

《操作系统原理》实验报告 2

姓名	郭雪菲	学号	U202112131	专业班级	网安 2104	时间	2023.11.28
----	-----	----	------------	------	---------	----	------------

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 Linux 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、实验过程

3.1 双线程循环输出数据

1) 定义线程函数

分别定义线程 A 和线程 B 需要执行的输出数据函数，一个升序输出一个降序

```
void *pthread_A(void *arg)
{
    for (int i = 0; i <= 1000; ++i)
    {
        printf("A:%04d\n", i);
        sleep(0.5);
    }
    pthread_exit(NULL);
}
```

```
void *pthread_B(void *arg)
{
    for (int i = 1000; i >= 0; --i)
    {
        printf("B:%04d\n", i);
        sleep(0.5);
    }
    pthread_exit(NULL);
}
```

2) 创建线程

```
// 创建线程
pthread_t tid_A, tid_B;
int res1 = pthread_create(&tid_A, NULL, pthread_A, NULL);
int res2 = pthread_create(&tid_B, NULL, pthread_B, NULL);
```

3) 回收线程

```
// 回收线程
pthread_join(tid_A, NULL);
pthread_join(tid_B, NULL);
return 0;
```

4) 测试程序

编译运行 c 程序，发现 A 和 B 线程循环输出，但是两者并非严格交替输出，当执行一段时间之后，可明显观察到某一个进程进度快于另一个进程，最终一个进程先行完成所有输出，只留下一个进程单独执行输出数据的操作。

```
shiftw@shiftw-virtual-machine:~/桌面/lab2$ pluma abput.c
shiftw@shiftw-virtual-machine:~/桌面/lab2$ gcc abput.c -o ab -lpthread
shiftw@shiftw-virtual-machine:~/桌面/lab2$ ./ab
A:0000
B:1000
B:0999
A:0001
A:0002
B:0998
A:0003
B:0997
```

```
A:0995
A:0996
A:0997
A:0998
A:0999
A:1000
shiftw@shiftw-virtual-machine:~/桌面/lab2$ s
```

3.2 Linux 实验 wait/exit 函数

1) 父进程不用 wait 函数

fork()创建子进程后，父进程休眠 3 秒后使用 exit()结束进程并打印进程 ID

```
if (pid > 0)
{
    sleep(3);
    printf("father_pid:%d exit\n", getpid());
    exit(0);
}
```

子进程则进入死循环，使得父进程先于子进程结束

```
if (pid == 0)
{
    int i = 0;
    while ((i++) < 10)
    {
        printf("son_pid:%d\n", getpid());
        sleep(1);
    }
}
```

编译运行程序，同时使用 ps 命令显示进程，观察到显示运行的进程与输出的子进程和父进程 ID 一致

```
shiftw@shiftw-virtual-machine:~/桌面$ ps -a
  PID TTY          TIME CMD
 155204 pts/0    00:00:00 ab
 307955 pts/0    00:00:00 ex
 307956 pts/0    00:00:00 ex
 308070 pts/1    00:00:00 ps
shiftw@shiftw-virtual-machine:~/桌面$
shiftw@shiftw-virtual-machine:~/桌面/lab2$ ./ex
son_pid:307956
son_pid:307956
son_pid:307956
father_pid:307955 exit
shiftw@shiftw-virtual-machine:~/桌面/lab2$ son_pid:307956
son_pid:307956
son_pid:307956
son_pid:307956
son_pid:307956
```

2) 父进程用 wait 函数

创建子进程之后，父进程不使用 sleep 休眠，而是用 wait 函数等待回收子进程。子进程休眠 5 秒后结束，exit()设置特定的返回参数。父进程中接收参数并 printf 子进程返回的参数。

```

pid_t pid;
int status;
int result;
pid = fork();
if (pid < 0)
{
    printf("create son failed!\n");
    return 0;
}
if (pid > 0)
{
    printf("father_pid:%d\n", getpid());
    result = wait(&status);
    if (result)
    {
        printf("son_pid:%d exit\n", result);
        printf("son exit code:%d\n", WEXITSTATUS(status));
    }
    else
    {
        printf("son exit failed\n");
    }
}
if (pid == 0)
{
    printf("create son_pid:%d\n", getpid());
    sleep(5);
    exit(131);
}

```

编译运行程序，可见五秒后子进程结束，父进程正确获取了子进程的返回参数并输出

```

shiftw@shiftw-virtual-machine:~/桌面/lab2$ gcc wait.c -o wa
shiftw@shiftw-virtual-machine:~/桌面/lab2$ ./wa
father_pid:405134
create son_pid:405135
son_pid:405135 exit
son exit code:131

```

3.3 Windows 并发线程画圆画方

1) 程序编写

在 Windows11 的 Visual Studio2022 平台上进行开发。

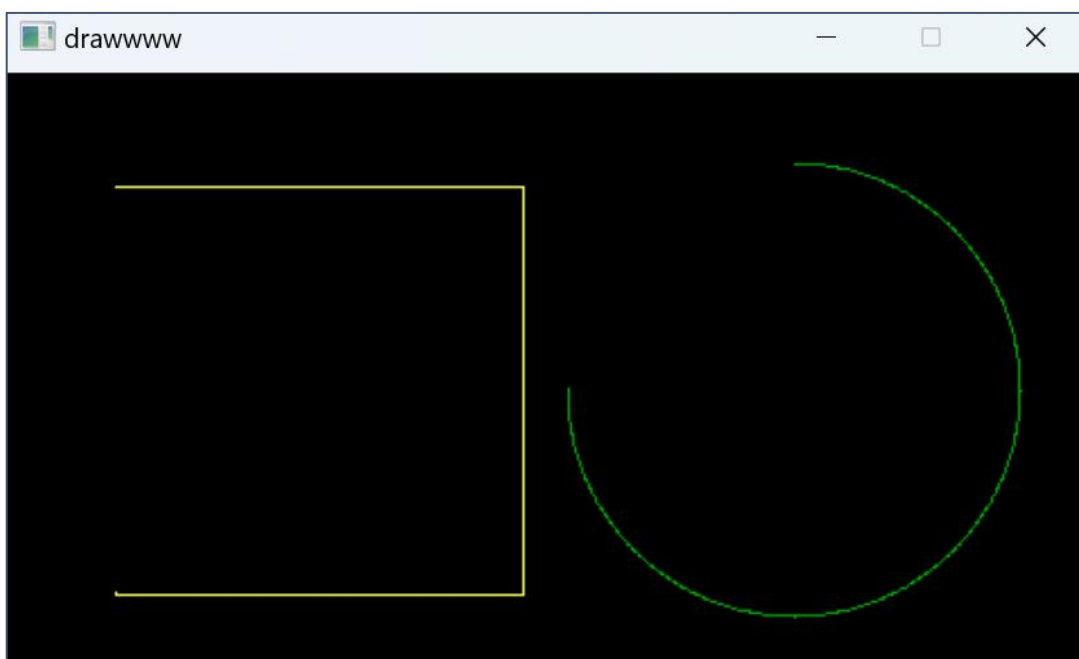
首先需要配置画图环境。画图需要使用库 graphics.h 中的 putpixel(x,y,color)等函数来绘制图像，网站下载 EasyX 头文件安装程序，运行后选择对应的开发环境安装，便可使用 Visual Studio 创建控制台项目完成程序的编写。EasyX 文档中有详细的绘图函数使用教程。

```
// 初始化图形模式
initgraph(480, 360);
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;
```

2) 运行测试

生成解决方案并运行，可观察到窗口中画圆和画方同时顺时针进行，并且绘制进度同步



3.4 Windows “生产者-消费者”同步控制

1) 开发环境

在 Windows11 下的 Visual Studio2022 平台上开发控制台项目。

2) 主程序编写

首先定义缓冲区长度、生产者个数、消费者个数、休眠时间等全局变量，并声明函数

```
#define BUFFLEN 10 // 缓冲区长度
#define PRODUCER_CNT 2 // 2个输入线程
#define CONSUMER_CNT 3 // 3个输出线程
#define SLEEP_TIME 1000 // 每次输出后的休眠时间，毫秒

int productid[2] = {1000, 2000}; // 2个生产者的生产产品起始号，1000-1999, 2000-2999
int consumeid; // 消费的产品号
int buf[BUFFLEN]; // 10个元素的缓冲区
int pidx = 0, cidx = 0; // 分别记录生产的序号和消费的序号
int consumerid[3] = {1, 2, 3}; // 消费者编号
int producerid[2] = {0, 1}; // 生产者编号
bool exitflag = true; // 用于退出的标志
HANDLE buffill, bufnull; // 信号量
CRITICAL_SECTION cse; // 临界区
```

编写 main 函数。先使用 CreateSemaphore()函数创建信号量 buffill 和 bufnull 分别表示缓冲区已有数据个数和缓冲区的空位个数，并设置信号量的初始值和最大值。然后使用 InitalizeCriticalSection() 函数初始化临界区变量 cse 用于缓冲区互斥使用。接着建立生产者和消费者的线程信息变量，再使用 CreateThead()函数分别创建生产者线程和消费者线程。最后设置程序终止方式，当用户输入任意字符时终止程序，否则程序一直执行。

```
srand(unsigned(time(nullptr)));
// 创建信号量和临界区
buffill = CreateSemaphore(NULL, 0, BUFFLEN, NULL); // 缓冲区数据个数
bufnull = CreateSemaphore(NULL, BUFFLEN, BUFFLEN, NULL); // 缓冲区空位个数
InitializeCriticalSection(&cse);
HANDLE hThread[PRODUCER_CNT + CONSUMER_CNT];
DWORD producers[PRODUCER_CNT], consumers[CONSUMER_CNT];
// 创建生产者线程
for (int i = 0; i < PRODUCER_CNT; ++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 < CONSUMER_CNT; ++i)
{
    hThread[PRODUCER_CNT + i] = CreateThread(NULL, 0, consumer, LPVOID(i), 0, &consumers[i]);
    if (!hThread[PRODUCER_CNT + i])
        return -1;
}
// 输入任意字符终止
while (exitflag)
{
    if (getchar())
        exitflag = false;
}
return 0;
```

编写生产者函数。首先根据线程号设定生产者编号，由此产生符合的产品编号。接着设置随机的间隔时间，然后判断缓冲是否有空位，有才执行生产操作。使用 `EnterCriticalSection()` 和 `LeaveCriticalSection()` 函数实现共享资源操作的互斥，通过限制有且只有一个函数进入 `CRITICAL_SECTION` 变量来实现代码段同步。具体来说，对于同一个 `CRITICAL_SECTION`，当一个线程执行了 `EnterCriticalSection()` 而没有执行 `LeaveCriticalSection()` 的时候，其它任何一个线程都无法完全执行 `EnterCriticalSection()` 而不得不处于等待状态，从而避免了两个线程同时对共享资源操作的情况。最后使用 `ReleaseSemaphore()` 函数将指定的生产信号灯对象的计数增加 1，用来标志缓冲区的填充状态。

```
DWORD WINAPI producer(LPVOID lpPara)
{
    int no = int(lpPara); // 生产者编号
    while (exitflag)
    {
        auto step = rand() % 900 + 100; // 随机一个100ms-1s的间隔时间
        WaitForSingleObject(bufnull, INFINITE); // 要求缓冲区有空位
        // INFINITE:对象被触发信号后函数才会返回
        EnterCriticalSection(&cse); // 互斥
        Sleep(step);
        produce(no);
        LeaveCriticalSection(&cse); // 离开
        ReleaseSemaphore(buffill, 1, NULL); // 数据+1
    }
    return 0;
}
```

同理完成消费者函数的编写

```
DWORD WINAPI consumer(LPVOID lpPara)
{
    int no = int(lpPara); // 消费者编号
    while (exitflag)
    {
        auto step = rand() % 900 + 100; // 随机一个100ms-1s的间隔时间
        WaitForSingleObject(buffill, INFINITE); // 要求缓冲区有产品
        EnterCriticalSection(&cse);
        Sleep(step);
        consume(no);
        LeaveCriticalSection(&cse);
        ReleaseSemaphore(bufnull, 1, NULL);
    }
    return 0;
}
```

接着编写生产函数。首先生成对应的产品编号，1号生产者生产1000-1999号产品，2号生产者生产2000-2999号产品。然后使用模数实现缓冲区的循环队列，遍历缓冲区找到合适的放入位置放入产品后休眠。打印相关信息。消费函数同理。

```
void produce(int no)
{
    auto productid = ++productid[no];
    cout << "生产产品:" << productid << endl;
    cout << "将产品放入缓冲区:" << pidx + 1 << endl;
    buf[pidx] = productid;
    cout << "缓冲区状态:";
    for (int i = 0; i < BUFFLEN; ++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) % BUFFLEN;
    Sleep(SLEEP_TIME);
}

void consume(int no)
{
    consumeid = buf[cidx];
    cout << "消费者" << consumerid[no] << "消费产品:" << consumeid << endl;
    cout << "消费产品缓冲区位置:" << cidx + 1 << endl;
    cout << "缓冲区状态:";
    for (int i = 0; i < BUFFLEN; ++i)
    {
        if (buf[i] != 0)
            cout << i + 1 << ':' << buf[i] << ' ';
        else
            cout << i + 1 << ':' << "NULL ";
        if (i == cidx)
            cout << "<-消费 ";
    }
    cout << endl
        << endl;
    buf[cidx] = 0;
    cidx = (cidx + 1) % BUFFLEN;
    Sleep(SLEEP_TIME);
}
```


3) 测试程序

生成解决方案，调试运行结果如下。生产者和消费者线程都能按照要求运行。

```
D:\visualstudiodoc\producerc x + v
生产产品:2005
将产品放入缓冲区:2
缓冲区状态:1:1007 2:2005 <-生产 3:NULL 4:NULL 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL
生产产品:1008
将产品放入缓冲区:3
缓冲区状态:1:1007 2:2005 3:1008 <-生产 4:NULL 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL
消费者1消费产品:1007
消费产品缓冲区位置:1
缓冲区状态:1:1007 <-消费 2:2005 3:1008 4:NULL 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL
生产产品:2006
将产品放入缓冲区:4
缓冲区状态:1:NULL 2:2005 3:1008 4:2006 <-生产 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL
消费者3消费产品:2005
消费产品缓冲区位置:2
缓冲区状态:1:NULL 2:2005 <-消费 3:1008 4:2006 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL
生产产品:1009
将产品放入缓冲区:5
缓冲区状态:1:NULL 2:NULL 3:1008 4:2006 5:1009 <-生产 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL
消费者2消费产品:1008
消费产品缓冲区位置:3
缓冲区状态:1:NULL 2:NULL 3:1008 <-消费 4:2006 5:1009 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL
```

3.5 Linux 信号机制实现进程通信

1) 共享内存实现

定义整型变量 `shm_id` 用于保存共享内存的 `id`，通过 `shmget()` 函数创建共享内存，参数 `IPC_CREATE | IPC_EXCL` 设置每次创建的都是新的一块内存，`S_IRUSR | S_IWUSR` 设置可以对共享内存进行读写

```
// 创建共享内存
int shm_id;
int *share_mem;
shm_id = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

2) 父进程

父进程通过 `shmat()` 函数获取共享内存指针，通过指针访问共享内存读取子进程 `pid`，之后断开共享内存链接并且关闭共享内存。然后进入死循环，每隔两秒询问用户是否终止子进程，同时发送 `SIGTOP` 信号暂时暂停子进程，避免子进程与父进程一同输出导致信息混乱。如果用户选择终止子进程，则发送 `SIGUSR1` 信号终止子进程并跳出循环；如果用户选择不关闭，则发送 `SIGCONT` 信号让子进程继续运行。

```

if (pid > 0)
{
    char input;
    int child_pid;
    share_mem = (int *)shmat(shm_id, 0, 0);
    child_pid = *share_mem;
    shmdt(share_mem);
    shmctl(shm_id, IPC_RMID, NULL);
    while (1)
    {
        kill(child_pid, SIGSTOP); // 停止子进程
        printf("Father:%d Do you want to kill Child Process:%d ?[Y/N]", getpid(), child_pid);
        input = getchar();
        getchar();
        if (input == 'y' || input == 'Y')
        {
            kill(child_pid, SIGCONT);
            kill(child_pid, SIGUSR1); // 终止子进程
            wait(NULL);
            break;
        }
        kill(child_pid, SIGCONT); // 让停止的子进程继续进行
        sleep(2);
    }
}

```

3) 子进程

子进程通过 signal 函数接受父进程发送的信号 SIGUSR1，并自定义接受信号后的操作为执行 handler 函数，打印信息后终止进程。

```

void handler(int arg)
{
    printf("child:%d Bye, World!\n", getpid());
    exit(0);
}

```

子进程同样通过 shmat()函数获得共享内存的指针，由指针访问共享内存通过 getpid()函数将子进程的 pid 保存至共享内存中，父进程可由此获得 pid，接着子进程断开共享连接，进入死循环每隔两秒打印一次存活信息。

```

pid = fork();
if (pid == 0)
{
    signal(SIGUSR1, handler); // 自定义信号操作
    share_mem = (int *)shmat(shm_id, 0, 0);
    *share_mem = getpid();
    shmdt(share_mem);
    while (1)
    {
        printf("Child:%d I am Child Process, alive!\n", getpid());
        sleep(2);
    }
}

```

4) 运行程序

编译运行程序,可以看到子进程打印信息,父进程询问用户操作时子进程的活动会中止,直到用户输入后才继续执行下一步操作。

```
shiftw@shiftw-virtual-machine:~/桌面/lab2$ ./co
Child:149260 I am Child Process, alive!
Father:149259 Do you want to kill Child Process:149260 ?[Y/N]n
Child:149260 I am Child Process, alive!
Father:149259 Do you want to kill Child Process:149260 ?[Y/N]n
Child:149260 I am Child Process, alive!
Father:149259 Do you want to kill Child Process:149260 ?[Y/N]y
child:149260 Bye, World!
```

3.6 Windows 下模拟哲学家就餐

1) 死锁解法

在 Windows11 的 Visual Studio2022 平台上进行开发。

首先定义全局变量,包括哲学家数量、休眠时长、哲学家当前筷子数等等。

```
#define PHIL_CNT 5 // 哲学家数量
#define SLEEP_TIME 3000

bool exitflag = true; // 用于退出的标志
int phichops[PHIL_CNT]; // 每个哲学家的筷子(0:思考,1:一只筷子,2:进餐)
CRITICAL_SECTION cse; // 临界区
HANDLE s[PHIL_CNT]; // 每根筷子对应的信号量
char status[3][20] = {"思考", "一只筷子", "进餐"}; // 哲学家的三种状态
```

编写主函数。首先初始化随机函数,以便每次运行都能产生随机数以生成随机间隔。然后使用 CreateSemaphore()函数创建每个哲学家的筷子数的信号量,接着创建每个哲学家的线程。最后构建终止程序的方法,当用户输入任意字符即终止程序,否则继续执行。

```
srand(unsigned(time(nullptr)));
for (int i = 0; i < PHIL_CNT; ++i)
    s[i] = CreateSemaphore(NULL, 1, 1, NULL);
HANDLE hThread[PHIL_CNT];
DWORD phs[PHIL_CNT];
for (int i = 0; i < PHIL_CNT; ++i)
{
    hThread[i] = CreateThread(NULL, 0, philosopher, LPVOID(i), 0, &phs[i]);
    if (!hThread[i])
        return -1;
}
while (exitflag)
{
    if (getchar())
        exitflag = false;
}
return 0;
```

接着编写哲学家函数 `philosopher()`，模拟哲学家就餐的过程。具体来说，每次先产生一个 100-500ms 的随机时间间隔，然后使用 `WaitForSingleObject()` 函数监视左侧筷子，直到左侧筷子空闲才进行就餐准备。接着继续监视右侧筷子，直到右侧筷子也空闲，执行就餐准备之后正式就餐。之后便是逆过程，释放筷子并结束用餐

```
DWORD WINAPI philosopher(LPVOID lpPara)
{
    int i = int(lpPara);
    while (exitflag)
    {
        auto stop = rand() % 400 + 100; // 产生100-500ms的随机时长
        Sleep(stop);
        WaitForSingleObject(s[i], INFINITE); // 等待左侧筷子可用
        beforedining(i, i);
        WaitForSingleObject(s[(i + PHIL_CNT - 1) % PHIL_CNT], INFINITE); // 等待右侧筷子可用
        beforedining(i, (i + PHIL_CNT - 1) % PHIL_CNT);
        dining(i);
        ReleaseSemaphore(s[(i + PHIL_CNT - 1) % PHIL_CNT], 1, NULL); // 放下右侧筷子
        afterdining(i, (i + PHIL_CNT - 1) % PHIL_CNT);
        ReleaseSemaphore(s[i], 1, NULL); // 放下左侧筷子
        afterdining(i, i);
    }
    return 0;
}
```

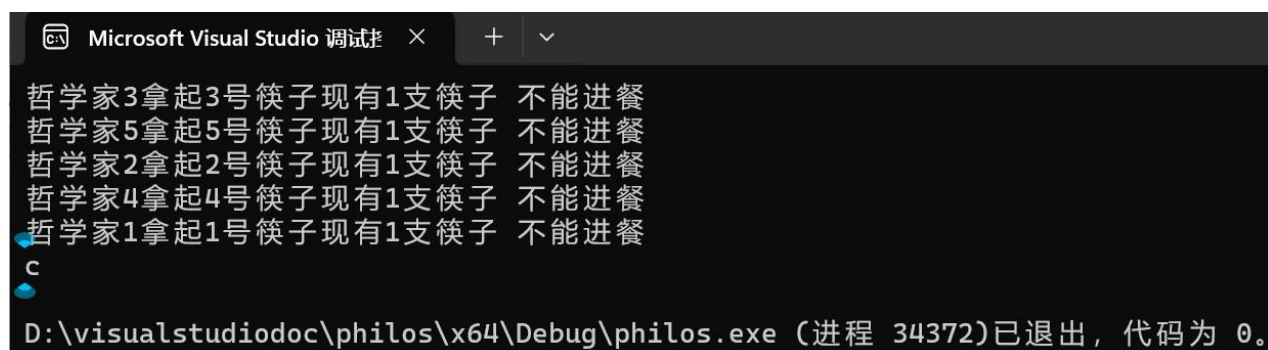
接着分别编写哲学家拿筷子、释放筷子、就餐三个函数，并在过程中输出状态信息，具体函数编写不难，如下所示

```
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 + 1, k + 1, ++phichops[i]);
    if (phichops[i] == 1)
        printf("不能进餐\n");
    else if (phichops[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 + 1, k + 1);
    --phichops[i];
}

inline void dining(int i)
{
    printf("哲学家%d就餐\n", i + 1);
    printf("哲学家1:%s 哲学家2:%s 哲学家3:%s 哲学家4:%s 哲学家5号:%s\n\n",
        status[phichops[0]], status[phichops[1]], status[phichops[2]], status[phichops[3]], status[phichops[4]]);
    Sleep(SLEEP_TIME);
}
```


Windows11 下在 Visual Studio2022 中生成解决方案，调试运行结果如下，观察到死锁立刻发生，没有哲学家能进餐



```
Microsoft Visual Studio 调试 × + v
哲学家3拿起3号筷子现有1支筷子 不能进餐
哲学家5拿起5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4拿起4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
c
D:\visualstudiodoc\philos\x64\Debug\philos.exe (进程 34372)已退出，代码为 0。
```

2) 非死锁解法

要避免产生死锁，可以通过避免产生环路实现，也就是最多允许 4 个哲学家取筷子，这样可以保证至少一个哲学家能够顺利进餐，就避免给了死锁情况的产生。

故需要设置一个新变量记录拿筷子的人数，最大为 4，同时这个变量是一个临界资源，所以要使用一个临界区 cse，以确保不同线程对其访问都是互斥的，故对比死锁情况下，只需要修改哲学家函数，以及在 main 函数中添加初始化临界值语句。



```
DWORD WINAPI philosopher(LPVOID lpPara)
{
    int i = int(lpPara);
    while (exitflag)
    {
        auto stop = rand() % 400 + 100; // 产生100-500ms的随机时长
        Sleep(stop);
        if (diningcnt == PHIL_CNT - 1)
            continue; // 最多4个人同时去拿筷子
        WaitForSingleObject(s[i], INFINITE); // 等待左侧筷子可用

        EnterCriticalSection(&cse);
        ++diningcnt; // 拿筷子人数+1
        LeaveCriticalSection(&cse);

        beforedining(i, i);
        WaitForSingleObject(s[(i + PHIL_CNT - 1) % PHIL_CNT], INFINITE); // 等待右侧筷子可用
        beforedining(i, (i + PHIL_CNT - 1) % PHIL_CNT);
        dining(i);
        ReleaseSemaphore(s[(i + PHIL_CNT - 1) % PHIL_CNT], 1, NULL); // 放下右侧筷子
        afterdining(i, (i + PHIL_CNT - 1) % PHIL_CNT);
        ReleaseSemaphore(s[i], 1, NULL); // 放下左侧筷子
        afterdining(i, i);

        EnterCriticalSection(&cse);
        --diningcnt; // 拿筷子人数-1
        LeaveCriticalSection(&cse);
    }
    return 0;
}
```

生成解决方案，调试运行，可见哲学家们总有人可以进餐，死锁情况没有发生

D:\visualstudiodoc\nophilos\ × + ∨

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5拿起5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2拿起1号筷子现有2支筷子 开始进餐
哲学家2就餐
哲学家1:思考 哲学家2:进餐 哲学家3:一只筷子 哲学家4:一只筷子 哲学家5号:一只筷子

哲学家2放下1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
哲学家1:思考 哲学家2:一只筷子 哲学家3:进餐 哲学家4:一只筷子 哲学家5号:一只筷子

哲学家2放下2号筷子
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3放下3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
哲学家1:一只筷子 哲学家2:思考 哲学家3:思考 哲学家4:进餐 哲学家5号:一只筷子

四、实验结果

4.1 双线程循环输出数据

Linux 下编译运行 c 程序，发现 A 和 B 线程循环输出，但是两者并非严格交替输出，当执行一段时间之后，可明显观察到某一个进程进度快于另一个进程，最终一个进程先行完成所有输出，只留下一个进程单独执行输出数据的操作。

```
shiftw@shiftw-virtual-machine:~/桌面/lab2$ pluma abput.c
shiftw@shiftw-virtual-machine:~/桌面/lab2$ gcc abput.c -o ab -lpthread
shiftw@shiftw-virtual-machine:~/桌面/lab2$ ./ab
A:0000
B:1000
B:0999
A:0001
A:0002
B:0998
A:0003
B:0997
```

```
A:0995
A:0996
A:0997
A:0998
A:0999
A:1000
shiftw@shiftw-virtual-machine:~/桌面/lab2$ s
```

Windows11 下同样编译运行 c 程序，结果和 Linux 下的一致

```
C:\Users\shiftw\Desktop\操作系统原理\实验二>ab
B:1000
A:0000
A:0001
B:0999
A:0002
B:0998
```

4.2 Linux 实验 wait/exit 函数

1) 父进程不用 wait 函数

Linux 下编译运行程序，同时使用 ps 命令显示进程，观察到显示运行的进程与输出的子进程和父进程 ID 一致，父进程结束后子进程单独运行，继续打印进程信息。

```

shiftw@shiftw-virtual-machine:~/桌面$ ps -a
  PID TTY          TIME CMD
 155204 pts/0    00:00:00 ab
 307955 pts/0    00:00:00 ex
 307956 pts/0    00:00:00 ex
 308070 pts/1    00:00:00 ps
shiftw@shiftw-virtual-machine:~/桌面$ 
shiftw@shiftw-virtual-machine:~/桌面/lab2$ ./ex
son_pid:307956
son_pid:307956
son_pid:307956
father_pid:307955 exit
shiftw@shiftw-virtual-machine:~/桌面/lab2$ son_pid:307956
son_pid:307956
son_pid:307956
son_pid:307956
son_pid:307956

```

2) 父进程用 wait 函数

创建子进程之后，父进程不使用 sleep 休眠，而是用 wait 函数等待回收子进程。子进程休眠 5 秒后结束，exit() 设置特定的返回参数 131 待验证。

```

pid_t pid;
int status;
int result;
pid = fork();
if (pid < 0)
{
    printf("create son failed!\n");
    return 0;
}
if (pid > 0)
{
    printf("father_pid:%d\n", getpid());
    result = wait(&status);
    if (result)
    {
        printf("son_pid:%d exit\n", result);
        printf("son exit code:%d\n", WEXITSTATUS(status));
    }
    else
    {
        printf("son exit failed\n");
    }
}
if (pid == 0)
{
    printf("create son_pid:%d\n", getpid());
    sleep(5);
    exit(131);
}

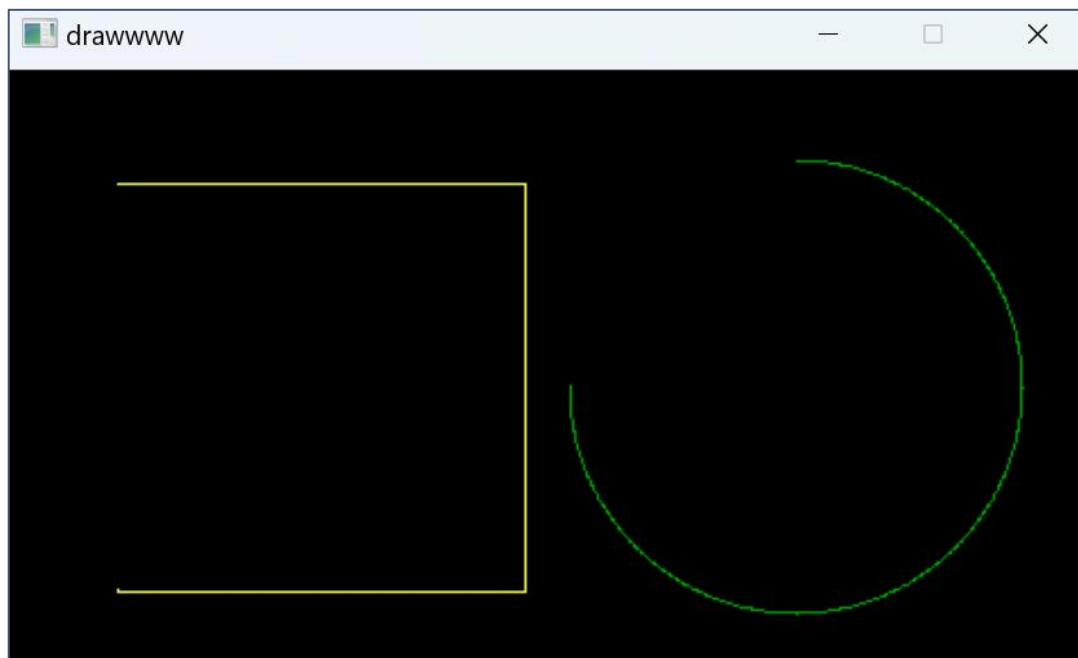
```

编译运行程序，观察到五秒后子进程结束，父进程正确获取子进程返回参数 131 并输出

```
shiftw@shiftw-virtual-machine:~/桌面/lab2$ gcc wait.c -o wa
shiftw@shiftw-virtual-machine:~/桌面/lab2$ ./wa
father_pid:405134
create son_pid:405135
son_pid:405135 exit
son_exit code:131
```

4.3 Windows 并发线程画圆画方

在 Windows11 下的 Visual Studio2022 生成项目解决方案，运行结果如下，可观察到窗口中画圆和画方同时顺时针进行，且两者进度统一



4.4 Windows “生产者-消费者” 同步控制

在 Windows11 下的 Visual Studio2022 生成项目解决方案，运行结果如下，可观察到生产者和消费者都能按照要求运行，只有当缓冲区空余的时候才进行生产，且只有当缓冲区有产品时才消费

```
D:\visualstudiodoc\producerc x + v
生产产品:2005
将产品放入缓冲区:2
缓冲区状态:1:1007 2:2005 <-生产 3:NULL 4:NULL 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL

生产产品:1008
将产品放入缓冲区:3
缓冲区状态:1:1007 2:2005 3:1008 <-生产 4:NULL 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL

消费者1消费产品:1007
消费产品缓冲区位置:1
缓冲区状态:1:1007 <-消费 2:2005 3:1008 4:NULL 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL

生产产品:2006
将产品放入缓冲区:4
缓冲区状态:1:NULL 2:2005 3:1008 4:2006 <-生产 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL

消费者3消费产品:2005
消费产品缓冲区位置:2
缓冲区状态:1:NULL 2:2005 <-消费 3:1008 4:2006 5:NULL 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL

生产产品:1009
将产品放入缓冲区:5
缓冲区状态:1:NULL 2:NULL 3:1008 4:2006 5:1009 <-生产 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL

消费者2消费产品:1008
消费产品缓冲区位置:3
缓冲区状态:1:NULL 2:NULL 3:1008 <-消费 4:2006 5:1009 6:NULL 7:NULL 8:NULL 9:NULL 10:NULL
```

4.5 Linux 信号机制实现进程通信

Linux 下编译运行程序，可以看到子进程打印信息，父进程询问用户操作时子进程的活动会中止，直到用户输入后才继续执行下一步操作。

```
shiftw@shiftw-virtual-machine:~/桌面/lab2$ ./co
Child:149260 I am Child Process, alive!
Father:149259 Do you want to kill Child Process:149260 ?[Y/N]n
Child:149260 I am Child Process, alive!
Father:149259 Do you want to kill Child Process:149260 ?[Y/N]n
Child:149260 I am Child Process, alive!
Father:149259 Do you want to kill Child Process:149260 ?[Y/N]y
child:149260 Bye, World!
```


4.6 Windows 下模拟哲学家就餐

1) 死锁解法

Windows11 下在 Visual Studio2022 中生成解决方案，调试运行结果如下，观察到死锁立刻发生，没有哲学家能进餐

```
Microsoft Visual Studio 调试 × + v
哲学家3拿起3号筷子现有1支筷子 不能进餐
哲学家5拿起5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4拿起4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
c
D:\visualstudiodoc\philos\x64\Debug\philos.exe (进程 34372)已退出，代码为 0。
```

2) 非死锁解法

采用最多允许 4 个哲学家拿筷子的方式避免环路产生，从而避免死锁，程序运行结果如下所示，可观察到死锁没有发生，每个阶段总有一名哲学家能够进餐

```
D:\visualstudiodoc\nophilos\ × + v
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5拿起5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2拿起1号筷子现有2支筷子 开始进餐
哲学家2就餐
哲学家1:思考 哲学家2:进餐 哲学家3:一只筷子 哲学家4:一只筷子 哲学家5号:一只筷子

哲学家2放下1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
哲学家1:思考 哲学家2:一只筷子 哲学家3:进餐 哲学家4:一只筷子 哲学家5号:一只筷子

哲学家2放下2号筷子
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3放下3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
哲学家1:一只筷子 哲学家2:思考 哲学家3:思考 哲学家4:进餐 哲学家5号:一只筷子
```

五、实验错误排查和解决方法

5.1 多线程循环输出数据

1. 使用 `gcc ab.c -o ab` 命令编译编写好的 c 代码报错。原因是使用了 `pthread` 库，需要加上参数 `-lpthread` 才能顺利编译。
2. 多线程休眠时间太短，运行速度大差异大，难以捕获初始打印信息。解决方法是延长 `sleep` 休眠时间为 1 秒。

5.2 Linux 实验 `wait/exit` 函数

1. 父进程设定的结束时间太快，没来得及在另一个终端使用 `ps` 命令查看进程父进程便终止。解决方案是延长父进程的存活时间，以便于观察。

5.3 Windows 并发线程画圆画方

1. 无 c 程序画图经历和知识。上网搜索发现只需要引用 `graphics.h` 库，且添加库的配置操作十分简单，只需要下载 `EasyX` 安装程序运行即可。且画图形可以抽象为画若干个点的过程，依赖通俗易懂的官方文档的帮助，最终实现了画图函数。
2. 一开始对于如何绘制圆形理不清楚思路，后来参考了网上的样例，以 $(\cos(-\pi/2 + (i * \pi) / 360)), \sin(-\pi/2 + (i * \pi) / 360))$ 为变换坐标点的基础公式。

5.4 Windows “生产者-消费者”同步控制

1. 多次运行发现结果基本相同。原因是运行时没有初始化随机数种子，导致随机函数每次喂的种子相同，由此产生的时间间隔总是相同。只需要在开始前将当前时间作为种子初始化随机函数，即可实现每次运行的随机结果。

5.5 Linux 信号机制实现进程通信

1. 一开始程序无法运行，检查后发现是父进程在访问共享内存的时候没有休眠，此时子进程还未将 `pid` 传入共享内存导致运行错误，应该休眠一段时间，等待子进程至少执行一轮循环之后再执行操作。
2. 用户在选择终止进程之后父进程和子进程均无输出和动作。原因是父进程在询问用户操作之前中止了子进程，以防止输出紊乱，但是在用户选择终止子进程之后，并没有恢复子进程的执行，导致即使发送了自定义终止信号，处于休眠态的子进程无法被唤醒执行相应的操作。正确的做法应该是在用户做出终止决策之后，先恢复子进程的执行再发送终止信号，这样子进程便能接收父进程的信号并做出反馈。

5.6 Windows 模拟哲学家就餐

1. 非死锁解法在使用临界值对象后调试报错。原因是没有进行临界值初始化操作，之后的进入和释放操作是非法的，在 main 函数中添加 `InitializeCriticalSection(&cse);` 语句即可

六、实验参考资料和网址

(1) 教学课件

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

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

(4) <https://blog.csdn.net/low5252/article/details/104800671>

(5) <https://blog.csdn.net/kxjryk/article/details/81603049>

(6) <https://blog.csdn.net/drdairen/article/details/51896141>

(7) https://blog.csdn.net/weixin_36440319/article/details/117163701

(8) <https://blog.csdn.net/humblehunger/article/details/106593577>

(9) <https://www.cnblogs.com/frisk/p/11602973.html>