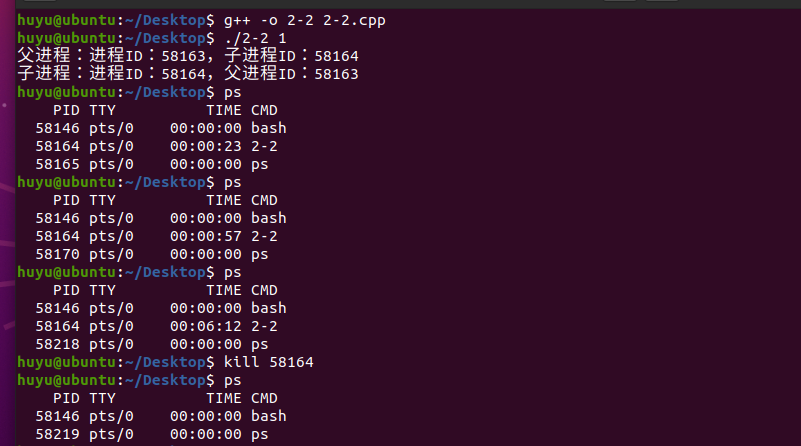
输出1-1000数据太大，遂决定两个线程交替输出1-100作为测试，测试结果正确，无任何异常，如图1。



**图1 测试正确案例**

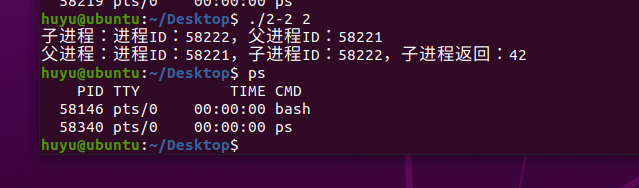
**4.2子进程建立并实现wait和exit函数**

根据题目要求，要实现两个效果，效果1如图2，效果2如图3。



**图2 效果1测试**

效果1中，父进程没有wait，而子进程进入了死循环。普通情况父子进程同时结束，但是父进程的运行比子进程快了一个fork函数的时间差，所以父进程先输出，之后子进程输出。父进程运行完就会自动结束，而子进程由于进入死循环会一直运行，测试中用ps命令在不同时间查看了三次进程，发现子进程（PID：58164）一直在运行，符合预期，知道手动kill了该进程后，子进程才结束。



**图3 效果2测试**

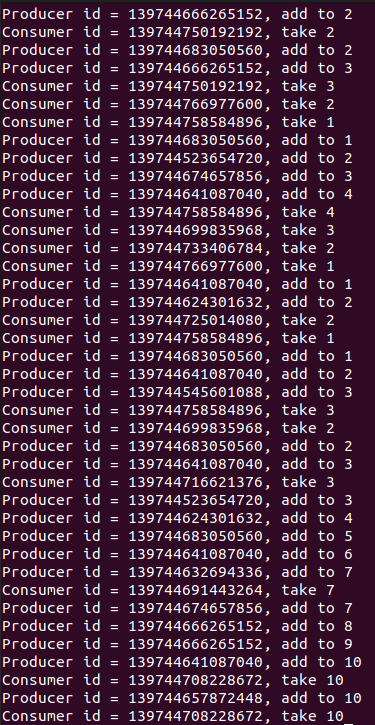
效果2中，父进程使用wait，子进程用exit返回。在此情况下，父进程wait函数等待子进程的终止，并获取子进程的退出状态。子进程在完成任务后，通过exit函数终止自己的执行，并返回退出状态码给父进程，所以会子进程先输出，父进程再输出，一切都是正常情况，符合预期。

**4.4 Linux实现“生产者-消费者”同步控制**

生产者线程通过生成随机数并将其放入缓冲区中，消费者线程从缓冲区中取出数据进行消费。生产者线程和消费者线程之间使用互斥锁pthread\_mutex\_t和信号量sem\_t进行同步。

在主函数中，首先初始化互斥锁和信号量，然后创建了两个生产者线程和三个消费者线程。每个线程都执行相应的生产者函数或消费者函数，并通过互斥锁和信号量进行同步操作。

最后，通过调用pthread\_join等待所有线程执行完毕，并销毁互斥锁和信号量。



**图4 生产者-消费者同步控制测试**

测试结果良好，如图4，通过多线程和信号量实现了生产者-消费者问题，展示了线程间的同步和互斥操作，以及如何处理共享资源的安全访问。

**4.6 Linux下模拟哲学家就餐**

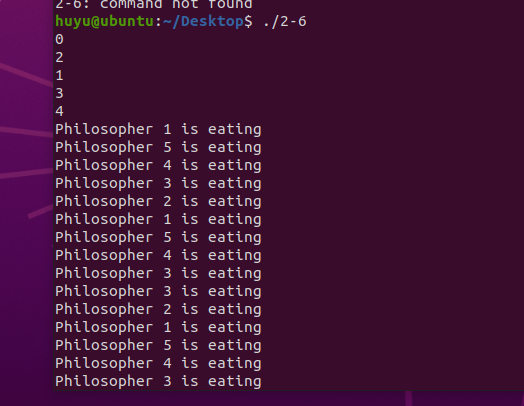
Linux下模拟哲学家就餐，提供死锁和非死锁解法。

死锁解法，为了体现死锁，在哲学家的函数中，取了第一根筷子后会sleep两秒，这样就一定会发生死锁。



**图5 死锁解法**

如图5，可以看到，在输出各自的编号后，就死锁了，程序无法正常运行。



**图6 非死锁解法**

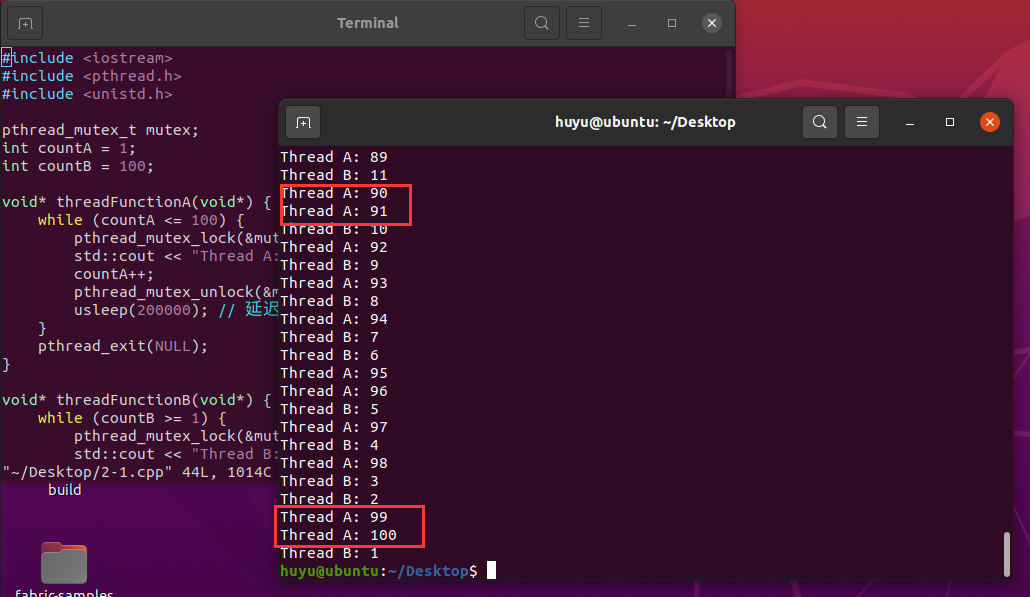
非死锁解法，如图6，输出各自编号后，程序还可以正常运行，且每个哲学家都吃到了饭。

**五、实验错误排查和解决方法**

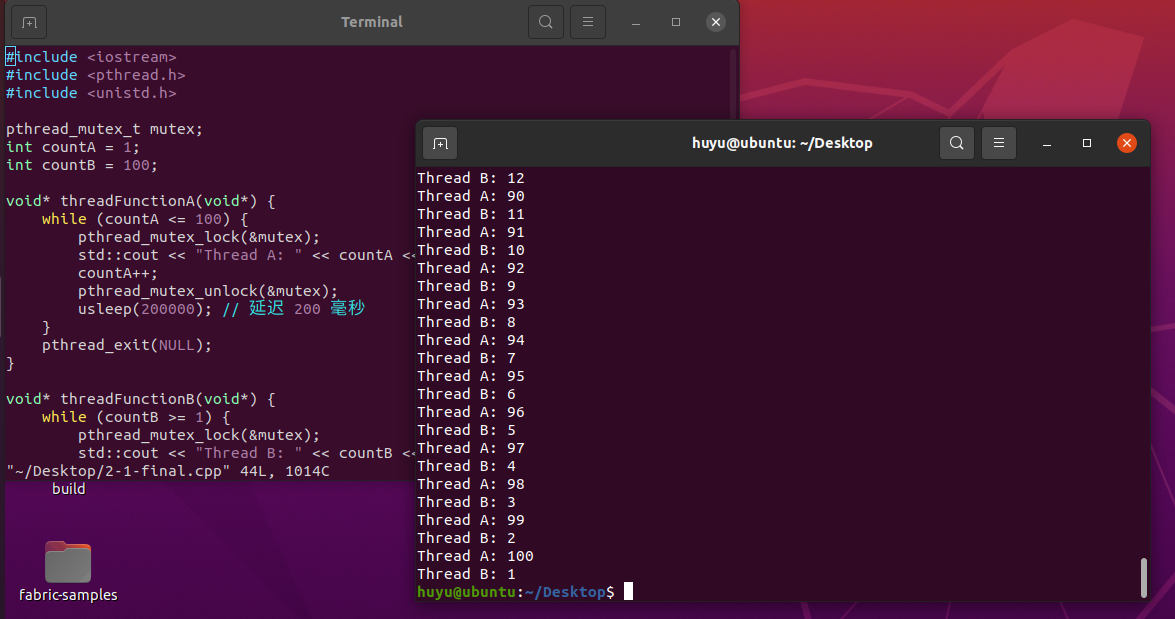
**5.1 两线程循环输出数据和字符串**

先开始没有用互斥锁，导致把线程A的100个数输出完之后，才开始输出线程B的数，很明显不符合题意。于是开始在网上查询资料，发现可以用互斥锁解决这种情况，其提供了一种互斥访问的机制，即同一时间只允许一个线程获取锁并访问被保护的资源，其他线程则需要等待锁的释放才能继续执行。然后便使用mutex编写代码，测试时却发现有些小问题，极个别数据输出顺序还是错误（如图7），于是开始仔细排查代码，发现代码中的互斥锁位置不正确，导致线程A连续输出两次。在示例代码中，我们应该在while循环内部使用互斥锁进行保护，以确保只有一个线程在访问共享变量countA和countB时进行修改，最终修改代码测试成功（如图8）。

编译的时候记得链接线程库：-lphtread。



**图7 测试错误案例**



**图8 测试正确案例**

**5.2子进程建立并实现wait和exit函数**

此实验较为简单，主要去理解了父子进程的并发过程，并没有什么大的问题。非要说的话：一是温习了一些linux命令行输入参数，argc的用法；二是熟悉了fork（）的用法，返回值为0表示子进程，大于0为父进程；三是在父进程中使用wait函数等待子进程的终止时，需要检查子进程的终止状态，可以使用WIFEXITED宏判断子进程是否正常终止。

**5.4 Linux实现“生产者-消费者”同步控制** 其实在完成实验1之后，多线程的操作就基本熟悉了，能够完成如何创建多线程并结束，通过互斥锁的加锁和解锁操作，保证了对缓冲区的访问是互斥的，避免竞态条件 。这里新接触的其实是几个信号量对象，通过上网查阅资料，可以得知：

sem\_wait(sem\_t \*sem)：等待信号量，该函数用于阻塞当前线程，并等待信号量的值变为大于 0。一旦信号量的值大于 0，函数将将其减一并返回；否则，当前线程将一直等待。sem\_post(sem\_t \*sem)：释放信号量。该函数用于增加信号量的值，并唤醒一个等待该信号量的线程。如果有线程正在等待该信号量，那么其中的一个线程将被唤醒。 由此，我们可以这样设计：生产者在放入一个物品之前，会调用sem\_wait(&emptySlots)来等待可用的空槽位。如果当前没有空槽位可用（即信号量的值为 0），生产者线程将被阻塞，直到有空槽位可用。当生产者放入一个物品后，会调用sem\_post(&emptySlots)释放一个空槽位。这将增加信号量的值，表示有一个空槽位可用。如果此时有消费者线程正在等待空槽位，那么其中的一个消费者线程将被唤醒。

**5.6 Linux下模拟哲学家就餐**

死锁现象很难触发，解决方法是在拿起第一根筷子后先sleep，这样就一定会触发死锁。

死锁问题通常在五个或更多的哲学家同时尝试获取筷子时出现，而我的非死锁解法代码限制了最多四个哲学家，所以有效解决了该问题。

**六、实验参考资料和网址**

1.实验ppt和上课课件

2.https://www.cnblogs.com/x\_wukong/p/5671137.html

3.<https://blog.csdn.net/l_searcing/article/details/83649962>