

《操作系统原理》实验报告

姓名	黄浩岩	学号		专业班级		时间	2020.11.24
----	-----	----	--	------	--	----	------------

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此处主要粘贴四次
- 注：其中 1、6、8 必做，其余任选 2，此处选择了 4 和 7。

三、实验过程

1) Linux 下创建父子进程 (pcProcess.c)

该实验通过 fork()函数在 Linux 下创建一对父子进程。通过判断 fork 函数的返回值来判断父子进程，并分别做相应操作：输出父/子进程表示字符串，以及进程号 (pid) 和父进程号 (ppid)，该部分代码如下。

```
//main 函数部分代码
pid_t pid;
pid=fork(); //创建子进程
if(pid==0) { //子进程
```

```

    sleep(ct);

    printf("\nChild Process: pid:%d ppid:%d\n", getpid(),getppid());

    //输出进程 id 和父进程 id
}
else if(pid>0) { //父进程

    sleep(pt);

    printf("\nParent Process: pid:%d ppid:%d\n", getpid(),getppid());

}

```

此外，为了查看父进程提前或后结束时子进程的父进程号的变化，在进程输出自己的进程号和父进程号前使用 `sleep()` 函数进行休眠。其中，`pt` 为父进程休眠时间，`ct` 为子进程休眠时间，通过调整 `pt` 和 `ct` 的相对大小来使父进程提前或后结束。此处通过添加 `main` 函数的一个参数：当参数为 0 时为普通情况，`ct==pt`，父子进程休眠时间相同，基本同时结束，可以查看一般情况下的父子进程的进程号关系；当参数为 1 时，`ct>pt`，子进程休眠时间更长，为父进程先结束；当参数为 2 时，`pt>ct`，父进程休眠时间更长，为子进程先结束。该部分代码如下。

```

#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int ct,pt; //设置父子进程休眠时间
int main(int argc, char** argv){
    pid_t pid;
    switch(argv[1][0]){
        case '0':
            printf("\n 普通情况: \n");
            ct=pt=5;
            break;
        case '1':
            printf("\n 父进程先结束: \n");
            pt=2,ct=8;

```

```
        break;
    case '2':
        printf("\n 子进程先结束: \n");
        pt=8,ct=2;
        break;
    ...    //上述父子进程代码
}
```

2) Linux 下父子进程同步 (syn.c)

使用 `fork()` 函数创建子进程，利用 `fork` 返回值来判断当前进程是父进程还是子进程。其中，子进程休眠 5 秒之后用 `exit()` 函数返回参数；而父进程不休眠，使用 `wait()` 函数等待子进程先结束，并将子进程的进程号以返回值的形式记录到变量 `p` 中，将子进程的返回参数记录到变量 `status` 中，并通过 `WEXITSTATUS()` 函数进程分析，最终由父进程输出子进程的返回信息。具体代码如下：

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

int main(){
    pid_t pid,p;
    int status,i;
    pid=fork();
    if(pid==0){ //子进程
        printf("This is Child Process\n");
        sleep(5);//休眠 5s 结束
        printf("Child Process will stop\n");
    }
```

```

        exit(6); //结束进程
    }
    else if(pid>0){
        p=wait(&status); //等待子进程结束，返回子进程进程号
        i=WEXITSTATUS(status); //分析子进程返回信息
        printf("This is Parent Process\nChild's pid=%d exit status=%d\n",pid,i);
    }
    return 0;
}

```

3) Windows 下“生产者-消费者”同步控制 (ProducerConsumer.c)

Windows 环境下，首先定义全局变量，包括缓冲区长度、生产者消费者个数、生产产品号消费产品号等，该部分代码如下：

```

const unsigned short BufLen = 10; //缓冲区长度
const unsigned short ProducerCnt = 2; //生产者个数
const unsigned short ConsumerCnt = 3; //消费者个数
const unsigned sleepTime = 1000; //每次输出后的休眠时间

int productId1 = 1000, productId2 = 2000; //2 个生产者的生产产品起始号
int consumeId; //消费的产品号
int buf[BufLen]; //缓冲区
int pIdx = 0, cIdx = 0; //分别记录生产的序号和消费的序号
bool goon = true; //用于退出的标志
HANDLE sFull, sEmpty; //信号量
CRITICAL_SECTION cs; //临界区

```

main 函数里使用 CreateSemaphore()函数创建信号量 sFull 和 SEmpty 分别表示缓冲区数据个数和缓冲区空位个数，并设置信号量的初始值和最大值，使用 InitalizeCriticalSection()函数初始化临界区变量 cs 用于缓冲区互斥使用。然后使用 CreateThead()函数分别创建消费

者线程和生产者线程。最后 main 函数主线程进入死循环，直到有输入时停止。该部分代码如下：

```
int main() {  
    srand(unsigned(time(nullptr)));  
    //创建信号量和临界区  
    sFull = CreateSemaphore(NULL, 0, BufLen, NULL);        //缓冲区数据个数  
    sEmpty = CreateSemaphore(NULL, BufLen, BufLen, NULL);   //缓冲区空位个数  
    //sMutex = CreateSemaphore(NULL, 1, 1, NULL);  
    InitializeCriticalSection(&cs);  
    //创建生产者线程  
    HANDLE hThread[ProducerCnt+ConsumerCnt];  
    DWORD producers[ProducerCnt], consumers[ConsumerCnt];  
    for (int i = 0; i < ProducerCnt; ++i) {  
        hThread[i] = CreateThread(NULL, 0, producer, LPVOID(i), 0, &producers[i]);  
        if (!hThread[i]) return -1;  
    }  
    //创建消费者线程  
    for (int i = 0; i < ConsumerCnt; ++i) {  
        hThread[ProducerCnt + i]  
            = CreateThread(NULL, 0, consumer, NULL, 0, &consumers[i]);  
        if (!hThread[ProducerCnt + i]) return -1;  
    }  
    //输入任意字符终止  
    while (goon) {  
        if (getchar()) goon = false;  
    }  
    return 0;  
}
```

生产者线程为函数 `producer()`，使用 `lpPara` 参数传递生产者的序号，1 号生产者生产 1000~2000 号产品，2 号生产者生产 2000-3000 号产品。每次生产产品的过程如下：生产产品前使用 `WaitForSingleObject()`函数作为 P 操作等待缓冲区有空位的信号量 `sEmpty`，使用 `EnterCriticalSection()`函数进入临界区，对缓冲区进行互斥使用。使用 `produce()`函数生产产品。生产产品后，使用函数 `LeaveCriticalSection()`离开临界区。最后使用 `ReleaseSemaphore()`函数作为 V 操作使缓冲区个数+1。

该部分代码如下：

```
DWORD WINAPI producer(LPVOID lpPara) {  
    int no = int(lpPara); //生产者编号  
    while (goon) {  
        auto step = rand() % 900 + 100;  
        WaitForSingleObject(sEmpty, INFINITE); //要求缓冲区有有空位  
        //INFINITE:对象被触发信号后函数才会返回  
        EnterCriticalSection(&cs);          //互斥使用缓冲区  
        Sleep(step);  
        produce(no);  
        LeaveCriticalSection(&cs);  
        ReleaseSemaphore(sFull, 1, NULL); //缓冲区数据个数+1  
    }  
    return 0;  
}
```

消费者线程为函数 `consumer()`。每次消费产品过程如下：消费产品前使用 `WaitForSingleObject()` 函数作为 P 操作等待缓冲区有产品的信号量 `sFull`，使用 `EnterCriticalSection()`函数进入临界区，对缓冲区进行互斥使用。使用 `consume()`函数生产产品。生产产品后，使用函数 `LeaveCriticalSection()`离开临界区。最后使用 `ReleaseSemaphore()`函数作为 V 操作使缓冲区空位+1。

该部分代码如下：

```
DWORD WINAPI consumer(LPVOID lpPara) {
```

```
while (goon) {  
    auto step = rand() % 900 + 100;  
    WaitForSingleObject(sFull, INFINITE);  
    EnterCriticalSection(&cs);  
    Sleep(step);  
    consume();  
    LeaveCriticalSection(&cs);  
    ReleaseSemaphore(sEmpty, 1, NULL);  
}  
return 0;  
}
```

生产函数 `produce()` 和消费函数 `consume()` 分别用于生产和消费产品，并在控制台上输出相应产品已经整个缓冲区的状态。其中，缓冲区采用的是循环队列，生产和消费产品都是从缓冲区的 0 号~9 号再循环回 0 号。

该部分代码如下：

```
void produce(int no) {  
    auto productIdx = no == 0 ? productIdx1++ : productIdx2++;  
    cout << "生产产品: " << ++productIdx << endl;  
    cout << "将产品放入缓冲区\n";  
    buf[pIdx] = productIdx;  
    cout << "缓冲区状态:";  
    for (int i = 0; i < BufLen; ++i) {  
        if (buf[i] != 0) cout << " (" << i + 1 << ')' << buf[i];  
        else cout << " (" << i + 1 << ')' << "NULL";  
        if (i == pIdx) cout << "<-生产";  
    }  
    cout << endl << endl;  
    pIdx = (pIdx + 1) % BufLen;  
}
```

```

        Sleep(sleepTime);
    }

void consume() {
    cout << "从缓冲区取出产品\n";
    consumeId = buf[cIdx];
    cout << "缓冲区状态:";
    for (int i = 0; i < BufLen; ++i) {
        if (buf[i] != 0) cout << " (" << i + 1 << ')' << buf[i];
        else cout << " (" << i + 1 << ')' << "NULL";
        if (i == cIdx) cout << "<-消费";
    }
    cout << endl;
    buf[cIdx] = 0;
    cIdx = (cIdx + 1) % BufLen;
    cout << "消费产品: " << consumeId << endl << endl;
    Sleep(sleepTime);
}

```

4) Linux 下利用信号机制进程通信 (signal.c)

使用 `fork()` 函数创建子进程，让子进程进入死循环，并利用 `sleep()` 函数每隔 2s 输出 “I’m Child Process, alive !\n”。

父进程询问用户是否终止子进程 “To terminate Child Process. Yes or No? \n”，若用户输入 n/N，则利用 `sleep()` 函数休眠 2s 再询问。

若用户回答 y/Y，则父进程利用 `kill(pid, SIGUSR1)` 函数向子进程发送信号 `SIGUSR1`，并使用 `wait()` 函数等待子进程。由于父进程 `fork()` 函数的返回值 `pid` 即为子进程的进程号，因此此处 `kill` 的第 1 个实参为 `pid`。而子进程使用函数 `signal(SIGUSR1, stop)` 来注册信号 `SIGUSR1` 的处理函数，当接收到父进程发送来的 `SIGUSR1` 信号时，调用函数 `stop`，输出 “Bye, world! \n”，并用 `exit()` 结束自己（子进程）。随后父进程跳出死循环也结束。

具体代码如下：

```
#include<signal.h>
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>

void stop(int signum){
    printf("Bye, world!\n");
    exit(0);
}

int main(){
    pid_t pid;
    pid =fork();
    if(pid==0){
        signal(SIGUSR1,stop);
        while(1){
            printf("I'm Child Process, alive!\n");
            sleep(2);
        }
    }
    else if(pid>0){
        while(1){
            char op;
            printf("To terminate Child Process. Yes or No?\n");
            scanf("%c",&op);
            if(op=='Y' || op=='y'){
                kill(pid,SIGUSR1);
            }
        }
    }
}
```

```

        wait();
        break;
    }
    sleep(2);
}
printf("Parent Process will stop.\n");
}
return 0;
}

```

5) Windows 下“哲学家就餐”问题

(1) 死锁解法 (DeadlockPhilosDining.c)

在 Windows 下，首先定义全局变量，包括哲学家数量，每个哲学家的筷子等，具体如下代码：

```

const unsigned pCnt = 5; //哲学家数量
const unsigned sleepTime = 800; //每次输出后的休眠时间

bool goon = true; //用于退出的标志
int pS[pCnt]; //每个哲学家的筷子(0:思考,1:一只筷子,2:进餐)
HANDLE s[pCnt]; //每根筷子对应的信号量

```

在 main 函数中，使用 CreateSemaphore() 函数为每个筷子初始化信号量 s[i]，表示每根筷子是否可用（1 可用，0 不可用），然后创建每个哲学家的线程。最后 main 函数主线程进入死循环，直到有输入时停止。该部分代码如下：

```

int main() {
    srand(unsigned(time(nullptr)));
    for (int i = 0; i < pCnt; ++i) {
        s[i] = CreateSemaphore(NULL, 1, 1, NULL);
    }
}

```

```

    }
    HANDLE hThread[pCnt];
    DWORD phs[pCnt];
    for (int i = 0; i < pCnt; ++i) {
        hThread[i] = CreateThread(NULL, 0, philosopher, LPVOID(i), 0, &phs[i]);
        if (!hThread[i]) return -1;
    }
    while (goon) {
        if (getchar()) goon = false;
    }
    return 0;
}

```

哲学家线程为函数 `philosopher`，进行循环思考和进餐。每次就餐过程，先要使用 `WaitForSingleObject()` 函数作为 P 操作，依次等待左侧筷子和右侧筷子的信号量可用，然后使用函数 `dining()` 进餐，接下来再使用 `ReleaseSemaphore()` 函数，依次放下右侧筷子和左侧筷子，作为 V 操作使两者对应信号量可用。

该部分代码如下：

```

DWORD WINAPI philosopher(LPVOID lpPara) {
    int i = int(lpPara);
    while (goon) {
        auto step = rand() % 400 + 100;
        Sleep(step);
        WaitForSingleObject(s[i], INFINITE); //等待左侧筷子可用
        beforeDining(i, i);
        WaitForSingleObject(s[(i + pCnt - 1) % pCnt], INFINITE); //等待右侧筷子可用
        beforeDining(i, (i + pCnt - 1) % pCnt);
        dining(i);
        ReleaseSemaphore(s[(i + pCnt - 1) % pCnt], 1, NULL); //放下右侧筷子
    }
}

```

```
        afterDining(i, (i + pCnt - 1) % pCnt);  
        ReleaseSemaphore(s[i], 1, NULL);    //放下左侧筷子  
        afterDining(i, i);  
    }  
    return 0;  
}
```

其中，beforeDining()函数用于拿筷子后输出信息，并对哲学家的筷子数+1，afterDining()函数用于放下筷子后输出信息，并对哲学家的筷子数-1。dining()函数表示哲学家在进餐，并输出全部哲学家的筷子数。

该部分代码如下：

```
inline void beforeDining(int i, int k) {  
    printf("哲学家%d 拿起%d 号筷子 现有%d 支筷子 ", i, k, ++pS[i]);  
    if (pS[i] == 1) printf("不能进餐\n");  
    else if (pS[i] == 2) printf("开始进餐\n");  
}  
  
inline void afterDining(int i, int k) {  
    printf("哲学家%d 放下%d 号筷子\n", i, k);  
    --pS[i];  
}  
  
inline void dining(int i) {  
    printf("哲学家%i 就餐\n", i);  
    printf("[0 号:%d] [1 号:%d] [2 号:%d] [3 号:%d] [4 号:%d]\n",  
        pS[0], pS[1], pS[2], pS[3], pS[4]);  
    Sleep(sleepTime);  
}
```

有以上分析和代码易知，若 0-4 号哲学家都拿起了左侧的筷子，则会造成“死锁”现象，即每个哲学家都在等待右手边的筷子被放下，而一直无法进餐。

（2）非死锁解法（PhilosDining.c）

非死锁解法即避免产生上述的所有哲学家都拿到一根筷子的情况，此处采用的解决方案是引入一个变量 `diningCnt`，用来记录当前拿到筷子（1 根或 2 根）的哲学家数，要求拿到筷子的哲学家数不能超过“总哲学家数-1”的个数，而若人数达到“总哲学家数-1”时，要求其他哲学家不能拿筷子。这样就确保至少有 1 个哲学家能够顺利进餐，从而不会产生死锁。

此外，对于该计数变量 `diningCnt`，由于可能存在多个哲学家线程同时拿筷子的情况，因此又引入了一个临界区变量 `cs`，以确保不同线程对 `diningCnt` 变量的访问是互斥的，以保证计数的正确性。

由于程序整体与死锁解法一致，仅在哲学家线程中需要加上拿筷子人数计数部分，因此以下仅列出该部分代码：

```
DWORD WINAPI philosopher(LPVOID lpPara) {
    int i = int(lpPara);
    while (goon) {
        auto step = rand() % 400 + 100;
        Sleep(step);
        //判断当前拿筷子的人数，若达到"总数-1"则不允许拿筷子
        if (diningCnt == pCnt - 1) continue;

        WaitForSingleObject(s[i], INFINITE); //等待左侧筷子可用

        EnterCriticalSection(&cs);
        ++diningCnt;    //拿筷子人数+1
        LeaveCriticalSection(&cs);

        beforeDining(i, i);
        WaitForSingleObject(s[(i + pCnt - 1) % pCnt], INFINITE); //等待右侧筷子可用
        beforeDining(i, (i + pCnt - 1) % pCnt);
    }
}
```

```

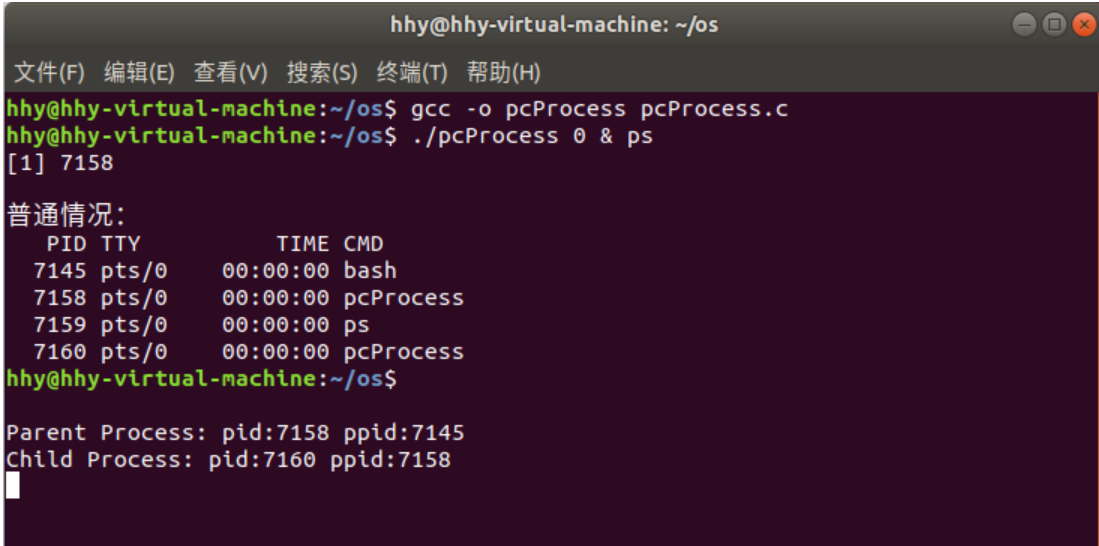
    dining(i);
    ReleaseSemaphore(s[(i + pCnt - 1) % pCnt], 1, NULL); //放下右侧筷子
    afterDining(i, (i + pCnt - 1) % pCnt);
    ReleaseSemaphore(s[i], 1, NULL);    //放下左侧筷子
    afterDining(i, i);

    EnterCriticalSection(&cs);
    --diningCnt; //拿筷子人数-1
    LeaveCriticalSection(&cs);
}
return 0;
}

```

四、实验结果

1) Linux 下创建父子进程 (pcProcess.c)



```


hhy@hhy-virtual-machine: ~/os
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
hhy@hhy-virtual-machine:~/os$ gcc -o pcProcess pcProcess.c
hhy@hhy-virtual-machine:~/os$ ./pcProcess 0 & ps
[1] 7158
普通情况:
  PID TTY          TIME CMD
  7145 pts/0        00:00:00 bash
  7158 pts/0        00:00:00 pcProcess
  7159 pts/0        00:00:00 ps
  7160 pts/0        00:00:00 pcProcess
hhy@hhy-virtual-machine:~/os$
Parent Process: pid:7158 ppid:7145
Child Process: pid:7160 ppid:7158

```

图 4-1 普通情况下父子进程及 ps 查看进程号

如图 4-1，为程序在普通情况，即父子进程休眠时间相同，结束时间接近的情况下的输出，可以看到 pcProcess 父进程的进程号为 7158，其父进程号为 7145，对应着 bash 程序；而 pcProcess 子进程的进程号为 7160，其父进程号为 7158，对应为 pcProcess 父进程。

通过 ps 命令也验证了进程号的正确性。



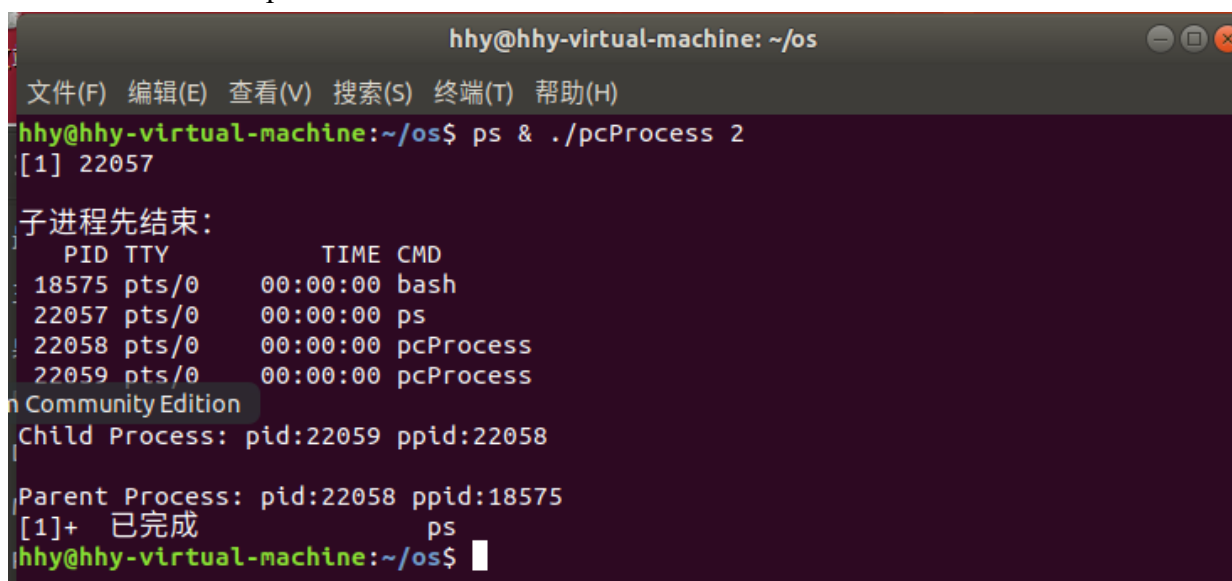
```
hhy@hhy-virtual-machine: ~/os
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
hhy@hhy-virtual-machine:~/os$ ps & ./pcProcess 1
[1] 21814

父进程先结束:
  PID TTY          TIME CMD
 18575 pts/0        00:00:00 bash
 21814 pts/0        00:00:00 ps
 21815 pts/0        00:00:00 pcProcess

Parent Process: pid:21815 ppid:18575
[1]+  已完成                  ps
hhy@hhy-virtual-machine:~/os$
Child Process: pid:21816 ppid:1535
```

图 4-2 父进程先结束情况下父子进程输出

如图 4-2，为父进程先结束情况下的输出，可以看到父进程结束后子进程的父进程变为了 1533，且未出现在 ps 的结果中，可推测该进程为系统进程。



```
hhy@hhy-virtual-machine: ~/os
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
hhy@hhy-virtual-machine:~/os$ ps & ./pcProcess 2
[1] 22057

子进程先结束:
  PID TTY          TIME CMD
 18575 pts/0        00:00:00 bash
 22057 pts/0        00:00:00 ps
 22058 pts/0        00:00:00 pcProcess
 22059 pts/0        00:00:00 pcProcess

Community Edition
Child Process: pid:22059 ppid:22058

Parent Process: pid:22058 ppid:18575
[1]+  已完成                  ps
hhy@hhy-virtual-machine:~/os$
```

图 4-3 子进程先结束情况下父子进程输出

如图 4-3，为子进程先结束情况下的输出，可以看到 PCProcess 的父子进程关系和普通情况下保存一致。

2) Linux 下父子进程同步 (syn.c)

如图 4-4，子进程休眠 5s 后终止，父进程输出了子进程的进程号以及其返回信息。

```
hhy@hhy-virtual-machine:~/os$ ./syn
This is Child Process
Child Process will stop
This is Parent Process
Child's pid=22897 exit status=6
hhy@hhy-virtual-machine:~/os$
```

图 4-4 父子进程同步输出

3) Windows 下“生产者-消费者”同步控制 (ProducerConsumer.c)

```
Microsoft Visual Studio 调试控制台
生产产品: 2001
将产品放入缓冲区
缓冲区状态: (1)2001<-生产 (2)NULL (3)NULL (4)NULL (5)NULL (6)NULL (7)NULL (8)NULL (9)NULL (10)NULL
生产产品: 1001
将产品放入缓冲区
缓冲区状态: (1)2001 (2)1001<-生产 (3)NULL (4)NULL (5)NULL (6)NULL (7)NULL (8)NULL (9)NULL (10)NULL
生产产品: 2002
将产品放入缓冲区
缓冲区状态: (1)2001 (2)1001 (3)2002<-生产 (4)NULL (5)NULL (6)NULL (7)NULL (8)NULL (9)NULL (10)NULL
从缓冲区取出产品
缓冲区状态: (1)2001<-消费 (2)1001 (3)2002 (4)NULL (5)NULL (6)NULL (7)NULL (8)NULL (9)NULL (10)NULL
消费产品: 2001
从缓冲区取出产品
缓冲区状态: (1)NULL (2)1001<-消费 (3)2002 (4)NULL (5)NULL (6)NULL (7)NULL (8)NULL (9)NULL (10)NULL
消费产品: 1001
生产产品: 1002
将产品放入缓冲区
缓冲区状态: (1)NULL (2)NULL (3)2002 (4)1002<-生产 (5)NULL (6)NULL (7)NULL (8)NULL (9)NULL (10)NULL
生产产品: 1003
将产品放入缓冲区
缓冲区状态: (1)NULL (2)NULL (3)2002 (4)1002 (5)1003<-生产 (6)NULL (7)NULL (8)NULL (9)NULL (10)NULL
生产产品: 1004
将产品放入缓冲区
缓冲区状态: (1)NULL (2)NULL (3)2002 (4)1002 (5)1003 (6)1004<-生产 (7)NULL (8)NULL (9)NULL (10)NULL
```

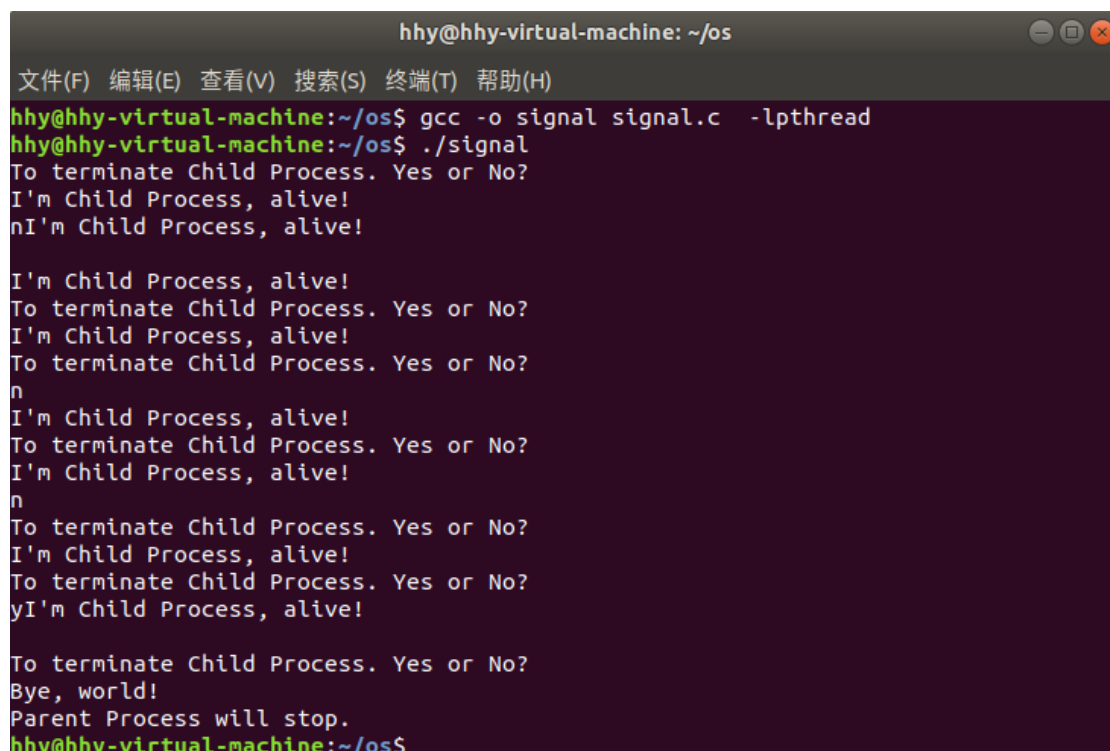
图 4-5 “生产者-消费者”同步控制输出 1

```
Microsoft Visual Studio 调试控制台
生产产品: 2003
将产品放入缓冲区
缓冲区状态: (1)NULL (2)NULL (3)2002 (4)1002 (5)1003 (6)1004 (7)2003<-生产 (8)NULL (9)NULL (10)NULL
生产产品: 2004
将产品放入缓冲区
缓冲区状态: (1)NULL (2)NULL (3)2002 (4)1002 (5)1003 (6)1004 (7)2003 (8)2004<-生产 (9)NULL (10)NULL
从缓冲区取出产品
缓冲区状态: (1)NULL (2)NULL (3)2002<-消费 (4)1002 (5)1003 (6)1004 (7)2003 (8)2004 (9)NULL (10)NULL
消费产品: 2002
从缓冲区取出产品
缓冲区状态: (1)NULL (2)NULL (3)NULL (4)1002<-消费 (5)1003 (6)1004 (7)2003 (8)2004 (9)NULL (10)NULL
消费产品: 1002
从缓冲区取出产品
缓冲区状态: (1)NULL (2)NULL (3)NULL (4)NULL (5)1003<-消费 (6)1004 (7)2003 (8)2004 (9)NULL (10)NULL
消费产品: 1003
从缓冲区取出产品
缓冲区状态: (1)NULL (2)NULL (3)NULL (4)NULL (5)NULL (6)1004<-消费 (7)2003 (8)2004 (9)NULL (10)NULL
消费产品: 1004
生产产品: 2005
将产品放入缓冲区
缓冲区状态: (1)NULL (2)NULL (3)NULL (4)NULL (5)NULL (6)NULL (7)2003 (8)2004 (9)2005<-生产 (10)NULL
从缓冲区取出产品
缓冲区状态: (1)NULL (2)NULL (3)NULL (4)NULL (5)NULL (6)NULL (7)2003<-消费 (8)2004 (9)2005 (10)NULL
消费产品: 2003
```

图 4-6 “生产者-消费者”同步控制输出 2

如图 4-5 和 4-6，显示了“生产者-消费者”同步控制的输出。2 个生产者将生产的产品从 1 号位置开始依次放入缓冲区中，而消费者从 1 号位置依次从缓冲区中取走产品，每次生产和消费操作仅有一个线程可以操作缓冲区。

4) Linux 下利用信号机制进程通信 (signal.c)



```
hhy@hhy-virtual-machine: ~/os
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
hhy@hhy-virtual-machine:~/os$ gcc -o signal signal.c -lpthread
hhy@hhy-virtual-machine:~/os$ ./signal
To terminate Child Process. Yes or No?
I'm Child Process, alive!
nI'm Child Process, alive!

I'm Child Process, alive!
To terminate Child Process. Yes or No?
I'm Child Process, alive!
To terminate Child Process. Yes or No?
n
I'm Child Process, alive!
To terminate Child Process. Yes or No?
I'm Child Process, alive!
n
To terminate Child Process. Yes or No?
I'm Child Process, alive!
To terminate Child Process. Yes or No?
yI'm Child Process, alive!

To terminate Child Process. Yes or No?
Bye, world!
Parent Process will stop.
hhy@hhy-virtual-machine:~/os$
```

图 4-7 信号进制进程通信输出

如图 4-7，显示了 Linux 下父子进程利用信号机制进行通信的输出：子进程循环输出“I’m Child Process”，父进程询问用户是否终止子进程，当用户输入“n”时，父进程休眠 2s 后继续询问，当输入为“y”时，父进程发送信号给子进程，子进程输出“Bye, world”后结束，随后父进程也结束。

5) Windows 下“哲学家就餐”问题

(1) 死锁解法 (DeadlockPhilosDining.c)

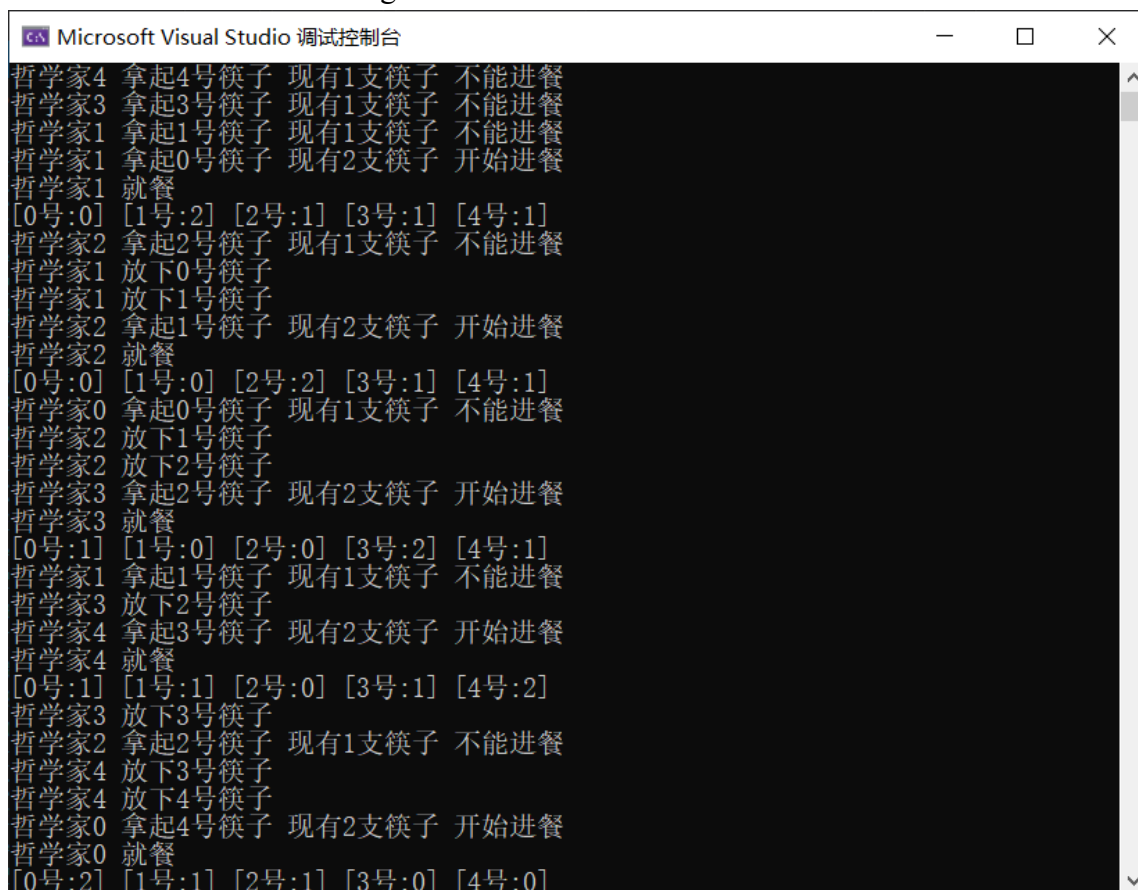


```
E:\Computer\C_Language\OExp\Debug\OExp.exe
哲学家0 拿起0号筷子 现有1支筷子 不能进餐
哲学家3 拿起3号筷子 现有1支筷子 不能进餐
哲学家2 拿起2号筷子 现有1支筷子 不能进餐
哲学家1 拿起1号筷子 现有1支筷子 不能进餐
哲学家4 拿起4号筷子 现有1支筷子 不能进餐
```

图 4-8 “哲学家就餐”死锁解法死锁时输出

如图 4-8,“哲学家问题”采用死锁解法时,容易产生如图 4-8 死锁的情形,所有哲学家均拿到了左手边的筷子而一直无法拿到右手边的筷子,造成所有哲学家线程“饥饿”,程序卡住。

(2) 非死锁解法 (PhilosDining.c)



```
Microsoft Visual Studio 调试控制台
哲学家4 拿起4号筷子 现有1支筷子 不能进餐
哲学家3 拿起3号筷子 现有1支筷子 不能进餐
哲学家1 拿起1号筷子 现有1支筷子 不能进餐
哲学家1 拿起0号筷子 现有2支筷子 开始进餐
哲学家1 就餐
[0号:0] [1号:2] [2号:1] [3号:1] [4号:1]
哲学家2 拿起2号筷子 现有1支筷子 不能进餐
哲学家1 放下0号筷子
哲学家1 放下1号筷子
哲学家2 拿起1号筷子 现有2支筷子 开始进餐
哲学家2 就餐
[0号:0] [1号:0] [2号:2] [3号:1] [4号:1]
哲学家0 拿起0号筷子 现有1支筷子 不能进餐
哲学家2 放下1号筷子
哲学家2 放下2号筷子
哲学家3 拿起2号筷子 现有2支筷子 开始进餐
哲学家3 就餐
[0号:1] [1号:0] [2号:0] [3号:2] [4号:1]
哲学家1 拿起1号筷子 现有1支筷子 不能进餐
哲学家3 放下2号筷子
哲学家4 拿起3号筷子 现有2支筷子 开始进餐
哲学家4 就餐
[0号:1] [1号:1] [2号:0] [3号:1] [4号:2]
哲学家3 放下3号筷子
哲学家2 拿起2号筷子 现有1支筷子 不能进餐
哲学家4 放下3号筷子
哲学家4 放下4号筷子
哲学家0 拿起4号筷子 现有2支筷子 开始进餐
哲学家0 就餐
[0号:2] [1号:1] [2号:1] [3号:0] [4号:0]
```

图 4-9 “哲学家就餐”非死锁解法输出

采用限制拿筷子人数的非死锁解法解决“哲学家就餐”问题,程序不会产生死锁线下,可以从输出中看到哪些哲学家拿了筷子,以及相应的状态,如图 4-9。

五、体会

1. 本次实验学习并实践了 Windows 和 Linux 编程,学习并使用了 Windows 中如何创建线程的函数 `CreateThread()`等,以及 Linux 中创建进程的函数 `fork()`等。

2. 在实验中弄清楚 Linux 下依据 `fork` 函数返回值确定父子进程的原理:`fork` 函数的返回值为该进程创建的子进程的进程号,对于父进程它会创建子进程,返回子进程的进程号,因此大于 0;而子进程不会再创建进程,因此子进程 `fork` 的返回值为 0。

3. 学习并实践了 Linux 下使用 `wait()`函数进行父子进程同步，`wait` 函数会等待子进程的信号或者结束信息，当子进程结束时会返回子进程的进程号，并将子进程的结束状态记录到形参的指针指向的地址中。再使用 `WEXITSTATUS()`函数能将 `wait` 记录的状态信息转换为状态码。在这个过程中，我也进一步了解了函数 `exit()`，该函数结束的不是“程序”，而是调用该函数的进程。

4. 学习并实践了 Windows 下的同步互斥机制，包括使用信号量、互斥锁和临界区等，比如使用函数 `WaitForSingleObject()`、`ReleaseSemaphore()`分别作为 P 和 V 操作来控制信号量，使用 `EnterCriticalSection()`、`ReleaseSemaphore()`函数实现进入临界区退出临界区，用于线程间的互斥操作等。并依照上述函数实现了“生产者-消费者”问题。

5. 学习并实践了 Linux 下的信号通信机制，使用 `kill(pid, sig)`函数完成向进程号为 `pid` 的进程发送信号 `sig`，而使用函数 `signal(sig, func)`可以用来接收信号 `sig`，并注册（触发）函数 `func`，并以此来实现进程间的通信。

6. 实践了“哲学家就餐”问题，更深入的了解了“死锁”现象产生的原理以及相应的解决方法。