《操作系统原理》实验报告

姓名	侯皓斐	学号	U202010851	专业班级	软工 2003 班	时间	2022.5.1

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；

二、实验内容

1）在 Linux 下创建一对父子进程。

2）在 Linux 下创建 2 个线程 A 和 B，循环输出数据或字符串。

3）在 Windows 下创建线程 A 和 B，循环输出数据或字符串。

4）在 Linux 下创建一对父子进程，实验 wait 同步函数。

5）在 Windows 下利用线程实现并发画圆/画方。

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

7）在 Linux 下利用信号机制实现进程通信。

8）在 Windows 或 Linux 下模拟哲学家就餐，提供死锁和非死锁解法。

三、实验过程

3.1 Linux 下父子进程的创建

开发环境：银河麒麟操作系统 v10 桌面版

Linux 使用 fork()创建进程，在子进程中 fork()函数返回值为 0。分别输出各自的进程号，父进程号和特别的提示字符串信息。让父进程提前结束或后结束，观察子进程的父进程 ID。使用 PS 命令查看进程列表信息，核对进程号，父进程号。

编写程序 createProcess.c，使用 sleep 函数暂时挂起父进程 10s，使子进程提前结束。在子进程挂起的时间内，使用 ps 命令，观察父进程 ID。

createProcess.c

```c
#include<unistd.h>
#include<stdio.h>
int main()
{
    printf("\"#\"代表父进程输出信息；\"@\"代表子进程输出信息。\n");
    pid_t pid=fork();
    if(pid){
        printf("# 1.进程号：%d\n",getpid());
        printf("# 2.其父进程的进程号：%d\n",getppid());
        printf("# 3.暂时挂起父进程10s\n");
        sleep(10);
        printf("# 4.父进程结束\n");
    }else{
        printf("@ 1.进程号：%d\n",getpid());
        printf("@ 2.父进程未结束时，子进程的父进程进程号：%d\n",getppid());
        printf("@ 3.暂时挂起子进程5s\n");
        sleep(5);
        printf("@ 4.子进程结束\n");
    }
    return 0;
}
```

```
moonlight@moonlight-VMware:~/test2/1$ ./createPr
"#"代表父进程输出信息；"@"代表子进程输出信息。
# 1.进程号：3736
# 2.其父进程的进程号：3497
# 3.暂时挂起父进程10s
@ 1.进程号：3737
@ 2.父进程未结束时，子进程的父进程进程号：3736
@ 3.暂时挂起子进程5s
@ 4.子进程结束
# 4.父进程结束
moonlight@moonlight-VMware:~/test2/1$
```

```
root        3310       2  0 20:03 ?        00:00:00 [kworker/1:2-events]
root        3406       2  0 20:08 ?        00:00:00 [kworker/1:1-events]
moonlig+    3486       1  0 20:12 ?        00:00:00 /usr/bin/mate-terminal --wo
moonlig+    3497    3486  0 20:12 pts/0    00:00:00 bash
moonlig+    3625    2466  0 20:12 ?        00:00:00 [sh] <defunct>
root        3640       2  0 20:13 ?        00:00:00 [kworker/1:0+pm]
moonlig+    3704    3486  0 20:14 pts/1    00:00:00 bash
moonlig+    3736    3497  0 20:15 pts/0    00:00:00 ./createProcess.out
moonlig+    3737    3736  0 20:15 pts/0    00:00:00 ./createProcess.out
moonlig+    3739    3704  0 20:15 pts/1    00:00:00 ps -ef
moonlight@moonlight-VMware:~/test2/1$
```

编写程序 createProcess2.c，使用 sleep 函数暂时挂起子进程 10s，使父进程提前结束。在子进程挂起的时间内，使用 ps 命令，观察其父进程 ID。

createProcess2.c

```c
#include<unistd.h>
#include<stdio.h>
int main()
{
    printf("\"#\"代表父进程输出信息；\"@\"代表子进程输出信息。\n");
    pid_t pid=fork();
    if(pid){
        printf("# 1.进程号：%d\n",getpid());
            printf("# 2.其父进程的进程号：%d\n",getppid());
        printf("# 3.父进程结束\n\n");
    }else{
        printf("@ 1.进程号：%d\n",getpid());
        printf("@ 2.暂时挂起子进程10s：\n");
        sleep(10);
        printf("\n@ 3.父进程结束后，子进程的父进程进程号：%d\n",getppid());
        printf("@ 4.暂时挂起子进程10s：\n");
        sleep(10);
        printf("@ 5.子进程结束\n");
    }
    return 0;
}
```

```
moonlight@moonlight-VMware:~/test2/1$ ./createProcess2
"#"代表父进程输出信息；"@"代表子进程输出信息。
# 1.进程号：4124
# 2.其父进程的进程号：4061
# 3.父进程结束

@ 1.进程号：4125
@ 2.暂时挂起子进程10s:
moonlight@moonlight-VMware:~/test2/1$
@ 3.父进程结束后，子进程的父进程进程号：1
@ 4.暂时挂起子进程10s:
@ 5.子进程结束
```

```
root        2834       2  0 19:47 ?        00:00:00 [kworker/2:0-cgroup_dest
root        2837       2  0 19:47 ?        00:00:00 [kworker/0:0-cgroup_dest
moonlig+    2974       1  0 19:58 ?        00:00:07 /usr/bin/peony computer:
root        3011       2  0 19:58 ?        00:00:00 [kworker/u256:2-events_u
moonlig+    3012    1604  0 19:58 ?        00:00:00 /usr/libexec/gvfsd-netwo
moonlig+    3094    1604  0 19:59 ?        00:00:00 /usr/libexec/gvfsd-dnssd
root        3406       2  0 20:08 ?        00:00:00 [kworker/1:1-events]
moonlig+    3625    2466  0 20:12 ?        00:00:00 [sh] <defunct>
root        3640       2  0 20:13 ?        00:00:00 [kworker/1:0+usb_hub_wq]
root        3774       2  0 20:17 ?        00:00:00 [kworker/3:0-cgroup_dest
root        3793       2  0 20:18 ?        00:00:00 [kworker/1:2-events]
root        3806       2  0 20:19 ?        00:00:00 [kworker/u256:0-events_u
moonlig+    4049       1  0 20:23 ?        00:00:00 /usr/bin/mate-terminal -
moonlig+    4061    4049  0 20:23 pts/0    00:00:00 bash
moonlig+    4125       1  0 20:23 pts/0    00:00:00 ./createProcess2.out
moonlig+    4142    4049  0 20:23 pts/1    00:00:00 bash
```

3.2 Linux 线程实验

开发环境：银河麒麟操作系统 v10 桌面版

使用 pthread 线程库在 Linux 环境下，使用 pthread_create()函数创建线程，使用 pthread_join()使主线程等待线程执行结束。线程 A 递增输出 1-1000；线程 B 递减输出 1000-1。

打开(O) createThread.c ~/test2/2 保存(S)

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>

void *A() {
    for(int i = 1; i <= 1000; i++) {
        printf("A : %d\n", i);
        usleep(100000);
    }
    printf("A Thread End\n");
    pthread_exit(NULL);
}

void *B() {
    for(int i = 1000; i >= 1; i--) {
        printf("B : %d\n", i);
        usleep(100000);
    }
    printf("B Thread End\n");
    pthread_exit(NULL);
}

int main(){
    pthread_t t1,t2;
    pthread_create(&t1,NULL,A,NULL);
    pthread_create(&t2,NULL,B,NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

应注意，编译时要增加编译选项，加入线程库。

```
moonlight@moonlight-VMware:~/test2/2$ gcc createThread.c -lpthread
moonlight@moonlight-VMware:~/test2/2$ ./a.out
```

3.3 Windows 线程实验

开发环境：Windows10，dev-C++

使用 CreateThread 在 Windows 环境下，创建线程，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 秒输出一个数。

编写程序 createThread.c，编译后运行。

createThread.c

```c
#include<Windows.h>
#include<stdio.h>

DWORD A() {
    for(int i = 1; i <= 1000; i++) {
        printf("A : %d\n", i);
        Sleep(100);
    }
    return 0;
}

DWORD B () {
    for(int i = 1000; i >= 1; i--) {
        printf("B : %d\n", i);
        Sleep(100);
    }
    return 0;
}

int main()
{
    HANDLE hThread[2];
    DWORD  threadId1, threadId2;

    hThread[0] = CreateThread(NULL, 0, A, 0, 0, &threadId1);
    hThread[1] = CreateThread(NULL, 0, B, 0, 0, &threadId1);
    WaitForMultipleObjects(2, hThread, 1, INFINITE);
    return 0;
}
```

3.4 Linux 使用 wait 同步

开发环境：银河麒麟操作系统 v10 桌面版

使用 fork()在 Linux 环境下创建子进程，使用 exit(status)函数结束子进程，使用 wait(&status)函数使父进程等待子进程结束，同时接受并分析子进程返回参数。

同时应注意的是，status 并不是子进程的返回值。WIFEXITED(status) 这个宏用来指出子进程是否为正常退出的，如果是，它会返回一个非零值。而当 WIFEXITED 返回非零值时，我们可以用这个宏来提取子进程的返回值，即如果子进程调用 exit(5)退出，WEXITSTATUS(status)就会返回 5；应注意的是，如果进程不是正常退出的，也就是说，WIFEXITED 返回 0，这个值就毫无意义。

wait.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main () {
    printf("\"#\"代表父进程输出信息；\"@\"代表子进程输出信息。\n");
    pid_t pid = fork();
    if(pid) {
        printf("# 1.父进程等待子进程结束\n");
        int status = 0;
        wait(&status);
        printf("# 2.子进程返回数值为 %d\n", WEXITSTATUS(status));
    }
    else {
        printf("@ 1.子进程休眠5s\n");
        sleep(5);
        exit(3);
    }
    return 0;
}
```

编译后运行。

3.5 Windows 并发画圆画方

开发环境：Windows10+Qt5.12.2+Qt Creator 4.8.2

使用 QThread 库提供的线程工具库，与 QTimer 定时器定时重绘界面，在 QPixmap 图层上用画笔作画。程序中一共 drawcircleandrectinthesametime，mainwindow，mythread，mythread2 四个实现类。分别完成：同时作画界面，起始界面，画方进程，画圆进程。

drawcircleandrectinthesametime 类的部分实现如下：声明了左右两部分画布，将其传递给新建的进程，并唤醒进程，声明计时器定时唤醒重绘界面。

```
DrawCircleAndRectInTheSameTime::DrawCircleAndRectInTheSameTime(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::DrawCircleAndRectInTheSameTime)
{
    ui->setupUi(this);
    pixmap1=new QPixmap(this->width()/2, this->height());
    pixmap1->fill(Qt::white);

    pixmap2=new QPixmap(this->width()/2, this->height());
    pixmap2->fill(Qt::white);

    myThread *m0=new myThread(pixmap1);
    QThreadPool::globalInstance()->start(m0);

    myThread2 *m1=new myThread2(pixmap2);
    QThreadPool::globalInstance()->start(m1);

    QTimer *qt0=new QTimer();
    qt0->setInterval(REPAINT_TIME);
    qt0->start();
    connect(qt0,SIGNAL(timeout()),this,SLOT(onTimeOut()));
}
```

```
void DrawCircleAndRectInTheSameTime::paintEvent(QPaintEvent *){
    QPainter painter(this);
    painter.drawPixmap(0,0,*pixmap1,0,0,0,0);
    painter.drawPixmap(pixmap1->width(),0,*pixmap2,0,0,0,0);
}
```

而界面跳转相关实现在 mainwindows 中，基于 Qt 的信号槽机制：

```
void MainWindow::on_btn_clicked()
{
    DrawCircleAndRectInTheSameTime *draw = new DrawCircleAndRectInTheSameTime();
    draw->show();
    this->close();
}
```

画方的进程如下，而画圆形的进程也可类似的定义：

```
void myThread::run(){
    QPainter* painter = new QPainter(pixmap);
    QColor red(0xFF,0,0);//设置颜色
    QPen pen(red);//定义画笔
    pen.setWidth(5);//
    painter->setPen(pen);
    for(int i = 1; i <= 200; i++) {
        if(1 <= i && i <= 50)
            painter->drawLine(100+8*(i-1),100,100+8*i,100);
        else if(51 <= i && i <= 100)
            painter->drawLine(500,100+8*(i-51),500,100+8*(i-50));
        else if(101 <= i && i <= 150)
            painter->drawLine(500-8*(i-101),500,500-8*(i-100),500);
        else
            painter->drawLine(100,500-8*(i-151),100,500-8*(i-150));

        QThread::msleep(SLEEP_TIME_MS);
        //emit rep();
    }
}
```

编译后运行。

3.6 Linux“生产者-消费者”同步控制

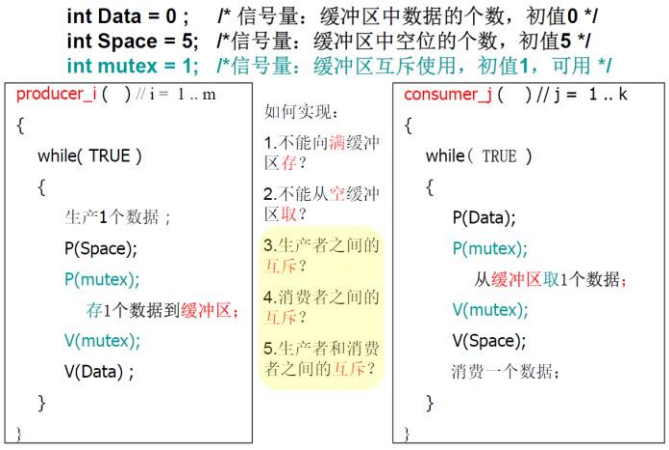
开发环境：银河麒麟操作系统 v10 桌面版

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。Linux 环境下使用互斥锁对象和轻量级信号量对象，主要函数，P 操作：sem_wait( )，V 操作：sem_post( )，上锁原语：pthread_mutex_lock( )，开锁原语：pthread_mutex_unlock( )。

其中生产者 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)。而消费者每休眠 100ms-1s 的随机时间消费一个数据。

为保证 1.缓冲区满时不能存；缓冲区空时不能取（消费）；2.每个时刻生产者或消费者只能有 1 个存或取缓冲区；

我们需要借助 PV 操作实现如下过程。



综上，生产者 1（produce1）可如下编写：

```
10  void *produce1() {
11      while(1) {
12          sem_wait(&space);
13          pthread_mutex_lock(&mutex);
14
15          int product = rand() % 1000 + 1000, pos = 0;
16          for(int i = 1; i <= 10; i++)
17              if(arr[i] == 0) {
18                  arr[i] = product;
19                  pos = i;
20                  break;
21              }
22          printf("生产者1生产了第%d号产品：%d\n",pos, product);
23
24          pthread_mutex_unlock(&mutex);
25          sem_post(&data);
26
27          int t = rand() % 901 + 100;
28          usleep(t * 1000);
29      }
30  }
```

生产者 2（produce2）类似，消费者（consume）如下编写：

```
53  void *consume() {
54      while(1) {
55          sem_wait(&data);
56          pthread_mutex_lock(&mutex);
57
58          int product = 0, pos = 0;
59          for(int i = 1; i <= 10; i++)
60              if(arr[i] != 0) {
61                  product = arr[i];
62                  arr[i] = 0;
63                  pos = i;
64                  break;
65              }
66          printf("消费者消费了第%d号产品：%d\n",pos, product);
67
68          pthread_mutex_unlock(&mutex);
69          sem_post(&space);
70
71          int t = rand() % 901 + 100;
72          usleep(t * 1000);
73      }
74  }
```

主进程创建进程并使主进程生命周期与其同步。

```
int main () {
    sem_init(&data,0,0);
    sem_init(&space,0,9);
    pthread_t p1, p2;
    pthread_t c1, c2, c3;
    pthread_create(&p1,NULL,produce1,NULL);
    pthread_create(&p2,NULL,produce2,NULL);
    pthread_create(&c1,NULL,consume,NULL);
    pthread_create(&c2,NULL,consume,NULL);
    pthread_create(&c3,NULL,consume,NULL);
    pthread_join(p1,NULL);
    pthread_join(p2,NULL);
    pthread_join(c1,NULL);
    pthread_join(c2,NULL);
    pthread_join(c3,NULL);
    return 0;
}
```

编译后运行。

3.7 Linux 信号通信

开发环境：银河麒麟操作系统 v10 桌面版

父进程创建子进程，并让子进程进入死循环。子进程每隔 2 秒输出“I am Child Process, alive !\n”，而父进程询问用户“To terminate Child Process. Yes or No? \n”，用户从键盘回答 Y 或 N.若用户回答 N，延迟 2 秒后再提问。若用户回答 Y，父进程使用 kill(pid , signal)函数向子进程发送用户信号，让子进程结束（子进程提前使用 signal()函数注册信号处理函数）。子进程结束之前打印字符串：“Bye, Wolrd !\n”。

父进程编写如下：

```
if(pid) {
    char c;
    while(1) {
        printf("To terminate Child Process. Yes or No? \n");
        scanf("%c", &c);
        if(c == 'Y') {
            kill(pid, SIGUSR1);
            break;
        }
        sleep(2);
    }
    sleep(4);
}
```

子进程编写如下：

```
else {
    signal(SIGUSR1, handler);
    while(1) {
        printf("I am Child Process, alive!\n");
        sleep(2);
    }
}
```

而被注册的信号处理函数 handler 即为先输出结束信息后使进程结束。

```
 9  void handler(int sig) {
10      printf("Bye,Wolrd!\n");
11      exit(0);
12      return ;
13  }
```

编译后运行。

3.8 Linux“哲学家就餐”模拟

开发环境：银河麒麟操作系统 v10 桌面版

五个哲学家围坐圆桌边，桌上有 1 盘面和 5 支筷子。哲学家的生活重复进行思考-休息-吃饭-思考-……。

若假设哲学家吃饭必须用 1 双筷子，每次取 1 支自己左边或右面的筷子，而且仅当吃完才放下。这个过程可能会产生死锁，例如当所有哲学家同时拿起了自己左边的筷子，便再也不会被放下，故每个哲学家都无限期地等待邻座放下筷子！每个进程陷入阻塞状态。

如若我们也为筷子进行编号，而且我们规定，哲学家取筷子一定先取编号较小的，而后取编号较大的筷子。这样破坏了死锁的环路条件。我们假设 5 个哲学家陷入死锁状态，由于死锁的性质，每个哲学家占有了一部分资源（而且必然一人一个），而在不断申请另一部分资源，而由于我们的规定，他申请的资源编号一定大于占有的资源，则会出现矛盾，即一个循环大于的不等式，故哲学家们不会进入死锁状态。

到目前为止，我们则提供了哲学家问题的死锁与非死锁问题的解决方案。我们在 Linux 环境下借助 pthread 库，使用互斥锁 pthread_mutex_lock, pthread_mutex_unlock 等函数完成了实现。

哲学家问题的死锁版本实现如下：

```
12  void *Philosopher(void *args) {
13      int num = *(int*)args;
14      while(1) {
15          int t = rand() % 401 + 100;
16          printf("philosopher%d思考%d ms\n", num, t);
17          usleep(t * 1000);
18          t = rand() % 401 + 100;
19          printf("philosopher%d休息%d ms\n", num, t);
20          usleep(t * 1000);
21          pthread_mutex_lock(&mutex[num]);
22          printf("philosopher%d获得了左手边的筷子%d\n", num, num);
23          usleep(1000 * 1000);
24          pthread_mutex_lock(&mutex[(num+4)%5]);
25          printf("philosopher%d获得了右手边的筷子%d\n", num, (num+4)%5);
26          t = rand() % 401 + 100;
27          printf("philosopher%d获得了两只筷子%d和%d，吃饭%d ms\n", num, num, (num+4)%5, t);
28          usleep(t * 1000);
29          pthread_mutex_unlock(&mutex[(num+4)%5]);
30          printf("philosopher%d放下了右手边的筷子%d\n", num, (num+4)%5);
31          pthread_mutex_unlock(&mutex[num]);
32          printf("philosopher%d放下了左手边的筷子%d\n", num, num);
33      }
34  }
```

而非死锁版本只需改动部分（第 20—27 行）：

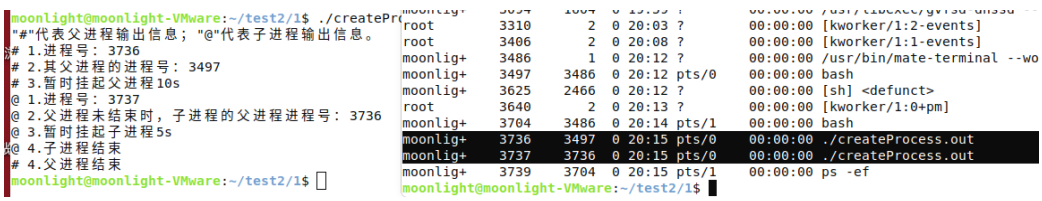
```
23          if(num != 0) {
24              pthread_mutex_lock(&mutex[num]);
25              printf("philosopher%d获得了左手边的筷子%d\n", num, num);
26              usleep(1000 * 1000);
27              pthread_mutex_lock(&mutex[(num+4)%5]);
28              printf("philosopher%d获得了右手边的筷子%d\n", num, (num+4)%5);
29          }
30          else {
31              pthread_mutex_lock(&mutex[(num+4)%5]);
32              printf("philosopher%d获得了右手边的筷子%d\n", num, num);
33              usleep(1000 * 1000);
34              pthread_mutex_lock(&mutex[num]);
35              printf("philosopher%d获得了左手边的筷子%d\n", num, (num+4)%5);
36          }
```

编译后运行。

四、实验结果

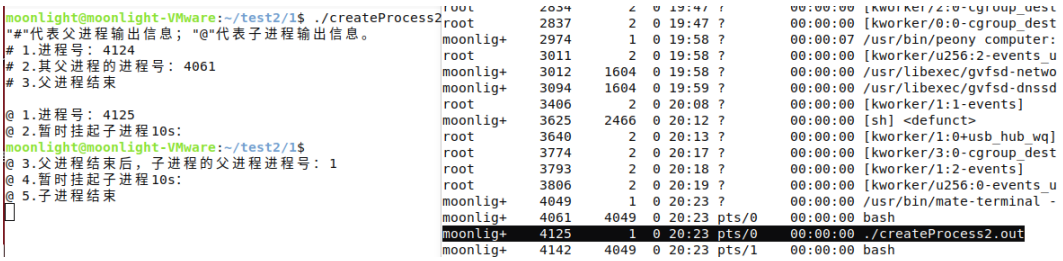
4.1 Linux 下父子进程的创建

在父进程后结束的程序运行起来后，在子进程结束前，使用 ps 命令，观察父进程 ID。



可以清楚看出，子进程（pid=3737）的父进程 pid 为 3736。符合规律。

程序 createProcess2.c，使父进程提前结束。在子进程挂起的时间内，使用 ps 命令，观察其父进程 ID。



可以清楚看出，父进程结束后，子进程（pid=3737）的父进程 pid 为 1（init 进程）。即父进程进入结束态后子进程被自动挂载到 init 进程下。

4.2 Linux 线程实验

编译后运行，其运行结果如下：

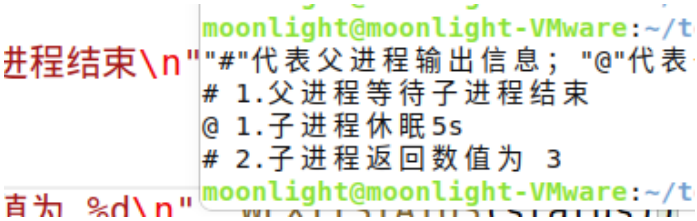
```
A : 988
B : 6
A : 989
B : 5
A : 990
B : 4
A : 991
B : 3
A : 992
B : 2
A : 993
B : 1
A : 994
B Thread End
A : 995
A : 996
A : 997
A : 998
A : 999
A : 1000
A Thread End
```

4.3 Windows 线程实验

编译后运行，其运行结果如下：（与 Linux 线程实验结果一致）

4.4 Linux 使用 wait 同步

编译后运行，其运行结果如下：



4.5 Windows 并发画圆画方

当我们运行程序后，点击起始界面的 PushButtom 键，即可观察同时画圆画方。（双击下方 PowerPoint 对象即可查看视频）

4.6 Linux“生产者-消费者”同步控制

编译后运行，其运行结果如下：

```
生产者1生产了第6号产品: 1649
生产者2生产了第7号产品: 2421
消费者消费了第1号产品: 1793
消费者消费了第2号产品: 2915
生产者2生产了第1号产品: 2362
生产者1生产了第2号产品: 1027
消费者消费了第1号产品: 2362
生产者2生产了第1号产品: 2690
生产者1生产了第8号产品: 1059
生产者1生产了第9号产品: 1763
消费者消费了第1号产品: 2690
生产者2生产了第1号产品: 2926
消费者消费了第1号产品: 2926
生产者1生产了第1号产品: 1540
消费者消费了第1号产品: 1540
生产者1生产了第1号产品: 1426
消费者消费了第1号产品: 1426
生产者2生产了第1号产品: 2172
消费者消费了第1号产品: 2172
生产者1生产了第1号产品: 1736
消费者消费了第1号产品: 1736
生产者1生产了第1号产品: 1211
消费者消费了第1号产品: 1211
```

4.7 Linux 信号通信

编译后运行，其运行结果如下：

```
moonlight@moonlight-VMware:~/test2/7$ ./a.out
To terminate Child Process. Yes or No?
I am Child Process, alive!
I am Child Process, alive!
N
I am Child Process, alive!
To terminate Child Process. Yes or No?
I am Child Process, alive!
YTo terminate Child Process. Yes or No?

Bye,Wolrd!
```

4.8 Linux“哲学家就餐”模拟

某一次执行过程中死锁版本出现了如下情况（死锁）：

```
philosopher2休息377 ms
philosopher1获得了左手边的筷子1
philosopher0休息435 ms
philosopher4获得了左手边的筷子4
philosopher2获得了左手边的筷子2
philosopher3获得了左手边的筷子3
philosopher0获得了左手边的筷子0
```

而非死锁版本中的哲学家会一直思考下去：

```
moonlight@moonlight-VMware:~/test2/8$ ./b.out
philosopher0思考409 ms
philosopher1思考139 ms
philosopher3思考313 ms
philosopher4思考129 ms
philosopher2思考330 ms
philosopher4休息485 ms
philosopher1休息261 ms
philosopher3休息477 ms
philosopher2休息377 ms
philosopher1获得了左手边的筷子1
philosopher1获得了右手边的筷子0
philosopher1获得了两只筷子1和0，吃饭435 ms
philosopher0休息248 ms
philosopher4获得了左手边的筷子4
philosopher4获得了右手边的筷子3
philosopher4获得了两只筷子4和3，吃饭322 ms
philosopher2获得了左手边的筷子2
philosopher1放下了右手边的筷子0
philosopher1放下了左手边的筷子1
philosopher1思考453 ms
philosopher2获得了右手边的筷子1
philosopher2获得了两只筷子2和1，吃饭333 ms
```

五、实验错误排查和解决方法

4.4 Linux 使用 wait 同步

一开始我遇到了 wait(&status)中 status 返回的参数值与子进程 exit 向父进程传递的参数不同的情况。

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main () {
    printf("\"#\"代表父进程输
    pid_t pid = fork();
    if(pid) {
        printf("# 1.父进程等待
        int status;
        wait(&status);
        printf("# 2.子进程返回
    }
    else {
        printf("@ 1.子进程休眠
        sleep(5);
        exit(520);
    }
    return 0;
}
```

```
moonlight@moonlight-VMware: ~/test2/4
文件(F)  编辑(E)  视图(V)  搜索(S)  终端(T)  帮助(H
moonlight@moonlight-VMware:~/test2/4$ gcc
moonlight@moonlight-VMware:~/test2/4$ ./a.
"#"代表父进程输出信息; "@"代表子进程输出信
# 1.父进程等待子进程结束
@ 1.子进程休眠5s
# 2.子进程返回数值为 2048
moonlight@moonlight-VMware:~/test2/4$
```

经过查阅相关网站，发现若要还原返回值，需要使用正文中使用的宏。WIFEXITED(status)。当 WIFEXITED 返回非零值时，我们可以用这个宏来提取子进程的返回值，即如果子进程调用 exit(5)退出，WEXITSTATUS(status)就会返回 5；应注意的是，如果进程不是正常退出的，也就是说，WIFEXITED 返回 0，这个值就毫无意义。

4.7 Linux 信号通信

当我一开始编写程序时，子进程总是接到父进程发出的信号后，没有执行信号处理函数直接被终止运行。

```c
void handler(int sig) {
    printf("Bye,Wolrd!\n");
    //exit(0);
    return ;
}

int main () {
    pid_t pid = fork();
    if(pid) {
        char c;
        while(1) {
            printf("To terminate Child Process. Yes or No? \n");
            scanf("%c", &c);
            if(c == 'Y') {
                kill(pid, SIGKILL);
                printf("SIGKILL!\n");
                break;
            }
            sleep(2);
        }
        sleep(4);
    }
    else {
        signal(SIGKILL, handler);
        //while(1) {
```

```
moonlight@moonlight-VMware: ~/test2/7
文件(F)  编辑(E)  视图(V)  搜索(S)  终端(T)  帮助(H)
moonlight@moonlight-VMware:~/test2/7$ gcc
moonlight@moonlight-VMware:~/test2/7$ ./a
bash: ./a: 没有那个文件或目录
moonlight@moonlight-VMware:~/test2/7$ ./a.
To terminate Child Process. Yes or No?
Y
SIGKILL!
moonlight@moonlight-VMware:~/test2/7$
```

后来发现，不能使用系统已经实现的信号，我当时望文生义，使用了 SIGKILL 信号，其信号处理函数已被系统固定，我们可以使用的仅有部分信号值。

4.8 Linux“哲学家就餐”模拟

当时出现了非常多怪异的情况，比如莫名其妙的出现了不应该出现的死锁，当我进行输出调试时惊人的发现了多个同一哲学家被创建。

```
--------3
philosopher3思考409 ms
--------4
philosopher4思考139 ms
--------3
philosopher3思考330 ms
--------5
--------4
philosopher4思考313 ms
philosopher5思考129 ms
philosopher5休息485 ms
philosopher4休息261 ms
```

仔细分析原因后发现，是临界资源访问的问题。当主线程中的 i 被改变时，可能其他线程并未成功创建，多个线程同时访问修改 i 这个临界资源，导致了并发错误。

```
for(int i = 0; i < philosophernum; i++) {
    pthread_create(&philosopher[i], NULL, Philosopher, &i);
}
```

我分配了多个资源，破坏了临界资源访问的必要条件，使其不再是临界资源。

```
for(int i = 0; i < philosophernum; i++) {
    philosopherId[i] = i;
    pthread_create(&philosopher[i], NULL, Philosopher, &philosopherId[i]);
}
```

从而解决了问题。

六、实验参考资料和网址

（1）教学课件：感谢华中科技大学软件学院苏曙光老师

（2）苏曙光. 操作系统原理（慕课版）[M]. 北京：人民邮电出版社，2022.

（2）https://blog.csdn.net/zangle260/article/details/34852789

（3）https://blog.csdn.net/WWWzq_/article/details/116029735

（4）https://blog.csdn.net/networkhunter/article/details/100218945

（5）https://blog.csdn.net/weixin_42518668/article/details/105905707

（6）https://blog.csdn.net/weixin_44732817/article/details/90143888

（7）https://zhuanlan.zhihu.com/p/112297714

（8）https://blog.csdn.net/briblue/article/details/87859716

（9）http://c.biancheng.net/view/8628.html

（10）https://blog.csdn.net/qq_46106285/article/details/121631962

（11）https://blog.csdn.net/zhizhengguan/article/details/107567449

（12）https://blog.csdn.net/qq_42606750/article/details/89296615

（13）https://blog.csdn.net/XZ2585458279/article/details/98474020

（14）https://blog.csdn.net/mm5670252/article/details/85161006

（15）https://blog.csdn.net/m0_60352504/article/details/122485247

（16）https://doc.qt.io/qt-5/

（17）https://www.cnblogs.com/lifexy/p/9203929.html

（18）https://www.cnblogs.com/fuhang/p/9900123.html

（19）https://www.coder.work/article/3323070

（20）https://blog.csdn.net/hechao3225/article/details/53033993

（21）https://blog.csdn.net/m0_52364631/article/details/111937373

（22）https://blog.csdn.net/qq_39054069/article/details/84257909

（23）https://blog.csdn.net/qq_45097019/article/details/105612663

（24）http://www.manongjc.com/detail/20-lblqhpfvaewizsg.html

（25）https://www.jianshu.com/p/72711d8a32ac

（26）https://blog.csdn.net/iteye_2733/article/details/82278538

（27）https://blog.csdn.net/zx3517288/article/details/52743011