

《操作系统原理》实验报告

姓名	侯竣	学号	U202116003	专业班级	密码 2101	时间	2023.12.06
----	----	----	------------	------	---------	----	------------

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

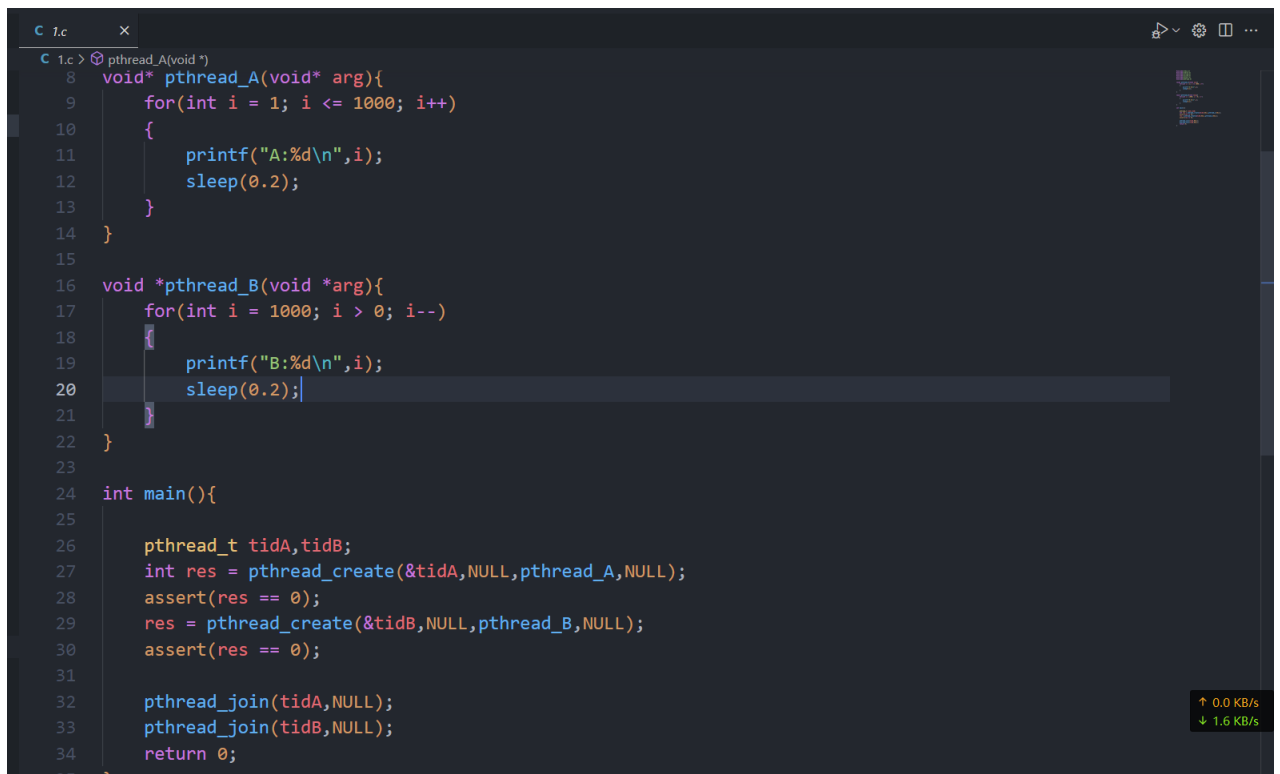
- 1) 在 Linux/Windows 下创建 2 个线程 A 和 B，循环输出数据或字符串。
- 2) 在 Linux 下创建（fork）一个子进程，实验 wait/exit 函数
- 3) 在 Windows/Linux 下，利用线程实现并发画圆画方。
- 4) 在 Windows 或 Linux 下利用线程实现“生产者-消费者”同步控制
- 5) 在 Linux 下利用信号机制(signal)实现进程通信
- 6) 在 Windows 或 Linux 下模拟哲学家就餐，提供死锁和非死锁解法。
- 7) 研读 Linux 内核并用 printk 调试进程创建和调度策略的相关信息。

三、实验环境和核心代码

所有环境均为优麒麟 20.04，内核版本是 5.15，编译工具 gcc11

3.1 创建 2 个线程 A 和 B，循环输出数据或字符串

如图编写如下代码，创建两个线程，分别打印数据，并且用 pthread_join 让 main 线程等待完成



```
1.c > pthread_A(void *)
8 void* pthread_A(void* arg){
9     for(int i = 1; i <= 1000; i++)
10    {
11        printf("A:%d\n",i);
12        sleep(0.2);
13    }
14 }
15
16 void *pthread_B(void *arg){
17     for(int i = 1000; i > 0; i--)
18     {
19         printf("B:%d\n",i);
20         sleep(0.2);
21     }
22 }
23
24 int main(){
25
26     pthread_t tidA,tidB;
27     int res = pthread_create(&tidA,NULL,pthread_A,NULL);
28     assert(res == 0);
29     res = pthread_create(&tidB,NULL,pthread_B,NULL);
30     assert(res == 0);
31
32     pthread_join(tidA,NULL);
33     pthread_join(tidB,NULL);
34     return 0;
35 }
```

图 3-1 实验 1 代码

3.2 在 Liunx 下创建子进程，实验 wait/exit 函数

I. 不用 wait 函数:

代码如图, fork 一个子进程，让子进程休眠 5 秒，父进程则打印 pid 后直接退出

```
C 2-1.c x
C 2-1.c > main()
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  int main() {
6      pid_t pid = fork();
7
8      if (pid < 0) {
9          printf("Fork failed\n");
10         return -1;
11     }
12
13     if (pid == 0) { // 子进程
14         printf("Child process: pid=%d, parent pid=%d\n", getpid(), getppid());
15         sleep(5); // 休眠 5 秒
16     } else { // 父进程
17         printf("Parent process: pid=%d, child pid=%d\n", getpid(), pid);
18     }
19
20     printf("pid=%d, exit\n", getpid());
21     return 0;
22 }
```

图 3-2 实验 2 代码

II. 使用 wait 函数

如图，在上面基础上，让父进程 **wait** 子进程退出，并获取子进程返回的参数

```
C 2-2.c X
C 2-2.c > main()
6
7 #define exitParam 111 // 子进程退出参数
8
9 int main() {
10     pid_t pid = fork();
11
12     if (pid < 0) {
13         printf("Fork failed\n");
14         return -1;
15     }
16
17     if (pid == 0) { // 子进程
18         printf("Child process: pid=%d, parent pid=%d, sleep 5 seconds\n", getpid(), getppid());
19         sleep(5); // 休眠 5 秒
20         exit(exitParam);
21     } else { // 父进程
22         printf("Parent process: pid=%d, child pid=%d, waiting for child to exit\n", getpid(), pid);
23         int status;
24         pid_t childPid = wait(&status);
25         if (WIFEXITED(status)) {
26             printf("Parent process: pid=%d, child pid=%d, child exit status=%d\n", getpid(), childPid,
27                 status);
28         }
29     }
30     return 0;
31 }
```

图 3-3 实验 3 代码

3.3 在 Linux 下利用线程实现“生产者-消费者”同步控制

代码如下图:

先用 buffer 数组作为缓存存储, inOffset 和 outOffset 作为生产者和消费者的读写偏移量(即下一个 item)在 buffer 的下标

信号量分别是 empty 和 full 以及 mutex, 分别表示队列是否空或满和生产者消费者的读写互斥

```
// 信号量
sem_t empty;
sem_t full;
int inOffset = 0; // 缓冲区的写入位置
int outOffset = 0; // 缓冲区的读取位置
int buffer[BufferSize]={0};

pthread_mutex_t mutex; // 读写锁
```

图 3-4 实验 4 信号量

生产者部分的代码如下:

主要是先判断是否 `empty`(P 操作), 则表示已满, 会被阻塞不能生产 `item`, 然后进入 `mutex` 防止多个生产者同时生产, `sleepTime` 是模拟生产者加大临界区的

```
void *producer(void *pno){
    int threadNum = *((int *)pno); // 生产者编号
    int item;
    while(1){
        if (threadNum == 1){
            // 生产者1: 1000-1999
            item = rand()%1000+1000;
        }else{
            // 生产者2: 2000-2999
            item = rand()%1000+2000;
        }

        sem_wait(&empty); // 等待empty信号量

        pthread_mutex_lock(&mutex); // 互斥锁, 防止多个生产者同时生产
        buffer[inOffset] = item;

        // 随机睡眠100ms-1000ms
        int sleepTime = rand()%900+100;

        printf("P_%d: Item=%d\tidx=%d\tsleep=%dms\n", *((int *)pno), buffer[inOffset], inOffset, sleepTime);
        inOffset = (inOffset+1)%BufferSize;

        pthread_mutex_unlock(&mutex); // 释放互斥锁
        sem_post(&full); // 发送full信号量

        usleep(sleepTime*1000); // 睡眠
    }
}
```

图 3-5 实验 4 生产者代码

消费者代码类似, 只是把 `empty` 换成了 `full`:

```

1 // 消费者线程
2 void *consumer(void *cno){
3     int thereadNum = *((int *)cno); // 消费者编号
4     while(1){
5         sem_wait(&full);
6         pthread_mutex_lock(&mutex);
7
8         int item = buffer[outOffset];
9
10        // 随机睡眠100ms-1000ms
11        int sleepTime = rand()%900+100;
12
13        printf("C_%d: Item=%d\tidx=%d\tsleep=%dms\n", thereadNum,item,outOffset,sleepTime);
14        outOffset = (outOffset+1)%BufferSize;
15
16        pthread_mutex_unlock(&mutex);
17        sem_post(&empty);
18
19        usleep(sleepTime*1000); // 睡眠
20    }
21 }
22
23 int main(){
24     pthread_t pro[countOfProducer], con[countOfConsumer]; // 生产者和消费者线程

```

图 3-6 实验 4 消费者代码

3.4 在 Linux 下利用信号机制(signal)实现进程通信

代码如下:

父进程 fork 子进程后进入死循环, 每隔两秒询问用户 Y 和 N, 而子进程先注册处理函数, 然后持续睡眠, 直到收到父进程的 kill 信号为止

```

1 void handle_sigint(int sig) {
2     printf("Bye, World!\n");
3     exit(0);
4 }
5
6 int main() {
7     pid_t pid = fork();
8
9     if (pid < 0) {
10        printf("Fork failed\n");
11        return -1;
12    }
13
14    if (pid == 0) { // 子进程
15        signal(SIGUSR1, handle_sigint); // 注册信号处理函数
16        while (1) {
17            printf("I am Child Process, alive!\n");
18            sleep(2);
19        }
20    } else { // 父进程
21        char answer[2];
22        while (1) {
23            printf("To terminate Child Process, Yes or No?\n");
24            scanf("%1s", answer);
25            if (answer[0] == 'Y' || answer[0] == 'y') {
26                kill(pid, SIGUSR1);
27                break;
28            } else {
29                sleep(2);
30            }
31        }
32    }
33
34    return 0;
35 }

```

图 3.7 实验 5 代码

3.5 在 Linux 下模拟哲学家就餐，提供死锁和非死锁解法

I. 死锁解法

如图:

筷子是临界资源，所以哲学家就餐前会请求筷子，这里让所有哲学家拿起第一个筷子就睡眠 1s，增大临界区时间，更容易进入死锁

```

6  pthread_mutex_t chopstick[5];
7  void * philosopher(void *);
8  void eat(int);
9
10 int main()
11 {
12     int i,a[5];
13     pthread_t tid[5];
14
15     for(i=0;i<5;i++)
16         pthread_mutex_init(&chopstick[i], NULL);
17
18     for(i=0;i<5;i++){
19         a[i]=i;
20         pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
21     }
22     for(i=0;i<5;i++)
23         pthread_join(tid[i],NULL);
24     for(i=0;i<5;i++)
25         pthread_mutex_destroy(&chopstick[i]);
26 }
27
28 void * philosopher(void * num)
29 {
30     int phil=*(int *)num;
31     pthread_mutex_lock(&chopstick[phil]);
32     printf("Philosopher %d picked up chopstick %d, waiting for chopstick %d\n",phil,phil,(phil+1)%5);
33
34     sleep(1); // sleep增加死锁几率
35     pthread_mutex_lock(&chopstick[(phil+1)%5]);
36     eat(phil);
37     sleep(2);
38     printf("Philosopher %d finished eating, unlocking chopstick %d and %d\n",phil,phil,(phil+1)%5);
39     pthread_mutex_unlock(&chopstick[(phil+1)%5]);
40     pthread_mutex_unlock(&chopstick[phil]);

```

图 3-8 实验 5 死锁解法

II. 非死锁解法

如图:

非死锁解法的关键是限制同时就餐数量，即最多只能让 4 个哲学家同时就餐，这样就无法形成死锁链，这里是加了一个 `para` 信号量，用来限制并发个数(初始值为 4，最多只能 4 人同时就餐)，从而避免了死锁


```
main0
int i,a[5];
pthread_t tid[5];

sem_init(&para,0,4);

for(i=0;i<5;i++)
    sem_init(&chopstick[i],0,1);

for(i=0;i<5;i++){
    a[i]=i;
    pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
}
for(i=0;i<5;i++)
    pthread_join(tid[i],NULL);

printf("All philosophers finished eating, exit\n");
}

void * philosopher(void * num)
{
    int phil=(int *)num;

    sem_wait(&para);
    sem_wait(&chopstick[phil]);
    printf("Philosopher %d picked up chopstick %d, waiting for chopstick %d\n",phil,phil,(phil+1)%5);
    sleep(1);
    sem_wait(&chopstick[(phil+1)%5]);
    sleep(2);

    eat(phil);
    sleep(1);
    printf("Philosopher %d finished eating, unlocking chopstick %d and %d\n",phil,phil,(phil+1)%5);

    sem_post(&chopstick[(phil+1)%5]);
    sem_post(&chopstick[phil]);
    sem_post(&para);
}
```

图 3-9 实验 5 非死锁解法

四、实验结果

4.1 创建 2 个线程 A 和 B，循环输出数据或字符串

运行两次，结果不同，如图：

```
A:996
B:8
A:997
B:7
A:998
B:6
A:999
B:5
A:1000
B:4
B:3
B:2
B:1
→ lab2
```

图 4-1 实验 1 结果 1

```
A:1000
B:12
B:11
B:10
B:9
B:8
B:7
B:6
B:5
B:4
B:3
B:2
B:1
➡ lab2
```

图 1-2 实验 1 结果 2

说明两个进程是并发执行的

4.2 在 Liunx 下创建子进程，实验 wait/exit 函数

I. 不用 wait 的情况

如图, ps 可见程序获取的子进程 pid=3890, 右边 ps 获得的也是 3890, 正确

```
B:7
B:6
B:5
B:4
B:3
B:2
B:1
• → lab2 ./2-1
Parent process: pid=3465, child pid=
3466
Child process: pid=3466, parent pid=
3465
pid=3465, exit
• → lab2 ./2-1
Parent process: pid=3498, child pid=
3499
pid=3498, exit
Child process: pid=3499, parent pid=
3498
○ → lab2 pid=3466, exit
:spid=3499, exit
○ → lab2
• → lab2 ./2-1
Parent process: pid=3752, child pid=
3753
pid=3752, exit
Child process: pid=3753, parent pid=
3752
• → lab2 pid=3753, exit
./2-1
Parent process: pid=3889, child pid=
3890
pid=3889, exit
Child process: pid=3890, parent pid=
3889
• → lab2
```

PID	TTY	TIME	CMD
1	hvc0	00:00:00	init(Ubuntu
4	hvc0	00:00:00	init
426	pts/1	00:00:00	sh
431	pts/1	00:00:00	sh
435	pts/1	00:00:05	node
465	pts/1	00:00:00	node
504	pts/1	00:00:21	node
554	pts/1	00:00:01	node
565	pts/1	00:00:02	cpptools
3040	pts/1	00:00:00	cpptools-sr
3858	pts/4	00:00:00	watch
3890	pts/6	00:00:00	2-1
3937	pts/4	00:00:00	watch
3938	pts/4	00:00:00	sh
3939	pts/4	00:00:00	ps

图 4-3 实验 2 结果 1

II. 用 wait 的情况

如图, 父进程获取到了子进程的参数 111, 实验成功

```
• → lab2 ./2-2
Parent process: pid=4859, child pid=
4860, waiting for child to exit
Child process: pid=4860, parent pid=
4859, sleep 5 seconds
Parent process: pid=4859, child pid=
4860, child exit status=111
○ → lab2
```

图 4-4 实验 2 结果 2

4.3 在 Linux 下利用线程实现“生产者-消费者”同步控制

如图:

P 代表生产者, C 代表消费者, 这里 P 的数量是 2, C 的数量是 3

对比 P 和 C 取出来的 Item 可见确实成功消费, 并且 P 的数量比 C 少, 能很明显的看

出程序的阻塞执行情况

```
lab2 ./4
P_1: Item=1383 idx=0 sleep=386ms
P_2: Item=2777 idx=1 sleep=215ms
C_1: Item=1383 idx=0 sleep=393ms
C_2: Item=2777 idx=1 sleep=935ms
P_2: Item=2386 idx=2 sleep=292ms
C_3: Item=2386 idx=2 sleep=349ms
P_1: Item=1421 idx=3 sleep=762ms
C_1: Item=1421 idx=3 sleep=527ms
P_2: Item=2690 idx=4 sleep=359ms
C_3: Item=2690 idx=4 sleep=663ms
P_2: Item=2926 idx=5 sleep=340ms
C_1: Item=2926 idx=5 sleep=226ms
P_1: Item=1172 idx=6 sleep=236ms
C_2: Item=1172 idx=6 sleep=711ms
P_2: Item=2368 idx=7 sleep=367ms
C_1: Item=2368 idx=7 sleep=529ms
P_1: Item=1782 idx=8 sleep=630ms
C_3: Item=1782 idx=8 sleep=162ms
P_2: Item=2123 idx=9 sleep=767ms
C_3: Item=2123 idx=9 sleep=335ms
P_1: Item=1929 idx=0 sleep=802ms
C_1: Item=1929 idx=0 sleep=622ms
P_2: Item=2058 idx=1 sleep=969ms
C_2: Item=2058 idx=1 sleep=967ms
```

图 4-5 实验 3 结果

4.4 在 Linux 下利用信号机制(signal)实现进程通信

如图:

父进程输入 N 后延迟 2s 再次提问, 而子进程一直循环打印输出 alive

当父进程输入 Y 后子进程终结

```
lab2 ./5
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!
Y
Bye, World!
```

图 4-6 实验 4 结果

4.5 在 Linux 下模拟哲学家就餐, 提供死锁和非死锁解法

1.死锁解法

如图:

由于用 `sleep` 增大了临界区，直接就进入死锁：

```
lab2 ➔ lab2 ./6-dead
Philosopher 0 picked up chopstick 0, waiting for chopstick 1
Philosopher 2 picked up chopstick 2, waiting for chopstick 3
Philosopher 4 picked up chopstick 4, waiting for chopstick 0
Philosopher 3 picked up chopstick 3, waiting for chopstick 4
Philosopher 1 picked up chopstick 1, waiting for chopstick 2
```

图 4-7 实验 5 结果 1

II. 非死锁解法

如图：

通过限制同时就餐的哲学家数量，进而避免的死锁成功运行完毕

```
lab2 ➔ lab2 ./6
Philosopher 0 picked up chopstick 0, waiting for chopstick 1
Philosopher 2 picked up chopstick 2, waiting for chopstick 3
Philosopher 1 picked up chopstick 1, waiting for chopstick 2
Philosopher 3 picked up chopstick 3, waiting for chopstick 4
Philosopher 3 is eating
Philosopher 3 finished eating, unlocking chopstick 3 and 4
Philosopher 4 picked up chopstick 4, waiting for chopstick 0
Philosopher 2 is eating
Philosopher 2 finished eating, unlocking chopstick 2 and 3
Philosopher 1 is eating
Philosopher 1 finished eating, unlocking chopstick 1 and 2
Philosopher 0 is eating
Philosopher 0 finished eating, unlocking chopstick 0 and 1
Philosopher 4 is eating
Philosopher 4 finished eating, unlocking chopstick 4 and 0
All philosophers finished eating, exit
lab2 ➔ lab2
```

图 4-8 实验 5 结果 2

五、实验错误排查和解决方法

3.2 在 Linux 下创建子进程，实验 `wait/exit` 函数

这个实验并不是很难，代码比较少，主要是查阅相关的 API 实现

3.3 在 Linux 下利用线程实现“生产者-消费者”同步控制

实现生产者-消费者模型时，我一开始信号量同步操作没写对，导致了竞态条件。

这种 bug 比较难发现, 而且很难复现, 我通过打印详细的每一步的日志, 再通过一步步分析解决了这个问题, 终于确保了当缓冲区为空时, 消费者会等待, 而当缓冲区满时, 生产者会等待的实验要求。

3.4 在 Linux 下利用信号机制(signal)实现进程通信

一开始不知道怎么注册信号, 这是难点. 在查阅相关资料后, 实际上比较简单, 通过调用 `signal` 函数注册信号值和处理函数即可

3.5 在 Linux 下模拟哲学家就餐, 提供死锁和非死锁解法

一开始做死锁解法的时候, 实际上也很难进入死锁, 因为这要求 5 个哲学家都恰好在拿起一根筷子的时候停止, 等待拿另外一根筷子, 导致很难复现死锁情况, 于是想到可以通过 `sleep` 扩大这个临界区, 让所有的哲学家在拿起一根筷子的时候 `sleep 1s` 后, 就基本上一定能进入死锁情况了

六、实验参考资料和网址

(1) 教学课件

(2) <https://www.runoob.com/linux/linux-tutorial.html> 菜鸟教程-linux 教程

(3) <https://www.runoob.com/w3cnote/shell-quick-start.html> 菜鸟教程-Shell 编程快速入门

(4) https://blog.csdn.net/qq_28602957/article/details/53538329 【操作系统】“哲学家进餐”问题