《操作系统原理》实验报告

姓名	汪闻韵	学号	U202012056	专业班级	网安 2002	时间	2022.11.21

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

三、实验过程

3.1 Linux 下循环输出数据

在 Linux 下利用库函数< pthread.h >进行程序的编写。在 main 函数中用函数 pthread_create 创建两个线程。之后使用函数 pthread_join 阻塞主线程的执行，直至目标线程 tid1 和 tid2 执行结束，实现 tid1 和 tid2 的同步。

```
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("can't create thread");
        }
        //创建 tid2线程
        if(err=pthread_create(&tid2,NULL,th_fn2,NULL)){
                perror("can't create thread");
        }

        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
        return 0;
}
```

3.2 Linux 下 fork 一个子进程

效果一：父进程不用 wait 函数，先于子进程结束。

首先用 fork 函数创建一个子进程，由于 fork 函数的返回值特性，利用返回值区分子进程和父进程。在父进程中，打印父进程和子进程的 PID，休眠五秒，同时利用在命令行中利用 ps 命令查看两个进程的 PID。在子进程中，利用死循环，使父进程先于子进程结束。

```
#include<unistd.h>
#include<stdio.h>
int main()
{
        pid_t p1=fork();
        if(p1>0){
                printf(" Parent Pid in Parent Pro-
cess：%d\n",getpid());
                printf(" Child Pid in Parent Process：%d\n",p1);
                printf(" 挂起父进程...\n");
                sleep(5);
                printf(" Parent process end! \n");
        }else if(p1==0){
                while(1)//进入长时间循环，观察父进程中的 pid
                    int i=1+1;
        }
        else{
                printf("创建子进程失败");
        }
        return 0;
}
```

效果二：父进程用 wait 函数，后于子进程结束。

首先用 fork 函数创建一个子进程，与上一程序思路相似，只是在父进程中不再休眠，而是利用 wait() 函数等待子进程执行完毕，并打印出子进程返回的值。在子进程中，利用 exit()函数退出该进程，并传递返回值，此处设定返回值为 0。

```
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<stdio.h>
int main()
{
        pid_t p1=fork();
        if(p1>0){
                int child_ret;
                printf("\n Parent Pid in Parent Pro-
cess: %d\n",getpid());
                printf(" Child Pid in Parent Process: %d\n",p1);
                printf("父进程等待子进程结束...\n");
                wait(&child_ret);
                printf("子进程返回参数: %d\n",child_ret);
                printf(" Parent process end! \n");
        }else if(p1==0){
                printf("\n Child Pid in Child Pro-
cess: %d\n",getpid());
                printf("Child Process start sleeping...(5s)");
                sleep(5);//在该系统中，sleep 以 s 为单位
                printf(" Child process end! \n");
                exit(0);
        }
        else{
                printf("创建子进程失败");
        }
        return 0;
}
```

3.3 Windows 下利用线程并发画圆画方

画圆画方需要用到库 graphics.h 中的函数，进入网站 easyX 进行头文件的安装下载。利用函数 putpixel 进行画点，之后连成线。如下代码所示，画圆时，设定像素点共 720 个，每个函数执行间隔 5ms，依据圆的形状坐标对圆图形进行绘制。

```
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)), GREEN);
    Sleep(5);
}
```

在 Windows 进行线程控制时，主要利用函数 CreateThread 与 WaitForSingleObject，与 Linux 下的并发过程类似，创建线程后利用函数阻塞目前的主线程，等待圆线程和方线程结束。

```
int main()
{
    // 初始化图形模式
    initgraph(640, 480);
    HANDLE square,circle;
    DWORD threadID;     //记录线程 ID

    if ((square = CreateThread(NULL, 0, drawSquare, 0, 0,
&threadID)) == NULL)
        printf("正方形线程创建失败!");
    if ((circle = CreateThread(NULL, 0, drawCircle, 0, 0,
&threadID)) == NULL)
        printf("圆线程创建失败!");

    //等待所有线程结束
    WaitForSingleObject(square,INFINITE);
    WaitForSingleObject(circle,INFINITE);
    CloseHandle(square);
    CloseHandle(circle);

    return 0;
}
```

3.4 Linux 下实现“生产者-消费者”同步控制

首先基于要求，定义生产者的数量为常量 2，消费者的数量为常量 3，同时设置大小为 10 的数组(缓冲区)用于存放“产品”；fullFlag 和 emptyFlag 声明为初始信号量，在 main 函数中通过 init 函数分别初始化为 10 和 0，分别表示当前数组中的空位和已存入的元素数量。 结合相应的 pv 保证：生产者在数组满时会阻塞等待消费者进行消费操作，而非继续生产； 消费者在数组空时会阻塞等待生产者进行生产操作，而非继续消费；声明一个大小为生产者与消费者数量之和的数组 tid，用于创建线程时存储指向线程标识符的指针；声明一个互斥锁对象 mutex，实现任一生产者和生产者、生产者和消费者、消费者之间对于数组访问操作的互斥关系；ProPos 和 ConPos 两个变量分别用于标记当前生产者和消费者操作的数组位置。

```
#define producerNum 2 // 生产者数量
#define consumerNum 3 // 消费者数量
#define bufferSize 10 // 缓冲区大小
sem_t fullFlag;
sem_t emptyFlag;
pthread_t tid[producerNum + consumerNum];
pthread_mutex_t mutex;
int buf[bufferSize] = {0};
int ProPos = 0;
int ConPos = 0;
```

对于生产者线程，使用 while(1)循环表明生产者不断地重复生产指定范围数的操作。下面代码列出不断循环的操作。在每次要开始真正的生产操作前，需要调用 sem_wait(&fullFlag)，保证不会出现数组满依然继续生产，之后调用 pthread_mutex_lock(&mutex)保证操作的互斥；之后根据函数传入的信息，输出对应生产者范围内的物品号码，及存放的数组下标位置，并更新当前生产者完成操作的位置（ProPos），注意所有生产者共用变量 ConPos；最后用函数 pthread_mutex_unlock 执行相应的解锁操作，用函数 sem_post 实现 v 操作。

```
sem_wait(&fullFlag);
pthread_mutex_lock(&mutex); // 上锁
int item;
if (order == 1)
    item = rand() % (1999 - 1000) + 1000;
else if (order == 2)
    item = rand() % (2999 - 2000) + 2000;

buf[ProPos] = item;
int tmp = ProPos;
ProPos = (ProPos + 1) % bufferSize;
cout << "生产者: " << order << "\t 放入物品: " << item << "\t 在位置:
" << tmp << endl;
pthread_mutex_unlock(&mutex);
sem_post(&emptyFlag);
sleep(1);
```

对于消费者线程，在每次要开始真正的消费操作前，调用 sem_wait(&emptyFlag)，保证不会出现数组空依然继续消费，之后调用 pthread_mutex_lock(&mutex)保证操作的互斥；之后输出提示信息：包括当前消费者的编号、消费的数组元素值、消费的数组元素下标。更新当前消费者完成操作的位置(ConPos)，最后用函数 pthread_mutex_unlock 执行相应的解锁操作，用函数 sem_post 实现 v 操作。

```
sem_wait(&emptyFlag);
pthread_mutex_lock(&mutex);
for (int i = 0; i < bufferSize; i++)
{
    if (i == ConPos)
        cout << "消费者: " << order << "\t 消费物品: " << buf[i] <<
"\t 位置: " << i << endl;
}
sleep(4);
ConPos = (ConPos + 1) % bufferSize;
pthread_mutex_unlock(&mutex);
sem_post(&emptyFlag);
sleep(1);
```

对于线程控制，在 main 函数中完成。首先完成互斥量的初始化，以及信号量的初始化；之后创建一个线程池子，分别为 2 个消费者、3 个生产者创建线程， 这里要特别注意 pthread_create 函数的参数及参数的含义，以免出现问题；调用 5 次 pthread_join()函数以阻塞的方式等待所有消费者和生产者线程结束；最后释放所有互斥量和信号量。

```
pthread_mutex_init(&mutex, NULL);
sem_init(&fullFlag, 0, bufferSize);
sem_init(&emptyFlag, 0, 0);
pthread_t threadPool[producerNum + consumerNum];

int i;
for (i = 0; i < producerNum; i++){
    pthread_t temp;
    if (pthread_create(&temp, NULL, producer, &i) == -1){
        ……}
    threadPool[i] = temp;
} // 创建生产者进程放入线程池
for (i = 0; i < consumerNum; i++){
    pthread_t temp;
    if (pthread_create(&temp, NULL, consumer, &i) == -1){
        ……}
    threadPool[i + producerNum] = temp;
} // 创建消费者进程放入线程池

void *result;
for (i = 0; i < producerNum + consumerNum; i++){
    if (pthread_join(threadPool[i], &result) == -1){
        ……}
}
pthread_mutex_destroy(&mutex);
sem_destroy(&fullFlag);
sem_destroy(&emptyFlag);
```

四、实验结果

4.1 Linux 下循环输出数据

编译运行之后，观察到标识为 B 的线程从 1000 开始递减 1 进行输出，而标识为 A 的线 程从 1 开始递增 1 进行输出，可以看到两个线程交替输出正确的预期结果，如图 4-1 所示。

```
willow@willow-VMware-Virtual-Platform:/data/usershare/OS$ gcc ./lab2.c -o ./lab2.out -lpthread
willow@willow-VMware-Virtual-Platform:/data/usershare/OS$ ./lab2.out
B:1000
A:1
A:2
B:999
A:3
B:998
B:997
A:4
A:5
B:996
A:6
B:995
A:7
B:994
SA:8
B:993
A:9
B:992
```

图 4-1 循环输出数据运行结果

4.2 Linux 下 fork 一个子进程

运行实现效果一的程序，可以看到输出部分。对于子进程，pid 为 8487，其父进程 pid 为 8486；对于父进程， pid 为 8486 (其父进程 pid 为 2197)；由于该程序中控制父进程在子进程后结束，因此子进程的父进程 pid 应当与创建该子进程的进程 pid 相同，如图 4-2，验证了这一结果。

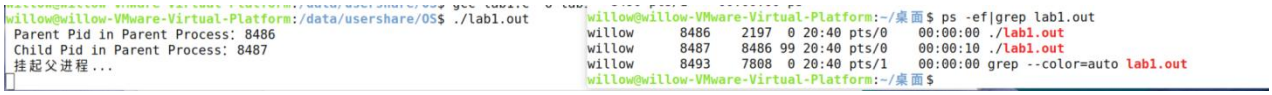

图 4-2 效果一运行结果

运行实现效果二的程序，可以看到，在父进程中打印出父进程 PID 为 8977，然后进入阻塞开始等待子进程的结束；子进程打印出自己的 PID 为 8978，睡眠五秒后结束；最后父进程结束。如图 4-3 所示。

```
willow@willow-VMware-Virtual-Platform:/data/usershare/OS$ gcc lab1_2.c -o lab1_2.out
willow@willow-VMware-Virtual-Platform:/data/usershare/OS$ ./lab1_2.out

 Parent Pid in Parent Process: 8977
 Child Pid in Parent Process: 8978
Start Waiting...

 Child Pid in Child Process: 8978
Start sleeping...(5s)
 Child process end!
子进程返回参数: 0
 Parent process end!
willow@willow-VMware-Virtual-Platform:/data/usershare/OS$
```

图 4-3 效果二运行结果

4.3 Windows 下利用线程并发画圆画方

如下图所示，该程序可以同时画圆画方，左侧为正方形线程画出的图案，右侧为圆形线程画出的图案，两者由于像素点个数相同，所以可以同时结束，如图 4-4 所示。

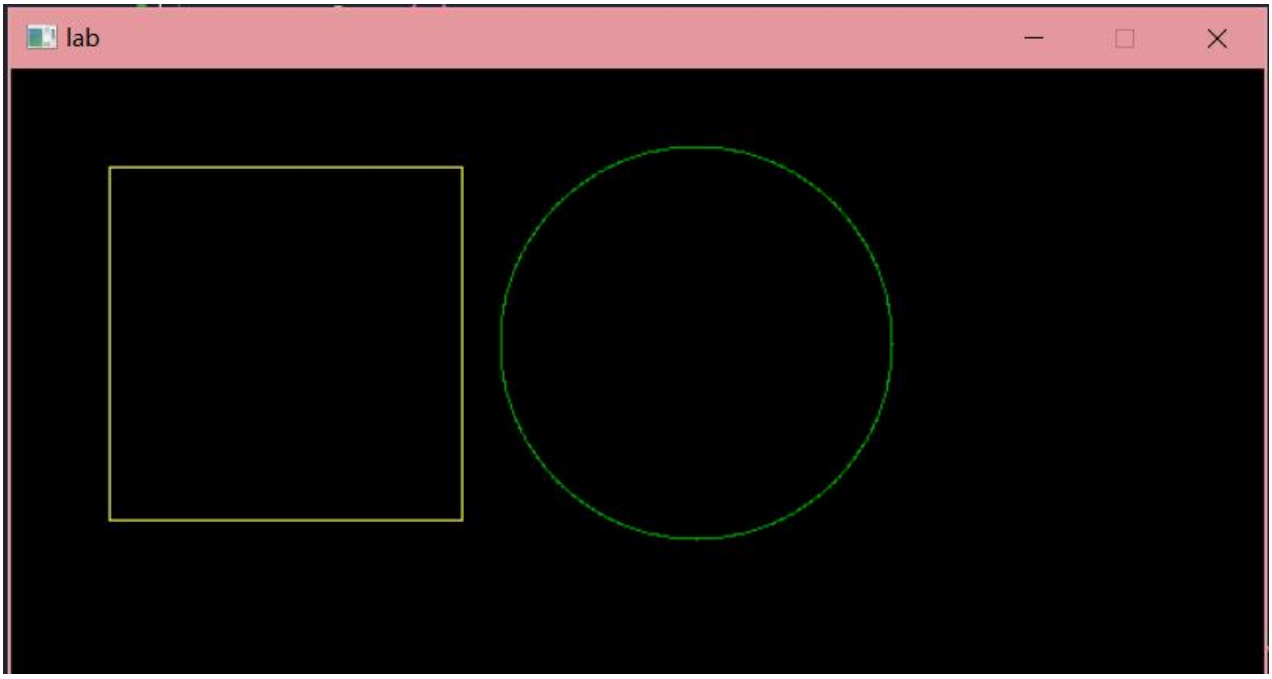

图 4-4 同时画圆画方运行结果

4.4 Linux 下实现“生产者-消费者”同步控制

编译运行，查看运行结果，可以看到在终端的输出中 2 个生产者依次生产数据，而 3 个消费者依次消费数据，并且都能够清楚地指出生产数据/消费数据的值和数组下标位置，依次查看对比，发现对于任意数据也都满足先生产后消费的规律，满足要求，如图 4-5 所示。

```
willow@willow-VMware-Virtual-Platform:/data/usershare/OS$ g++ ProCon.cpp -o ProCon.out -lpthread
willow@willow-VMware-Virtual-Platform:/data/usershare/OS$ ./ProCon.out
生产者：2       放入物品：2744  在位置：0
生产者：1       放入物品：1744  在位置：1
消费者：3       消费物品：2744  位置：0
消费者：1       消费物品：1744  位置：1
生产者：1       放入物品：1020  在位置：2
生产者：2       放入物品：2368  在位置：3
消费者：2       消费物品：1020  位置：2
消费者：3       消费物品：2368  位置：3
消费者：1       消费物品：0     位置：4
生产者：1       放入物品：1216  在位置：4
生产者：2       放入物品：2624  在位置：5
消费者：2       消费物品：2624  位置：5
消费者：3       消费物品：0     位置：6
消费者：1       消费物品：0     位置：7
生产者：1       放入物品：1182  在位置：6
生产者：2       放入物品：2451  在位置：7
消费者：2       消费物品：0     位置：8
```

图 4-5 “生产者-消费者”运行结果

五、实验错误排查和解决方法

5.1 Linux 下循环输出数据

Q：对库 pthread.h 里的函数不熟悉。

A：查阅相关资料，学习各个参数的表达的意思。

对函数 int pthread_create (pthread_t * thread , const pthread_attr_t * attr ,void *(* start_routine) (void *),void * arg)—— thread 指向的地址存储创建的线程号；attr 是一个 pthread_attr_t 结构体的指针，用来在线程创建的时候指定新线程的属性。如果在创建线程时，这个参数指定为 NULL, 那么就会使用默认属性。新线程通过调用 start_routine（）开始执行; arg 作为 start_routine（）的唯一参数传递。

对函数 int pthread_join(pthread_t thread, void ** retval) —— thread 用于指定接收哪个线程的返回值；第二参数表示接收到的返回值，如果 thread 线程没有返回值，或不需要接收 thread 线程的返回值，可以将 retval 参数置为 NULL。

5.2 Linux 下 fork 一个子进程

Q：sleep(5*1000)等待时间过长终端无反应。

A：与 Windows 系统不同，在 Linux 系统中，使用的是 gcc 的库，sleep 是以秒为单位的，所以需要休眠五秒时应该使用 sleep(5)。

5.3 Windows 下利用线程并发画圆画方

Q：不会使用画图函数库？

A：网上查阅资料后，学习了相关的库函数之后，选择使用函数 putpixel 进行绘制。对函数 void putpixel(int x,int y,COLORREF color)中，前两参数为点的横纵坐标，第三参数为点的颜色。

Q：绘制圆的公式？

A：以坐标点(cos(-pi / 2 + ((i * pi) / 360)), sin(-pi / 2 + ((i * pi) / 360)为基础，进行变换。

5.4 Linux 下实现“生产者-消费者”同步控制

Q：rand 函数每次产生的随机值相同？

A：在 C 语言中，rand 函数可以用来产生随机数，但并不是真正意义上的随机数。它是返回介于 0 和 RAND_ MAX 之间的伪随机整数。这个数字是由一个算法生成的，该算法

每次调用它时都返回一个显然不相关的数字序列。该算法是使用一个种子来生成序列，当计算机正常开机后，这个种子的值是固定的，因此产生的伪随机整数也是固定的，除非你为了改变这个值破坏了系统。为了初始化的值不同，C 语言提供了 srand 函数。

rand 函数每次调用前都会查询是否调用过 srand(seed)，是否给 seed 设定了一个值，如果没有，种子的值就默认为 1，直接用 1 来初始化种子，那生成的随机数每次就会重复，为了防止生成的随机数重复，一般使用时间戳作为时间种子，采用系统时间来初始化，使用 time 函数来获得系统时间，它的返回值为从 00:00:00 GMT, January 1, 1970 到现在所持续的秒数，然后将 time_t 型数据转化为(unsigned)型再传给 srand 函数。

所以要产生真正的随机数需要在使用 rand 函数前先用 srand 随机化种子。

六、实验参考资料和网址

（1）教学课件

（2）URL：https://blog.csdn.net/weixin_44518102/article/details/124622003

（3）URL：https://blog.csdn.net/m0_47988201/article/details/116332597