

实验三：同步问题

袁少随 16281054 安全 1601

Task 1:

实验要求:

通过 fork 的方式，产生 4 个进程 P1, P2, P3, P4，每个进程打印输出自己的名字，例如 P1 输出 “I am the process P1”。要求 P1 最先执行，P2、P3 互斥执行，P4 最后执行。通过多次测试验证实现是否正确。

程序代码:

```
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <fcntl.h>
#include <semaphore.h>

sem_t *mySem = NULL;    //进程 2 和进程 3 互斥信号量
sem_t *mySem2 = NULL;   //进程 2 结束信号量
sem_t *mySem3 = NULL;   //进程 3 结束信号量
void *createP2(void *arg) {
    pid_t p2;
    sem_wait(mySem);
    while((p2=fork())!=-1);
    if(p2>0)
        printf("I am the process P2!\n");
    sem_post(mySem);
    sem_post(mySem2);
    return NULL;
}
void *createP3(void *arg) {
    pid_t p3;
    sem_wait(mySem);
    while((p3=fork())!=-1);
    if(p3>0)
        printf("I am the process P3!\n");
    sem_post(mySem);
    sem_post(mySem3);
    return NULL;
}
void *createP4(void *arg) {
    pid_t p4;
```

```

    sem_wait(mySem2);
    sem_wait(mySem3);
    while((p4=fork())!=-1);
    if(p4>0)
        printf("I am the process P4!\n");
    sem_post(mySem2);
    sem_post(mySem3);
    return NULL;
}
int main(int argc, char* argv[])
{
    pid_t p1;
    pthread_t pp2, pp3, pp4;
    while((p1=fork())!=-1);
    if(p1>0){
        printf("I am the process P1!\n");
        mySem = sem_open("mySem", O_CREAT, 0666, 1);
        mySem2 = sem_open("mySem2", O_CREAT, 0666, 0);
        mySem3 = sem_open("mySem3", O_CREAT, 0666, 0);
        pthread_create(&pp2, NULL, createP2, NULL);
        pthread_create(&pp3, NULL, createP3, NULL);
        pthread_create(&pp4, NULL, createP4, NULL);
        pthread_join(pp2, NULL);
        pthread_join(pp3, NULL);
        pthread_join(pp4, NULL);
        sem_close(mySem);    //线程结束
        sem_close(mySem2);
        sem_close(mySem3);
        sem_unlink("mySem");
        sem_unlink("mySem2");
        sem_unlink("mySem3");
    }
    return 0;
}

```

代码解析：

主函数产生进程 P1, 在进程 P1 中通过三个线程 PP2、PP3、PP4 分别执行子函数 createP2、createP3、createP4 产生 3 个进程 P2、P3、P4。

设置三个信号量： mySem（进程 2 和进程 3 互斥信号量）、mySem2（进程 2 结束信号量）、mySem3（进程 3 结束信号量）用来实现 P2、P3 互斥执行，P4 最后执行。

实验结果：

```

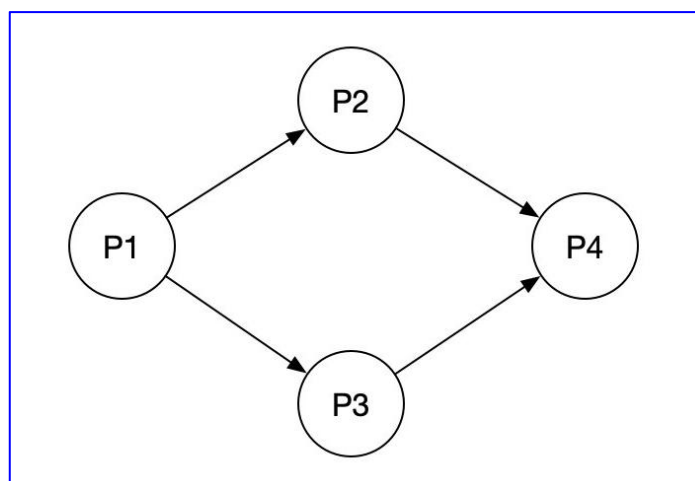
yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o task1 task1.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./task1
I am the process P1!
I am the process P3!
I am the process P2!
I am the process P4!
yuan@ubuntu:~/Desktop/OS/lab3$ ./task1
I am the process P1!
I am the process P3!
I am the process P2!
I am the process P4!
yuan@ubuntu:~/Desktop/OS/lab3$ ./task1
I am the process P1!
I am the process P3!
I am the process P2!
I am the process P4!

```

代码执行结果顺序为：P1、P3、P2、P4，符合实验要求。

原理解析：

进程前趋图为：



P1 为在主函数中直接通过 fork 的方式产生，最先执行；

```

void *createP2(void *arg) {
    pid_t  p2;
    sem_wait(mySem);
    while((p2=fork())!=-1);
    if(p2>0)
        printf("I am the process P2!\n");
    sem_post(mySem);
    sem_post(mySem2);
    return NULL;
}

void *createP3(void *arg) {
    pid_t  p3;
    sem_wait(mySem);
    while((p3=fork())!=-1);

```

```

    if(p3>0)
        printf("I am the process P3!\n");
    sem_post(mySem);
    sem_post(mySem3);
    return NULL;
}

```

由上述子函数可知：两个线程同时阻塞等待 `sem_wait(mySem)`，则 P2、P3 互斥执行；

```

void *createP4(void *arg) {
    pid_t p4;
    sem_wait(mySem2);
    sem_wait(mySem3);
    while((p4=fork())!=-1);
    if(p4>0)
        printf("I am the process P4!\n");
    sem_post(mySem2);
    sem_post(mySem3);
    return NULL;
}

```

由子函数 `createP4` 可知：必须等待 `sem_wait(mySem2)`、`sem_wait(mySem3)`；都实现后，才可创建进程 P4，P4 最后执行。

Task 2:

实验要求:

火车票余票数 `ticketCount` 初始值为 1000，有一个售票线程，一个退票线程，各循环执行多次。添加同步机制，使得结果始终正确。要求多次测试添加同步机制前后的实验效果。（说明：为了更容易产生并发错误，可以在适当的位置增加一些 `pthread_yield()`，放弃 CPU，并强制线程频繁切换。）

程序代码:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>
int pthread_yield(void);
int ticketCount = 1000;
sem_t *mySem = NULL;
void *ticket(void *arg) { //售票
    int temp;
    while(1){

```

```

        if(ticketCount>0){
            sem_wait(mySem); //同步操作
            temp=ticketCount;
            pthread_yield();
            temp=temp-1;
            pthread_yield();
            ticketCount=temp;
            sleep(1);
            printf("售票 1 张,现有票数: %d!\n",ticketCount);
            sem_post(mySem); //同步操作
        }
    }
    return NULL;
}

void *refund(void *arg) {    //退票
    int temp;
    while(1){
        if(ticketCount<1000){
            sem_wait(mySem); //同步操作
            temp=ticketCount;
            pthread_yield();
            temp=temp+1;
            pthread_yield();
            ticketCount=temp;
            sleep(1);
            printf("退票 1 张,现有票数: %d!\n",ticketCount);
            sem_post(mySem); //同步操作
        }
    }
    return NULL;
}

int main(void)
{
    pthread_t p1, p2;
    printf("现有票数: %d 张,售票退票线程开始!\n",ticketCount);
    mySem = sem_open("mySem", O_CREAT, 0666, 1);
    pthread_create(&p1, NULL, ticket, NULL);
    pthread_create(&p2, NULL, refund, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    sem_close(mySem); //线程结束
    sem_unlink("mySem"); //主程序结尾
    return 0;
}

```

代码解析：

注：

带有“//同步操作”的程序代码行为实现同步机制所添加，在不想要同步前应注释掉；

主函数产生两个线程 P1、P2 分别执行子函数 ticke（售票）、refund（退票）用来售票、退票操作。程序代码应注意：在没有售出票以前，不允许退票。在售票 1000 张后，不在允许售票。售票、退票两个线程同时执行，并在适当的位置增加一些 pthread_yield()，放弃 CPU，强制线程频繁切换。

通过设置信号量 mySem 用来实现同步机制，即不能同时对余票量 ticketCount 进行写操作。

实验结果：

添加同步前：

```
yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o task2 task2.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./task2
现有票数：1000张,售票退票线程开始!
售票1张,现有票数：1000!
退票1张,现有票数：999!
售票1张,现有票数：1000!
退票1张,现有票数：999!
售票1张,现有票数：1000!
退票1张,现有票数：999!
售票1张,现有票数：1000!
退票1张,现有票数：1000!
售票1张,现有票数：1000!
退票1张,现有票数：999!
售票1张,现有票数：1000!
退票1张,现有票数：999!
售票1张,现有票数：1000!
退票1张,现有票数：999!
售票1张,现有票数：1000!
退票1张,现有票数：999!
```

添加同步后：

```

yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o task2 task2.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./task2
现有票数：1000张,售票退票线程开始!
售票1张,现有票数：999!
售票1张,现有票数：998!
售票1张,现有票数：997!
售票1张,现有票数：996!
售票1张,现有票数：995!
售票1张,现有票数：994!
售票1张,现有票数：993!
售票1张,现有票数：992!
售票1张,现有票数：991!
售票1张,现有票数：990!
售票1张,现有票数：989!
售票1张,现有票数：988!
售票1张,现有票数：987!
售票1张,现有票数：986!

```

```

售票1张,现有票数：969!
售票1张,现有票数：968!
售票1张,现有票数：967!
售票1张,现有票数：966!
退票1张,现有票数：967!
退票1张,现有票数：968!
退票1张,现有票数：969!
退票1张,现有票数：970!
退票1张,现有票数：971!
退票1张,现有票数：972!
退票1张,现有票数：973!
退票1张,现有票数：974!

```

```

退票1张,现有票数：996!
退票1张,现有票数：997!
退票1张,现有票数：998!
退票1张,现有票数：999!
退票1张,现有票数：1000!
售票1张,现有票数：999!
售票1张,现有票数：998!
售票1张,现有票数：997!
售票1张,现有票数：996!
售票1张,现有票数：995!
售票1张,现有票数：994!
售票1张,现有票数：993!
售票1张,现有票数：992!
售票1张,现有票数：991!
售票1张,现有票数：990!

```



```

退票1张,现有票数:988!
退票1张,现有票数:989!
退票1张,现有票数:990!
退票1张,现有票数:991!
退票1张,现有票数:992!
售票1张,现有票数:991!
售票1张,现有票数:990!
售票1张,现有票数:989!
售票1张,现有票数:988!
售票1张,现有票数:987!
售票1张,现有票数:986!
售票1张,现有票数:985!
售票1张,现有票数:984!

```

实验结果解析:

添加同步前，两个线程同时执行，当售票一张后，还未打印余票量 ticketCount 时，退票线程又退票一张，使得余票量 ticketCount 又加一，打印余票量时 ticketCount 为退票执行后，则打印出结果为：售票 1 张, 现有票数：1000!；明显结果错误！因此，未添加同步前，两个线程可同时对余票量 ticketCount 进行读写操作，则导致显示结果不可信，数据不准确！

添加同步后，两个线程共享信号量 mysem，互斥执行，即当售票时，退票阻塞等待；即当退票时，售票阻塞等待。只有当售票完全结束后，余票量 ticketCount 变化后，打印余票量 ticketCount 后，才可进行下一线程操作；退票时也相同。因此，添加同步后，两个线程不可同时对余票量 ticketCount 进行读写操作，则数据具有准确性！退票，售票进程是处理机随机调度，不具有可预测性！当退票数达到 1000 后，不可在进行退票操作，符合现实情况，待有票售出后，才可继续进行退票操作（结果演示请看添加同步后第 3 张截图）。

Task3:

实验要求:

一个生产者一个消费者线程同步。设置一个线程共享的缓冲区，char buf[10]。一个线程不断从键盘输入字符到 buf, 一个线程不断的把 buf 的内容输出到显示器。要求输出的和输入的字符和顺序完全一致。（在输出线程中，每次输出睡眠一秒钟，然后以不同的速度输入测试输出是否正确）。要求多次测试添加同步机制前后的实验效果。

程序代码:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>

```



```

#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>

char buffer[10];
int i=0,j=0;
sem_t *full = NULL;
sem_t *empty = NULL;
void *worker1(void *arg) { //生产者
    while(1){
        sem_wait(empty);
        scanf("%c",&buffer[j]);j++;
        sleep(1);
        if(j>=10)j%=10;
        sem_post(full);
    }
    return NULL;
}
void *worker2(void *arg) { //消费者
    while(1){
        sem_wait(full);
        printf("输出:%c\n",buffer[i]);i++;
        sleep(1);
        if(i>=10)i%=10;
        sem_post(empty);
    }
    return NULL;
}
int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    full = sem_open("full", O_CREAT, 0666, 0);
    empty = sem_open("empty", O_CREAT, 0666, 10);
    pthread_create(&p1, NULL, worker1, NULL);
    pthread_create(&p2, NULL, worker2, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    sem_close(full);//线程结束
    sem_close(empty);//线程结束
    sem_unlink("full");//主程序结尾
    sem_unlink("empty");//主程序结尾
    return 0;
}

```

代码解析：

主函数产生两个线程 P1、P2 分别执行子函数 worker1（生产者）、worker2（消费者）用来往 buffer 缓冲区中读入、读出数据操作。设置一个共享信号量 full，用来表示 buffer 缓冲区中数据个数；设置一个共享信号量 empty，用来表示 buffer 缓冲区中空闲位置个数；

需注意：在没有数据以前，不允许读出，worker2() 函数阻塞等待；在数据存满以后，不允许读入，worker1() 函数阻塞等待。

实验结果：

第一次测试：

```
yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o task3 task3.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./task3
qwert
输出:q
输出:w
输出:e
输出:r
输出:t
输出:
```

第二次测试：

```
asdfgghh
输出:a
输出:s
输出:d
输出:f
输出:g
输出:g
输出:h
输出:h
输出:
```

第三次测试（输入数据大于 10）：

```
1234567890asd
输出:1
输出:2
输出:3
输出:4
输出:5
输出:6
输出:7
输出:8
输出:9
输出:0
输出:a
输出:s
输出:d
输出:
```

第四次测试（连续输入）：

```
123
45输出:1
6
输出:2
qwe输出:3

as输出:

d
zx输出:4
c
输出:5
输出:6
输出:

输出:q
输出:w
输出:e
输出:

输出:a
输出:s
输出:d
输出:

输出:z
输出:x
输出:c
输出:
```

实验结果解析：

一个生产者一个消费者线程共享的一个缓冲区：char buf[10]；一个线程不断从键盘输入字符到 buf, 一个线程不断的把 buf 的内容输出到显示器。在实验结果截图中：“输出：”前为输入的字符，后为输出的字符。

empty 的初始值为 10，full 的初始值为 0，输入字符最多可以连续读入 10 个，其他的在 I/O 缓冲区等待输入；当 full 的值大于 0 时，输出线程便可进行输出，每输出一个字符便会 post 一个 empty 信号量，此时输入线程接收到 empty 信号量便可开始从 I/O 缓冲区继续读取数据。根据实验结果可知：当输入数据很长或间断输入时，都可保证输出的和输入的字符和顺序完全一致，实验结果准确。

Task4:

实验要求：

进程通信问题。阅读并运行共享内存、管道、消息队列三种机制的代码

(参见 <https://www.cnblogs.com/Jimmy1988/p/7706980.html>;
<https://www.cnblogs.com/Jimmy1988/p/7699351.html>;
<https://www.cnblogs.com/Jimmy1988/p/7553069.html>)

实验测试:

a) 通过实验测试, 验证共享内存的代码中, receiver 能否正确读出 sender 发送的字符串? 如果把其中互斥的代码删除, 观察实验结果有何不同? 如果在发送和接收进程中打印输出共享内存地址, 他们是否相同, 为什么?

程序代码:

```
/*
 * Filename: Sender.c
 * Description:
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char *argv[])
{
    key_t    key;
    int shm_id;
    int sem_id;
    int value = 0;
    //1.Product the key
    key = ftok(".", 0xFF);
    //2. Creat semaphore for visit the shared memory
    sem_id = semget(key, 1, IPC_CREAT|0644);
    if(-1 == sem_id)
    {
        perror("semget");
        exit(EXIT_FAILURE);
    }
    //3. init the semaphore, sem=0
    if(-1 == (semctl(sem_id, 0, SETVAL, value)))
    {
```

```

        perror("semctl");
        exit(EXIT_FAILURE);
    }
    //4. Creat the shared memory(1K bytes)
    shm_id = shmget(key, 1024, IPC_CREAT|0644);
    if(-1 == shm_id)
    {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
    //5. attach the shm_id to this process
    char *shm_ptr;
    shm_ptr = shmat(shm_id, NULL, 0);
    if(NULL == shm_ptr)
    {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    //6. Operation procedure
    struct sembuf sem_b;
    sem_b.sem_num = 0;          //first sem(index=0)
    sem_b.sem_flg = SEM_UNDO;
    sem_b.sem_op = 1;           //Increase 1,make sem=1
    while(1)
    {
        if(0 == (value = semctl(sem_id, 0, GETVAL)))
        {
            printf("\nNow, snd message process running:\n");
            printf("\tInput the snd message:  ");
            scanf("%s", shm_ptr);

            if(-1 == semop(sem_id, &sem_b, 1))
            {
                perror("semop");
                exit(EXIT_FAILURE);
            }
        }
        //if enter "end", then end the process
        if(0 == (strcmp(shm_ptr, "end")))
        {
            printf("\nExit sender process now!\n");
            break;
        }
    }
}

```

```

        shmdt(shm_ptr);
        return 0;
    }

/*
 * Filename: Receiver.c
 * Description:
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char *argv[])
{
    key_t    key;
    int shm_id;
    int sem_id;
    int value = 0;
    //1.Product the key
    key = ftok(".", 0xFF);
    //2. Creat semaphore for visit the shared memory
    sem_id = semget(key, 1, IPC_CREAT|0644);
    if(-1 == sem_id)
    {
        perror("semget");
        exit(EXIT_FAILURE);
    }
    //3. init the semaphore, sem=0
    if(-1 == (semctl(sem_id, 0, SETVAL, value)))
    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }
    //4. Creat the shared memory(1K bytes)
    shm_id = shmget(key, 1024, IPC_CREAT|0644);
    if(-1 == shm_id)
    {
        perror("shmget");
    }
}

```

```

        exit(EXIT_FAILURE);
    }
    //5. attach the shm_id to this process
    char *shm_ptr;
    shm_ptr = shmat(shm_id, NULL, 0);
    if(NULL == shm_ptr)
    {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    //6. Operation procedure
    struct sembuf sem_b;
    sem_b.sem_num = 0;          //first sem(index=0)
    sem_b.sem_flg = SEM_UNDO;
    sem_b.sem_op = -1;          //Increase 1,make sem=1
    while(1)
    {
        if(1 == (value = semctl(sem_id, 0, GETVAL)))
        {
            printf("\nNow, receive message process running:\n");
            printf("\tThe message is : %s\n", shm_ptr);

            if(-1 == semop(sem_id, &sem_b, 1))
            {
                perror("semop");
                exit(EXIT_FAILURE);
            }
        }
        //if enter "end", then end the process
        if(0 == (strcmp(shm_ptr, "end")))
        {
            printf("\nExit the receiver process now!\n");
            break;
        }
    }
    shmdt(shm_ptr);
    //7. delete the shared memory
    if(-1 == shmctl(shm_id, IPC_RMID, NULL))
    {
        perror("shmctl");
        exit(EXIT_FAILURE);
    }
    //8. delete the semaphore
    if(-1 == semctl(sem_id, 0, IPC_RMID))

```



```

    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }
    return 0;
}

```

代码执行：

打开两个终端，分别执行编译命令：

```

gcc -o Sender Sender.c -lpthread
gcc -o Receiver Receiver.c -lpthread

```

实验结果：

```

yuan@ubuntu:~/Desktop/OS/lab3$ ./Sender
Now, snd message process running:
    Input the snd message: Hello!
Now, snd message process running:
    Input the snd message: I am YSS!
Now, snd message process running:
    Input the snd message:
Now, snd message process running:
    Input the snd message:
Now, snd message process running:
    Input the snd message: end
Exit sender process now!

```

```

yuan@ubuntu:~/Desktop/OS/lab3$ ./Receiver
Now, receive message process running:
    The message is : Hello!
Now, receive message process running:
    The message is : I
Now, receive message process running:
    The message is : am
Now, receive message process running:
    The message is : YSS!
Exit the receiver process now!
yuan@ubuntu:~/Desktop/OS/lab3$

```

实验结果解析：

可见 Sender 进程发出的消息 Receiver 进程准确无误的接收；但 Sender 读入信息时用的是 `scanf("%s", shm_ptr)` 机制，遇到空格自动结束，则出现上述截图效果。

删除互斥的代码：

代码变化处仅以下两个地方：

Sender.c

```
while(1)
{
    printf("\nNow, snd message process running:\n");
    printf("\tInput the snd message: ");
    scanf("%s", shm_ptr);

    //if enter "end", then end the process
    if(0 == (strcmp(shm_ptr, "end")))
    {
        printf("\nExit sender process now!\n");
        break;
    }
}
```

Receiver.c

```
while(1)
{
    printf("\nNow, receive message process running:\n");
    printf("\tThe message is : %s\n", shm_ptr);
    sleep(1);
    //if enter "end", then end the process
    if(0 == (strcmp(shm_ptr, "end")))
    {
        printf("\nExit the receiver process now!\n");
        break;
    }
}
```

代码执行：

打开两个终端，分别执行编译命令：

```
gcc -o Sender2 Sender2.c -lpthread
```

```
gcc -o Receiver2 Receiver2.c -lpthread
```

实验结果：

```
Now, snd message process running:
    Input the snd message:  Hello

Now, snd message process running:
    Input the snd message:  I am YSS!

Now, snd message process running:
    Input the snd message:

Now, snd message process running:
    Input the snd message:

Now, snd message process running:
    Input the snd message:  end

Exit sender process now!
```

```
Now, receive message process running:  
    The message is :  
  
Now, receive message process running:  
    The message is :  
  
Now, receive message process running:  
    The message is :  
  
Now, receive message process running:  
    The message is :  
  
Now, receive message process running:  
    The message is :
```

```
Now, receive message process running:
The message is : Hello

Now, receive message process running:
The message is : Hello

Now, receive message process running:
The message is : Hello

Now, receive message process running:
The message is : Hello

Now, receive message process running:
The message is : Hello

Now, receive message process running:
The message is : YSS!

Now, receive message process running:
The message is : YSS!
```

```
Now, receive message process running:
The message is : YSS!

Now, receive message process running:
The message is : YSS!

Now, receive message process running:
The message is : YSS!

Now, receive message process running:
The message is : YSS!

Now, receive message process running:
The message is : YSS!

Now, receive message process running:
The message is : YSS!

Exit the receiver process now!
yuan@ubuntu:~/Desktop/OS/lab3$
```

实验结果解析:

可见 Sender 进程发出的消息 Receiver 进程接收出现错误;
当 Sender 进程未发送信息前, Receiver 进程一直输出空信息;
当 Sender 进程发送信息后, Receiver 进程一直输出刚接收到的信息;
因 Sender 读入信息时用的是 `scanf("%s", shm_ptr)` 机制, 遇到空格自动结

束，则读入“ I am YSS!”时，“YSS!”会覆盖前面的消息，Receiver 进程没有接收到“ I am ”。

在发送和接收进程中打印输出共享内存地址：


代码变动：

只需分别在 Sender、Receiver 代码中添加：

```
printf("共享内存地址:%p\n", shm_ptr);
```

（具体代码请看 Sender3.c、Receiver3.c）

实验结果：



```
yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o Sender3 Sender3.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./Sender3
共享内存地址:0x7fbe733c9000

Now, snd message process running:
Input the snd message: ^Z
[47] Stopped (core dumped) ./Sender3

yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o Receiver3 Receiver3.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./Receiver3
共享内存地址:0x7fe281c57000
```

可见共享内存地址不同。

原因分析：

共享内存映射函数 `shmat()`：

作用：将共享内存空间挂载到进程中

头文件：`#include <sys/shm.h>`

函数原型：

```
void *shmat(int shmid, const void *shmaddr, int shmflg)
```

参数：

`shmid`：`shmget()` 返回值

`shmaddr`：共享内存的映射地址，一般为 0（由系统自动分配地址）

`shmflg`：访问权限和映射条件

在执行共享内存映射函数 `shm_ptr(shm_id, NULL, 0)` 时：参数 `shmaddr` 为 `NULL`，则由系统自动分配地址，则两个进程共享内存地址可能不同。

实验测试：

b) 有名管道和无名管道通信系统调用是否已经实现了同步机制？通过实验验证，发送者和接收者如何同步的。比如，在什么情况下，发送者会阻塞，什么情况下，接收者会阻塞？

有名管道:

程序代码:

```
/*
 *File: fifo_send.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <fcntl.h>

#define FIFO "/tmp/my_fifo"
int main()
{
    char buf[] = "hello,world";
    //1. check the fifo file existed or not
    int ret;
    ret = access(FIFO, F_OK);
    if(ret == 0)    //file /tmp/my_fifo existed
    {
        system("rm -rf /tmp/my_fifo");
    }
    //2. creat a fifo file
    if(-1 == mkfifo(FIFO, 0766))
    {
        perror("mkfifo");
        exit(EXIT_FAILURE);
    }
    //3.Open the fifo file
    int fifo_fd;
    fifo_fd = open(FIFO, O_WRONLY);
    if(-1 == fifo_fd)
    {
        perror("open");
        exit(EXIT_FAILURE);
    }
    //4. write the fifo file
    int num = 0;
    num = write(fifo_fd, buf, sizeof(buf));
    if(num < sizeof(buf))
    {
        perror("write");
    }
}
```

```

        exit(EXIT_FAILURE);
    }
    printf("write the message ok!\n");
    close(fifo_fd);
    return 0;
}

/*
 *File: fifo_rcv.c
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <fcntl.h>

#define FIFO "/tmp/my_fifo"
int main()
{
    char buf[20] ;
    memset(buf, '\0', sizeof(buf));
    /*. check the fifo file existed or not
    int ret;
    ret = access(FIFO, F_OK);
    if(ret != 0)    //file /tmp/my_fifo existed
    {
        fprintf(stderr, "FIFO %s does not existed", FIFO);
        exit(EXIT_FAILURE);
    }
    //2.Open the fifo file
    int fifo_fd;
    fifo_fd = open(FIFO, O_RDONLY);
    if(-1 == fifo_fd)
    {
        perror("open");
        exit(EXIT_FAILURE);
    }
    //4. read the fifo file
    int num = 0;
    num = read(fifo_fd, buf, sizeof(buf));

```



```

printf("Read %d words: %s\n", num, buf);
close(fifo_fd);
return 0;
}

```

实验结果：

```

yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o fifo_rcv fifo_rcv.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./fifo_rcv
Read 12 words: hello,world
yuan@ubuntu:~/Desktop/OS/lab3$ █

yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o fifo_send fifo_send.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./fifo_send
write the message ok!
yuan@ubuntu:~/Desktop/OS/lab3$ █

```

实验结果分析：

有名管道通信系统调用已经实现了同步机制。

阻塞情况见下表：

读进程	写进程	FIFO 无数据	FIFO 有数据
	√	阻塞	阻塞
√		阻塞	阻塞
√	√	写入	读出(未满，读写同时)
√	√x	即写中途退出，读直接返回 0	same
√x	√	读中途退出，写返回 SIGPIPE	same

无名管道：

程序代码：

```

/*
 * Filename: pipe.c
 */

#include <stdio.h>
#include <unistd.h>    //for pipe()
#include <string.h>    //for memset()
#include <stdlib.h>    //for exit()

int main()

```

```

{
    int fd[2];
    char buf[20];
    if(-1 == pipe(fd))
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    write(fd[1], "hello,world", 12);
    memset(buf, '\0', sizeof(buf));

    read(fd[0], buf, 12);
    printf("The message is: %s\n", buf);
    return 0;
}

```

实验结果：

```

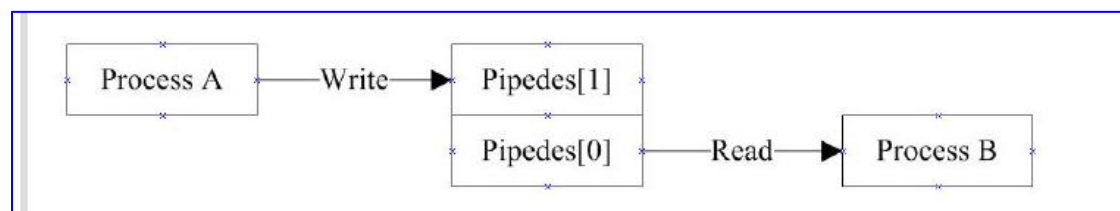
yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o pipe pipe.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./pipe
The message is: hello,world
yuan@ubuntu:~/Desktop/OS/lab3$

```

实验结果分析：

无名管道通信系统调用已经实现了同步机制。

无名管道存储文件描述符：



阻塞情况见下表：

读进程	写进程	管道无数据	管道有数据
阻塞✓		立刻返回 0	返回数据
阻塞✓	✓	读阻塞	读数据并返回
	阻塞✓	收到 SIGPIPE 信号，write 返回-1	
✓	阻塞✓	写入	若满，阻塞等待

实验测试：

c) 消息通信系统调用是否已经实现了同步机制？通过实验验证，发送者和接收者如何同步的。比如，在什么情况下，发送者会阻塞，什么情况下，接收者会阻塞？

程序代码：

client.c

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

#define BUF_SIZE 128
//Rebuild the struct (must be)
struct msgbuf
{
    long mtype;
    char mtext[BUF_SIZE];
};

int main(int argc, char *argv[])
{
    //1. creat a mseg queue
    key_t key;
    int msgld;
    printf("The process(%s),pid=%d started~\n", argv[0], getpid());
    key = ftok(".", 0xFF);
    msgld = msgget(key, IPC_CREAT|0644);
    if(-1 == msgld)
    {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    //2. creat a sub process, wait the server message
    pid_t pid;
    if(-1 == (pid = fork()))
    {
        perror("vfork");
        exit(EXIT_FAILURE);
    }
    //In child process
```

```

if(0 == pid)
{
    while(1)
    {
        alarm(0);
        alarm(100);    //if doesn't receive message in 100s, timeout & exit
        struct msgbuf rcvBuf;
        memset(&rcvBuf, '\0', sizeof(struct msgbuf));
        msgrcv(msgId, &rcvBuf, BUF_SIZE, 2, 0);
        printf("Server said: %s\n", rcvBuf.mtext);
    }
    exit(EXIT_SUCCESS);
}
else    //parent process
{
    while(1)
    {
        usleep(100);
        struct msgbuf sndBuf;
        memset(&sndBuf, '\0', sizeof(sndBuf));
        char buf[BUF_SIZE];
        memset(buf, '\0', sizeof(buf));
        printf("\nInput snd msg: ");
        scanf("%s", buf);
        strncpy(sndBuf.mtext, buf, strlen(buf)+1);
        sndBuf.mtype = 1;
        if(-1 == msgsnd(msgId, &sndBuf, strlen(buf)+1, 0))
        {
            perror("msgsnd");
            exit(EXIT_FAILURE);
        }
        //if scanf "end~", exit
        if(!strcmp("end~", buf))
            break;
    }
    printf("The process(%s),pid=%d exit~\n", argv[0], getpid());
}
return 0;
}

```

server.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

#include <unistd.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <signal.h>

#define BUF_SIZE 128
//Rebuild the struct (must be)
struct msgbuf
{
    long mtype;
    char mtext[BUF_SIZE];
};

int main(int argc, char *argv[])
{
    //1. creat a mseg queue
    key_t key;
    int msgld;
    key = ftok(".", 0xFF);
    msgld = msgget(key, IPC_CREAT|0644);
    if(-1 == msgld)
    {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    printf("Process (%s) is started, pid=%d\n", argv[0], getpid());
    while(1)
    {
        alarm(0);
        alarm(600); //if doesn't receive messge in 600s, timeout & exit
        struct msgbuf rcvBuf;
        memset(&rcvBuf, '\0', sizeof(struct msgbuf));
        msgrcv(msgld, &rcvBuf, BUF_SIZE, 1, 0);
        printf("Receive msg: %s\n", rcvBuf.mtext);
        struct msgbuf sndBuf;
        memset(&sndBuf, '\0', sizeof(sndBuf));
        strncpy((sndBuf.mtext), (rcvBuf.mtext), strlen(rcvBuf.mtext)+1);
        sndBuf.mtype = 2;
        if(-1 == msgsnd(msgld, &sndBuf, strlen(rcvBuf.mtext)+1, 0))
        {
            perror("msgsnd");
            exit(EXIT_FAILURE);
        }
    }
}

```

```

        //if scanf "end~", exit
        if(!strcmp("end~", rcvBuf.mtext))
            break;
    }
    printf("The process(%s),pid=%d exit~\n", argv[0], getpid());
    return 0;
}

```

代码解析：

server.c:

等待接收客户端发送的数据，若时间超出 600s，则自动 exit；
当收到信息后，打印接收到的数据；并原样的发送给客户端，由客户端显示。

client.c:

启动两个进程（父子进程），父进程用于发送数据，子进程接收由 server 发送的数据；

发送数据：由使用者手动输入信息，回车后发送；当写入“end~”后，退出本进程。

接收数据：接收由 Server 端发送的数据信息，并打印。

实验结果：

```

yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o client client.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./client
The process(./client),pid=6083 started~

Input snd msg: Hello

Input snd msg: YSS!

Input snd msg: end~
The process(./client),pid=6083 exit~
yuan@ubuntu:~/Desktop/OS/lab3$

```

```

yuan@ubuntu:~/Desktop/OS/lab3$ gcc -o server server.c -lpthread
yuan@ubuntu:~/Desktop/OS/lab3$ ./server
Process (./server) is started, pid=6098
Receive msg: Hello
Receive msg: YSS!
Receive msg: end~
The process(./server),pid=6098 exit~
yuan@ubuntu:~/Desktop/OS/lab3$

```

实验结果分析：

消息通信系统调用已经实现了同步机制。

阻塞机制：当 client 不阻塞时，接受 server 消息的时候会一直打印空消息；
当 server 不阻塞时，server 会一直接受空消息并转发给 client。

Task5:

实验要求:

阅读 Pintos 操作系统，找到并阅读进程上下文切换的代码，说明实现的保存和恢复的上下文内容以及进程切换的工作流程。

进程上下文切换流程分析:

因本人才疏学浅，很难理解 Pintos 操作系统进程上下文切换的相关代码，寻遍网上相关资料，获取到以下相关进程切换的工作流程相关知识，并理解学习，自身提高，将相关知识粘贴如下：

我们先来看一下 **devices** 目录下 **timer.c** 中的 **timer_sleep** 实现:

```
1 /* Sleeps for approximately TICKS timer ticks. Interrupts must 2 be turned
on. */ 3 void 4 timer_sleep (int64_t ticks) 5 { 6 int64_t start = timer_ticks
(); 7 ASSERT (intr_get_level () == INTR_ON); 8 while (timer_elapsed (start)
< ticks) 9 thread_yield();10 }
```

第 6 行：调用了 **timer_ticks** 函数，让我们来看看这个函数做了什么。

```
1 /* Returns the number of timer ticks since the OS booted. */ 2 int64_t 3 timer_ticks
(void) 4 { 5 enum intr_level old_level = intr_disable (); 6 int64_t t = ticks; 7
intr_set_level (old_level); 8 return t; 9 }
```

然后我们注意到这里有个 **intr_level** 的东西通过 **intr_disable** 返回了一个东西，没关系，我们继续往下找。

```
1 /* Interrupts on or off? */ 2 enum intr_level 3 { 4 INTR_OFF, /*
Interrupts disabled. */ 5 INTR_ON /* Interrupts enabled. */ 6 };
```

```
1 /* Disables interrupts and returns the previous interrupt status. */ 2 enum
intr_level 3 intr_disable (void) 4 { 5 enum intr_level old_level =
intr_get_level (); 6 7 /* Disable interrupts by clearing the interrupt flag.
8 See [IA32-v2b] "CLI" and [IA32-v3a] 5.8.1 "Masking Maskable 9 Hardware
Interrupts". */ 10 asm volatile ("cli" : : : "memory"); 11 12 return
old_level; 13 }
```


这里很明显, `intr_level` 代表能否被中断, 而 `intr_disable` 做了两件事情: 1. 调用 `intr_get_level()` 2. 直接执行汇编代码, 调用汇编指令来保证这个线程不能被中断。

注意: 这个 `asm volatile` 是在 C 语言中内嵌了汇编语言, 调用了 `CLI` 指令, `CLI` 指令不是 `command line interface`, 而是 `clear interrupt`, 作用是将标志寄存器的 `IF` (`interrupt flag`) 位置为 0, `IF=0` 时将不响应可屏蔽中断。

好, 让我们继续来看 `intr_get_level`。

```
1 /* Returns the current interrupt status. */ 2 enum intr_level 3 intr_get_level
(void) 4 { 5     uint32_t flags; 6     7     /* Push the flags register on the processor
stack, then pop the 8         value off the stack into `flags'. See [IA32-v2b] "PUSHF"
9         and "POP" and [IA32-v3a] 5.8.1 "Masking Maskable Hardware10
Interrupts". */11     asm volatile ("pushfl; popl %0" : "=g" (flags));12 13     return
flags & FLAG_IF ? INTR_ON : INTR_OFF;14 }
```

这里就是 `intr_disable` 函数调用的最深的地方了!

这个函数一样是调用了汇编指令, 把标志寄存器的东西放到处理器栈上, 然后把值 `pop` 到 `flags` (代表标志寄存器 `IF` 位) 上, 通过判断 `flags` 来返回当前终端状态(`intr_level`)。

好, 到这里。函数嵌套了这么多层, 我们整理一下逻辑:

1. `intr_get_level` 返回了 `intr_level` 的值

2. `intr_disable` 获取了当前的中断状态, 然后将当前中断状态改为不能被中断, 然后返回执行之前的中断状态。

有以上结论我们可以知道: `timer_ticks` 第五行做了这么一件事情: 禁止当前行为被中断, 保存禁止被中断前的中断状态 (用 `old_level` 储存)。

让我们再来看 `timer_ticks` 剩下的做了什么, 剩下的就是用 `t` 获取了一个全局变量 `ticks`, 然后返回, 其中调用了 `set_level` 函数。

```
1 /* Enables or disables interrupts as specified by LEVEL and2     returns the
previous interrupt status. */3 enum intr_level4 intr_set_level (enum intr_level
level) 5 {6     return level == INTR_ON ? intr_enable () : intr_disable ();7 }
```

有了之前的基础, 这个函数就很容易看了, 如果之前是允许中断的 (`INTR_ON`) 则 `enable` 否则就 `disable`。

而 `intr_enable` 正如你们所想, 实现和之前基本一致:

```
1 /* Enables interrupts and returns the previous interrupt status. */ 2 enum
intr_level 3 intr_enable (void) 4 { 5     enum intr_level old_level =
```

```
intr_get_level (); 6  ASSERT (!intr_context ()); 7  8  /* Enable interrupts by
setting the interrupt flag. 9 10      See [IA32-v2b] "STI" and [IA32-v3a] 5.8.1
"Masking Maskable11      Hardware Interrupts". */12  asm volatile ("sti");13 14
return old_level;15 }
```

说明一下， `sti` 指令就是 `cli` 指令的反面，将 `IF` 位置为 1。

然后有个 `ASSERT` 断言了 `intr_context` 函数返回结果的 `false`。

再看 `intr_context`

```
1 /* Returns true during processing of an external interrupt2 and false at all
other times. */3 bool4 intr_context (void) 5 {6  return in_external_intr;7 }
```

这里直接返回了是否外中断的标志 `in_external_intr`，就是说 `ASSERT` 断言这个中断不是外中断（IO 等，也称为硬中断）而是操作系统正常线程切换流程里的内中断（也称为软中断）。

好的，至此，我们总结一下：

这么多分析其实分析出了 `pintos` 操作系统如何利用中断机制来确保一个原子性的操作的。

我们来看，我们已经分析完了 `timer_ticks` 这个函数，它其实就是获取 `ticks` 的当前值返回而已，而第 5 行和第 7 行做的其实只是确保这个过程是不能被中断的而已。

那么我们来达成一个共识，被以下两个语句包裹的内容目的是为了保证这个过程不被中断。

```
1 enum intr_level old_level = intr_disable ();2 ...3 intr_set_level (old_level);
```

好的，那么 `ticks` 又是什么？来看 `ticks` 定义。

```
1 /* Number of timer ticks since OS booted. */2 static int64_t ticks;
```

从 `pintos` 被启动开始，`ticks` 就一直在计时，代表着操作系统执行单位时间的前进计量。

好，现在回过来看 `timer_sleep` 这个函数，`start` 获取了起始时间，然后断言必须可以被中断，不然会一直死循环下去，然后就是一个循环

```
1 while (timer_elapsed (start) < ticks)2  thread_yield();
```

注意这个 `ticks` 是函数的形参不是全局变量，然后看一下这两个函数：

```
1 /* Returns the number of timer ticks elapsed since THEN, which2 should be
a value once returned by timer_ticks(). */3 int64_t4 timer_elapsed (int64_t then)5
{6  return timer_ticks () - then;7 }
```

很明显 `timer_elapsed` 返回了当前时间距离 `then` 的时间间隔，那么这个循环实质就是在 `ticks` 的时间内不断执行 `thread_yield`。

那么我们最后来看 `thread_yield` 是什么就可以了：

```
1 /* Yields the CPU. The current thread is not put to sleep and 2 may be
scheduled again immediately at the scheduler's whim. */ 3 void 4 thread_yield
(void) 5 { 6 struct thread *cur = thread_current (); 7 enum intr_level old_level;
8 9 ASSERT (!intr_context ()); 10 11 old_level = intr_disable (); 12 if
(cur != idle_thread) 13 list_push_back (&ready_list, &cur->elem); 14
cur->status = THREAD_READY; 15 schedule (); 16 intr_set_level (old_level); 17 }
```

第 6 行 `thread_current` 函数做的事情已经可以顾名思义了，不过具有钻研精神和强迫症的你还是要确定它的具体实现：

```
1 /* Returns the running thread. 2 This is running_thread() plus a couple of
sanity checks. 3 See the big comment at the top of thread.h for details. */
4 struct thread * 5 thread_current (void) 6 { 7 struct thread *t = running_thread
(); 8 9 /* Make sure T is really a thread. 10 If either of these assertions
fire, then your thread may 11 have overflowed its stack. Each thread has less
than 4 kB 12 of stack, so a few big automatic arrays or moderate 13 recursion
can cause stack overflow. */ 14 ASSERT (is_thread (t)); 15 ASSERT (t->status
== THREAD_RUNNING); 16 17 return t; 18 }
```

```
1 /* Returns the running thread. */ 2 struct thread * 3 running_thread (void)
4 { 5 uint32_t *esp; 6 7 /* Copy the CPU's stack pointer into `esp', and then
round that 8 down to the start of a page. Because `struct thread' is 9
always at the beginning of a page and the stack pointer is 10 somewhere in
the middle, this locates the current thread. */ 11 asm ("mov %%esp, %0" : "=g"
(esp)); 12 return pg_round_down (esp); 13 }
```

```
1 /* Returns true if T appears to point to a valid thread. */ 2 static bool 3 is_thread
(struct thread *t) 4 { 5 return t != NULL && t->magic == THREAD_MAGIC; 6 }
```

先来看 `thread_current` 调用的 `running_thread`，把 CPU 栈的指针复制到 `esp` 中，然后调用 `pg_round_down`

```
1 /* Round down to nearest page boundary. */ 2 static inline void *pg_round_down
(const void *va) { 3 return (void *) ((uintptr_t) va & ~PGMASK); 4 }
```

好，这里又涉及到这个操作系统是怎么设计页面的了：

```

1 /* Page offset (bits 0:12). */2 #define PGSHIFT 0 /* Index
of first offset bit. */3 #define PGBITS 12 /* Number of
offset bits. */4 #define PGSIZE (1 << PGBITS) /* Bytes in a page. */5
#define PGMASK BITMASK(PGSIFT, PGBITS) /* Page offset bits (0:12). */

1 /* Functions and macros for working with virtual addresses.2 3 See pte.h for
functions and macros specifically for x864 hardware page tables. */5 6 #define
BITMASK(SHIFT, CNT) (((1ul << (CNT)) - 1) << (SHIFT))

```

一个页面 12 位，PGMASK 调用 BITMASK 其实就是一个页面全部位都是 1 的这么一个 MASK，注意 1ul 的意思是 unsigned long 的 1。

然后来看 pg_round_down，对 PGMASK 取反的结果就是一个页面大小全部为 0 的这么个数，然后和传过来的指针做与操作的结果就是清 0 指针的靠右 12 位。

这里有什么效果呢？我们知道一个页面 12 位，而 struct thread 是在一个页面的最开始的位置，所以对任何一个页面的指针做 pg_round_down 的结果就是返回到这个页面最开始线程结构体的位置。

好，我们现在分析出了 pg_round_down 其实就是返回了这个页面线程的最开始指针，那么 running_thread 的结果返回当前线程起始指针。

再来看 thread_current 里最后的两个断言，一个断言 t 指针是一个线程，一个断言这个线程处于 THREAD_RUNNING 状态。

然后 is_thread 用的 t->magic 其实是用于检测时候有栈溢出的这么个元素。

```

1 /* Owned by thread.c. */2 unsigned magic; /* Detects stack
overflow. */

```

好，现在 thread_current 分析完了，这个就是返回当前线程起始指针位置。

我们继续看 thread_yield，然后剩下的很多东西其实我们已经分析过了，在分析的过程其实是对这个操作系统工作过程的剖析，很多地方都是相通的。

第 9 断言这是个软中断，第 11 和 16 包裹起来的就是我们之前分析的线程机制保证的一个原子性操作。

然后我们来看 12-15 做了什么：

```

1 if (cur != idle_thread)2 list_push_back (&ready_list, &cur->elem);3
cur->status = THREAD_READY;4 schedule ();

```

如何当前线程不是空闲的线程就调用 list_push_back 把当前线程的元素扔到就绪队列里面，并把线程改成 THREAD_READY 状态。

关于队列 list 的相关操作 mission2 会涉及到，这里先不作解释，顾名思义即可。

然后再调用 schedule:

```

1 /* Schedules a new process. At entry, interrupts must be off and 2 the running
process's state must have been changed from 3 running to some other state. This

```

```

function finds another 4 thread to run and switches to it. 5 6 It's not safe
to call printf() until thread_schedule_tail() 7 has completed. */ 8 static void
9 schedule (void) 10 { 11 struct thread *cur = running_thread (); 12 struct thread
*next = next_thread_to_run (); 13 struct thread *prev = NULL; 14 15 ASSERT
(intr_get_level () == INTR_OFF); 16 ASSERT (cur->status != THREAD_RUNNING); 17
ASSERT (is_thread (next)); 18 19 if (cur != next) 20 prev = switch_threads (cur,
next); 21 thread_schedule_tail (prev); 22 }

```

首先获取当前线程 `cur` 和调用 `next_thread_to_run` 获取下一个要 run 的线程:

```

1 /* Chooses and returns the next thread to be scheduled. Should 2 return a
thread from the run queue, unless the run queue is 3 empty. (If the running
thread can continue running, then it 4 will be in the run queue.) If the run
queue is empty, return 5 idle_thread. */ 6 static struct thread * 7
next_thread_to_run (void) 8 { 9 if (list_empty (&ready_list)) 10 return
idle_thread; 11 else 12 return list_entry (list_pop_front (&ready_list),
struct thread, elem); 13 }

```

如果就绪队列空闲直接返回一个空闲线程指针， 否则拿就绪队列第一个线程出来返回。

然后 3 个断言之前讲过就不多说了， 确保不能被中断， 当前线程是 `RUNNING_THREAD` 等。

如果当前线程和下一个要跑的线程不是同一个的话调用 `switch_threads` 返回给 `prev`。

```

1 /* Switches from CUR, which must be the running thread, to NEXT, 2 which must
also be running switch_threads(), returning CUR in 3 NEXT's context. */ 4 struct
thread *switch_threads (struct thread *cur, struct thread *next);

```

注意， 这个函数实现是用汇编语言实现的在 `threads/switch.S` 里:

```

1 ##### struct thread *switch_threads (struct thread *cur, struct thread *next);
2 ##### 3 ##### Switches from CUR, which must be the running thread, to NEXT, 4
##### which must also be running switch_threads(), returning CUR in 5 ##### NEXT's
context. 6 ##### 7 ##### This function works by assuming that the thread we're
switching 8 ##### into is also running switch_threads(). Thus, all it has to do
is 9 ##### preserve a few registers on the stack, then switch stacks and 10 #####
restore the registers. As part of switching stacks we record the 11 ##### current
stack pointer in CUR's thread structure. 12 13 .globl switch_threads 14 .func
switch_threads 15 switch_threads: 16 # Save caller's register state. 17 # 18
# Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx, 19 # but requires
us to preserve %ebx, %ebp, %esi, %edi. See 20 # [SysV-ABI-386] pages 3-11 and
3-12 for details. 21 # 22 # This stack frame must match the one set up by

```

```

thread_create()23    # in size.24    pushl %ebx25    pushl %ebp26
pushl %esi27    pushl %edi28 29    # Get offsetof (struct thread,
stack).30 .globl thread_stack_ofs31    mov thread_stack_ofs, %edx32 33    # Save
current stack pointer to old thread's stack, if any.34    movl
SWITCH_CUR(%esp), %eax35    movl %esp, (%eax,%edx,1)36 37    # Restore stack
pointer from new thread's stack.38    movl SWITCH_NEXT(%esp), %ecx39    movl
(%ecx,%edx,1), %esp40 41    # Restore caller's register state.42    popl %edi43
popl %esi44    popl %ebp45    popl %ebx46    ret47 .endfunc

```

分析一下这个汇编代码： 先 4 个寄存器压栈保存寄存器状态（保护作用）， 这 4 个寄存器是 `switch_threads_frame` 的成员：

```

1 /* switch_thread()'s stack frame. */ 2 struct switch_threads_frame 3 { 4
uint32_t edi; /* 0: Saved %edi. */ 5 uint32_t esi; /*
4: Saved %esi. */ 6 uint32_t ebp; /* 8: Saved %ebp. */ 7
uint32_t ebx; /* 12: Saved %ebx. */ 8 void (*eip) (void); /*
16: Return address. */ 9 struct thread *cur; /* 20: switch_threads()'s
CUR argument. */10 struct thread *next; /* 24: switch_threads()'s NEXT
argument. */11 };

```

然后全局变量 `thread_stack_ofs` 记录线程和栈之间的间隙， 我们都知道线程切换有个保存现场的过程，

来看 34,35 行， 先把当前的线程指针放到 `eax` 中， 并把线程指针保存在相对基地址偏移量为 `edx` 的地址中。

38,39: 切换到下一个线程的线程栈指针， 保存在 `ecx` 中， 再把这个线程相对基地址偏移量 `edx` 地址（上一次保存现场的时候存放的）放到 `esp` 当中继续执行。

这里 `ecx`, `eax` 起容器的作用， `edx` 指向当前现场保存的地址偏移量。

简单来说就是保存当前线程状态， 恢复新线程之前保存的线程状态。

然后再把 4 个寄存器拿出来， 这个是硬件设计要求的， 必须保护 `switch_threads_frame` 里面的寄存器才可以 `destroy` 掉 `eax`, `edx`, `ecx`。

然后注意到现在 `eax`(函数返回值是 `eax`)就是被切换的线程栈指针。

我们由此得到一个结论， `schedule` 先把当前线程丢到就绪队列， 然后把线程切换如果下一个线程和当前线程不一样的话。

然后再看 `shedule` 最后一行的函数 `thread_schedule_tail` 做了什么鬼， 这里参数 `prev` 是 `NULL` 或者在下一个线程的上下文中的当前线程指针。

```

1 /* Completes a thread switch by activating the new thread's page 2 tables,
and, if the previous thread is dying, destroying it. 3 4 At this function's
invocation, we just switched from thread 5 PREV, the new thread is already
running, and interrupts are 6 still disabled. This function is normally invoked
by 7 thread_schedule() as its final action before returning, but 8 the first
time a thread is scheduled it is called by 9 switch_entry() (see switch.S).10
11 It's not safe to call printf() until the thread switch is12 complete. In
practice that means that printf()'s should be13 added at the end of the
function.14 15 After this function and its caller returns, the thread switch16
is complete. */17 void18 thread_schedule_tail (struct thread *prev)19 {20 struct
thread *cur = running_thread ();21 22 ASSERT (intr_get_level () == INTR_OFF);23
24 /* Mark us as running. */25 cur->status = THREAD_RUNNING;26 27 /* Start
new time slice. */28 thread_ticks = 0;29 30 #ifdef USERPROG31 /* Activate the
new address space. */32 process_activate ();33 #endif34 35 /* If the thread
we switched from is dying, destroy its struct36 thread. This must happen late
so that thread_exit() doesn't37 pull out the rug under itself. (We don't
free38 initial_thread because its memory was not obtained via39
palloc().) */40 if (prev != NULL && prev->status == THREAD_DYING && prev !=
initial_thread)41 {42 ASSERT (prev != cur);43 palloc_free_page
(prev);44 }45 }

```

先是获得当前线程 `cur`，注意此时是已经切换过的线程了（或者还是之前 `run` 的线程，因为 `ready` 队列为空）。

然后把线程状态改成 `THREAD_RUNNING`，然后 `thread_ticks` 清零开始新的线程切换时间片。

然后调用 `process_activate` 触发新的地址空间。

```

1 /* Sets up the CPU for running user code in the current 2 thread. 3 This
function is called on every context switch. */ 4 void 5 process_activate (void)
6 { 7 struct thread *t = thread_current (); 8 9 /* Activate thread's page tables.
*/10 pagedir_activate (t->pagedir);11 12 /* Set thread's kernel stack for use
in processing13 interrupts. */14 tss_update ();15 }

```

这里先是拿到当前线程，调用 `pagedir_activate`:

```

1 /* Loads page directory PD into the CPU's page directory base 2 register.
*/ 3 void 4 pagedir_activate (uint32_t *pd) 5 { 6 if (pd == NULL) 7 pd =
init_page_dir; 8 9 /* Store the physical address of the page directory into
CR310 aka PDBR (page directory base register). This activates our11 new
page tables immediately. See [IA32-v2a] "MOV--Move12 to/from Control

```



```
Registers" and [IA32-v3a] 3.7.5 "Base13 Address of the Page Directory". */14
asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");15 }
```

这个汇编指令将当前线程的页目录指针存储到 **CR3**（页目录表物理内存基地址寄存器）中，也就是说这个函数更新了现在的页目录表。

最后来看 **tss_update**:

```
1 /* Sets the ring 0 stack pointer in the TSS to point to the end2 of the thread
stack. */3 void4 tss_update (void) 5 {6 ASSERT (tss != NULL);7 tss->esp0 =
(uint8_t *) thread_current () + PGSIZE;8 }
```

首先要弄清楚 **tss** 是什么，**tss** 是 **task state segment**，叫任务状态段，任务（进程）切换时的任务现场信息。

这里其实是把 **TSS** 的一个栈指针指向了当前线程栈的尾部，也就是更新了任务现场的信息和状态。

好，到现在 **process_activate** 分析完了，总结一下：其实就是做了 2 件事情：1.更新页目录表 2.更新任务现场信息（TSS）

我们现在继续来看 **thread_schedule_tail**，最后这是 4 行：

```
1 /* If the thread we switched from is dying, destroy its struct 2 thread.
This must happen late so that thread_exit() doesn't 3 pull out the rug under
itself. (We don't free 4 initial_thread because its memory was not obtained
via 5 palloc().) */ 6 if (prev != NULL && prev->status == THREAD_DYING &&
prev != initial_thread) 7 { 8 ASSERT (prev != cur); 9
palloc_free_page (prev);10 }
```

这里是如果我们切换的线程状态是 **THREAD_DYING**（代表欲要销毁的线程）的话，调用 **palloc_free_page**:

```
1 /* Frees the page at PAGE. */2 void3 palloc_free_page (void *page) 4 {5
palloc_free_multiple (page, 1);6 }
```

```
1 /* Frees the PAGE_CNT pages starting at PAGES. */ 2 void 3 palloc_free_multiple
(void *pages, size_t page_cnt) 4 { 5 struct pool *pool; 6 size_t page_idx;
7 8 ASSERT (pg_ofs (pages) == 0); 9 if (pages == NULL || page_cnt == 0)10
return;11 12 if (page_from_pool (&kernel_pool, pages))13 pool =
&kernel_pool;14 else if (page_from_pool (&user_pool, pages))15 pool =
&user_pool;16 else17 NOT_REACHED ();18 19 page_idx = pg_no (pages) - pg_no
```

```
(pool->base);20 21 #ifndef NDEB22  memset (pages, 0xcc, PGSIZE * page_cnt);23
#endif24 25  ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));26
bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);27 }
```

这里创建了一个 **pool** 的结构体:

```
1 /* A memory pool. */2 struct pool3  {4      struct lock lock;          /*
Mutual exclusion. */5      struct bitmap *used_map;      /* Bitmap of free
pages. */6      uint8_t *base;          /* Base of pool. */7  };
```

首先 **pallocc** 实现的是一个页分配器，这里 **pool** 的角色就是记忆分配的内容。这里结构体用位图记录空的页，关键是在这里又有一个操作系统很重要的知识概念出现了，就是 **lock**:

```
1 /* Lock. */2 struct lock3  {4      struct thread *holder;      /* Thread holding
lock (for debugging). */5      struct semaphore semaphore; /* Binary semaphore
controlling access. */6  };
```

然后锁其实是由二值信号量实现的:

```
1 /* A counting semaphore. */2 struct semaphore3  {4      unsigned value;
/* Current value. */5      struct list waiters;      /* List of waiting threads.
*/6  };
```

具体信号量方法实现在 **threads/synch.c** 中，这里不作更多讲解了，毕竟函数分析还没涉及到这里。

继续看 **pallocc_free_multiple**，第 8 行其实就是截取后 12 位，即获得当前页偏差量，断言为 0 就是说页指针应该指向线程结构体

```
1 /* Offset within a page. */2 static inline unsigned pg_ofs (const void *va)
{3      return (uintptr_t) va & PGMASK;4 }
```

然后分析 12-17 行，这里要弄清楚一点是系统 **memory** 分成 2 个池，一个是 **kernel pool**，一个是 **user pool**，**user pool** 是提供给用户页的，别的都是 **kernel pool**。

然后看下这里调用的 **page_from_pool** 函数:

```
1 /* Returns true if PAGE was allocated from POOL, 2 false otherwise. */3
static bool4 page_from_pool (const struct pool *pool, void *page)5 {6      size_t
page_no = pg_no (page);7      size_t start_page = pg_no (pool->base);8      size_t
end_page = start_page + bitmap_size (pool->used_map);9 10      return page_no >=
start_page && page_no < end_page;11 }
```

pg_no 是获取虚拟页数的，方法其实就是直接指针右移 12 位就行了:

```
1 /* Virtual page number. */2 static inline uintptr_t pg_no (const void *va) {3
return (uintptr_t) va >> PGBITS;4 }
```

然后这里获取当前池中的起始页和结束页位置，然后判断页面时候在这个池的 Number 范围之类来判断时候属于某个池。

再看 NOT_REACHED 函数,这个函数博主找了半天，最后用全文件搜索才找着在哪，在 lib/debug.h 中：

```
1 /* This is outside the header guard so that debug.h may be 2 included multiple
times with different settings of NDEBUG. */ 3 #undef ASSERT 4 #undef NOT_REACHED
5 6 #ifndef NDEBUG 7 #define ASSERT(CONDITION)
\ 8 if (CONDITION) { } else { \ 9
PANIC ("assertion '%s' failed.", #CONDITION); \10 }11 #define
NOT_REACHED() PANIC ("executed an unreachable statement");12 #else13 #define
ASSERT(CONDITION) ((void) 0)14 #define NOT_REACHED() for (;;);15 #endif /*
lib/debug.h */
```

```
1 /* GCC lets us add "attributes" to functions, function 2 parameters, etc.
to indicate their properties. 3 See the GCC manual for details. */ 4 #define
UNUSED __attribute__ ((unused)) 5 #define NO_RETURN __attribute__ ((noreturn))
6 #define NO_INLINE __attribute__ ((noinline)) 7 #define PRINTF_FORMAT(FMT, FIRST)
__attribute__ ((format (printf, FMT, FIRST))) 8 9 /* Halts the OS, printing the
source file name, line number, and10 function name, plus a user-specific message.
*/11 #define PANIC(...) debug_panic (__FILE__, __LINE__, __func__,
__VA_ARGS__)12 13 void debug_panic (const char *file, int line, const char
*function,14 const char *message, ...) PRINTF_FORMAT (4, 5)
NO_RETURN;
```

这里根据 NDEBUG 状态分两种 define，一个是 ASSERT 空函数，NOT_REACHED 执行死循环，一个是如果 ASSERT 参数 CONDITION 为 false 的话就调用 PANIC 输出文件，行数，函数名和用户信息，NOT_REACHED 也会输出信息。

有些童鞋在跑测试的时候会出现卡在一个地方不动的状态，其实不是因为你电脑的问题，而是当一些错误触发 NOT_REACHED 之类的问题的时候，因为非 debug 环境就一直执行死循环了，反映出来的行为就是命令行卡住不动没有输出。

注意这里的语法类似 __attribute__ 和 ((format(printf, m, n))) 是面向 gcc 编译器处理的写法，这里做的事情其实是参数声明和调用匹配性检查。

好，继续来看 pallocc_free_multiple，用 page_idx 保存了计算出来了页 id，清空了页指针，然后还剩下最后两行：

```
1 ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));2
bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
```

第一个断言:

```
1 /* Returns true if every bit in B between START and START + CNT, 2 exclusive,
is set to true, and false otherwise. */ 3 bool 4 bitmap_all (const struct bitmap
*b, size_t start, size_t cnt) 5 { 6 return !bitmap_contains (b, start, cnt,
false); 7 }
```

```
1 /* Returns true if any bits in B between START and START + CNT, 2 exclusive,
are set to VALUE, and false otherwise. */ 3 bool 4 bitmap_contains (const struct
bitmap *b, size_t start, size_t cnt, bool value) 5 { 6 size_t i; 7 8 ASSERT
(b != NULL); 9 ASSERT (start <= b->bit_cnt); 10 ASSERT (start + cnt <=
b->bit_cnt); 11 12 for (i = 0; i < cnt; i++) 13 if (bitmap_test (b, start +
i) == value) 14 return true; 15 return false; 16 }
```

bitmap_contains 首先做断言对参数正确性确认，然后如果所有位处于 **start** 到 **start+cnt** 都是 **value** 的话，别的都是 **~value** 的话，返回 **true**，从我们的函数调用来看就是断言位图全是 0。

```
1 /* Returns the value of the bit numbered IDX in B. */ 2 bool 3 bitmap_test (const
struct bitmap *b, size_t idx) 4 { 5 ASSERT (b != NULL); 6 ASSERT (idx <
b->bit_cnt); 7 return (b->bits[elem_idx (idx)] & bit_mask (idx)) != 0; 8 } 9
```

```
1 /* Returns the index of the element that contains the bit 2 numbered BIT_IDX.
*/ 3 static inline size_t 4 elem_idx (size_t bit_idx) 5 { 6 return bit_idx /
ELEM_BITS; 7 } 8 9 /* Returns an elem_type where only the bit corresponding to 10
BIT_IDX is turned on. */ 11 static inline elem_type 12 bit_mask (size_t bit_idx)
13 { 14 return (elem_type) 1 << (bit_idx % ELEM_BITS); 15 }
```

来看 **bit_test** 的实现，这里直接返回某一位的具体值。

这里直接用 **elem_idx** 获取 **idx** 对应的 **index** 取出位，然后和 **bit_mask** 做与操作，**bit_mask** 就是返回了一个只有 **idx** 位是 1 其他都是 0 的一个数，也就是说 **idx** 必须为 1 才返回 **true** 对 **bit_test** 来说，否则 **false**。

好，至此，对 **palloc_free_multiple** 只剩一行了：

```
1 bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
```

```
/* Sets the CNT bits starting at START in B to VALUE. */ void
bitmap_set_multiple (struct bitmap *b, size_t start, size_t cnt, bool value)
{
```

```

size_t i;

ASSERT (b != NULL);

ASSERT (start <= b->bit_cnt);

ASSERT (start + cnt <= b->bit_cnt);

for (i = 0; i < cnt; i++)

    bitmap_set (b, start + i, value);
}

```

这里对位图所有位都做了 `bitmap_set` 设置:

```

1 /* Atomically sets the bit numbered IDX in B to VALUE. */ 2 void 3 bitmap_set
(struct bitmap *b, size_t idx, bool value) 4 { 5     ASSERT (b != NULL); 6     ASSERT
(idx < b->bit_cnt); 7     if (value) 8         bitmap_mark (b, idx); 9     else 10
bitmap_reset (b, idx); 11 }

```

很明显这里 `mark` 就是设为 1, `reset` 就是置为 0。

来看一下实现:

```

1 /* Atomically sets the bit numbered BIT_IDX in B to true. */ 2 void 3 bitmap_mark
(struct bitmap *b, size_t bit_idx) 4 { 5     size_t idx = elem_idx (bit_idx); 6
elem_type mask = bit_mask (bit_idx); 7 8     /* This is equivalent to `b->bits[idx]
|= mask' except that it 9         is guaranteed to be atomic on a uniprocessor machine.
See 10         the description of the OR instruction in [IA32-v2b]. */ 11     asm
("orl %1, %0" : "=m" (b->bits[idx]) : "r" (mask) : "cc"); 12 } 13 14 /* Atomically
sets the bit numbered BIT_IDX in B to false. */ 15 void 16 bitmap_reset (struct
bitmap *b, size_t bit_idx) 17 { 18     size_t idx = elem_idx (bit_idx); 19     elem_type
mask = bit_mask (bit_idx); 20 21     /* This is equivalent to `b->bits[idx] &= ~mask'
except that it 22         is guaranteed to be atomic on a uniprocessor machine. See 23
the description of the AND instruction in [IA32-v2a]. */ 24     asm ("andl %1, %0" :
"=m" (b->bits[idx]) : "r" (~mask) : "cc"); 25 }

```

一样，最底层的实现依然是用汇编语言实现的，两个汇编语言实现的就是两个逻辑：1. `b->bits[idx] |= mask` 2. `b->bits[idx] &= ~mask`，这里 `mask` 都是只有 `idx` 位为 1，其他为 0 的 `mask`。

好，到现在位置 `palloc_free_multiple` 已经分析完了，整理一下逻辑：

其实就是把页的位图全部清 0 了，清 0 代表这个页表的所有页都是 `free` 的，等于清空了页目录表中的所有页面。

逻辑继续向上回溯：

`thread_schedule_tail` 其实就是获取当前线程， 分配恢复之前执行的状态和现场， 如果当前线程死了就清空资源。

`schedule` 其实就是拿下一个线程切换过来继续 `run`。

`thread_yield` 其实就是把当前线程扔到就绪队列里， 然后重新 `schedule`， 注意这里如果 `ready` 队列为空的话当前线程会继续在 `cpu` 执行。

最后回溯到我们最顶层的函数逻辑： `timer_sleep` 就是在 `ticks` 时间内， 如果线程处于 `running` 状态就不断把他扔到就绪队列不让他执行。

如有需要可以搜索：

http://www.cnblogs.com/laiy/p/pintos_project1_thread.html

<https://wenku.baidu.com/view/9e94430671fe910ef02df884.html>