华 中 科 技 大 学

网络空间安全学院

本科：《操作系统原理实验》

实验报告

题目：进程管理实验

姓　　名 _____

班　　级 _____

学　　号 _____

联系方式 _____

分　　数 _____

评 分 人 _____

2024 年 12 月 03 日

报告要求

1. 报告不可以抄袭，发现雷同者记为 **0** 分。

2. 报告中不可以只粘贴大段代码，应是文字与图、表结合的，需要说明流程的时候，也应该用流程图或者伪代码来说明；如果发现有大段代码粘贴者，报告需重写。

3. 报告格式严格按照要求规范，并作为评分标准。

课程目标评价标准

课程目标	评价环节	评价标准		
		高于预期（优良）	达到预期（及格）	低于预期(不及格)
目标 1.能够依据实际工程问题中软硬件的约束，应用操作系统的基础理论、抽象和分层设计思想，对特定操作系统进行评价、分析，并能进行合理的优化和裁减。具备设计、实现、开发小型或简化的操作系统的能力，并能体现一定程度的创新性。能熟练应用操作系统的开发和调试技术和工具。	当堂验收	能很合理地设计程序结构、算法和主要数据结构；完成 80%以上的预定功能。	能比较合理地设计程序结构、算法和关键的数据结构；完成 80%以上的预定功能。	程序结构、算法和主要数据结构设计不很合理；仅完成不到50%的预定功能。
	实验报告	预定功能全部完成；截止日期前提交；源代码完整，且可编译；源代码注释清晰可读。	预定功能 80%以上完成；截止日期前提交；源代码完整，且可编译；源代码注释比较清晰可读。	预定功能仅完成不到 50%以上完成；截止日期以后提交；内容单薄；图文排版杂乱无章；源代码缺失或无法编译；源代码无注释或注释不清晰。
目标 2.具备操作系统国产化和自主创新意识，掌握国产操作系统，例如麒麟操作系统，鸿蒙操作系统的应用和开发环境，支持和推广国产操作系统，积极建设国产操作系统的技术生态。	当堂验收	完全使用国产操作系统；代码规范；能正确回答老师的绝大部分提问。	完全使用国产操作系统；代码比较规范；能正确回答老师的半数以上提问。	仅部分功能使用国产操作系统；代码不够规范，注释缺乏；仅能正确回答老师的极少数提问。
	实验报告	技术框架非常合理高效；符合用户硬件环境和约束参数。报告内容充实；图文排版规范；使用国产操作系统完成全部实验。	技术框架基本合理高效；符合用户硬件环境和约束参数。报告内容基本充实；图文排版基本规范；大部分实验过程使用国产操作系统。	技术框架不合理，代码臃肿；不完全符合用户硬件环境和约束参数。报告内容单薄；图文排版杂乱无章；仅小部分实验过程使用国产操作系统，或完全没用。

报告评分表

评 分 项 目		满分	得分	评分标准
进程管理实验	总体设计（目标 1）	15		15-11：能够给出明确需求，系统功能完整、正确和适当。 10-6：能够给出需求，但不够完整，能够阐述系统的设计，但不够完整、恰当和准确。 5-0：需求不够明确，系统设计不够完整、正确和恰当。
	详细设计（目标 1）	15		15-11：函数和数据结构描述完整，关系清晰，流程设计正确规范。 10-6：函数和数据结构描述基本完整，流程设计基本正确。 5-0：函数和数据结构描述不完整，流程设计有错误。
	代码实现（目标 1，目标 2）	10		10-8：代码能够实现设计的功能要求，考虑错误处理和边界条件。有充分的注释，代码格式规范。 7-5：代码能够实现基本的功能要求，但可能缺少错误处理和边界条件的考虑。关键部分有简单注释，代码格式较为规范。 4-0：代码未能完全实现功能要求，缺少错误处理和边界条件的考虑。注释不足，代码格式不够规范。
	测试及结果分析（目标 1，目标 2）	20		20-14：测试方法科学、完整，结果分析准确完备。 13-8：测试方法描述基本正确、完整，结果分析准确完备。 7-0：测试仅针对数据集的通过性进行描述。
	问题描述及解决方案（目标 1）	10		10-8：遇到的问题及解决方案真实具体 7-5：遇到描述不够详细，解决方案不够具体 4-0：没有写什么内容。
感想（含思政）（目标 2）		10		10-8：感想真实具体。 7-5：感想比较空洞。 4-0：没有写什么感想。
遇到的问题和解决方案，及意见和建议（目标 1）		10		10-8：意见和建议有的放矢。 7-5：意见和建议不够明确。 4-0：没有写什么内容。
文档格式（段落、行间距、缩进、图表、编号等）（目标 1）		10		基本要求：目录、标题、行间距、缩进、正文字体字号按照模板要求执行，图、表清晰且有标号。 10-8：格式规范美观，满足要求。 7-5：基本满足要求。 4-0：格式较为混乱。
总分		**100**		
教师签名			日期	

目　　录

1 实验概述

1.1 进程管理实验

实验二：第 4 章进程管理，第 5 章死锁

1.2 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。

1.3 实验环境

操作平台：VMware Workstation Pro 16

操作系统：优麒麟 24.04，Linux 6.6.22，Window 11

编辑工具：gedit 46.2

1.4 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 下创建一对父子进程，实验 wait 同步函数；

(3) 在 Windows 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。

1.5 实验要求

1、4、6 必做，其余任选 1。

现场检查：任意 2 个，通过得 2 分。

本实验我将前六个任务均完成，下面我将更加具体展开阐述完成过程和结果。

2 实验过程

2.1 总体设计

2.1.1 任务 1 双线程循环输出数据

该任务我使用 pthread 线程库创建两个线程，线程 A 递增输出 1-1000，线程 B 递减输出 1000-1，每隔 0.2 秒输出一个数，并标注线程标识，需定义线程函数及创建和回收线程。

2.1.2 任务 2 Linux 创建父子进程，实现 wait 同步函数

该任务先创建子进程并检测异常，让子进程打印提示语句和进程号后休眠 5 秒，父进程同时打印相关信息后先结束，用 wait 函数等待回收子进程，子进程休眠完后 exit 设置返回参数，父进程检测子进程退出状态并打印对应输出。

2.1.3 任务 3 Windows 并发线程画圆画方

该任务首先初始化 Pygame，包括导入模块及窗口等初始化操作。然后定义绘制正方形和圆形的线程任务，使用 pygame.draw.circle()绘制像素点来形成图形。最后通过 threading.Thread 创建并运行线程，用 join 等待线程执行完毕，关闭 pygame 退出程序。

2.1.4 任务 4 Linux“生产者-消费者”同步控制

该任务设计通过信号量和互斥锁实现生产者 - 消费者模型。先进行宏定义和全局变量声明，用数组作缓冲区，定义生产者和消费者线程函数，生产者生成随机数放入缓冲区，消费者从缓冲区取数输出，主函数进行变量初始化、创建线程、等待线程结束及销毁资源等操作。

2.1.5 任务 5 Linux 信号机制实现进程通信

该任务先创建共享内存，子进程注册 SIGUSR1 信号处理函数，通过 fork 创建子进程，子进程将 pid 写入共享内存后分离，进入循环输出状态信息。父进程等待 2 秒后获取子进程 pid，根据用户输入决定是否终止子进程。

2.1.6 任务 6 Linux 下模拟哲学家就餐

该任务我设计了死锁解法和非死锁解法。死锁解法是定义互斥锁数组代表筷子，在哲学家行为函数中通过无限循环模拟思考、进餐、放筷子等行为，使用互斥锁保证对筷子资源访问的互斥性。非死锁解法则使用 pthread_mutex_trylock 函数尝试获取互斥锁，避免死锁情况发生。

2.2 详细设计

2.2.1 任务 1 双线程循环输出数据

（1）定义线程函数

可以使用 pthread 线程库，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。输出数据时，同时输出 A 或 B 以标示是哪个线程输出的，并注意格式化输出信息。

```c
// 线程A的函数，递增输出
void* pthread_a(void* arg) {
    for (int i = 0; i < 1000; i++) {
        printf("A:%04d\n", i);
        sleep(0.2);
    }
    return NULL;
}
```

```c
// 线程B的函数，递减输出
void* pthread_b(void* arg) {
    for (int i = 1000; i > 0; i--) {
        printf("B:%04d\n", i);
        sleep(0.2);
    }
    return NULL;
}
```

图 2 - 1 当前内核版本截图

（2）创建和回收线程

```c
    // 创建线程
    if (pthread_create(&tid_a, NULL, pthread_a, NULL) != 0) {
        perror("Failed to create thread A");
        return 1;
    }
    if (pthread_create(&tid_b, NULL, pthread_b, NULL) != 0) {
        perror("Failed to create thread B");
        return 1;
    }

    // 回收线程
    pthread_join(tid_a, NULL);
    pthread_join(tid_b, NULL);
```

2.2.2 任务 2 Linux 创建父子进程，实现 wait 同步函数

创建代码如下，首先创建子进程并进行异常情况检测。

```c
    // 创建子进程
    pid = fork();

    if (pid < 0) {  // 如果 fork() 失败
        perror("fork失败");
        exit(1);
    }
```

然后先让子进程打印提示语句和进程号，然后休眠五秒，父进程同时打印提示语句和进程号，父进程先结束，用 wait 函数等待回收子进程，然后子进程休眠完后 exit()设置特定的返回参数（这里**取我学号的后两位即 15**），父进程检测子进程退出状态，并打印相对应的输出。

```c
    if (pid == 0) {  // 子进程
        printf("子进程的PID为%d\n", getpid());
        // 子进程休眠 5 秒
        sleep(5);
        printf("子进程结束休眠.此时状态码为15\n");
        exit(15);  // 子进程返回状态码15
    } else {  // 父进程
        printf("父进程PID为%d, 正在等待子进程%d结束...\n", getpid(), pid);

        // 父进程等待子进程结束
        wait(&status);  // 等待子进程退出

        // 检查子进程的退出状态
        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);  // 获取子进程的退出状态
            printf("父进程: 子进程存在且状态码为%d.\n", exit_status);
        } else {
            printf("父进程: 子进程不存在.\n");
        }
    }
```

2.2.3 任务 3 Windows 并发线程画圆画方

任务 3 的核心函数在于定义**绘制正方形和圆形的线程任务**。

下面两个函数分别用于绘制正方形和圆形。将正方形的每条边分成 180 个点来绘制，我们使用 pygame.draw.circle() 绘制单个像素点，位置会随 i 增长，形成一条边。我们将圆的中心设为 $(350, 140)$，半径为 100，然后通过三角函数 cos 和 sin 来计算每个点的 (x, y) 坐标。time.sleep(0.005) 用于模拟 Sleep(0.5)，让绘制过程变得可见。

```python
# 画正方形的线程函数
def draw_square():
    for i in range(180):
        screen.set_at((50 + i, 50), YELLOW)  # 画第一条边
        pygame.display.update()
        time.sleep(0.005)

    for i in range(180):
        screen.set_at((50 + 180, 50 + i), YELLOW)  # 画第二条边
        pygame.display.update()
        time.sleep(0.005)

    for i in range(180):
        screen.set_at((50 + 180 - i, 50 + 180), YELLOW)  # 画第三条边
        pygame.display.update()
        time.sleep(0.005)

    for i in range(180):
        screen.set_at((50, 50 + 180 - i), YELLOW)  # 画第四条边
        pygame.display.update()
        time.sleep(0.005)
```

```python
# 画圆的线程函数
def draw_circle():
    for i in range(720):
        x = int(350 + 100 * math.cos(-PI / 2 + (i * PI) / 360))
        y = int(140 + 100 * math.sin(-PI / 2 + (i * PI) / 360))
        screen.set_at((x, y), GREEN)
        pygame.display.update()
        time.sleep(0.005)
```

2.2.4 任务 4 Linux“生产者-消费者”同步控制

该任务主要实现了一个生产者 - 消费者模型，通过信号量和互斥锁的协同工作，确保了多个生产者和消费者线程能够安全、有序地访问共享缓冲区，避免了数据竞争和资源冲突。生

产者不断生产数据放入缓冲区，消费者从缓冲区中取出数据进行消费，并且缓冲区的大小是有限的，通过信号量的控制来实现生产者和消费者之间的同步和协调。

而任务 4 的核心部分为**生产者和消费者线程函数**，我将针对其设计逻辑展开阐述。

同时 Linux 使用互斥锁对象和轻量级信号量对象，主要函数：**①同步操作。**sem_wait(&empty)等待 empty 信号量，表示等待缓冲区有空闲空间。如果 empty 的值大于 0，说明有空闲空间，信号量值减 1，线程继续执行；如果 empty 的值为 0，线程会阻塞在这里，直到有其他线程释放 empty 信号量。pthread_mutex_lock(&mutex)获取互斥锁，确保在写入缓冲区时没有其他线程同时访问。**②释放资源。**pthread_mutex_unlock(&mutex)用来释放互斥锁，允许其他线程访问缓冲区（如果有等待的线程）。sem_post(&full)用来增加 full 信号量的值，表示缓冲区中有了新的数据，可供消费者消费。

```
// 生产者线程函数
void *producer(void *pno) {
    int item;
    while (1) {
        // 生成一个随机数作为生产的物品，范围在1000到1999之间，并根据生产者编号区分
        item = rand() % 1000 + 1000 * (*((int *)pno));
        sleep(1); // 模拟生产过程中的延迟

        // 等待缓冲区有空闲空间
        sem_wait(&empty);
        // 获取互斥锁，保护缓冲区写入操作
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        // 输出生产者生产物品的信息
        printf("生产者 %d: 在位置 %d 生产物品 %d\n", *((int *)pno), in, buffer[in]);
        in = (in + 1) % BufferSize; // 更新写入位置索引
        // 释放互斥锁
        pthread_mutex_unlock(&mutex);
        // 增加缓冲区中已有数据数量的信号量
        sem_post(&full);
    }
}
```

同时完成消费者线程函数，同理，不过在最后需要加上 sleep(2)为了模拟消费过程中的延迟，使消费者线程暂停 2 秒，模拟消费物品所需的时间。

```
// 消费者线程函数
void *consumer(void *cno) {
    while (1) {
        // 等待缓冲区有数据可供消费
        sem_wait(&full);
        // 获取互斥锁，保护缓冲区读取操作
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        // 输出消费者消费物品的信息
        printf("消费者 %d: 从位置 %d 消费物品 %d\n", *((int *)cno), out, item);
        out = (out + 1) % BufferSize; // 更新读取位置索引
        // 释放互斥锁
        pthread_mutex_unlock(&mutex);
        // 增加缓冲区空闲空间数量的信号量
        sem_post(&empty);
        sleep(2); // 模拟消费过程中的延迟
    }
}
```

2.2.5 任务 5 Linux 信号机制实现进程通信

（1）共享内存创建与进程创建

定义整型变量 shmid 用于保存共享内存的 id，通过 shmget()函数创建共享内存。IPC_PRIVATE 表示创建一个新的私有共享内存段，其键值由系统自动生成。IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR 用于创建具有特定权限（用户可读可写）的共享内存段。如果共享内存段已经存在（IPC_EXCL 标志），则创建失败。

```
24    // 创建共享内存段
25    pid_t pid;
26    int shm_id;
27    int *share_mem;
28    shm_id = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

（2）子进程操作

子进程注册了 SIGUSR1 信号的处理函数 handler。当子进程接收到 SIGUSR1 信号时，会执行 handler 函数，输出再见信息并退出。

```
11 // 信号处理函数
12 void handler(int arg) {
13     // 当接收到信号时，输出子进程的pid并退出进程
14     printf("Child%d:Bye, World!\n", getpid());
15     exit(0);
16 }
```

使用 fork 函数创建了一个子进程。然后，子进程将自己的 pid 写入共享内存段，通过 shmat 函数将共享内存段连接到自己的地址空间，写入 pid 后再使用 shmdt 函数分离共享内存段。最后，子进程进入一个无限循环，每隔 2 秒输出自己的状态信息，表明自己正在运行。

```
29    pid = fork();
30    if (pid == 0) {  // 子进程执行的代码块
31        // 注册信号处理函数，当接收到 SIGUSR1 信号时，调用 handler 函数
32        signal(SIGUSR1, handler);
33        // 将共享内存段连接到子进程的地址空间，返回指向共享内存段的指针
34        share_mem = (int *)shmat(shm_id, 0, 0);
35        // 将子进程的pid写入共享内存段
36        *share_mem = getpid();
37        // 分离共享内存段与子进程的地址空间（但并不删除共享内存段）
38        shmdt(share_mem);
39        // 子进程的主循环，每隔2秒输出一次自己的状态信息
40        while (1) {
41            printf("Child %d:I am Child Process, alive!\n", getpid());
42            sleep(2);
43        }
44    }
```

（3）父进程操作

父进程创建子进程后，先等待 2 秒，再连接共享内存获取子进程 pid 后分离，接着标记共享内存以便自动删除。随后进入主循环，先暂停子进程，询问用户是否终止，若用户选择终止（输入 y 或 Y），父进程则先恢复子进程，再发送信号触发子进程终止并等待其结束；若用

户不终止，父进程仅恢复子进程，等待 2 秒后再次循环。

```
51    if (pid > 0) {
52        // 父进程执行的代码块
53        char input;
54        int child_pid;
55        sleep(2);
56        // 将共享内存段连接到父进程的地址空间，获取子进程写入的pid
57        share_mem = (int *)shmat(shm_id, 0, 0);
58        child_pid = *share_mem;
59        // 分离共享内存段与父进程的地址空间（但并不删除共享内存段）
60        shmdt(share_mem);
61        // 标记共享内存段，使其在所有进程都不再使用后被系统自动删除
62        shmctl(shm_id, IPC_RMID, NULL);
63        // 父进程的主循环
64        while (1) {
65            // 向子进程发送 SIGSTOP 信号，使子进程暂停执行
66            kill(child_pid, SIGSTOP);
67            // 提示父进程用户是否要终止子进程
68            printf("Father %d:To terminate Child Process %d.Yes or No?[Y/N] ", getpid(), child_pid);
69            // 获取用户输入的字符，这里调用两次 getchar 是为了处理换行符
70            input = getchar();
71            getchar();
72            if (input == 'y' || input == 'Y') {
73                // 如果用户输入 'y' 或 'Y'，则向子进程发送 SIGCONT 信号恢复子进程执行
74                kill(child_pid, SIGCONT);
75                // 向子进程发送 SIGUSR1 信号，触发子进程的信号处理函数来终止子进程
76                kill(child_pid, SIGUSR1);
77                // 等待子进程结束
78                wait(NULL);
79                break;
80            }
81            // 如果用户输入不是 'y' 或 'Y'，则向子进程发送 SIGCONT 信号恢复子进程执行
82            kill(child_pid, SIGCONT);
83            sleep(2);
```

2.2.6 任务 6 Linux 下模拟哲学家就餐

（1）死锁解法

死锁解法则直接只需考虑思考、进餐、放筷子等行为，由于每个哲学家都占有了一根筷子且等待另一根筷子被释放，而被等待的筷子又被其他哲学家占有，导致所有哲学家都无法继续进餐，程序陷入僵局，即产生了死锁. 在这个场景中，对筷子资源的竞争以及不恰当的获取资源顺序，共同导致了死锁的产生 。

（2）非死锁解法

而针对非死锁解法，如果在尝试获取筷子的过程中，不能同时获得两根筷子（即有一根筷子获取失败），那么就会释放已经获取到的筷子（如果有的话）。这样就避免了所有哲学家都占有一根筷子然后无限等待另一根筷子的死锁情况。例如，如果一个哲学家获取左边筷子成功，但获取右边筷子失败，他会立即释放左边的筷子，这样其他哲学家就有机会获取这根筷子，从而保证了程序能够持续运行，不会因为资源占用的僵持而陷入死锁。

具体代码实现将在 2.3.3 进行逻辑阐述。

2.3 代码实现

本实验的任务均及多线程或多进程操作，需要解决资源共享与同步问题以及强调并发执行的控制和协调，基于这些共同点，在代码实现部分，都需要使用相应的操作系统提供的并发编

程接口和同步机制来实现具体的功能。例如，在 Linux 中使用 pthread 库来创建和管理线程，使用信号量和互斥锁来实现线程间的同步和资源的互斥访问；在 Windows 中使用 threading 模块等类似的机制。并且，每个任务都需要根据具体的业务逻辑，合理地组织代码结构，处理好线程或进程的创建、启动、暂停、恢复、终止等操作，以及资源的分配、使用和释放，以确保程序的正确性和高效性。由于相似性，便不一一进行编写逻辑描述，我选取了其中三个有代表性的题目对代码整体进行更加详细地介绍。

2.3.1 任务 3 Windows 并发线程画圆画方

（1）初始化 Pygame

先导入需要的模块，其中 Pygame 是一个用于创建图形界面和绘制图形的库。我们在这个程序中使用它来绘制方形和圆形，接着是一些初始化的操作，如窗口大小，窗口标题，颜色等。

（2）创建线程并运行

使用 threading.Thread(target=...)来创建一个线程对象，target 参数指定线程的目标函数。通过 start()启动线程，开始执行目标函数进行同步画圆和画方，通过 join()等待线程执行完毕，而 join() 会阻塞主线程，直到相应的线程执行完毕。在 pygame.quit()之前，程序等待所有线程完成工作。一旦所有绘制任务完成，就关闭 pygame，退出程序。

```
52      # 创建线程
53      square_thread = threading.Thread(target=draw_square)
54      circle_thread = threading.Thread(target=draw_circle)
55
56      square_thread.start()
57      circle_thread.start()
58
59      # 等待所有线程结束
60      square_thread.join()
61      circle_thread.join()
62
63      # 保持窗口直到用户关闭
64      pygame.time.wait(2000)  # 等待2秒钟关闭窗口
65      pygame.quit()
```

2.3.2 任务 4 Linux“生产者-消费者”同步控制

（1）宏定义和全局变量声明

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。

```c
#define BufferSize 10   // 缓冲区大小
#define countOfProducer 2 // 生产者数量
#define countOfConsumer 3 // 消费者数量

sem_t empty;    // 表示缓冲区空闲空间数量的信号量
sem_t full;     // 表示缓冲区中已有数据数量的信号量
int in = 0;      // 用于指示缓冲区写入位置的索引
int out = 0;     // 用于指示缓冲区读取位置的索引
int buffer[BufferSize] = {0}; // 存储数据的缓冲区

pthread_mutex_t mutex; // 用于保护缓冲区操作的互斥锁
```

（2）生产者和消费者线程函数

Linux 使用互斥锁对象和轻量级信号量对象，编写使用的函数主要有 sem_wait( )，sem_post( )，pthread_mutex_lock( )，pthread_mutex_unlock( ) 等。我们可以通过 rand()和 pno 编号区分，来实现生产者 1 的数据在 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)。消费者每休眠 100ms-1s 的随机时间消费一个数据。

（3）主函数初始化和调用

进行线程相关变量声明和初始化，并创建线程，然后等待线程结束，使用 pthread_join 函数等待每个生产者和消费者线程结束，这可以确保主线程在所有子线程完成任务后再继续执行后续的销毁操作，结束后销毁资源。代码我均进行了详细的注释，细节便不再赘述。

```c
    // 初始化互斥锁
    pthread_mutex_init(&mutex, NULL);
    // 初始化表示缓冲区空闲空间数量的信号量，初始值为缓冲区大小
    sem_init(&empty, 0, BufferSize);
    // 初始化表示缓冲区中已有数据数量的信号量，初始值为0
    sem_init(&full, 0, 0);

    int a[3] = {1, 2, 3}; // 仅用于给生产者和消费者编号

    // 创建生产者线程
    for (int i = 0; i < countOfProducer; i++) {
        pthread_create(&pro[i], NULL, (void *)producer, (void *)&a[i]);
    }
    // 创建消费者线程
    for (int i = 0; i < countOfConsumer; i++) {
        pthread_create(&con[i], NULL, (void *)consumer, (void *)&a[i]);
    }

    // 等待生产者线程结束
    for (int i = 0; i < countOfProducer; i++) {
        pthread_join(pro[i], NULL);
    }
    // 等待消费者线程结束
    for (int i = 0; i < countOfConsumer; i++) {
        pthread_join(con[i], NULL);
    }

    // 销毁互斥锁
    pthread_mutex_destroy(&mutex);
    // 销毁表示缓冲区空闲空间数量的信号量
    sem_destroy(&empty);
    // 销毁表示缓冲区中已有数据数量的信号量
```

2.3.3 任务 6 Linux 下模拟哲学家就餐

（1）死锁解法

首先在全局变量定义部分，定义一个互斥锁数组，数组大小为 PHILOSOPHERS（即 5 个），每个互斥锁代表一根筷子。互斥锁用于保证在同一时刻只有一个线程（对应一个哲学家）能够访问共享资源（这里的共享资源就是筷子），避免出现数据竞争等并发问题。

```
#define PHILOSOPHERS 5

pthread_mutex_t chopsticks[PHILOSOPHERS]; // 5根筷子的互斥锁
pthread_t philosophers[PHILOSOPHERS];
```

然后设计哲学家行为函数部分，整个函数体在一个无限循环 while(1)中，不断模拟哲学家的思考、进餐、放筷子等行为。**①思考状态。**首先输出哲学家正在思考的信息，然后使用 usleep(rand() % 401 + 100)让线程暂停一段随机的时间，模拟哲学家思考所花费的时间。**②拿筷子操作。**按照先左后右的顺序拿起筷子，在拿起每根筷子时，通过 pthread_mutex_lock 函数对对应的筷子互斥锁进行加锁操作，这意味着当一个哲学家拿起某根筷子（加锁成功）后，其他哲学家就不能同时拿起这根筷子了，从而保证了对筷子资源访问的互斥性。**③进餐状态。**成功拿起两根筷子后，输出哲学家正在进餐的信息，同样使用 usleep(rand() % 401 + 100)让线程暂停一段随机的进餐时间，模拟进餐花费的时长。**④放筷子操作。**进餐完毕后，按照先右后左的顺序放下筷子，通过 pthread_mutex_unlock 函数对对应的筷子互斥锁进行解锁操作，释放筷子资源，使得其他哲学家可以获取这些筷子，同样每个放筷子操作都有对应的输出提示信息。

```
// 哲学家的思考、进餐过程
void* philosopher(void* arg) {
    int id = *(int*)arg;
    int left = id;
    int right = (id + 1) % PHILOSOPHERS;
    while (1) {
        // 思考状态
        printf("哲学家 %d 正在思考\n", id);
        usleep(rand() % 401 + 100); // 随机思考时间
        // 拿筷子：首先拿左边的筷子，然后拿右边的筷子
        printf("哲学家 %d 正在拿起筷子 %d\n", id, left);
        pthread_mutex_lock(&chopsticks[left]);  // 拿左边筷子
        printf("哲学家 %d 正在拿起筷子 %d\n", id, right);
        pthread_mutex_lock(&chopsticks[right]); // 拿右边筷子
        // 进餐状态
        printf("哲学家 %d 正在进餐\n", id);
        usleep(rand() % 401 + 100); // 随机进餐时间
        // 放筷子：先放右边的筷子，然后放左边的筷子
        printf("哲学家 %d 正在放下筷子 %d\n", id, right);
        pthread_mutex_unlock(&chopsticks[right]);
        printf("哲学家 %d 正在放下筷子 %d\n", id, left);
        pthread_mutex_unlock(&chopsticks[left]);
    }
    return NULL;
}
```

在 Main 函数部分，初始化随机数种子和互斥锁，然后和先前的题目的逻辑一致，创建哲学家线程，等待线程结束后互斥锁销毁。

（2）非死锁解法

在这个非死锁的解法中，我使用 pthread_mutex_trylock 函数来尝试获取互斥锁。哲学家会先尝试获取左边的筷子，如果获取成功，再尝试获取右边的筷子。如果两根筷子都能获取到（left_taken == 0 && right_taken == 0），那么哲学家就可以进餐。

```
26          if (left_taken == 0 && right_taken == 0) {
27              // 如果两根筷子都拿到了
28              printf("哲学家 %d 正在进餐\n", id);
29              usleep(rand() % 401 + 100); // 随机进餐时间
30
31              // 放筷子：先放右边的筷子，然后放左边的筷子
32              printf("哲学家 %d 正在放下筷子 %d\n", id, right);
33              pthread_mutex_unlock(&chopsticks[right]);
34
35              printf("哲学家 %d 正在放下筷子 %d\n", id, left);
36              pthread_mutex_unlock(&chopsticks[left]);
37          } else {
38              // 如果没有成功拿到两根筷子，放回已拿到的筷子
39              if (left_taken == 0) {
40                  printf("哲学家 %d 正在放下筷子 %d（左边）\n", id, left);
41                  pthread_mutex_unlock(&chopsticks[left]);
42              }
43              if (right_taken == 0) {
44                  printf("哲学家 %d 正在放下筷子 %d（右边）\n", id, right);
45                  pthread_mutex_unlock(&chopsticks[right]);
46              }
47          }
```

3 测试与分析

3.1 系统测试及结果说明

3.1.1 任务 1 双线程循环输出数据

可以通过$ gcc -o threads threads.c -lpthread 命令实现双线程编译运行，由图 3-1 可以发现线程 A 和线程 B 成功循环输出数据，同时其并不是严格交替输出，当执行到结尾时，我们可以看到线程 B 先结束，输出 0001，最后线程 A 单独执行。

```
hyt@hyt915-VMware:~/Desktop/lab2$ gcc -o threads threads.c -lpthread
hyt@hyt915-VMware:~/Desktop/lab2$ ./threads
A:0000
B:1000
B:0999
A:0001
B:0998
B:0002
A:0992
B:0001
A:0993
A:0994
A:0995
A:0996
A:0997
A:0998
A:0999
hyt@hyt915-VMware:~/Desktop/lab2$
```

图 3 - 1 任务 1 结果截图

3.1.2 任务 2 Linux 创建父子进程，实现 wait 同步函数

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，同时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。

```
hyt@hyt915-VMware:~/Desktop/lab2$ gcc parent_child.c -o parent_child
hyt@hyt915-VMware:~/Desktop/lab2$ ./parent_child
父进程PID为59987，正在等待子进程59988结束...
子进程的PID为59988
子进程结束休眠.此时状态码为15
父进程：子进程存在且状态码为15.
hyt@hyt915-VMware:~/Desktop/lab2$ ps -a
    PID TTY          TIME CMD
  59987 pts/3    00:00:00 parent_child
  59988 pts/3    00:00:00 parent_child
  59997 pts/4    00:00:00 ps
```

图 3 - 2 任务 2 结果截图

3.1.3 任务 3 Windows 并发线程画圆画方

运行.py 代码，如图 3-3 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。

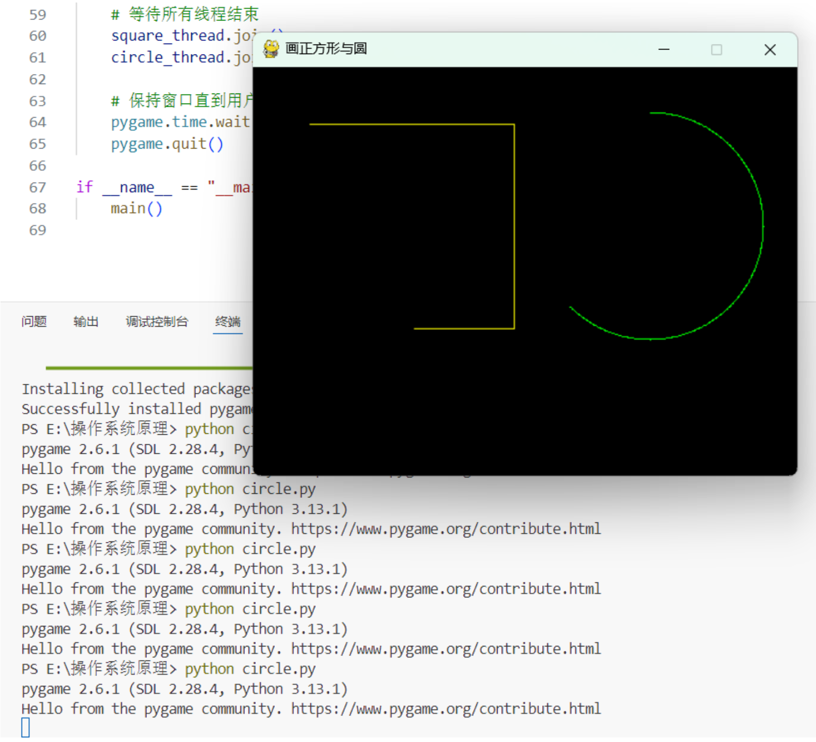

图 3-3 任务 3 结果截图

3.1.4 任务 4 Linux“生产者-消费者”同步控制

编译运行程序脚本，如图 3-4 可以看到成功同步实现多个生产者线程不断生产数据并放入缓冲区，多个消费者线程从缓冲区中取出数据进行消费。需要注意因为 consumer 比 producer 多一人，所以让 consumer 休眠时间长一些，否则就会刚生产出来就被消耗掉的情况。

```
hyt@hyt915-VMware:~/Desktop/lab2$ gcc -o Producer_Consumer Producer_Consumer.c -lpthread
hyt@hyt915-VMware:~/Desktop/lab2$ ./Producer_Consumer
生产者 1: 在位置 0 生产物品 1383
生产者 2: 在位置 1 生产物品 2886
消费者 2: 从位置 0 消费物品 1383
消费者 1: 从位置 1 消费物品 2886
生产者 1: 在位置 2 生产物品 1777
消费者 3: 从位置 2 消费物品 1777
生产者 2: 在位置 3 生产物品 2915
消费者 2: 从位置 3 消费物品 2915
生产者 1: 在位置 4 生产物品 1793
消费者 1: 从位置 4 消费物品 1793
生产者 2: 在位置 5 生产物品 2335
消费者 3: 从位置 5 消费物品 2335
生产者 1: 在位置 6 生产物品 1386
生产者 2: 在位置 7 生产物品 2492
消费者 2: 从位置 6 消费物品 1386
生产者 1: 在位置 8 生产物品 1649
```

图 3-4 任务 4 结果截图

3.1.5 任务 5 Linux 信号机制实现进程通信

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。如图 3-5，当输入 n 子进程不会终止，查看进程发现可以和父子 PID 对应；输入 y 后，子进程终止，可以注意到进程均结束。

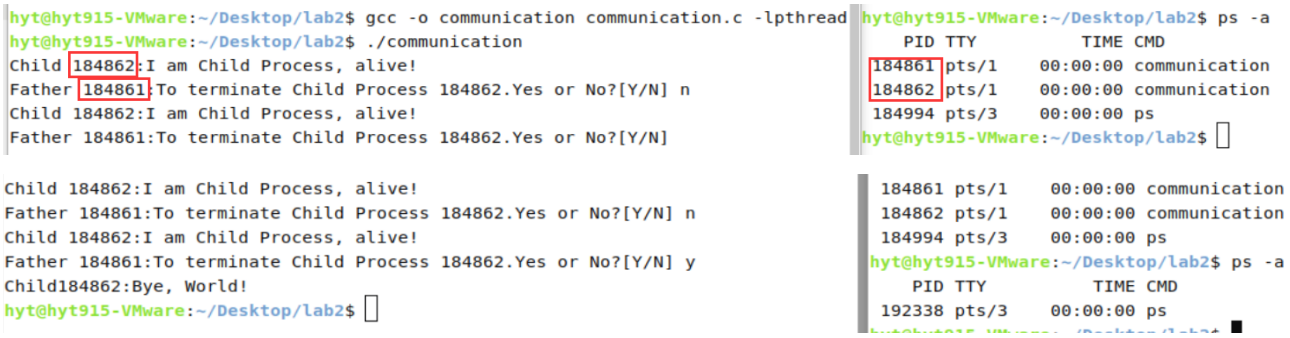

图 3 - 5 任务 5 结果截图

3.1.6 任务 6 Linux 下模拟哲学家就餐

- **死锁解法**

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，同时注意我们拿完筷子休眠一秒更容易发生死锁。

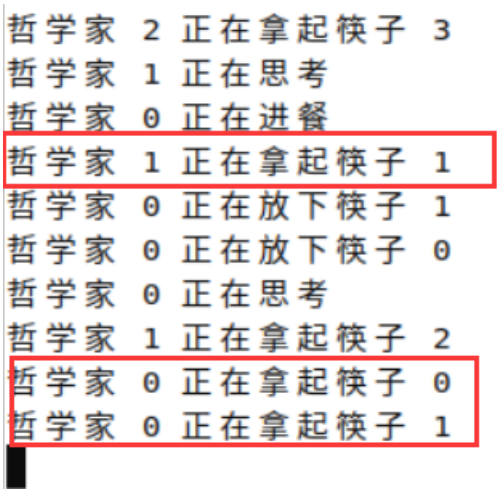

图 3 - 6 任务 6 死锁解法结果截图

- **非死锁解法**

采用每个哲学家要么同时取 2 只筷子，要么不取，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，我们通过 CTRL+C 停止进程。

```
哲学家 3 正在进餐
哲学家 0 正在放下筷子 1
哲学家 0 正在放下筷子 0
哲学家 0 正在思考
哲学家 1 正在放下筷子 2（右边）
哲学家 1 正在思考
哲学家 4 正在思考
哲学家 2 正在放下筷子 2（左边）
哲学家 2 正在思考
哲学家 4 正在思考
哲学家 2 正在思考
^C
hyt@hyt915-VMware:~/Desktop/lab2$
```

图 3 - 7 任务 6 非死锁解法结果截图

综上所述，实验二的前六个任务均可以准确完成测试要求。

3.2 遇到的问题及解决方法

（1）问题一：使用 gcc ab.c -o ab 命令编译编写好的 c 代码报错。

解决方法：原因是使用了 pthread 库，需要加上参数-lphread 才能顺利编译。

（2）问题二：双线程运行速度大差异大，难以捕获初始打印信息。

解决方法：原因是休眠时间太短，延长 sleep()休眠时间，除此之外很多实验部分都需要对休眠时间进行灵活调整，以便查看实验效果。

（3）问题三：Linux 信号机制实现进程通信的程序父进程在访问共享内存出现问题。

解决方法：父进程需要休眠等待子进程执行至少一轮，以便于让子进程将 pid 传入共享内容，之后父进程访问共享内存才可以读取到内容。

（4）问题四：模拟死锁时，由于每位哲学家拿筷子的时间并不固定，出现死锁的时间点时而靠前，时而比较靠后。

解决方法：每次取完筷子后停顿一小段时间，使得死锁更容易发生。

3.3 设计方案存在的不足

可以充分利用 Windows 和 Linux 不同优势来完成实验，而在本实验中我尝试尽量使用 Linux 系统来完成实验要求，利于加强对 Linux 系统的熟练度。

4 实验总结

4.1 实验感想

4.1.1 任务 1 双线程循环输出数据实验感想

执行该任务时，我深切体会到线程在操作系统中的并发执行机制，如同网络中多个连接共享带宽传输数据分组。但线程间的执行顺序和资源访问冲突问题，类似网络设备竞争信道资源，处理不当可能被攻击者利用，引发程序异常，甚至成为系统安全漏洞，如导致死循环，造成拒绝服务攻击隐患。这让我深知编写多线程代码时防范安全风险、遵循安全规范对维护系统稳定性和安全性的重要性。

4.1.2 任务 2 Linux 创建父子进程，实现 wait 同步函数实验感想

此任务让我明晰进程间的层级关系和同步协调的关键意义，类似网络安全中的访问控制和权限管理。父子进程如同网络中不同权限的用户或节点，wait 函数像身份验证与授权机制，可避免子进程成为 “孤儿进程” 等问题。若进程同步失控，可能被黑客篡改子进程数据，影响系统安全，如导致敏感信息泄露。所以，严格遵循进程同步规则和强化通信安全检查，是保障操作系统安全的重要防线。

4.1.3 任务 3 Windows 并发线程画圆画方实验感想

该任务融合多线程并发与图形绘制，展现多线程在可视化应用中的效能，也暴露安全挑战。精准调控线程绘图类似网络安全监控系统中协调传感器线程收集处理数据。若线程管理不善，如执行顺序混乱或资源分配不当，可能导致图形绘制错误，在安全敏感的可视化应用中会误导决策，危及系统安全。因此，掌握线程管理和绘图 API 的安全使用方法，确保线程协调与资源分配可靠，是防范安全风险的关键。

4.1.4 任务 4 Linux“生产者-消费者”同步控制实验感想

在“生产者 - 消费者” 模型中，生产者与消费者线程通过共享缓冲区交互数据，类似网络中的数据传输与处理，其中的同步问题对操作系统安全影响深远。借助信号量、互斥锁等同步工具如同网络通信中的加密、校验和流量控制机制，可防止数据竞争等问题。代码编写调试中，我体会到细微逻辑错误可能引发系统混乱，被攻击者利用，如篡改数据、注入恶意代码。所以，掌握资源共享与同步控制技术并融入安全理念，是构建安全操作系统的核心要素。

4.1.5 任务 5 Linux 信号机制实现进程通信实验感想

在此任务种我见识到其高效性，也意识到潜在安全风险。信号如同网络通信协议中的控制信号和数据包，准确收发和处理信号类似确保网络数据包的正确处理，信号被篡改或伪造可能引发进程异常，如执行非法指令、泄露敏感信息，如同网络攻击中的数据篡改。因此，研究信号机制的安全特性，强化信号验证和防护，是保障进程通信安全的关键。

4.1.6 任务 6 Linux 下模拟哲学家就餐实验感想

模拟哲学家就餐实验，让我在多线程编程的资源管理和同步协调方面树立安全意识。该问题呈现的资源分配与同步不当引发的死锁，类似网络中多个节点竞争共享资源导致的死锁，会使网络通信受阻。编写非死锁解法时设计资源获取策略，如同网络安全中的资源访问控制和流量调度策略，可避免系统故障。此过程让我深知多线程编程中资源管理和同步协调的安全复杂性，稍有疏忽就可能引发严重问题。所以，在多线程应用开发中，进行死锁分析预防并融入安全考量，是确保操作系统安全稳定运行的必要前提。

4.2 意见和建议

无。

原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

已阅读并同意以下内容。

判定为不合格的一些情形：

（1）请人代做或冒名顶替者；

（2）替人做且不听劝告者；

（3）实验报告内容抄袭或雷同者；

（4）实验报告内容与实际实验内容不一致者；

（5）实验代码抄袭者。

作者签名：