电子科技大学信息与软件工程学院

实 验 报 告

学　　号 2018091618008

姓　　名 袁昊男

（实验）课程名称 操作系统基础

理论教师 任立勇

实验教师 任立勇

电子科技大学教务处制表

电 子 科 技 大 学

实　　验　　报　　告

学生姓名：袁昊男　　学号：2018091618008　　指导教师：任立勇

实验地点：在线实验　　实验时间：2020.05.25

一、实验室名称：信息与软件工程学院实验中心

二、实验名称：信号量经典问题的实现

三、实验学时：4 学时

四、实验原理：

1、 哲学家就餐问题

由 Dijkstra 提出并解决的哲学家进餐问题（The Dinning Philosophers Problem）是典型的同步问题。该问题是描述有五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五只筷子，他们的生活方式是交替地进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐完毕，放下筷子继续思考。

当哲学家饥饿时，总是先去拿他左边的筷子，即执行 wait(chopsticks[i]); 成功后，再去拿他右边的筷子，即执行 wait(chopsticks[(i+1)%5]); 又成功后便可进餐。进餐毕，又放下他左边的筷子，然后再放他右边的筷子。虽然，上述解法可保证不会有两个相邻的哲学家同时进餐，但却有可能引起死锁。假如五位哲学家同时饥饿而各自拿起左边的筷子时，就会使五个信号量 chopsticks 为 0；当他们再试图去拿右边的筷子时，都将因无筷子可拿而无限期地等待。对于这样的死锁问题，可采取以下几种解决方法：

(1) 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。

(2) 仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。

(3) 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子；而偶数号哲学家则相反。按此规定，将是 1、2 号哲学家竞争 1 号筷子；3、4 号哲学家竞争 3 号筷子。即五位哲学家都先竞争奇数号筷子，获得后，

再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

2、 生产者/消费者问题

n 个缓冲区的缓冲池作为一个共享资源，当生产者进程从数据源—文件中读取数据后将会申请一个缓冲区，并将此数据放入缓冲区中。消费者从一个缓冲区中取走数据，并将其中的内容打印输出。当生产者进程正在访问缓冲区时，消费者进程不能同时访问缓冲区，因此缓冲区是个互斥资源。

可利用记录型信号量解决生产者/消费者问题，还可用信号量集机制来解决。

五、实验目的：

1、 哲学家就餐问题：实现哲学家就餐问题，要求不能出现死锁。通过本实验熟悉 Linux 系统的基本环境，了解 Linux 下进程和线程的实现。

2、 生产者/消费者问题

(1) 掌握进程、线程的概念，熟悉相关的控制语。
(2) 掌握进程、线程间的同步原理和方法。
(3) 掌握进程、线程间的互斥原理和方法。
(4) 掌握使用信号量原语解决进程、线程间互斥和同步方法。

六、实验内容：

1、 哲学家就餐问题：在 Unix 系统下实现教材 2.4.2 节中所描述的哲学家就餐问题。要求显示出每个哲学家的工作状态，如吃饭，思考。连续运行 30 次以上都未出现死锁现象。

2、 生产者/消费者问题

(1) 有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有 n 个缓冲区的缓冲池：生产者进程从文件中读取一个数据，并将它存放到一个缓冲区中；消费者进程从一个缓冲区中取走数据，并输出此数据。生产者和消费者之间必须保持同步原则：不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。
(2) 创建 3 个进程（或者线程）作为生产者，4 个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为数据。

(3) 生产者和消费者进程（或者线程）都具有相同的优先级。

3、 熟悉 Ubuntu 系统下的多线程编程

(1) 使用“Ctrl+Alt+T”打开终端。

(2) 使用 gedit 或 vim 命令打开文本编辑器进行编码：gedit 文件名.c。

(3) 编译程序：“gcc 文件名.c -o 可执行程序名”（如果只输入 gcc 文件名.c，默认可执行程序名为 a.out）使用线程库时，gcc 编译需要添加-l pthread。

(4) 执行程序：./可执行程序名。

七、实验器材（设备、元器件）：

1、 学生每人一台 PC，安装 WindowsXP/2000 操作系统。

2、 局域网络环境。

3、 个人 PC 安装 VMware 虚拟机和 Ubuntu 系统。

八、实验步骤：

1、 哲学家就餐问题

(1) 实现教材 2.4.2 节中所描述的哲学家就餐问题，要求显示出每个哲学家的工作状态，如吃饭，思考。

(2) 在 Ubuntu 系统下的进行编译与运行，要求连续运行 30 次以上都未出现死锁现象。

2、 生产者/消费者问题

(1) 分配具有 n 个缓冲区的缓冲池，作为共享资源。

(2) 定义两个资源型信号量 empty 和 full，empty 信号量表示当前空的缓冲区数量，full 表示当前满的缓冲区数量。

(3) 定义互斥信号量 mutex，当某个进程访问缓冲区之前先获取此信号量，在对缓冲区的操作完成后再释放此互斥信号量。以此实现多个进程对共享资源的互斥访问。

(4) 创建 3 进程（或者线程）作为生产者，4 个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为内容。

(5) 编写代码实现生产者进程的工作内容，即从文件中读取数据，然后申请一个 empty 信号量，和互斥信号量，然后进入临界区操作将读取的数据放入此缓冲区中。并释放 empty 信号量和互斥信号量。

九、实验数据及结果分析

1、 哲学家就餐问题

(1) 代码

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define PHI 5       //哲学家人数

int count = 1;

/*全局变量*/
int chopsticks[PHI];
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  //初始化条件变量
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //初始化互斥锁

void eat(long p_num)
{
    int s_time = rand() % 5 + 1;
    pthread_mutex_lock(&mutex);         //加锁

    if(p_num == 4)
        while(chopsticks[p_num] == 1 || chopsticks[0] == 1)
            pthread_cond_wait(&cond, &mutex);   //等待
    else
        while(chopsticks[p_num] == 1 || chopsticks[p_num+1] == 1)

            pthread_cond_wait(&cond, &mutex);   //等待

    /*拿到筷子后,将互斥量设置为占用*/
    chopsticks[p_num] = 1;
    if(p_num == 4)
        chopsticks[0] = 1;
    else
        chopsticks[p_num+1] = 1;

    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);

    printf("[第%2d 轮] %1d 号哲学家：吃饭%d 秒。
\n", count++, p_num, s_time);
    sleep(s_time);      //吃饭时间
}

void think(long p_num)
{
    int s_time = rand() % 5 + 1;
    pthread_mutex_lock(&mutex);         //加锁

    /*开始思考后，将互斥量释放*/
    chopsticks[p_num] = 0;
    if(p_num == 4)
        chopsticks[0] = 0;
```

```
51.     else
52.         chopsticks[p_num+1] = 0;
53.
54.     pthread_cond_signal(&cond);
55.     pthread_mutex_unlock(&mutex);
56.
57.     printf("[第%2d 轮] %ld 号哲学家：思考%d 秒。
    \n", count++, p_num, s_time);
58.     sleep(s_time);      //思考时间
59. }
60.
61. void* thread_exec(void* i)
62. {
63.     long p_num = (long)i;
64.     while(1)
65.     {
66.         eat(p_num);
67.         think(p_num);
68.     }
69. }
70.
71. int main()
72. {
73.     pthread_t philosophers[PHI];  //  线程 ID
74.     long i;     //哲学家序号
75.
76.     for(i=0; i<PHI; i++)    //筷子状态初始化
77.         chopsticks[i] = 0;
78.
79.     for(i=0; i<PHI; i++)    //创建线程
80.         pthread_create(&philosophers[i], NULL, &thread_exec, (voi
    d *)i);
81.
82.     for(i=0; i<PHI; i++)    //等待线程结束
83.         pthread_join(philosophers[i], NULL);
84.
85.     return 0;
86. }
```

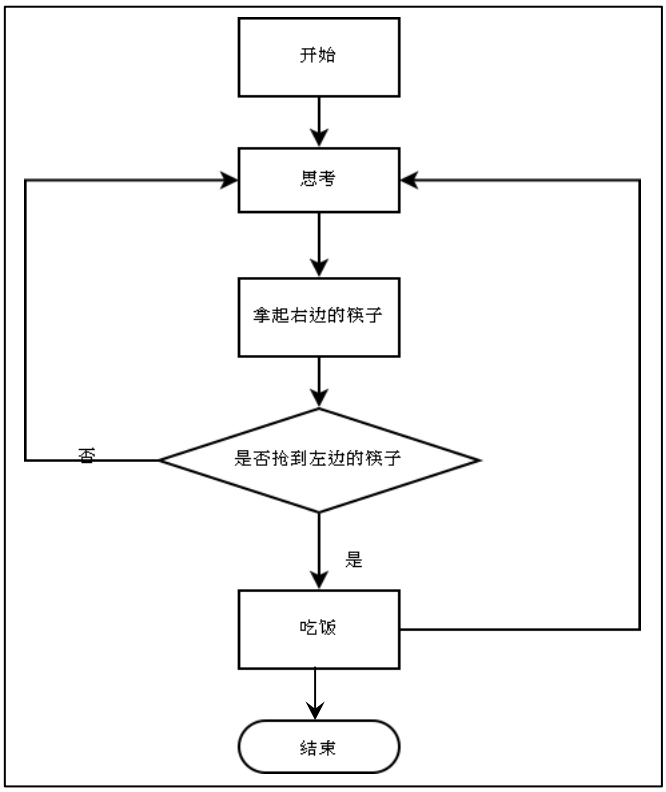
说明：主要函数是哲学家进行的动作 eat()和 think()。

- 在 eat()函数中，首先将互斥量 mutex 上锁，规定当且仅当哲学家的左、右筷子均可用时，才允许哲学家拿起筷子进餐。当左、右筷子中任意一个不可用时，调用 pthread_cond_wait()函数等待；当哲学家同时获得筷子后，将 chopsticks 数组中对应序号的筷子设置为 1（占用）。完成动作后将互斥量 mutex 解锁，并让该序号哲学家 sleep 一小段随机时间。

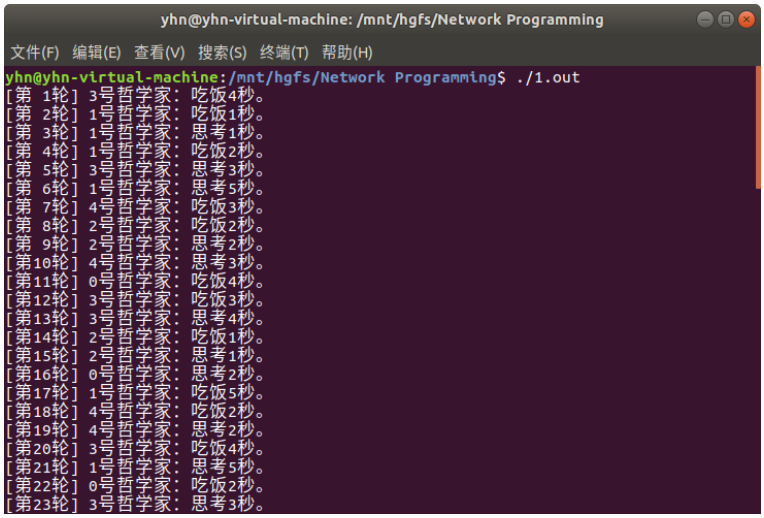
- 在 think()函数中，首先将互斥量 mutex 上锁。开始思考后，将哲学家占用的筷子互斥量释放，即：将 chopsticks 数组中对应序号的筷子设置为 0（空闲）。完成动作后将互斥量 mutex 解锁，并让该序号哲学家 sleep 一小段随机时间。

- 将 eat()函数与 think()函数封装为 thread_exec()例程函数，无限循环按序执行 eat()函数、think()函数。
- 在 main()函数中，首先初始化哲学家线程数组。第一个 for 循环将筷子的状态初始化；第二个 for 循环创建 5 个哲学家线程，其中调用到封装好的例程函数，参数为哲学家序号；第三个 for 循环调用 pthread_join()函数，等待 5 个哲学家线程结束。

(2) 程序流程图



(3) 运行截图



2、 生产者/消费者问题

(1) 代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define PROD 3      //生产者数目
#define CONSU 4     //消费者数目
#define BUFF 10     //缓冲区数目

/*全局变量*/
int in = 0;          //放产品位置
int out = 0;         //取产品位置
int buff[BUFF] = {0};    //缓冲区初始化为0(无产品)
int producer_id = 0;     //生产者 id
int consumer_id = 0;     //消费者 id
int count = 1;
sem_t empty_sem;         //同步信号量，满时阻止生产者放产品
sem_t full_sem;          //同步信号量，空时阻止消费者取产品
pthread_mutex_t mutex;   //互斥信号量

int get_data()
{
    int buf[10];
    int fd = open("data.txt",O_RDONLY);
    if(fd<0)
    {
        printf("打开文件失败!\n");
        exit(1);
    }
    else
    {
        int size = read(fd, buf, 1);
        if(size<0)
        {
            printf("文件读取失败!\n");
            exit(1);
        }
        else
            return (buf[0]+rand()) % 10;
    }
}

void print_buff()
{
    int i;
    for(i = 0; i < BUFF; i++)
        printf("%d ", buff[i]);
    printf("\n");
}

```

```
55. void* producer(void* id)
56. {
57.     long p_id = (long)id;
58.     while(1)
59.     {
60.         sleep(1);
61.         int data = get_data();
62.         sem_wait(&empty_sem);  //信号量-1，成功执行则说明可以放产品
63.         pthread_mutex_lock(&mutex);     //互斥锁
64.
65.         in = in % BUFF;
66.         buff[in++] = data;
67.         printf("[第%2d 轮] %ld 号生产者向缓冲区放产品[%d]      缓冲
    区: ", count++, p_id, data);
68.         print_buff();
69.
70.         pthread_mutex_unlock(&mutex);
71.         sem_post(&full_sem);            //信号量+1
72.     }
73. }
74.
75. void* consumer(void* id)
76. {
77.     long c_id = (long)id;
78.     while(1)
79.     {
80.         sleep(3);
81.         sem_wait(&full_sem); //信号量-1，成功执行则说明可以取产品
82.         pthread_mutex_lock(&mutex);     //互斥锁
83.
84.         out = out % BUFF;
85.         int data = buff[out];
86.         buff[out++] = 0;
87.         printf("[第%2d 轮] %ld 号消费者从缓冲区取产品[%d]      缓冲
    区: ", count++, c_id, data);
88.         print_buff();
89.
90.         pthread_mutex_unlock(&mutex);
91.         sem_post(&empty_sem);            //信号量+1
92.     }
93. }
94.
95. int main()
96. {
97.     pthread_t prod_id[PROD];        //生产者线程 ID
98.     pthread_t consu_id[CONSU];      //消费者线程 ID
99.     long i;
100.     int prod_ret[PROD];             //生产者线程创建结果
101.     int consu_ret[CONSU];           //消费者线程创建结果
102.
103.     int empty_init = sem_init(&empty_sem, 0, BUFF); //初始值为
    BUFF
104.     int full_init = sem_init(&full_sem, 0, 0);   //初始值为 0
105.     if(empty_init && full_init != 0)             //初始化失败
106.     {
107.         printf("初始化失败!\n");
108.         exit(1);
109.     }
```

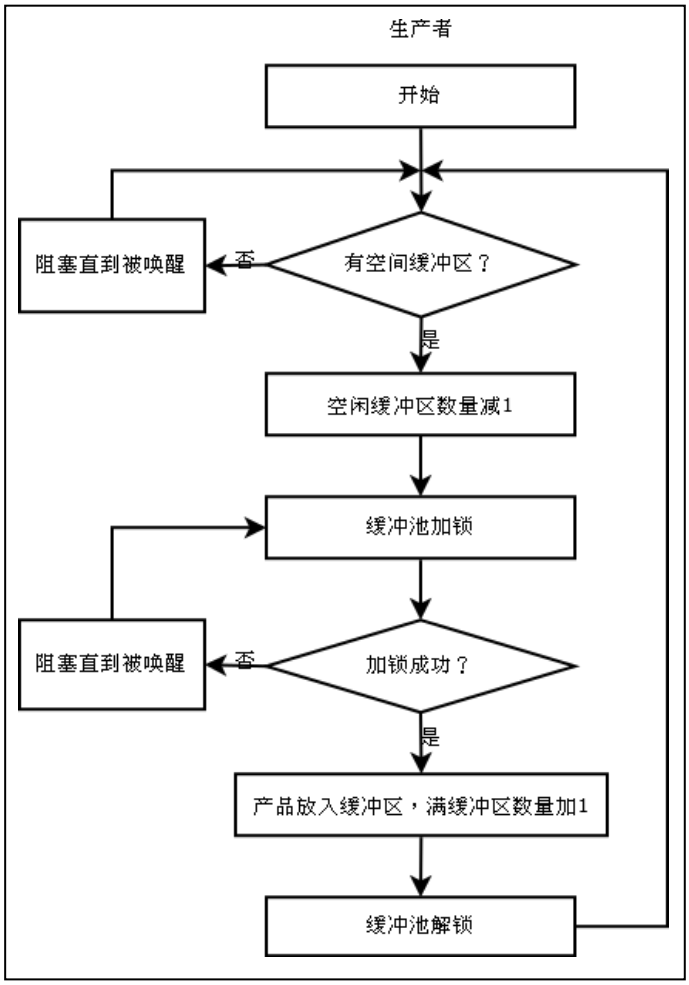
```
110.     for(i=0; i<PROD; i++)            //创建生产者线程
111.     {
112.         prod_ret[i] = pthread_create(&prod_id[i], NULL, &produce
    r, (void*)i);
113.         if(prod_ret[i] != 0)
114.         {
115.             printf("%ld 号生产者进程创建失败。\n", i);
116.             exit(1);
117.         }
118.     }
119.
120.     for(i=0; i<CONSU; i++)            //创建消费者线程
121.     {
122.         consu_ret[i] = pthread_create(&consu_id[i], NULL, &consu
    mer, (void*)i);
123.         if(consu_ret[i] != 0)
124.         {
125.             printf("%ld 号消费者进程创建失败。\n", i);
126.             exit(1);
127.         }
128.     }
129.
130.     for(i=0; i<PROD; i++)
131.         pthread_join(prod_id[i], NULL);
132.
133.     for(i=0; i<CONSU; i++)
134.         pthread_join(consu_id[i], NULL);
135.
136.     exit(0);
137. }
```

说明：主要函数是生产者函数 producer()及消费者函数 consumer()。

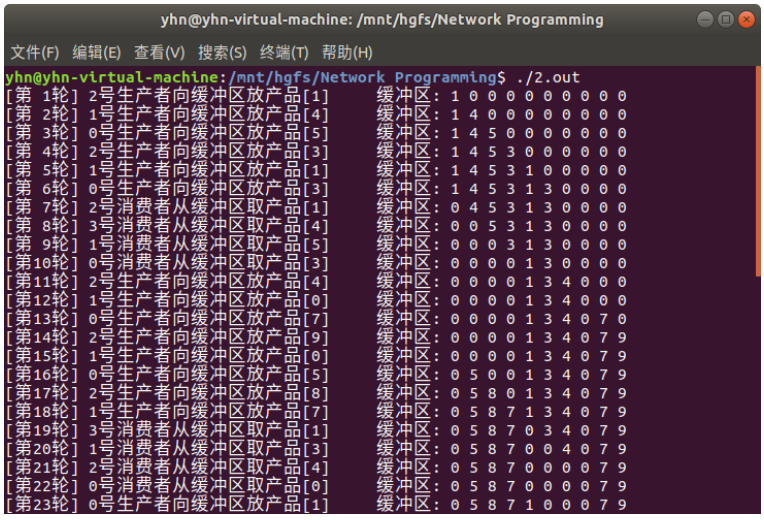
- 在 producer()函数中，首先调用 get_data()函数从文件中读取 1 字节准备好的数据作为生产出的产品。然后尝试使用 sem_wait()函数操作 empty_sem 信号量：若成功，则说明当前缓冲区还未满，可以向其中放产品，并对互斥信号量 mutex 加锁；否则当前线程阻塞，等待缓冲区空闲。全局变量 in 作为缓冲区指针，指示放产品的位置。将产品放入缓冲区后对互斥信号量 mutex 解锁，并调用 sem_post()函数对 full_sem 信号量执行+1 操作。
- 在 consumer()函数中，首先尝试使用 sem_wait()函数操作 full_sem 信号量：若成功，则说明当前缓冲区非空，可以从中取产品，并对互斥信号量 mutex 加锁；否则当前线程阻塞，等待产品放入缓冲区。全局变量 out 作为缓冲区指针，指示取产品的位置。将产品从缓冲区中取出后对互斥信号量 mutex 解锁，并调用 sem_post()函数对 empty_sem 信号量执行+1 操作。
- 在 main()函数中，首先初始化生产者线程与消费者线程、初始化 empty_sem 信号量与 full_sem 信号量。第一个 for 循环创建 3 个生

产者线程，以 producer()函数作为例程函数，参数 i 为生产者序号；第二个 for 循环创建 4 个消费者线程，以 consumer()函数作为例程函数，参数 i 为消费者序号；第三个 for 循环调用 pthread_join()函数，等待所有生产者与消费者线程结束。

(2) 程序流程图



(3) 运行截图



十、实验结论

1、 **哲学家就餐问题：**通过规定当且仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐，可有效解决哲学家进餐引起的死锁问题。程序在连续运行 30 轮后没有产生死锁。

2、 **生产者/消费者问题：**利用记录型信号量 full_sem、empty_sem 控制共享缓冲区的读与写。仅当 empty_sem 非负时，生产者可向缓冲区中放产品；仅当 full_sem 非负时，消费者可从缓冲区中取产品。程序在运行过程中没有出现由于同步问题造成的数据错误。

十一、总结及心得体会

1、 **哲学家就餐问题**

哲学家就餐问题是典型的访问临界资源导致的死锁问题，其解决方法有多种，如按哲学家序号奇偶不同而拿筷子顺序不同、当且仅当左右筷子均空闲时才进餐、使用 AND 信号量机制同时分配左右筷子等。本实验中采用第二种方法，通过互斥信号量 mutex 对 eat()之前取左侧和右侧筷子的操作进行保护，可以防止死锁的出现。在连续运行 30 次及以上后，没有产生死锁。

通过哲学家就餐问题的实践，我对操作系统中临界资源的访问、死锁问题有了更深刻的理解与认识。

2、 **生产者/消费者问题**

生产者/消费者问题是典型的进程互斥与同步问题，其解决方法有多种，如使用记录型信号量、AND 信号量、管程等。在多生产者、多消费者情况下，通过互斥信号量 mutex 的上锁与解锁，将共享缓冲区保护起来，防止其他生产者线程或消费者线程对缓冲区进行操作，从而实现任意时刻只有一个生产者线程或消费者线程操作缓冲区，避免了同步问题带来的数据错误。

通过生产者/消费者问题的实践，我对操作系统中同步问题有了更深刻的理解与认识。

十二、对本实验过程及方法、手段的改进建议

本实验设计与教材结合紧密、难度适中，通过对操作系统中两个信号量经典问题的实现，强化了学生对有关同步问题、死锁问题、临界区资源访问等知识点的理解与掌握。此外，还使学生熟悉 Linux 系统的基本环境，了解 Linux 下进程

和线程的实现，对操作系统上的深入学习打下了坚实的基础。

报告评分：

指导教师签字：