

**课 程 实 验 报 告**

**课程名称： 操作系统原理实验**

**专业班级： 软件2202班**

**学 号： U202217216**

**姓 名： 郑德凯**

**报告日期： 2024/4/11**

**软件学院**

目录

[实验二：第4章进程管理 2](#_Toc163739595)

[一、 实验目的 2](#_Toc163739596)

[二、 实验内容 2](#_Toc163739597)

[三、实验要求 2](#_Toc163739598)

[四、实验指南 2](#_Toc163739599)

[（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](#_Toc163739600)

[（2）在Liunx下创建（fork）一个子进程，实验wait/exit函数，理解父子进程的并发过程 2](#_Toc163739601)

[（3）在Windows/Linux下，利用线程实现并发画圆画方 3](#_Toc163739602)

[（4）在Windows或Linux下利用线程实现“生产者-消费者”同步控制 3](#_Toc163739603)

[（5）在Linux下利用信号机制(signal)实现进程通信 3](#_Toc163739604)

[（6）在Windows或Linux下模拟哲学家就餐，提供死锁和非死锁解法 4](#_Toc163739605)

[（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 4](#_Toc163739606)

[五、实验过程 4](#_Toc163739607)

[（1）在Linux/Windows下创建2个线程A和B，循环输出数据或字符串 4](#_Toc163739608)

[（2）在Liunx下创建（fork）一个子进程，实验wait/exit函数，理解父子进程的并发过程 6](#_Toc163739609)

[（3）在Windows/Linux下，利用线程实现并发画圆画方。 9](#_Toc163739610)

[（4）在Windows或Linux下利用线程实现“生产者-消费者”同步控制 9](#_Toc163739611)

[（5）在Linux下利用信号机制(signal)实现进程通信 12](#_Toc163739612)

[（6）Windows或Linux下模拟哲学家就餐，提供死锁和非死锁解法 12](#_Toc163739613)

[（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 12](#_Toc163739614)

# 实验二：第4章进程管理

## 实验目的

* 理解进程/线程的概念和应用编程过程；
* 理解进程/线程的同步机制和应用编程；
* 掌握和推广国产操作系统（推荐银河麒麟或优麒麟，建议）

## 实验内容

* 在Linux/Windows下创建2个线程A和B，循环输出数据或字符串。
* 在Liunx下创建（fork）一个子进程，实验wait/exit函数
* 在Windows/Linux下，利用线程实现并发画圆画方。
* 在Windows或Linux下利用线程实现“生产者-消费者”同步控制
* 在Linux下利用信号机制(signal)实现进程通信
* 在Windows或Linux下模拟哲学家就餐，提供死锁和非死锁解法。
* 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。

## 四、实验指南

### （1）在Linux/Windows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线程库或CreateThread函数
* 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”标示是哪个线程输出的，并注意格式化输出信息。

### （2）在Liunx下创建（fork）一个子进程，实验wait/exit函数，理解父子进程的并发过程

* 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。

### （3）在Windows/Linux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。

### （4）在Windows或Linux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。

### （5）在Linux下利用信号机制(signal)实现进程通信

* 提示1：父进程创建(fork)子进程，并让子进程进入死循环。
* 提示2：子进程每隔2秒输出”I am Child Process, alive !\n”
* 提示3：父进程询问用户”To terminate Child Process. Yes or No? \n”要求用户从键盘回答Y或N.若用户回答N，延迟2秒后再提问.
* 提示4：若用户回答Y，向子进程发送用户信号，让子进程结束。
* 提示5：子进程结束之前打印字符串：”Bye,Wolrd !\n”
* 提示6：函数：kill( )，signal( )，利用用户信号，编写信号处理函数

### （6）在Windows或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见课件。Windows尝试使用临界区对（EnterCriticalSection，LeaveCriticalSection）；Linux尝试使用互斥(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。
* 提示4：为增强随机性，各状态间维持100ms-500ms内的随机时长。
* 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。

### （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

要求：编写应用程序Hello.c，调用fork创建进程，在内核中跟踪该新建子进程的fork过程和显示与调度策略相关的PCB成员变量。

* 提示1：编写应用程序Hello.c，在其中调用fork创建子进程(功能不限)，打印出父子进程的ID号；
* 提示2：在内核中合适的位置（譬如do\_fork函数内的某处）用printk输出“当前正创建的进程对应cmd，进程ID和父进程ID”等调试信息。
* 提示3：为避免do\_fork函数频繁地输出上述调试信息，须限定仅在Hello程序中调用fork时才输出上述调试信息，请思考要如何实现。
  + 参考方法：内核设计全局变量bool flag和系统调用SetDebug(bool)，SetDebug可以修改flag的值为true或false。在Hello程序中的调用fork函数前后，分别调用SetDebug(true)和SetDebug(false)修改flag。在printk调试信息时检查flag以确定是否要使用printk输出调试信息

## 五、实验过程

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

采用pthread库创建线程，代码如下：

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

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

int count1 = 1;

int count2 = 1000;

void\* threadA(void\* arg) {

while (count1 <= 1000) {

pthread\_mutex\_lock(&mutex);

printf("A: %04d\n", count1++);

pthread\_mutex\_unlock(&mutex);

usleep(200000); // 0.2秒

}

return NULL;

}

void\* threadB(void\* arg) {

while (count2 >= 1) {

pthread\_mutex\_lock(&mutex);

printf("B: %04d\n", count2--);

pthread\_mutex\_unlock(&mutex);

usleep(200000); //

}

return NULL;

}

int main() {

pthread\_t tidA, tidB;

pthread\_create(&tidA, NULL, threadA, NULL);

pthread\_create(&tidB, NULL, threadB, NULL);

pthread\_join(tidA, NULL);

pthread\_join(tidB, NULL);

return 0;

}

运行结果：图形用户界面, 应用程序, Word

描述已自动生成

### （2）在Liunx下创建（fork）一个子进程，实验wait/exit函数，理解父子进程的并发过程

#### 1.效果1

1. 效果1代码：

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

return 1;

} else if (pid == 0) {

printf("Child process: PID=%d, PPID=%d\n", getpid(), getppid());

while(1);

} else {

printf("Parent process: PID=%d, PPID=%d\n", getpid(), getppid());

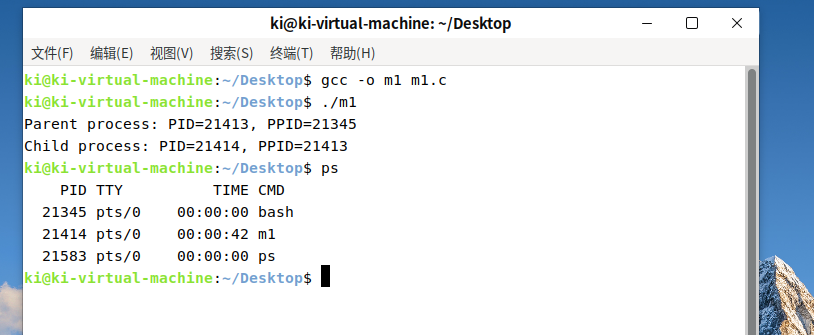
sleep(1);

}

return 0;

}

1. 运行结果：



1. 结果解释：

21414是正在运行的效果1程序的进程ID。由于在效果1的程序中，父进程先于子进程结束，而子进程进入了一个无限循环中。这意味着在父进程休眠期间，子进程会持续运行，直到父进程结束或者被其他方式终止。ps命令显示了子进程的进程ID，因为子进程仍在运行，而父进程已经在休眠1秒后结束

1. 效果2代码

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

return 1;

} else if (pid == 0) {

printf("Child process: PID=%d, PPID=%d\n", getpid(), getppid());

sleep(5);

return 123;

} else {

printf("Parent process: PID=%d, PPID=%d\n", getpid(), getppid());

int status;

wait(&status);

printf("Child process returned: %d\n", WEXITSTATUS(status));

}

return 0;

}

1. 运行结果

文本

描述已自动生成

### （3）在Windows/Linux下，利用线程实现并发画圆画方。

环境：win11，vscode，easyx图形库

代码：

#include <graphics.h>

#include <conio.h>

DWORD Circle\_Bresenham(LPVOID lpParam)

{

int x = 320, y = 240, r = 90, color = LIGHTBLUE;

int tx = 0, ty = r, d = 3 - 2 \* r;

while (tx <= ty)

{

putpixel(x + tx, y + ty, color);

putpixel(x + tx, y - ty, color);

putpixel(x - tx, y + ty, color);

putpixel(x - tx, y - ty, color);

putpixel(x + ty, y + tx, color);

putpixel(x + ty, y - tx, color);

putpixel(x - ty, y + tx, color);

putpixel(x - ty, y - tx, color);

if (d < 0) /

d += 4 \* tx + 6;

else

d += 4 \* (tx - ty) + 10, ty--;

Sleep(100);

tx++;

}

return 0;

}

DWORD Square(LPVOID lpParam)

{

int x = 320, y = 240, r = 90, color = RED;

int tx = 0, ty = r;

while (tx <= ty)

{

putpixel(x + tx, y + ty, color);

putpixel(x + tx, y - ty, color);

putpixel(x - tx, y + ty, color);

putpixel(x - tx, y - ty, color);

putpixel(x + ty, y + tx, color);

putpixel(x + ty, y - tx, color);

putpixel(x - ty, y + tx, color);

putpixel(x - ty, y - tx, color);

Sleep(70);

tx++;

}

return 0;

}

int main()

{

HANDLE hThread[2];

DWORD ThreadID;

initgraph(640, 480);

hThread[0] = CreateThread(NULL, 0, Circle\_Bresenham, NULL, 0, &ThreadID);

hThread[1] = CreateThread(NULL, 0, Square, NULL, 0, &ThreadID);

WaitForMultipleObjects(2, hThread, TRUE, INFINITE);

\_getch();

closegraph();

return 0;

}

运行结果：

形状, 维恩图, 圆圈

描述已自动生成

### （4）在Windows或Linux下利用线程实现“生产者-消费者”同步控制

任务代码：

#include<pthread.h>

#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<time.h>

#include<sys/syscall.h>

#include<semaphore.h>

int share[10]; //共享缓冲区

int indexc=0; //生产者处理的标号(临界资源)

int indexp=0; //消费者处理的标号(临界资源)

sem\_t is\_empty; //缓冲区中存在空位

sem\_t is\_full; //缓冲区已满

pthread\_mutex\_t mutexc = PTHREAD\_MUTEX\_INITIALIZER; //互斥锁

pthread\_mutex\_t mutexp = PTHREAD\_MUTEX\_INITIALIZER; //互斥锁

void\* consume(void\* arg){

int add=(int)arg;

while(1){

sem\_wait(&is\_full);

pthread\_mutex\_lock(&mutexc);

struct timespec ts;

clock\_gettime(CLOCK\_MONOTONIC, &ts);

int data=ts.tv\_nsec%1000+add;

share[indexc]=data;

float sleepTime=0.001\*(ts.tv\_nsec%900)+0.1;

printf("生产者%ld填第%d块：%d，即将沉睡%fs\n",syscall(SYS\_gettid),indexc,data,sleepTime);

indexc++;

if(indexc>9)indexc-=10;

pthread\_mutex\_unlock(&mutexc);

sem\_post(&is\_empty);

sleep(sleepTime);

}

}

void\* produce(void){

while(1){

sem\_wait(&is\_empty);

struct timespec ts;

clock\_gettime(CLOCK\_MONOTONIC, &ts);

float sleepTime=0.001\*(ts.tv\_nsec%900)+0.1;

pthread\_mutex\_lock(&mutexp);

printf("消费者%ld取第%d块：%d, 即将沉睡%fs\n",syscall(SYS\_gettid),indexp,share[indexp],sleepTime);

indexp++;

if(indexp>9)indexp-=10;

pthread\_mutex\_unlock(&mutexp);

sem\_post(&is\_full);

sleep(sleepTime);

}

}

int main(){

sem\_init(&is\_empty,0,0);//初值为0，用于同步生产者和消费者

sem\_init(&is\_full,0,9); //初值为9，表示初始缓冲区填10个满

pthread\_t idc1,idc2;

pthread\_t idp1,idp2,idp3;

pthread\_create(&idc1,NULL,(void\*)consume,(void\*)1000);

pthread\_create(&idc2,NULL,(void\*)consume,(void\*)2000);

pthread\_create(&idp1,NULL,(void\*)produce,NULL);

pthread\_create(&idp2,NULL,(void\*)produce,NULL);

pthread\_create(&idp3,NULL,(void\*)produce,NULL);

pthread\_join(idc1,NULL);

pthread\_join(idc2,NULL);

pthread\_join(idp1,NULL);

pthread\_join(idp2,NULL);

pthread\_join(idp3,NULL);

return 0;

}

图形用户界面, 应用程序

描述已自动生成

图中的生产者是58715、58714，消费者是58717、58718。沉睡时间为100ms到1s，填入数据随机。使用了两个信号量is\_empty/is\_full，两个互斥锁mutexc和mutexp。其中信号量用于监测缓冲区是否还能继续填入或已满，互斥锁分别用于生产者、消费者的互斥使用临界资源indexc和indexp操作。

### （5）在Linux下利用信号机制(signal)实现进程通信

### （6）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

1.在系统内核中添加set\_debug系统调用和debug\_flag全局变量：

图形用户界面, 文本, 应用程序

描述已自动生成

图片包含 文本

描述已自动生成



文本

低可信度描述已自动生成

2.在fork.c文件中修改kernel\_clone函数

文本

中度可信度描述已自动生成

1. 编写hello.c文件

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/syscall.h>

#include <stdbool.h>

#define \_\_NR\_set\_debug 335

long set\_debug(bool flag) {

return syscall(\_\_NR\_set\_debug, flag);

}

int main() {

set\_debug(true);

pid\_t pid;

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

pid = fork();

if (pid < 0) {

fprintf(stderr, "Fork failed\n");

return 1;

} else if (pid == 0) {

printf("Child process ID: %d, Parent process ID: %d\n", getpid(), getppid());

} else {

printf("Parent process ID: %d, Child process ID: %d\n", getpid(), pid);

}

set\_debug(false);

return 0;

}

1. 运行结果

