北 京 邮 电 大 学

计算机学院

《操作系统》课程设计指导书

Linux环境下进程/线程的管理通信与同步互斥机制

2020年5月

**目 录**

[1. 实验目的 3](#_Toc42014720)

[2. 实验环境 3](#_Toc42014721)

[3. 实验内容 3](#_Toc42014722)

[3.1 第一组 进程的创建与管理 3](#_Toc42014723)

[3.2 第二组 线程的创建与管理 4](#_Toc42014724)

[3.3 第三/四组 进程/线程通信 4](#_Toc42014725)

[3.4 第五组 进程/线程同步互斥 4](#_Toc42014726)

[4. 实验设计原理及设计方案 5](#_Toc42014727)

[5. 实验步骤及结果分析 6](#_Toc42014728)

[1. OS及Pthread线程库安装 6](#_Toc42014729)

[2. 第一组 进程的创建与管理 7](#_Toc42014730)

[3. 第二组 线程的创建与管理 8](#_Toc42014731)

[4. 第三组 进程间通信 9](#_Toc42014732)

[5. 第四组 线程间通信 10](#_Toc42014733)

[6. 第五组 进程/线程间同步与互斥（二选一） 10](#_Toc42014734)

[附录(源代码) 12](#_Toc42014735)

[第一组 进程的创建与管理 12](#_Toc42014736)

[第二组 线程的创建与管理 12](#_Toc42014737)

[第三组 进程间通信 14](#_Toc42014738)

[第四组 线程间通信 19](#_Toc42014739)

[第五组 进程/线程间同步与互斥 20](#_Toc42014740)

# 实验目的

1. 加深对Linux进程、线程概念的理解，掌握利用Linux系统调用创建、管理进程的方法，掌握利用POSIX线程（Pthread）库创建管理线程的方法，认识进程、线程并发执行的实质；
2. 深入理解Linux内核提供的消息队列、共享内存、管道、软中断四种进程间通信机制，掌握利用系统调用实现进程间通信；
3. 了解Pthread线程库提供的线程间通信机制，掌握使用Pthread API实现线程间通信的方法；
4. 深入理解Linux系统提供的多种进程同步互斥机制，掌握使用信号量实现进程间的同步互斥的方法；
5. 了解Pthread提供的线程同步互斥机制，掌握使用互斥变量和条件变量实现多线程间的同步互斥的方法

# 实验环境

硬件：Intel i7 CPU，内存=8G；

软件：Ubuntu的Linux发行版本，Linux内核版本5.3.0；

Pthread线程库，POSIX接口；gcc编译；

C语言。

# 实验内容

在Linux环境下，采用C语言编程，完成以下实验内容。

## 第一组 进程的创建与管理

1. 阅读Linux内核源码，分析Linux进程的组成，观察进PCB/task\_struc等进程管理数据结构；
2. 利用Linux内核提供的fork()、exec()、wait()等系统调用，创建管理多个进程，观察父子进程的结构和并发行为，掌握睡眠、撤销等进程控制方法；
3. 掌握ps、top、pstree –h、vmstat、strace、ltrace、sleep x、kill、jobs等命令的功能和使用方式；

## 第二组 线程的创建与管理

1. 了解POSIX 线程标准库（Pthread线程库）定义的线程结构和提供的线程管理API；
2. 利用Pthread线程库API，创建管理多个线程，观察线程的结构和并发执行行为；

## 第三/四组 进程/线程通信

1. 了解Linux提供的消息队列、共享内存、管道、软中断Signal等四种通信机制，编程实现进程间通信；
2. 线程通信

了解Linux所支持的线程机制，在一个进程内创建多个主从线程，采用参数传递机制，实现线程间通信；

## 第五组 进程/线程同步互斥

【要求：参照2019-2020学年操作系统期末考试信号量题目，设计实现三个进程/线程A、B、C，分别模拟题目所描述的生产产品A、B、C的三个worker，观察并记录进程/线程的创建和同步互斥行为。重点分析信号量设计方案是否合理、符合预期，避免设计方案导致死锁或不符合题目要求。

期末考试试题如下：

An assembly line is to produce a product C with four part As, and three part Bs. The worker of machining(加工) A and worker of machining B will produce two part As and one part B independently each time. Then the two part As or one part B will be moved to the station(工作台), which can hold at most 12 of part As and part Bs altogether. Two part As must be put onto the station simultaneously. The workers must exclusively put a part on the station or get it from the station. In addition, the worker to make C must get all part of As and Bs for one product once.

Using semaphores to coordinate the three workers who are machining part A, part B and the product C to manufacture the product without deadlock.

It is required that

(1) definition and initial value of each semaphore, and

(2) the algorithm to coordinate the production process for the three workers

should be given.

**】**

1. 进程同步互斥

创建3个Linux进程，分别模拟生产产品A、B、C的三个worker的行为，利用Linux内核信号量，实现三者间正确的同步互斥；

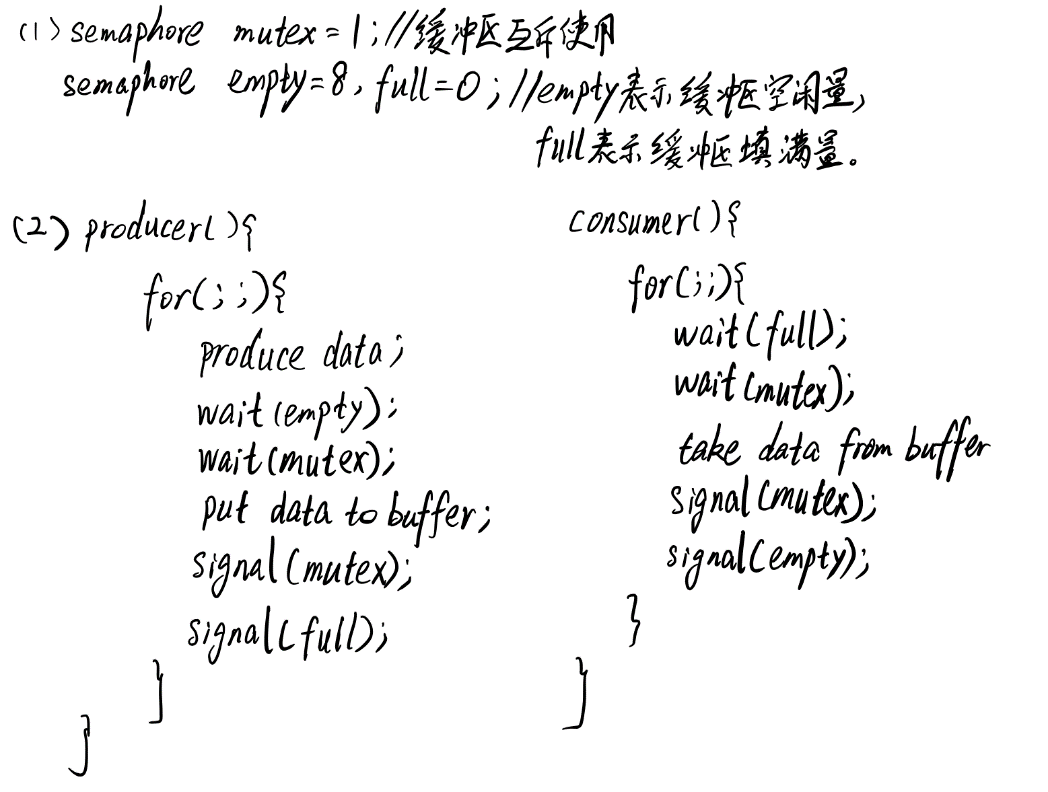
1. 线程同步互斥

利用Pthread API，创建3个Linux线程，分别模拟生产产品A、B、C的三个worker的行为，采用Pthread提供的信号量/管程机制，现三者间正确的同步互斥；

# 实验设计原理及设计方案

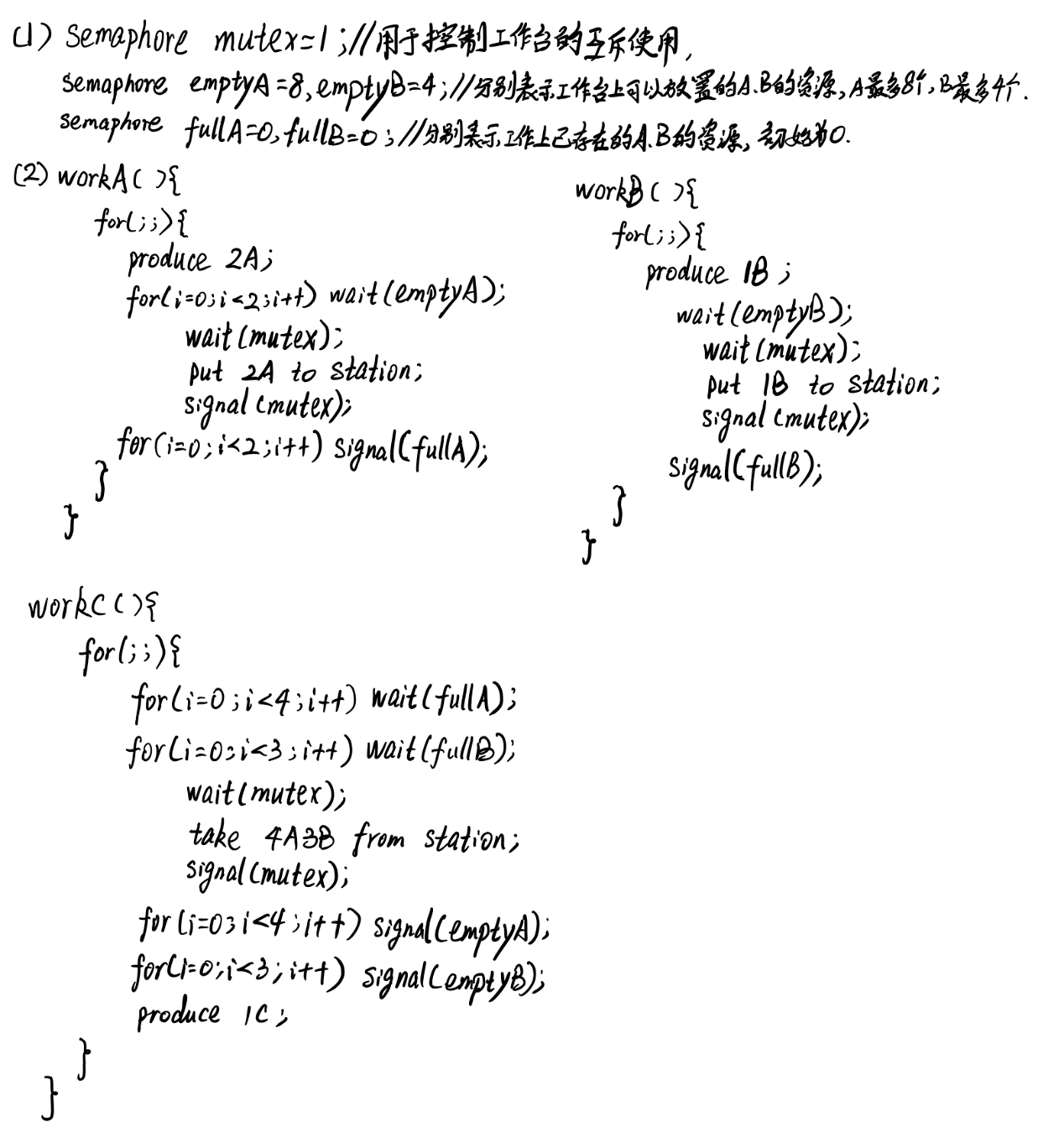
前四组实验设计原理分别为进程创建及管理、进程创建及管理、进程间通信、线程间通信。第五组实验原理则是通过信号量控制进程间同步互斥。

第三组实验中通过共享内存的通信方式和信号量实现了生产者与消费者之间的同步互斥，设计方案如下图：



第五组实验本质上是生产者与消费者的问题，工作台有12个资源，可以分配给A 8个，分配给B 4个，这样既可以避免产生死锁，又方便了工人A与工人C、工人B与工人C之间进行同步。

第五组的设计方案如下图所示：



# 实验步骤及结果分析

分步骤完成以下实验内容。

## OS及Pthread线程库安装

选择合适的Linux发行版本，观察所采用的内核版本，采用硬盘分区模式安装Linux系统；也可以先在本机操作系统上安装VitualBox、VMware、Virtual PC等虚拟机软件，在虚拟机之上安装Linux系统；

观察确认所安装的Linux发行版本带有Pthread线程库，注意：

1. 某些版本Ubuntu Linux默认不带Pthread线程库，即使在编译的度时候 加上 -lpthread 也不行，man不到相关Pthread函知数。此时，需要在/usr/lib/…下导入动态库libpthread.a，具体方法可以查阅网上相关资料。
2. 后续编程时导入头文件:#include <pthread.h>

## 第一组 进程的创建与管理

**实验1-1. 进程观察**

查阅相关资料，阅读Linux内核源码，分析Linux进程的组成，了解进程状态，观察进程PCB/task\_struc等进程管理数据结构；

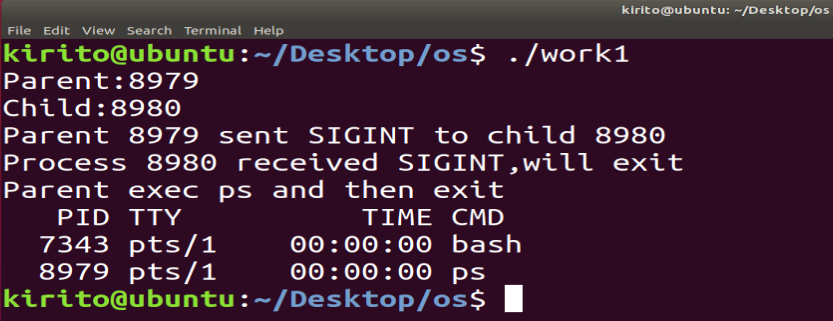
**实验1-2. 进程创建与管理**

参照“**【实验指导】1.进程创建及管理示例”**中的程序，结合所查阅的参考资料，利用Linux内核提供的fork()、exec()、wait()、exit()、kill()等系统调用，创建管理多个进程，观察父子进程的结构和并发行为，掌握睡眠、撤销等进程控制方法；

**要求：本组实验至少用到fork()、exec()、wait()、exit()、kill()、getpid五个系统调用。**

编写代码，父进程通过fork()创建子进程，然后父进程睡眠3s再通过kill()后向子进程发送信号。通过signal()设置子进程收到信号后，通过getpid()打印进程信息再执行exit()退出。父进程通过join()等待子进程结束，再通过exec()执行ps，这时应该只剩父进程ps了。详细源代码见【[附录](#_第一组_进程的创建与管理)】。

执行代码截图如下：



结果分析：有截图可见父进程pid为8979，fork()出的子进程pid为8980，3s后父进程发送信号SIGINT给子进程，子进程收到后退出，父进程再执行ps，打印结果只剩父进程8979，符合预期。该程序正确执行了进程的创建、退出、睡眠以及进程间信号的通信。

**实验1-3. 进程管理命令**

了解ps、top、pstree –h、vmstat、strace、ltrace、sleep x、kill、jobs等命令的功能，使用这些命令观察进程结构和行为；

## 第二组 线程的创建与管理

**实验2-1. Pthread线程库背景知识**

了解POSIX 线程标准库（Pthread线程库）相关知识，分析Pthread线程结构，掌握所提供的线程管理API，如pthread\_create, pthread\_join, pthread\_t pthread\_self, pthread\_detach, pthread\_exit；

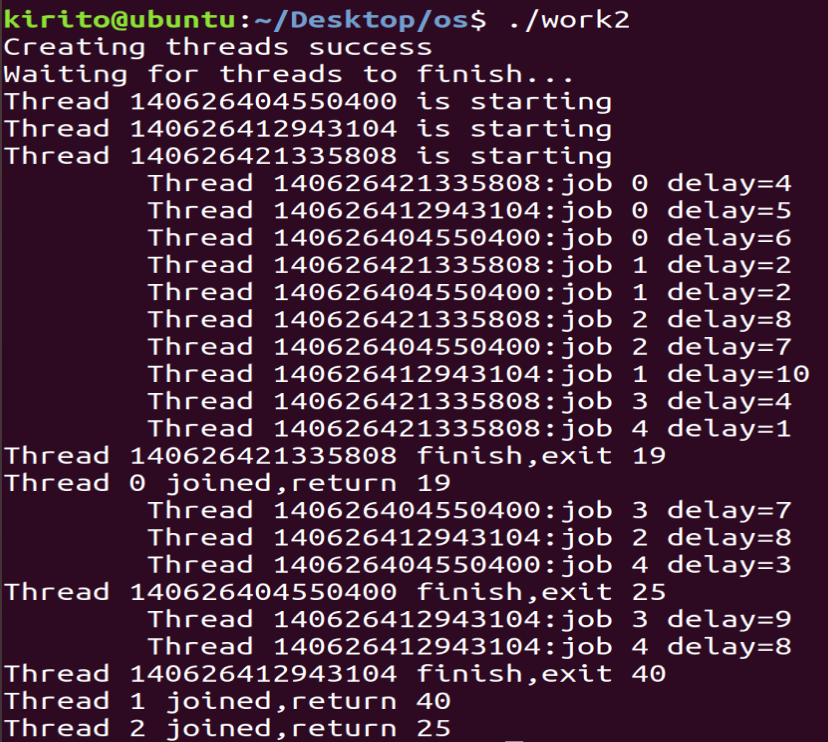
**实验2-2. 线程创建与管理**

参照“**【实验指导】 2. 线程创建及管理程序示例”**，查阅参考资料，利用Pthread API，创建和管理线程，观察线程的结构和并发执行行为；

要求：本组实验至少用到pthread\_create, pthread\_exit, pthread\_join, pthread\_self等四个API。

编写代码，通过pthread\_create创建子线程，在子线程中通过pthread\_self获取线程id并打印，再通过pthread\_exit退出子线程，主线程中通过pthread\_join等待子线程结束并打印其返回值。详细源代码见【[附录](#_第二组_线程的创建与管理)】。

运行结果截图如下：



结果分析：有截图可知该程序创建了3个子线程，每个子线程执行5个任务。以id为14…00的线程为例，执行job的时间分别为6s、2s、7s、7s、3s一共25s，退出时返回25，与主线程打印的线程2向对应。可见该程序成功创建了多个线程并退出线程，在子线程中成功获取线程id，在主线程中成功获取子线程返回值。

## 第三组 进程间通信

了解Linux提供的消息队列（消息传递）、共享内存、管道/命名管道、信号(signal)/软中断四种进程间通信机制的实现原理和方法；

参照“**【实验指导】 6.3进程间通信示例”**，查阅参考资料，**选择上述四种通信方式中的一种，**编程实现进程间通信，观察进程间通信过程。

本组实验中我选择的通信方式为共享内存，编写代码实现了生产者与消费者的同步互斥。

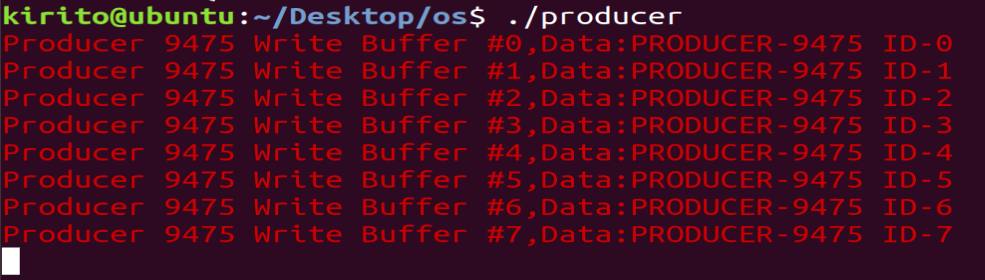
以共享内存作为生产者与消费者之间的缓冲区，同时用信号量控制两者的同步互斥，避免了忙等待与死锁。详细源代码与注释见【[附录](#_第三组_进程间通信)】。

实验运行结果：

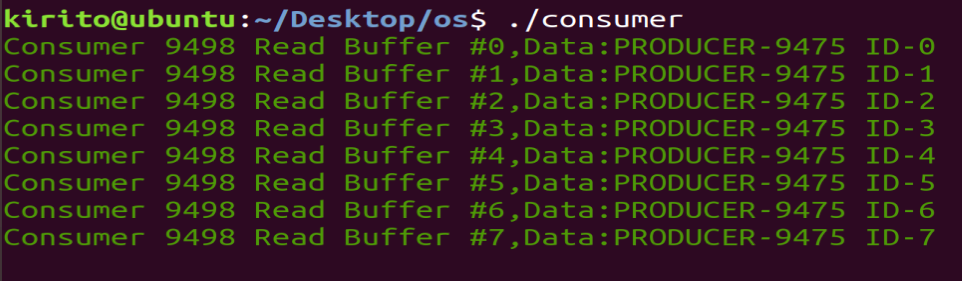
执行create后再执行producer与consumer：



remove后重新create，这次只执行producer，缓冲区满后producer阻塞，如下图：



再结束producer开启consumer，缓冲区空后consumer阻塞：



实验结果分析：执行create后成功在内核中创建信号量和共享内存，再执行producer与consumer，生产者与消费者正常工作，并未发生死锁。当只有producer在运行时，缓冲区满后producer就因wait()而挂起，当只有consumer运行时，缓冲区的数据取完后consumer因wait()而挂起，符合预期。

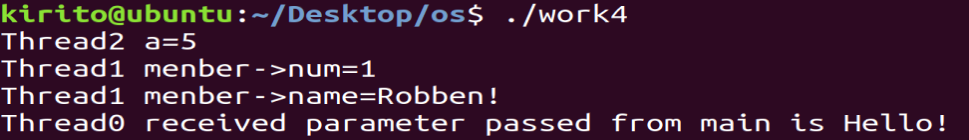
## 第四组 线程间通信

了解Linux中的线程概念、线程通信机制、线程间同步互斥模式，以及多线程编程方式；

参照“**【实验指导】 6.2.7 Linux c/c++线程间参数传递”中的【示例2-3-3】【示例2-3-5】**，编程实现线程间参数传递。

编程实现线程间参数传递，包括字符串、结构体以及全局变量，源代码见【[附录](#_第四组_线程间通信)】。

运行结果截图如下：



结果分析：由截图可知主线程创建了3个子线程，向线程0传递字符串，向线程1传递结构体，向线程2传递全局变量。该程序结果符合预期，完成了线程间的参数传递。

## 第五组 进程/线程间同步与互斥（二选一）

1. 针对2019-2020学年操作系统期末考试信号量题目，定义合理的锁和信号量，设计生产产品A、B、C的三个worker三个进程/线程A、B、C的业务流程和同步互斥机制。**要求：给出具体的设计方案，并单独提交设计方案，类似期末考试答题形式；**
2. 参照“**【实验指导】 6.5 进程间同步互斥示例”**，根据上一步的设计方案，利用semget、semctl、semop等信号量原语，以进程方式编程实现该设计方案；

观察分析程序运行结果，重点分析信号量设计方案是否合理、程序运行是否符合预期，应避免设计方案导致死锁或不符合题目要求。

1. 参照“**【实验指导】 6.6 线程间同步互斥示例”**，根据第一步的设计方案，利用Pthread提供的pthread\_mutex\_init()、pthread\_mutex\_lock()、pthread\_mutex\_unlock()等线程同步互斥API，以线程方式编程实现该设计方案，观察分析程序运行结果。重点分析信号量设计方案是否合理、程序运行是否符合预期，应避免设计方案导致死锁或不符合题目要求。

要求：

* **基于进程和基于线程的同步互斥实现方案二选一，完成其中一个即可；**
* 不论采用进程方式、还是线程方式，模拟A、B、C三个worker的三个进程/线程A、B、C应当多次循环反复执行，便于观察同步互斥效应；
* 根据设计方案，给出程序代码，程序运行结果，结果分析。

在该组实验中我选择的是基于进程的同步互斥的方式。

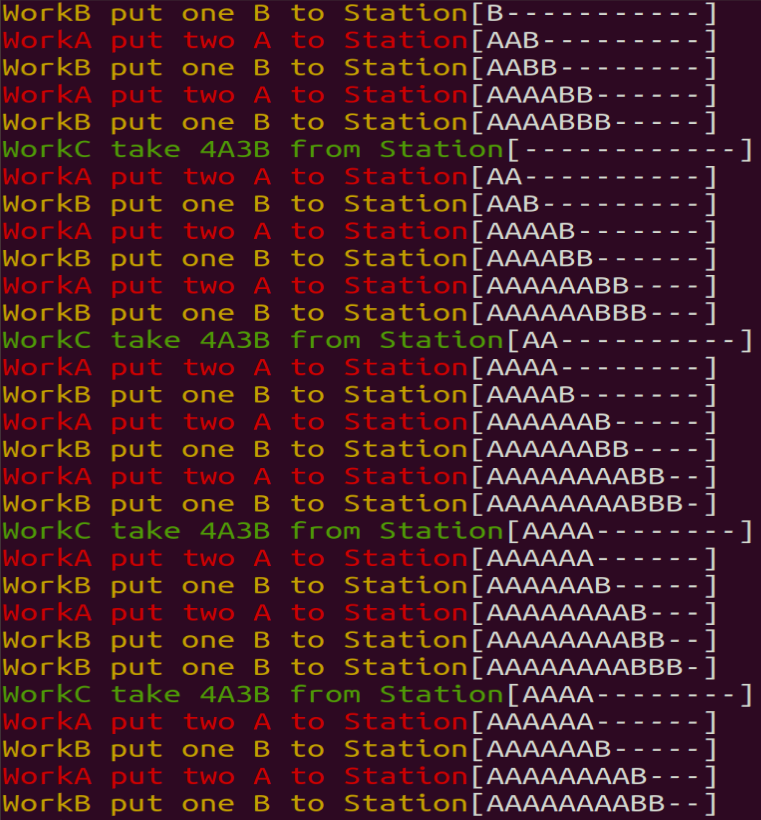
实验步骤如下：

1. 根据题目设计方案，具体方案见报告中【[实验设计原理及设计方案](#_实验设计原理及设计方案)】的[第五组设计方案](#第五组设计方案)。

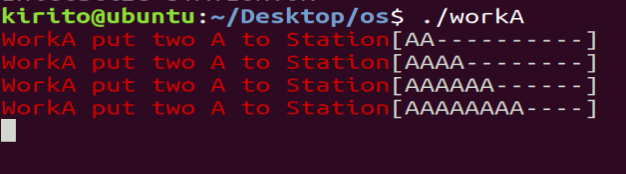
2. 根据方案编写代码，详细代码见【[附录](#_第五组_进程/线程间同步与互斥)】。

3. 运行结果截图如下：

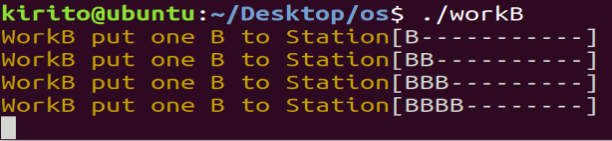
运行create后再运行workA、workB、workC：



只运行workA时：



只运行workB时：



只运行workC时：



4. 结果分析：由实验结果可知，工人A、工人B、工人C互斥地对工作台进行取放部件的操作，工人A一次生产2个部件A放入工作台，且当工作台中放满8个部件A时，工人A阻塞，等待工作台中的部件A被取走。工人B一次生产1个部件B放入工作台，且当工作台中放满4个部件B时，工人B阻塞，等待工作台中的部件B被取走。工人C在工作台不足4个部件A和3个部件B时阻塞，等待满足条件时一次性从工作台取出4个部件A和3个部件B。符合题意且未产生死锁。

# 附录(源代码)

## 第一组 进程的创建与管理

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

void quit(){

//收到信号SIGINT后，结束进程

printf("Process %d received SIGINT,will exit\n",getpid());

exit(0);

}

int main(){

int ret;

if(ret=fork()){

//父进程睡眠秒后向子进程发送信号SIGINT

printf("Parent:%d\n",getpid());

sleep(3);

kill(ret,SIGINT);

printf("Parent %d sent SIGINT to child %d\n",getpid(),ret);

//父进程等待子进程结束执行ps

wait(0);

printf("Parent exec ps and then exit\n");

execlp("ps","ps",NULL);

}else{

//设置子进程收到信号SIGINT的处理函数

printf("Child:%d\n",getpid());

signal(SIGINT,quit);

//无限循环

for(;;){}

}

}

## 第二组 线程的创建与管理

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#define THREAD\_NUMBER 3 //线程数

#define REPEAT\_NUMBER 5 //任务数

#define DELAY\_TIME\_MAX 10 //最大执行时间

void \*thread\_func(void){

pthread\_t thread\_id=pthread\_self();

int delay\_time=0;

int count=0;

int sum=0;

printf("Thread %lu is starting\n",thread\_id);

//每个线程执行REPEAT\_NUMBER个任务，每个任务即睡眠一段时间，且时间不超过DELAY\_TIME\_MAX秒

for(count=0;count<REPEAT\_NUMBER;count++){

delay\_time=(int)(rand()%DELAY\_TIME\_MAX)+1;

sleep(delay\_time);

printf("\tThread %lu:job %d delay=%d\n",thread\_id,count,delay\_time);

sum+=delay\_time;

}

printf("Thread %lu finish,exit %d\n",thread\_id,sum);

//线程退出时返回所有任务的总执行时间

pthread\_exit((void\*)sum);

}

int main(){

pthread\_t thread[THREAD\_NUMBER];

int no,res;

void\* ret[THREAD\_NUMBER];

srand(time(NULL));

//创建THREAD\_NUMBER个线程

for(no=0;no<THREAD\_NUMBER;no++){

res=pthread\_create(&thread[no],NULL,(void\*)thread\_func,NULL);

if(res!=0){

printf("Create thread %d failed\n",no);

exit(res);

}

}

printf("Creating threads success\nWaiting for threads to finish...\n");

//等待THREAD\_NUMBER个线程结束并打印其返回值

for(no=0;no<THREAD\_NUMBER;no++){

res=pthread\_join(thread[no],&ret[no]);

if(!res){

printf("Thread %d joined,return %d\n",no,(int)ret[no]);

}else{

printf("Thread %d join failed\n",no);

}

}

return 0;

}

## 第三组 进程间通信

//share memory

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<sys/shm.h>

#define N 8 //缓冲区能容纳的数据个数

#define BUF\_SIZE 128 //数据大小

//缓冲区队列

typedef struct{

int id;

int head,tail;

char buf[N][BUF\_SIZE];

}QUEUE;

#define NUM\_SEMAPHORE 3 //信号量个数

#define MUTEX 0 //MUTEX用于缓冲区队列的互斥使用

#define FULL 1 //FULL表示缓冲区中数据个数

#define EMPTY 2 //EMPTY表示缓冲区中空闲量

#define SEM\_KEY 0x2233

#define SHM\_KEY 0x6666

#define WHITE "\033[0;37m"

#define RED "\033[0;31m"

#define YELLOW "\033[0;33m"

#define GREEN "\033[0;32m"

#define PINK "\033[0;35m"

//创建信号量与共享内存并初始化

void create\_ipc(void){

int sem\_id,shm\_id;

struct sembuf ops[3];

QUEUE \*q;

sem\_id=semget(SEM\_KEY,NUM\_SEMAPHORE,IPC\_CREAT|IPC\_EXCL|0666);//创建信号量

if(sem\_id==-1){

perror("Create Semaphores");

exit(1);

}else{

printf("Create Semaphores:OK\n");

//MUTEX初始为1

ops[0].sem\_num=MUTEX;

ops[0].sem\_op=1;

ops[0].sem\_flg=0;

//FULL初始为0

ops[1].sem\_num=FULL;

ops[1].sem\_op=0;

ops[1].sem\_flg=0;

//EMPTY初始为N

ops[2].sem\_num=EMPTY;

ops[2].sem\_op=N;

ops[2].sem\_flg=0;

}

if(semop(sem\_id,ops,3)==-1){

perror("semop");

exit(1);

}else{

printf("Initialize Semaphores:OK\n");

}

shm\_id=shmget(SHM\_KEY,sizeof(QUEUE),IPC\_CREAT|IPC\_EXCL|0666);//创建共享内存

if(shm\_id==-1){

perror("Create Share Memory");

exit(1);

}else{

printf("Create Share Memory:OK\n");

q=(QUEUE\*)shmat(shm\_id,0,0);//获得共享内存的指针

if(q==(QUEUE\*)-1){

perror("Attach Share Memory");

exit(1);

}else{

q->head=q->tail=0;//初始化缓冲区队列

printf("Initialize QUEUE:OK\n");

}

}

}

//删除信号量与共享内存

void remove\_ipc(void){

int sem\_id,shm\_id;

sem\_id=semget(SEM\_KEY,0,0);

if(sem\_id==-1){

perror("Get Semaphores");

}else{

if(semctl(sem\_id,0,IPC\_RMID,0)==-1){

perror("Remove Semaphores");

}else{

printf("Remove Semaphores:OK\n");

}

}

shm\_id=shmget(SHM\_KEY,0,0);

if(shm\_id==-1){

perror("Get Share Memory");

}else{

if(shmctl(shm\_id,IPC\_RMID,0)==-1){

perror("Remove Share Memory");

}else{

printf("Remove Share Memory:OK\n");

}

}

}

static int sem\_id;

static QUEUE \*q;

//获取信号量与共享内存设置到全局变量中

void get\_ipc(void){

int shm\_id;

sem\_id=semget(SEM\_KEY,0,0);

if(sem\_id==-1){

perror("Get Semaphore ID");

exit(1);

}

shm\_id=shmget(SHM\_KEY,0,0);

if(shm\_id==-1){

perror("Get Share Memory ID");

exit(1);

}

q=(QUEUE\*)shmat(shm\_id,0,0);

if(q==(QUEUE\*)-1){

perror("Attach Share Memory");

exit(1);

}

}

//信号量wait操作

void Wait(int sem\_num){

struct sembuf op;

op.sem\_num=sem\_num;

op.sem\_op=-1;

op.sem\_flg=0;

if(semop(sem\_id,&op,1)==-1){

perror("Wait(semaphore)");

exit(1);

}

}

//信号量signal操作

void Signal(int sem\_num){

struct sembuf op;

op.sem\_num=sem\_num;

op.sem\_op=1;

op.sem\_flg=0;

if(semop(sem\_id,&op,1)==-1){

perror("Signal(semaphore)");

exit(1);

}

}

//生产数据

void produce(char \*data){

int data\_id;

sleep(1+random()%5);

Wait(MUTEX);

data\_id=q->id++;

Signal(MUTEX);

sprintf(data,"PRODUCER-%d ID-%d",getpid(),data\_id);

}

//生产者进程

void producer(void){

char data[BUF\_SIZE];

for(;;){

produce(data);

Wait(EMPTY);

Wait(MUTEX);

strcpy(q->buf[q->tail],data);

printf(RED"Producer %d Write Buffer #%d,Data:%s"WHITE"\n",getpid(),q->tail,q->buf[q->tail]);

q->tail=(q->tail+1)%N;

Signal(MUTEX);

Signal(FULL);

}

}

//消费数据

void consume(char \*data){

sleep(1+random()%5);

}

//消费者进程

void consumer(void){

char data[BUF\_SIZE];

for(;;){

Wait(FULL);

Wait(MUTEX);

strcpy(data,q->buf[q->head]);

printf(GREEN"Consumer %d Read Buffer #%d,Data:%s"WHITE"\n",getpid(),q->head,q->buf[q->head]);

q->head=(q->head+1)%N;

Signal(MUTEX);

Signal(EMPTY);

consume(data);

}

}

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

//根据不同的程序名执行不同的操作

if(strstr(argv[0],"create")){

create\_ipc();

}else if(strstr(argv[0],"remove")){

remove\_ipc();

}else if(strstr(argv[0],"producer")){

get\_ipc();

producer();

}else if(strstr(argv[0],"consumer")){

get\_ipc();

consumer();

}else{

printf("Use name: {create | remove | producer | consumer }\n");

}

return 0;

}

## 第四组 线程间通信

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

typedef struct{

int num;

char \*name;

}menber;

static int a=5;

//向线程传递字符串

void thread0(char \*arg){

char \*str;

str=arg;

printf("Thread0 received parameter passed from main is %s\n",str);

}

//向线程传递结构体

void thread1(void \*arg){

menber\* temp=(menber\*)arg;

printf("Thread1 menber->num=%d\n",temp->num);

printf("Thread1 menber->name=%s\n",temp->name);

}

//通过全局变量向线程传参

void thread2(void\* arg){

printf("Thread2 a=%d\n",a);

}

int main(){

int res;

pthread\_t id[3];

char \*str1="Hello!";

res=pthread\_create(&id[0],NULL,(void\*)thread0,(void\*)str1);

if(res!=0){

printf("Thread0 id not create!\n");

exit(1);

}

menber \*p=(menber\*)malloc(sizeof(menber));

p->num=1;

p->name="Robben!";

res=pthread\_create(&id[1],NULL,(void\*)thread1,(void\*)p);

if(res!=0){

printf("Thread1 id not create!\n");

exit(1);

}

res=pthread\_create(&id[2],NULL,(void\*)thread2,NULL);

if(res!=0){

printf("Thread2 id not create!\n");

exit(1);

}

for(int i=0;i<3;i++){

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

}

return 0;

}

## 第五组 进程/线程间同步与互斥

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<sys/shm.h>

#define WHITE "\033[0;37m"

#define RED "\033[0;31m"

#define YELLOW "\033[0;33m"

#define GREEN "\033[0;32m"

#define PINK "\033[0;35m"

#define SEM\_KEY 0x2233

#define SHM\_KEY 0x6666

#define NUM\_SEMAPHORE 5//信号量个数

#define MUTEX 0

#define EMPTYA 1

#define EMPTYB 2

#define FULLA 3

#define FULLB 4

#define N 12

//工作台，只记录A与B的个数

typedef struct{

int numA,numB;

}Station;

static int sem\_id;

static Station \*s;

//信号量初始化，将sem\_num初始为value

void init\_sem(int sem\_num,int value){

struct sembuf op;

op.sem\_num=sem\_num;

op.sem\_op=value;

op.sem\_flg=0;

if(semop(sem\_id,&op,1)==-1){

perror("init(semaphore)");

exit(1);

}

}

//在Linux内核中创建信号量与共享内存

void create\_ipc(void){

int shm\_id;

sem\_id=semget(SEM\_KEY,NUM\_SEMAPHORE,IPC\_CREAT|IPC\_EXCL|0666);

if(sem\_id==-1){

perror("Create Semaphores");

exit(1);

}else{

printf("Create Semaphores:OK\n");

//初始化信号量

init\_sem(MUTEX,1);

init\_sem(EMPTYA,8);

init\_sem(EMPTYB,4);

init\_sem(FULLA,0);

init\_sem(FULLB,0);

printf("Initialize Semaphores:OK\n");

}

shm\_id=shmget(SHM\_KEY,sizeof(Station),IPC\_CREAT|IPC\_EXCL|0666);

if(shm\_id==-1){

perror("Create Share Memory");

exit(1);

}else{

printf("Create Share Memory:OK\n");

s=(Station\*)shmat(shm\_id,0,0);

if(s==(Station\*)-1){

perror("Attach Share Memory");

exit(1);

}else{

s->numA=s->numB=0;//初始化工作站，A与B的个数初始为零

printf("Initialize STATION:OK\n");

}

}

}

//删除内核中的信号量与共享内存

void remove\_ipc(void){

int shm\_id;

sem\_id=semget(SEM\_KEY,0,0);

if(sem\_id==-1){

perror("Get Semaphores");

}else{

if(semctl(sem\_id,0,IPC\_RMID,0)==-1){

perror("Remove Semaphores");

}else{

printf("Remove Semaphores:OK\n");

}

}

shm\_id=shmget(SHM\_KEY,0,0);

if(shm\_id==-1){

perror("Get Share Memory");

}else{

if(shmctl(shm\_id,IPC\_RMID,0)==-1){

perror("Remove Share Memory");

}else{

printf("Remove Share Memory:OK\n");

}

}

}

//获取内核中的信号量与共享内存，并设置到全局变量中

void get\_ipc(void){

int shm\_id;

sem\_id=semget(SEM\_KEY,0,0);

if(sem\_id==-1){

perror("Get Semaphore ID");

exit(1);

}

shm\_id=shmget(SHM\_KEY,0,0);

if(shm\_id==-1){

perror("Get Share Memory ID");

exit(1);

}

s=(Station\*)shmat(shm\_id,0,0);

if(s==(Station\*)-1){

perror("Attach Share Memory");

exit(1);

}

}

//信号量wait操作

void Wait(int sem\_num){

struct sembuf op;

op.sem\_num=sem\_num;

op.sem\_op=-1;

op.sem\_flg=0;

if(semop(sem\_id,&op,1)==-1){

perror("Wait(semaphore)");

exit(1);

}

}

//信号量signal操作

void Signal(int sem\_num){

struct sembuf op;

op.sem\_num=sem\_num;

op.sem\_op=1;

op.sem\_flg=0;

if(semop(sem\_id,&op,1)==-1){

perror("Signal(semaphore)");

exit(1);

}

}

//生产部件，类型为tyoe，个数为num

void produce(char type,int num){

sleep(1+random()%5);

}

//检查工作站状态并打印

void pstation(){

if(s->numA>=0&&s->numB>=0&&s->numA+s->numB<N){ //检查工作站状态是否正确

printf("[");

int i;

for(i=0;i<s->numA;i++){ printf("A"); }

for(i=0;i<s->numB;i++){ printf("B"); }

int blank=N-s->numA-s->numB;

for(i=0;i<blank;i++){ printf("-"); }

printf("]\n");

}else{

printf("Station error\n");

exit(1);

}

}

//将个数为num，类型tyoe的部件放入工作站

void put(char type,int num){

if(type=='A'){

s->numA+=num;

}

if(type=='B'){

s->numB+=num;

}

}

//将个数为num，类型tyoe的部件从工作站取出

void take(char type,int num){

if(type=='A'){

s->numA-=num;

}

if(type=='B'){

s->numB-=num;

}

}

//工人A的进程

void workA(void){

int i;

get\_ipc();

for(;;){

produce('A',2);

for(i=0;i<2;i++){ Wait(EMPTYA); }

Wait(MUTEX);

put('A',2);

printf(RED"WorkA put two A to Station"WHITE);

pstation();

Signal(MUTEX);

for(i=0;i<2;i++){ Signal(FULLA); }

}

}

//工人B的进程

void workB(void){

int i;

get\_ipc();

for(;;){

produce('B',1);

Wait(EMPTYB);

Wait(MUTEX);

put('B',1);

printf(YELLOW"WorkB put one B to Station"WHITE);

pstation();

Signal(MUTEX);

Signal(FULLB);

}

}

//工人C的进程

void workC(void){

int i;

get\_ipc();

for(;;){

for(i=0;i<4;i++){ Wait(FULLA); }

for(i=0;i<3;i++){ Wait(FULLB); }

Wait(MUTEX);

take('A',4);

take('B',3);

printf(GREEN"WorkC take 4A3B from Station"WHITE);

pstation();

Signal(MUTEX);

for(i=0;i<4;i++){ Signal(EMPTYA); }

for(i=0;i<3;i++){ Signal(EMPTYB); }

produce('C',1);

}

}

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

//根据程序名运行相应操作

if(strstr(argv[0],"create")){

create\_ipc();

}

if(strstr(argv[0],"remove")){

remove\_ipc();

}

if(strstr(argv[0],"workA")){

workA();

}

if(strstr(argv[0],"workB")){

workB();

}

if(strstr(argv[0],"workC")){

workC();

}

return 0;

}