# 《操作系统原理》实验2

## 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 实验要求

1,2,4必做（当堂演示2或4得1分）。若选做7总分加2分。课前预做。

## 实验1 在Linux下创建2个线程A和B，循环输出数据或字符串

* 提示 1：使用 pthread 线程库。
* 提示 2：线程 A 递增输出 1-1000；线程 B 递减输出 1000-1。为避免输出太快，每隔 0.2 秒（可自行调节）输出一个数。
* 提示 3：输出数据时，同时输出 A 或 B 以标示是哪个线程输出的，并注意格式化输出信息。

#### 1 编写代码

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

//定义线程1运行函数

void\* th\_fn1(void \*arg)

{

for(int i=1;i<=1000;i++){

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

sleep(1);

}

}

//定义线程2运行函数

void\* th\_fn2(void \*arg)

{

for(int i=1000;i>0;i--){

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

sleep(1);

}

}

int main()

{

int err;//定义错误存储

pthread\_t tid1,tid2;//定义线程标识符

//创建tid1线程

if(err=pthread\_create(&tid1,NULL,th\_fn1,NULL)){

perror("pthread\_create error");

}

//创建tid2线程

if(err=pthread\_create(&tid2,NULL,th\_fn2,NULL)){

perror("pthread\_create error");

}

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

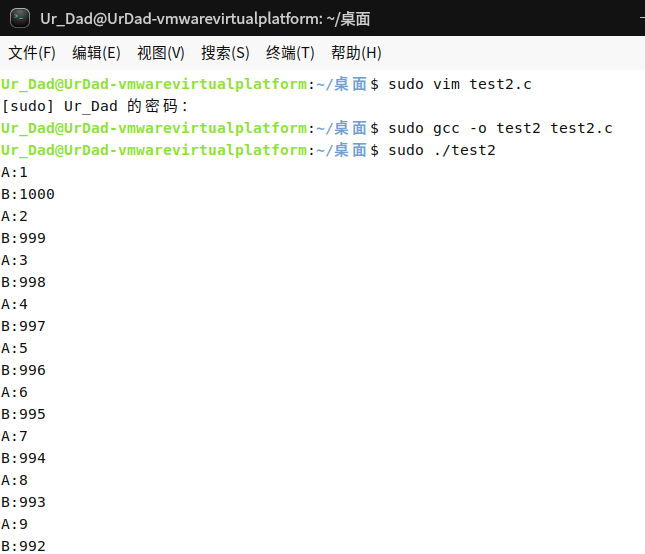
return 0;

}

#### 2 编译运行



#### 3 结果展示



### 总结

【线程创建函数：int pthread\_create(a,b,c,d)】

返回值：

* 成功：返回0
* 失败：返回错误编号，并不设置errno

参数：

* a：事先创建好的pthread\_t类型的参数。成功时a指向的内存单元被设置为新创建线程的线程ID。
* b：用于定制各种不同的线程属性，通常直接设为NULL。
* c：新创建线程从此函数开始运行。无参数是arg设为NULL即可。
* d：函数c的参数。无参数时设为NULL即可，有参数时输入参数的地址，当多于一个参数时应当使用结构体传入。

【线程等待函数：int pthread\_join ( pthread\_t thread, void \*\*rval\_ptr ) 】

返回值：

* 成功：返回0
* 失败：返回错误编号

参数：

* 参数1：指定等待的线程的ID
* 参数2：

填NULL：获取等待线程的终止信息，如果对线程的终止信息并不敢兴趣，可以设置为NULL。

非空：如果线程简单地从它的返回例程中返回，rval\_ptr 就包含返回码。

如果线程被取消，由rval\_ptr指定的内存单元就被设置为PTHREAD\_CANCELED

【休眠函数】

* sleep()单位为秒
* Sleep()单位为毫秒
* usleep()单位为微秒

## 实验2 在Liunx下创建（fork）一个子进程，实验wait/exit函数

* 设计1个或2个程序，实现下面的效果（不一定同时实现），也不一定每个程序都要调用wait或/和exit。
* 效果1：父进程不用wait函数，让父进程先于子进程结束，子进程进入死循环或较长时间的循环，观察父子进程的进程ID和父进程ID。

程序中printf各进程的进程号和父进程号。注意，父进程和子进程的输出请给出相应的提示字符串以便相互区分，后同。

同时，用ps命令显示进程列表，观察指定进程的进程ID和父进程ID，和printf输出的这些ID是否一致，并解释。

* 效果2：父进程用wait函数。子进程休眠5秒，父进程不休眠。子进程用exit返回参数。父进程中printf子进程返回的参数。

### 1 效果1的实现

#### 1.1 代码编写（C语言）

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include<stdio.h>

#include<sys/types.h>

#include<sys/wait.h>

int main()

{

pid\_t id=fork();

if(id<0)

{

printf("fork error!\n");

}

else if(id==0)

{

printf("我是子进程（进程号：%d）,父进程号：%d\n",getpid(),getppid());

sleep(6);

}

else if(id>0)

{

printf(“我是父进程（进程号：%d），父进程号：%d\n”,getpid(),getppid());

sleep(5);

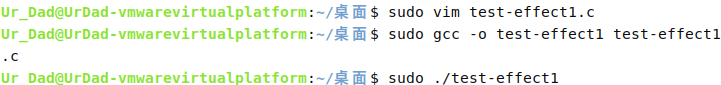
exit(0);

}

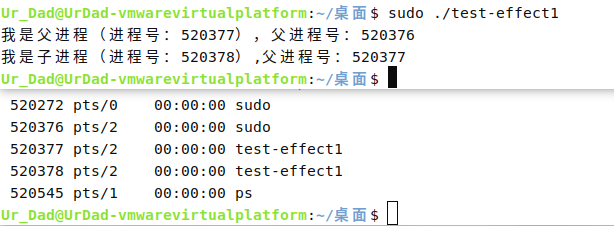
return 0;

}

#### 1.2 编译运行



#### 1.3 结果展示



### 2 效果2的实现

#### 2.1 代码编写（C语言）

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include<stdio.h>

#include<sys/types.h>

#include<sys/wait.h>

int main()

{

pid\_t id=fork();

if(id<0)

{

printf("fork error!\n");

return 1;

}

if(id==0)

{

printf("child\n");

sleep(5);

exit(2);

}

if(id>0)

{

printf("father\n");

wait(NULL);

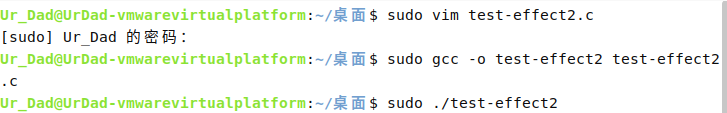
printf("子进程返回的参数为2\n");

}

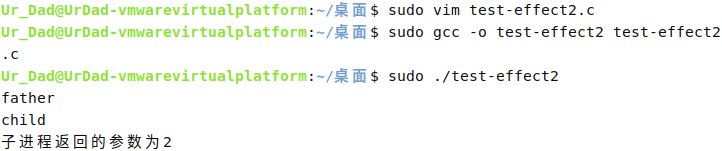
return 0;

}

#### 2.2 编译运行



#### 2.3 结果展示



### 总结

【函数 pid\_t fork(void) 】

返回值：

* 如果成功创建一个子进程，对父进程来说返回子进程ID
* 如果成功创建一个子进程，对子进程来说返回值为0
* 如果创建失败，返回值为-1

【函数 void exit(int status)】

执行return n等同于执行exit(n)

【函数 pid\_t wait(int \*status)】

**使用wait让父进程阻塞等待，只有子进程运行完毕退出后，父进程才退出。**

返回值：

* 成功：返回被等待进程pid
* 失败：返回-1

参数：

输出型参数，获取子进程退出状态，不关心则可以设置成NULL

## 实验3 在Windows下，利用线程实现并发画圆画方

* 提示1：圆心，半径，颜色，正方形中心，边长，颜色自己确定。
* 提示2：圆和正方形边界建议都取720个点。为直观展示绘制过程，每个点绘制后睡眠0.2秒~0.5秒。
* 提示3：建议使用VS和MFC或QT对话框类型程序来绘制窗口和图形。

#### 1 准备工作

在Visual studio 2022环境下进行编译，需提前下好Easy绘图库。

#### 2 代码编写（C++）

#include<graphics.h>

#include<stdio.h>

#include<Windows.h>

#include<math.h>

#define pi 3.1415926535897932

//定义画方的线程 顺时针画

DWORD WINAPI drawSquare(LPVOID)

{

//分为720个点 180\*4

for (int i = 0; i < 180; i++)

{

putpixel(50 + i, 50, YELLOW);

Sleep(30);

}

for (int i = 0; i < 180; i++)

{

putpixel(50 + 180, 50 + i, YELLOW);

Sleep(30);

}

for (int i = 0; i < 180; i++)

{

putpixel(50 + 180 - i, 50 + 180, YELLOW);

Sleep(30);

}

for (int i = 0; i < 180; i++)

{

putpixel(50, 50 + 180 - i, YELLOW);

Sleep(30);

}

return 0;

}

//定义画圆的函数

DWORD WINAPI drawCircle(LPVOID)

{

//同样分为720个点

for (int i = 0; i < 720; i++)

{

putpixel(350 + 100 \* cos(-pi / 2 + (double)((i \* pi) / 360)), 140 + 100 \* sin(-pi / 2 + (double)((i \* pi) / 360)), WHITE);

Sleep(30);

}

return 0;

}

int main()

{

// 初始化图形模式

initgraph(640, 480);

HANDLE hThread[2];//记录新线程句柄

DWORD threadID;//记录线程ID

if ((hThread[0] = CreateThread(NULL, 0, drawSquare, 0, 0, &threadID)) == NULL)

{

printf("线程创建失败!");

}

if ((hThread[1] = CreateThread(NULL, 0, drawCircle, 0, 0, &threadID)) == NULL)

{

printf("线程创建失败!");

}

//等待所有线程结束

WaitForMultipleObjects(2, hThread, TRUE, INFINITE);

CloseHandle(hThread[0]);

CloseHandle(hThread[1]);

getchar();

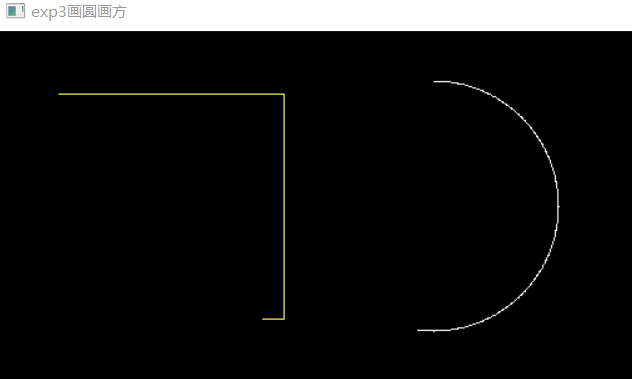
// 关闭图形模式

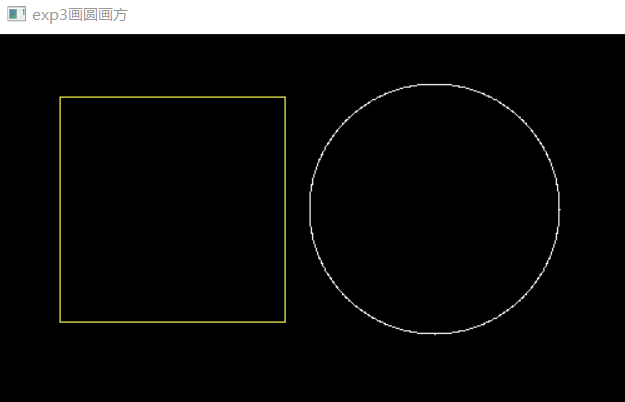
closegraph();

return 0;

}

#### 3 结果展示





## 实验4 在Windows下利用线程实现“生产者-消费者”同步控制

* 提示1：使用数组（10个元素）代替缓冲区。2个输入线程产生产品（随机数）存到数组中；3个输出线程从数组中取数输出。
* 提示2：Windows使用临界区对象和信号量对象，主要函数EnterCriticalSection | LeaveCriticalSection |WaitForSingleObject |ReleaseSemaphore
* 提示3：Linux使用互斥锁对象和轻量级信号量对象，主要函数:sem\_wait()，sem\_post(),pthread\_mutex\_lock(),pthread\_mutex\_unlock()
* 提示4：生产者1的数据：1000-1999（每个数据随机间隔100ms-1s)，生产者2的数据：2000-2999（每个数据随机间隔100ms-1s)
* 提示5：消费者每休眠100ms-1s的随机时间消费一个数据。
* 提示6：屏幕打印（或日志文件记录）每个数据的生产和消费记录。

#### 1 代码编写(C语言)

#include <windows.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define BUFFER\_SIZE 10

#define PRODUCER\_COUNT 2

#define CONSUMER\_COUNT 3

int buffer[BUFFER\_SIZE] = { 0 };

int in = 0, out = 0;

CRITICAL\_SECTION cs;

HANDLE semEmpty, semFull;

DWORD WINAPI Producer(LPVOID id) {

int producerId = (int)id;

while (TRUE) {

WaitForSingleObject(semEmpty, INFINITE);

EnterCriticalSection(&cs);

int value = (producerId == 1) ? (rand() % 1000 + 1000) : (rand() % 1000 + 2000);

buffer[in] = value;

printf("Producer %d produced %d\n", producerId, value);

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

LeaveCriticalSection(&cs);

ReleaseSemaphore(semFull, 1, NULL);

Sleep(rand() % 900 + 100); // Random delay between 100ms and 1s

}

return 0;

}

DWORD WINAPI Consumer(LPVOID id) {

while (TRUE) {

WaitForSingleObject(semFull, INFINITE);

EnterCriticalSection(&cs);

int value = buffer[out];

printf("Consumer %d consumed %d\n", (int)id, value);

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

LeaveCriticalSection(&cs);

ReleaseSemaphore(semEmpty, 1, NULL);

Sleep(rand() % 900 + 100); // Random delay between 100ms and 1s

}

return 0;

}

int main() {

InitializeCriticalSection(&cs);

semEmpty = CreateSemaphore(NULL, BUFFER\_SIZE, BUFFER\_SIZE, NULL);

semFull = CreateSemaphore(NULL, 0, BUFFER\_SIZE, NULL);

HANDLE producerThreads[PRODUCER\_COUNT];

for (int i = 0; i < PRODUCER\_COUNT; i++) {

producerThreads[i] = CreateThread(NULL, 0, Producer, (void\*)i, 0, NULL);

}

HANDLE consumerThreads[CONSUMER\_COUNT];

for (int i = 0; i < CONSUMER\_COUNT; i++) {

consumerThreads[i] = CreateThread(NULL, 0, Consumer, (void\*)i, 0, NULL);

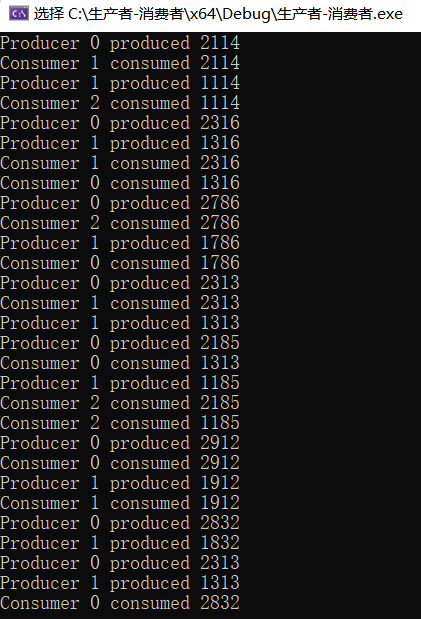
}

WaitForMultipleObjects(PRODUCER\_COUNT, producerThreads, TRUE, INFINITE);

WaitForMultipleObjects(CONSUMER\_COUNT, consumerThreads, TRUE, INFINITE);

}

#### 2 结果展示



## 实验6 在Linux下模拟哲学家就餐，提供死锁和非死锁解法

* 提示 1：同时提供提供可能会带来死锁的解法和不可能死锁的解法。
* 提示 2：可能会带来死锁的解法参⻅课件。Windows 尝试使用临界区对象（ EnterCriticalSection ， LeaveCriticalSection ）；Linux 尝 试 使 用 互 斥 锁(pthread\_mutex\_lock, pthread\_mutex\_unlock)
* 提示 3：完全不可能产生死锁的解法，例如：尝试拿取两只筷子，两只都能拿则拿，否则都不拿。
  + Windows 尝 试 使 用 WaitForMultipleObjects, WaitForSingleObject 和互斥量对象 ReleaseMutex 等相关函数) 。
  + Linux 尝试使用互斥锁 pthread\_mutex\_lock，pthread\_mutex\_trylock 等函数。
* 提示 4：为增强随机性，各状态间维持 100ms-500ms 内的随机时⻓。
* 提示 5：[可选] 图形界面显示哲学家取筷，吃饭，放筷，思考等状态。

### 1 实验思路

#### 1.1 问题描述

5个位哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有5个盘子和5双筷子，他们的生活方式是交替的进行思考和进餐。平时哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐完成之后，放下筷子继续思考。

#### 1.2 解决死锁的办法

* 法1：从0开始依次对每个哲学家进行编号，并规定：奇数编号哲学家先拿左边筷子再拿右边筷子，偶数编号哲学家先拿右边筷子再拿左边筷子。
* 法2：再创建一个任务，哲学家拿起筷子时向该任务发起申请，由该任务对当前筷子的分配情况进行判断，判定系统是否由安全状态向不安全状态转换，从而允许或拒绝该次申请。

### 2 可能死锁的解法

##### 2.1 代码编写

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include <unistd.h>

//规定奇数哲学家先拿左边筷子再拿右边筷子，偶数哲学家先拿右边筷子再拿左边筷子

pthread\_mutex\_t chopstick[5];

void \*func(void\*);// 线程函数

void init();

int Random(void) {

int a = time(NULL);

srand(a);

return (rand() % 400 + 100);

}//每种状态维持的随机时间

int main(){

init();

pthread\_t p1,p2,p3,p4,p5; // 定义哲学家

pthread\_create(&p1,NULL,func,"0");

pthread\_create(&p2,NULL,func,"1");

pthread\_create(&p3,NULL,func,"2");

pthread\_create(&p4,NULL,func,"3");

pthread\_create(&p5,NULL,func,"4");

pthread\_join(p1,NULL);

pthread\_join(p2,NULL);

pthread\_join(p3,NULL);

pthread\_join(p4,NULL);

pthread\_join(p5,NULL);

return 0;

}

void init(){

int i = 0;

for(;i < 5; i++){

chopstick[i] = (pthread\_mutex\_t)PTHREAD\_MUTEX\_INITIALIZER;

}

}

void \*func(void \*arg){

int i = 0;

int num = atoi((char \*)arg);

do{

//思考

int time =Random();

printf("哲学家%d开始思考，将思考%dms\n",num + 1,time); usleep(time\*1000);

//吃饭

if(num % 2 == 0){ //偶数哲学家

pthread\_mutex\_lock(&chopstick[(num + 1) % 5]);

pthread\_mutex\_lock(&chopstick[num]);

time=Random();

printf("哲学家%d开始吃饭，将吃饭%dms\n",num + 1,time);

usleep(time\*1000);

} else{ //奇数哲学家

pthread\_mutex\_lock(&chopstick[num]);

pthread\_mutex\_lock(&chopstick[(num + 1) % 5]);

time=Random();

printf("哲学家%d开始吃饭，将吃饭%dms\n",num + 1,time);

usleep(time\*1000);

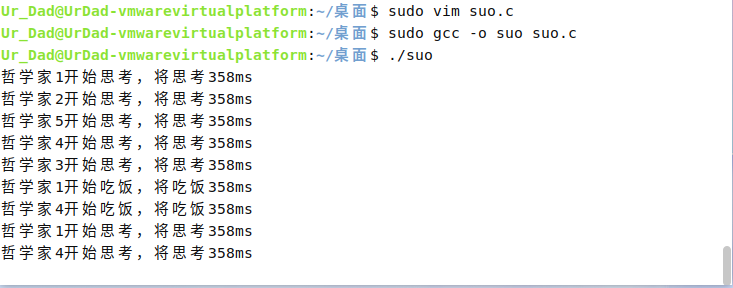
}

i++;

} while(i < 30);

}

##### 2.2结果展示



### 3 不可能死锁的解法

#### 3.1 奇偶编号避免死锁

##### 3.1.1 代码编写

为解决哲学家就餐问题中出现死锁的情况，从0开始依次对每个哲学家进行编号，并规定：奇数编号哲学家先拿左边筷子再拿右边筷子，偶数编号哲学家先拿右边筷子再拿左边筷子，这样就不会出现死锁现象了，奇偶编号预防死锁发生代码如下所示：

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include <unistd.h>

//规定奇数哲学家先拿左边筷子再拿右边筷子，偶数哲学家先拿右边筷子再拿左边筷子

pthread\_mutex\_t chopstick[5];

void \*func(void\*);// 线程函数

void init();

int Random(void) {

int a = time(NULL);

srand(a);

return (rand() % 400 + 100);

}//每种状态维持的随机时间

int main(){

init();

pthread\_t p1,p2,p3,p4,p5; // 定义哲学家

pthread\_create(&p1,NULL,func,"0");

pthread\_create(&p2,NULL,func,"1");

pthread\_create(&p3,NULL,func,"2");

pthread\_create(&p4,NULL,func,"3");

pthread\_create(&p5,NULL,func,"4");

pthread\_join(p1,NULL);

pthread\_join(p2,NULL);

pthread\_join(p3,NULL);

pthread\_join(p4,NULL);

pthread\_join(p5,NULL);

return 0;

}

void init(){

int i = 0;

for(;i < 5; i++){

chopstick[i] = (pthread\_mutex\_t)PTHREAD\_MUTEX\_INITIALIZER;

}

}

void \*func(void \*arg){

int i = 0;

int num = atoi((char \*)arg);

do{

//思考

int time =Random();

printf("哲学家%d开始思考，将思考%dms\n",num + 1,time);

usleep(time\*1000);

//吃饭

if(num % 2 == 0){ //偶数哲学家

pthread\_mutex\_lock(&chopstick[(num + 1) % 5]);

pthread\_mutex\_lock(&chopstick[num]);

int time=Random();

printf("哲学家%d开始吃饭，将吃饭%dms\n",num + 1,time);

usleep(time\*1000);

pthread\_mutex\_unlock(&chopstick[(num + 1) % 5]);

pthread\_mutex\_unlock(&chopstick[num]);

} else{ //奇数哲学家

pthread\_mutex\_lock(&chopstick[num]);

pthread\_mutex\_lock(&chopstick[(num + 1) % 5]);

int time=Random();

printf("哲学家%d开始吃饭，将吃饭%dms\n",num + 1,time);

usleep(time\*1000);

pthread\_mutex\_unlock(&chopstick[num]);

pthread\_mutex\_unlock(&chopstick[(num + 1) % 5]);

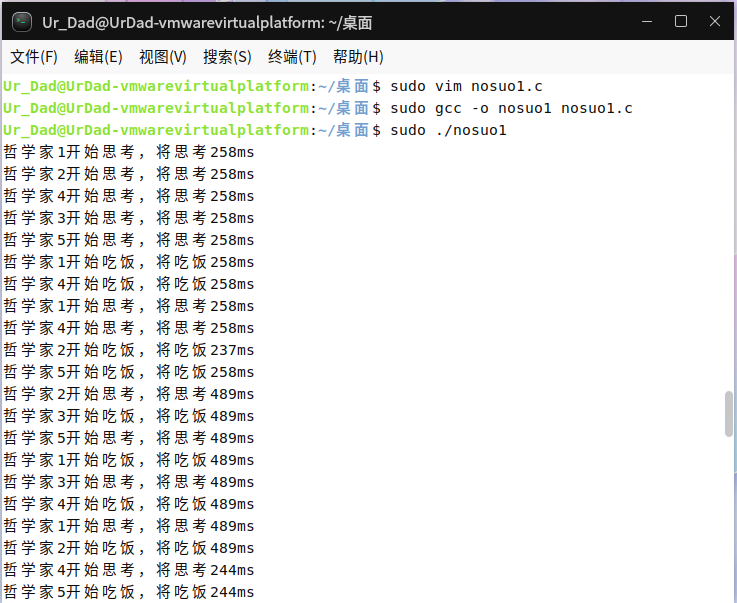
}

i++;

} while(i < 30);

}

##### 3.1.2 结果展示



#### 3.2 trylock避免死锁

##### 3.2.1 代码编写

哲学家如果无法同时拿起两支筷子，则放下已经拿起的筷子，等待一段时间再尝试，利用非阻塞操作pthread\_mutex\_trylock实现对能否拿起筷子的判断。破坏请求和保持条件预防死锁代码如下所示：

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include <unistd.h>

pthread\_mutex\_t chopstick[5];

void \*philosopher1(void);// 哲学家1

void \*philosopher2(void);// 哲学家2

void \*philosopher3(void);// 哲学家3

void \*philosopher4(void);// 哲学家4

void \*philosopher5(void);// 哲学家5

void init();

long int random(void) {

int a = time(NULL);

srand(a);

return (rand() % 400 + 100);

}

int main(){

init();//初始化互斥量

pthread\_t p1,p2,p3,p4,p5;

pthread\_create(&p1,NULL,(void\*)philosopher1,NULL);

pthread\_create(&p2,NULL,(void\*)philosopher2,NULL); pthread\_create(&p3,NULL,(void\*)philosopher3,NULL);

pthread\_create(&p4,NULL,(void\*)philosopher4,NULL);

pthread\_create(&p5,NULL,(void\*)philosopher5,NULL);

pthread\_join(p1,NULL);

pthread\_join(p2,NULL);

pthread\_join(p3,NULL);

pthread\_join(p4,NULL);

pthread\_join(p5,NULL);

}

void init(){

int i = 0;

for(;i < 5; i++){

chopstick[i] = (pthread\_mutex\_t)PTHREAD\_MUTEX\_INITIALIZER;

}

}

void \*philosopher1(void){

int i = 0;

do{

int time1=random();

printf("哲学家1开始思考，将思考%dms\n",time1);

usleep(time1\*1000);

pthread\_mutex\_trylock(&chopstick[0]);

if(pthread\_mutex\_trylock(&chopstick[1]) != 0){

pthread\_mutex\_unlock(&chopstick[0]);

continue;

}

time1=random();

printf("哲学家1开始吃饭，将吃饭%dms\n",time1);

usleep(time1\*1000);

pthread\_mutex\_unlock(&chopstick[0]);

pthread\_mutex\_unlock(&chopstick[1]);

i++;

} while(i < 30);

}

void \*philosopher2(void){

int i = 0;

do{

int time2=random();

printf("哲学家2开始思考，将思考%dms\n",time2);

usleep(time2\*1000);

pthread\_mutex\_trylock(&chopstick[1]);

if(pthread\_mutex\_trylock(&chopstick[2]) != 0){

pthread\_mutex\_unlock(&chopstick[1]);

continue;

}

time2 =random();

printf("哲学家2开始吃饭，将吃饭%dms\n",time2);

usleep(time2\*1000);

pthread\_mutex\_unlock(&chopstick[1]);

pthread\_mutex\_unlock(&chopstick[2]);

i++;

} while(i < 30);

}

void \*philosopher3(void){

int i = 0;

do{

int time3=random();

printf("哲学家3开始思考，将思考%dms\n",time3);

usleep(time3\*1000);

pthread\_mutex\_trylock(&chopstick[2]);

if(pthread\_mutex\_trylock(&chopstick[3]) != 0){

pthread\_mutex\_unlock(&chopstick[2]);

continue;

}

time3=random();

printf("哲学家3开始吃饭，将吃饭%dms\n",time3);

usleep(time3\*1000);

pthread\_mutex\_unlock(&chopstick[2]);

pthread\_mutex\_unlock(&chopstick[3]);

i++;

} while(i < 30);

}

void \*philosopher4(void){

int i = 0;

do{

int time4=random();

printf("哲学家4开始思考，将思考%dms\n",time4); usleep(time4\*1000);

pthread\_mutex\_trylock(&chopstick[3]);

if(pthread\_mutex\_trylock(&chopstick[4]) != 0){

pthread\_mutex\_unlock(&chopstick[3]);

continue;

}

time4=random();

printf("哲学家4开始吃饭，将吃饭%dms\n",time4);

usleep(time4\*1000);

pthread\_mutex\_unlock(&chopstick[3]);

pthread\_mutex\_unlock(&chopstick[4]);

i++;

} while(i < 30);

}

void \*philosopher5(void){

int i = 0;

do{

int time5=random();

printf("哲学家5开始思考，将思考%dms\n",time5);

usleep(time5\*1000);

pthread\_mutex\_trylock(&chopstick[4]); if(pthread\_mutex\_trylock(&chopstick[0]) != 0){

pthread\_mutex\_unlock(&chopstick[4]);

continue;

}

time5=random();

printf("哲学家5开始吃饭，将吃饭%dms\n",time5);

usleep(time5\*1000);

pthread\_mutex\_unlock(&chopstick[4]);

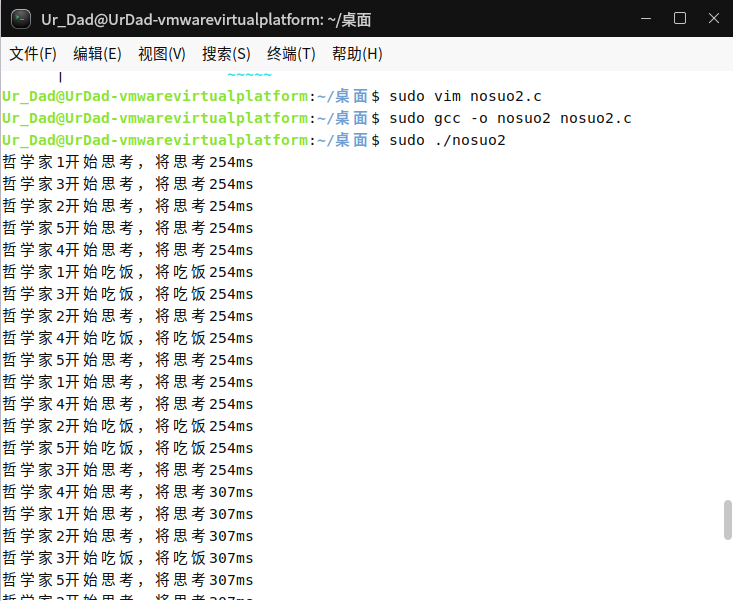
pthread\_mutex\_unlock(&chopstick[0]);

i++;

} while(i < 30);

}

##### 3.2.2 结果展示



### 总结

【int pthread\_mutex\_trylock()】

通俗的说法：**trylock就是尝试锁一下，锁不到就拉倒，不会影响自己进行下一步操作。lock就比较犟，锁不到的话，我就等着，等到我能锁了，再进行一下步操作。**

返回值：锁成功则返回0，其他值都表示锁失败。