

## 一.实验目的

- 系统调用的进一步理解。
- 进程上下文切换。
- 同步的方法。

## 二.实验题目

1. 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。
2. 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。)
3. 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。
4. 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？

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？

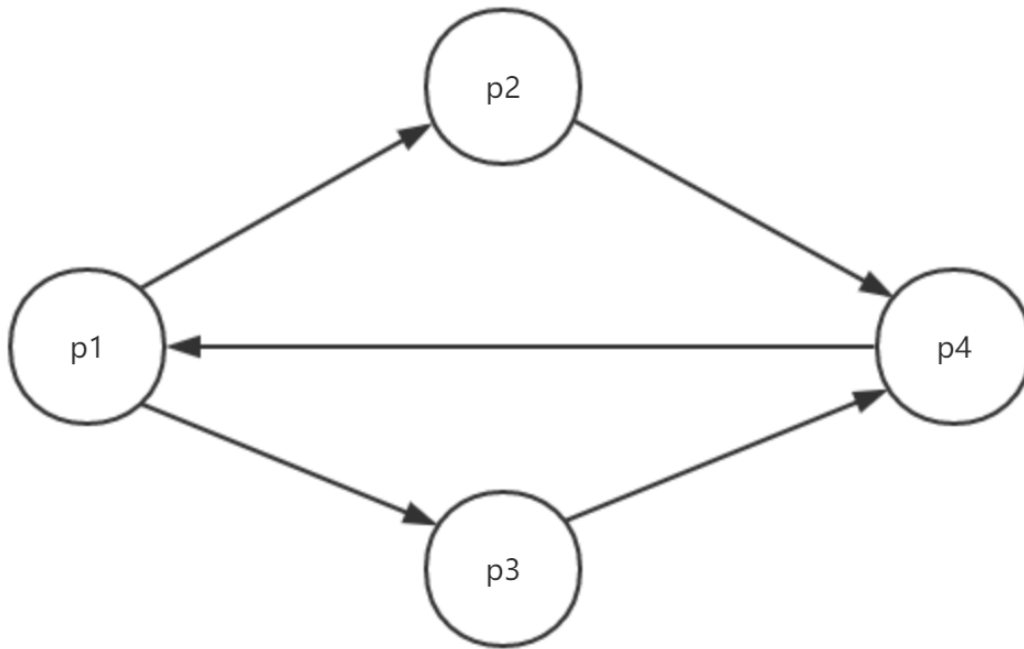
c) 消息通信系统调用是否已经实现了同步机制？通过实验验证，发送者和接收者如何同步的。比如，在什么情况下，发送者会阻塞，什么情况下，接收者会阻塞？

5. 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。

## 三.实验解答

### 1.实验1

据题意画出流程：



代码如下：

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

```
sem_t* A = NULL;
sem_t* B = NULL;
sem_t* C = NULL;
```

```
int main(int argc, char const *argv[])
{
    A = sem_open("A_name",O_CREAT,0666,1);
    B = sem_open("B_name",O_CREAT,0666,0);
    C = sem_open("C_name",O_CREAT,0666,0);
    int p1,p2,p3,p4;
    while((p1=fork())== -1);
    if(p1==0){
        while(1){
            sem_wait(A);
            printf("I am p1\n");
            sleep(1);
            sem_post(B);
        }
    }
    while((p2=fork())== -1);
    if(p2==0){
```

```

        while(1){
            sem_wait(B);
            printf("I am p2\n");
            sleep(1);
            sem_post(C);
        }
    }
    while((p3=fork())!=-1);
    if(!p3){
        while(1){
            sem_wait(B);
            printf("I am p3\n");
            sleep(1);
            sem_post(C);
        }
    }
    while((p4=fork())!=-1);
    if(!p4){c
        while(1){
            sem_wait(C);
            printf("I am p4\n");
            sleep(1);
            sem_post(A);
        }
    }
    return 0;
}

```

测试结果如下：

```

emrick@ubuntu: ~/Desktop/lab3
lab3.1.c:(.text+0x150): undefined reference to `sem_post'
lab3.1.c:(.text+0x175): undefined reference to `sem_wait'
lab3.1.c:(.text+0x198): undefined reference to `sem_post'
collect2: error: ld returned 1 exit status
emrick@ubuntu:~/Desktop/lab3$ ^C
emrick@ubuntu:~/Desktop/lab3$ gcc lab3.1.c -o lab3.1 -lpthread
emrick@ubuntu:~/Desktop/lab3$ ./lab3.1
emrick@ubuntu:~/Desktop/lab3$ I am p1
I am p3
I am p4
I am p1
I am p2
I am p4
I am p1
I am p3
I am p4
I am p1
I am p2
I am p4
I am p1
I am p3
I am p4
I am p1
I am p2

```

可以看到程序的每个循环的输出都是按照1、2、4或者1、3、4的顺序进行，与预期的相同。

**2、实验二：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。)**

实验过程：

在这里面主要有两件事情需要注意，一是需要做到 **无票不售，满票不退**，二是对全局变量的修改进行保护。

有同步机制的实验代码如下：

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

sem_t* ts_r = NULL;//1000剩余票数信号量
sem_t* ts_s = NULL;//0已售票数信号量
sem_t* a = NULL;//1, 对全局变量修改的保护
sem_t* b = NULL;//1, 同上
int ts_rr = 1000;//全局变量：剩余票数
int ts_ss = 0;//已售票数

void* worker1(void *arg)//出售车票进程
{
while(1){
    sem_wait(ts_r);
    sem_wait(a);
    ts_rr--;
    sem_post(a);
    printf("after sell tickets remaining : %d\n",ts_rr);
    sem_wait(b);
    ts_ss++;
    sem_post(b);
    printf("after sell tickets sold : %d\n",ts_ss);
    sem_post(ts_s);
    sleep(0.54);
}

}

void* worker2(void *arg)//退车票进程
{
while(1){
    sem_wait(ts_s);
    sem_wait(a);
    ts_rr++;
    sem_post(a);
    printf("after return tickets remaining : %d\n",ts_rr);
    sem_wait(b);
    ts_ss--;
    printf("after return tickets sold : %d\n",ts_ss);
    sem_post(b);
    sem_post(ts_r);
    sleep(0.56);//设置延时让票售出速度偏快，方便观察
}
```

```

}

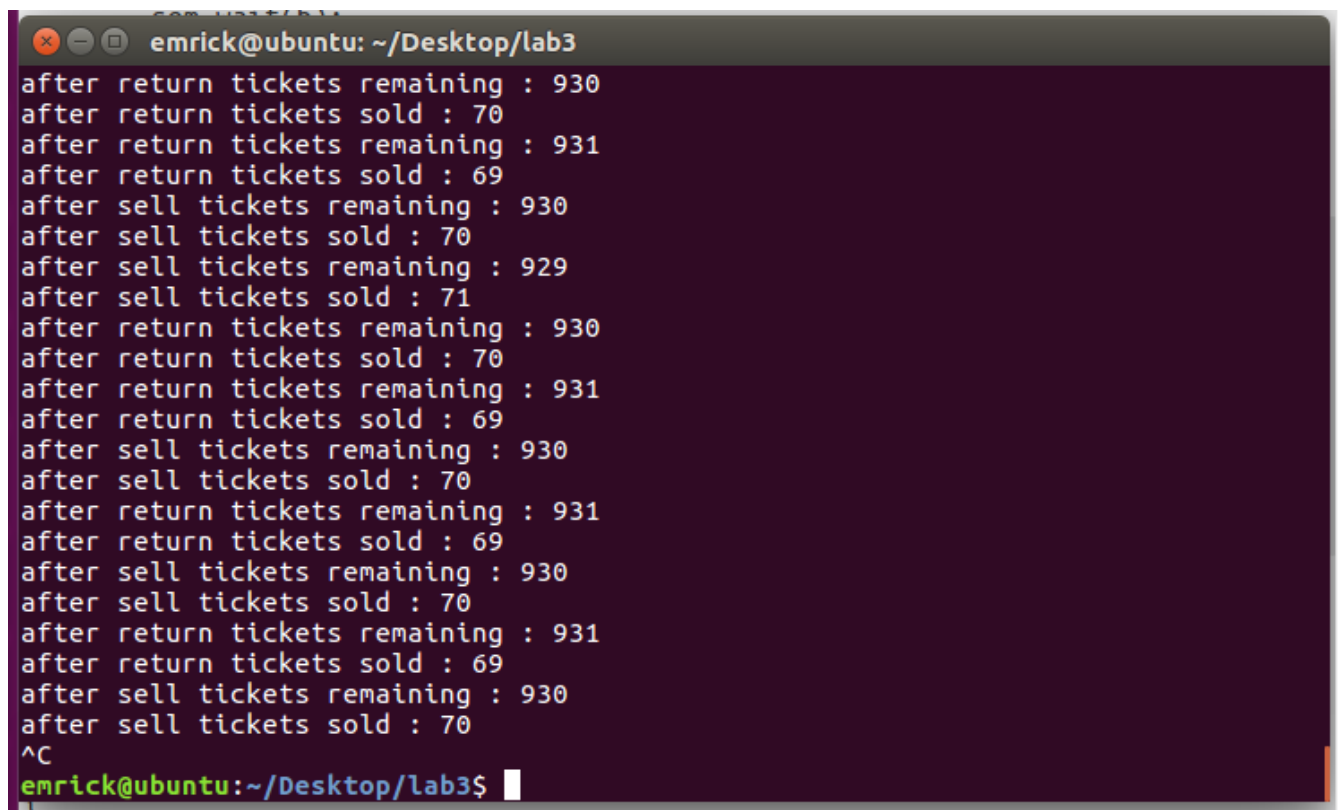
}

int main(){

    ts_r = sem_open("tsr", O_CREAT, 0666, 1000);
    ts_s = sem_open("tss", O_CREAT, 0666, 0);
    a = sem_open("aa", O_CREAT, 0666, 1);
    b = sem_open("bb", O_CREAT, 0666, 1);
    pthread_t p1,p2;
    pthread_create(&p1,NULL,worker1,NULL);
    pthread_create(&p2,NULL,worker2,NULL);
    pthread_join(p1,NULL);
    pthread_join(p2,NULL);
    sem_close(ts_r);
    sem_close(ts_s);
    sem_close(a);
    sem_close(b);
    sem_unlink("tsr");
    sem_unlink("tss");
    sem_unlink("aa");
    sem_unlink("bb");
    return 0;
}

```

有同步机制的运行结果如下图：



```

emrick@ubuntu: ~/Desktop/lab3
after return tickets remaining : 930
after return tickets sold : 70
after return tickets remaining : 931
after return tickets sold : 69
after sell tickets remaining : 930
after sell tickets sold : 70
after sell tickets remaining : 929
after sell tickets sold : 71
after return tickets remaining : 930
after return tickets sold : 70
after return tickets remaining : 931
after return tickets sold : 69
after sell tickets remaining : 930
after sell tickets sold : 70
after return tickets remaining : 931
after return tickets sold : 69
after sell tickets remaining : 930
after sell tickets sold : 70
after return tickets remaining : 931
after return tickets sold : 69
after sell tickets remaining : 930
after sell tickets sold : 70
^C
emrick@ubuntu:~/Desktop/lab3$

```

可以看到总票数、售出票数、剩余票数都一直保持正确的状态：售出+剩余=总票数（1000）；

然后将信号量部分代码进行注释并且根据文档提示加入了pthread\_yield()函数：

```

void* worker1(void *arg){//售票
while(1){
    sem_wait(ts_r);
    //sem_wait(a);
    ts_rr--;
    //sem_post(a);
    printf("after sell tickets remaining : %d\n",ts_rr);
    //sem_wait(b);
    ts_ss++;
    //sem_post(b);
    printf("after sell tickets sold : %d\n",ts_ss);
    sem_post(ts_s);
    sleep(0.54);
}

}

void* worker2(void *arg){//退票
while(1){
    sem_wait(ts_s);
    //sem_wait(a);
    int temp = ts_rr;
    pthread_yield();
    temp++;
    pthread_yield();
    ts_rr = temp;
    //sem_post(a);
    printf("after return tickets remaining : %d\n",ts_rr);
    //sem_wait(b);
    ts_ss--;
    printf("after return tickets sold : %d\n",ts_ss);
    //sem_post(b);
    sem_post(ts_r);
    sleep(0.56);
}
}

```

再次运行：

```
emrick@ubuntu: ~/Desktop/lab3
fter return tickets remaining : 7563
fter return tickets sold : 160
fter sell tickets remaining : 7562
fter sell tickets sold : 161
fter return tickets remaining : 7564
fter return tickets sold : 160
fter sell tickets remaining : 7563
fter sell tickets sold : 161
fter return tickets remaining : 7565
fter return tickets sold : 160
fter sell tickets remaining : 7564
fter sell tickets sold : 161
fter return tickets remaining : 7566
fter return tickets sold : 160
fter sell tickets remaining : 7565
fter sell tickets sold : 161
fter return tickets remaining : 7567
fter return tickets sold : 160
fter sell tickets remaining : 7566
fter sell tickets sold : 161
fter return tickets remaining : 7568
fter return tickets sold : 160
C
mricks@ubuntu:~/Desktop/lab3$
t(ts_s);
```

可以看到这时车票售出的票数很快就出现了错误。这是由于对全局变量的修改没有加以保护导致两个进程一直在读脏数据导致的结果。

**3、实验三：一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。

实验过程：

未添加同步机制时：

```
emrick@ubuntu: ~/Desktop/lab3
q
q
q
qoutput:q
q
q
q
q
q
qoutput:q
a
output:q
a
a
a
a
a
a
a
a
a
a
aoutput:a
output:a
^Z
```

开始迅速输入8个q然后迅速输入一堆a，结果q只输出了三个就被后面的输入的a重新占据了内存。开始输出a。

**添加同步机制：**

代码：

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

sem_t* buf_space = NULL;
sem_t* buf_data = NULL;
char buf[10];

void*worker1(void *arg){
    int i = 0;
    while(1){
        sem_wait(buf_space);
```



```

        scanf("%c",&buf[i]);
        i = (i+1)%10;
        sem_post(buf_data);
    }
    return NULL;
}

void*worker2(void *arg){
    int j = 0;
    while(1){
        sem_wait(buf_data);
        sleep(1);
        if (buf[j%10]!='\n') {
            printf("output:%c\n",buf[j%10]); //吃掉回车
        }
        j=(j+1)%10;
        sem_post(buf_space);
    }
    return NULL;
}

int main(int argc, char *argv[]){
    buf_space = sem_open("mySemName_1", O_CREAT, 0666, 10);
    buf_data = sem_open("mySemName_2", O_CREAT, 0666, 0);
    pthread_t p1,p2;
    printf("Input str length must less than 10.\n");
    pthread_create(&p1,NULL,worker1,NULL);
    pthread_create(&p2,NULL,worker2,NULL);
    pthread_join(p1,NULL);
    pthread_join(p2,NULL);
    sem_close(buf_space);
    sem_close(buf_data);
    sem_unlink("mySemName_1");
    sem_unlink("mySemName_2");
    return 0;
}

```

运行结果如下：

```
emrick@ubuntu: ~/Desktop/lab3
q
w
e
r
t
output:q
output:w
output:e
output:r
output:t
a
s
d
foutput:a
output:s
output:d

output:f
```

这次输的速度过快的话会发生阻塞，直到每次输出一行释放一个内存信号量才能再次输入，输入输出的结果顺序总是正确的。

#### 4、实验四：

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？

实验过程：

不删互斥代码时：

```
emrick@ubuntu: ~/Desktop/lab3
Processing triggers for hicolor-icon-theme (0.15-0ubuntu1.1) ...
Setting up unity-tweak-tool (0.0.7ubuntu2) ...
emrick@ubuntu:~$ cd Desktop/lab3
emrick@ubuntu:~/Desktop/lab3$ gcc sender
sender sender.c
emrick@ubuntu:~/Desktop/lab3$ gcc sender.c -o sender -pthread
emrick@ubuntu:~/Desktop/lab3$ gcc receiver.c.c -o receiver -pthread
gcc: error: receiver.c.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
emrick@ubuntu:~/Desktop/lab3$ gcc receiver.c -o receiver -pthread
emrick@ubuntu:~/Desktop/lab3$ ./sender

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

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

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

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

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

emrick@ubuntu:~/Desktop/lab3$ ./receiver

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

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

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

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

可以看到接受进程可以正确的接收到发送进程发送的信息。

### 对互斥代码进行注释:

```
while(1)
{
    //在这里将互斥代码进行注释
    //if(1 == (value = semctl(sem_id, 0, GETVAL)))
    if(1)
    {
        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;
}
}

```

再次运行：

```

emrick@ubuntu: ~/Desktop/lab3
Now, receive message process running:
The message is : 90
Now, receive message process running:
The message is : 90
Now, receive message process running:
The message is : 90
Now, receive message process running:
The message is : 90
Now, receive message process running:
The message is : 90
□

emrick@ubuntu: ~/Desktop/lab3
Input the snd message: 5
Now, snd message process running:
Input the snd message: 6
Now, snd message process running:
Input the snd message: 7
Now, snd message process running:
Input the snd message: 8
Now, snd message process running:
Input the snd message: 90
Now, snd message process running:
Input the snd message: □

```

可以看到没了互斥代码后，接收端不管发送端是否发送信息，总会读取最后发送端发送的信息，这就出现了错误。

添加打印地址代码：

```
printf("memory address is %x\n",shm_ptr);
```

运行：

```
emrick@ubuntu: ~/Desktop/lab3
Sender memory address is b4cf5000

Now, snd message process running:
    Input the snd message: 43
Sender memory address is b4cf5000

Now, snd message process running:
    Input the snd message: 45
Sender memory address is b4cf5000

Now, snd message process running:
    Input the snd message: 56
Sender memory address is b4cf5000

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

emrick@ubuntu: ~/Desktop/lab3

Now, receive message process running:
    The message is : 3
Receiver memory address is acde4000

Now, receive message process running:
    The message is : 43
Receiver memory address is acde4000

Now, receive message process running:
    The message is : 45
Receiver memory address is acde4000

Now, receive message process running:
    The message is : 56

```

可以看到接收端和发送端的共享内存的地址并不同，结合所学知识猜测，这里面的地址并不是实际的物理地址，应该是进程分配的虚拟地址，通过各自的映射关系映射到同一块物理地址上，达到共享内存的作用。

**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？**

对**无名管道**样例代码进行修改：增加了循环和父子进程（方面观察是否同步）

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

```

#include <assert.h>
#include <unistd.h>

int main()
{
    int fd[2];
    pipe(fd); // fd[0]是读, fd[1]是写
    while(1){
        if(fork() != 0)
        {
            close(fd[0]);
            write(fd[1], "helloworld", 10);
            printf("parent have writen msg:helloworld\n");
        }
        else
        {
            close(fd[1]);
            char buf[128] = {0};
            read(fd[0], buf, 127);
            printf("child have receive msg:");
            printf("%s\n", buf);
        }
        sleep(3);
    }
    return 0;
}

```

运行结果:

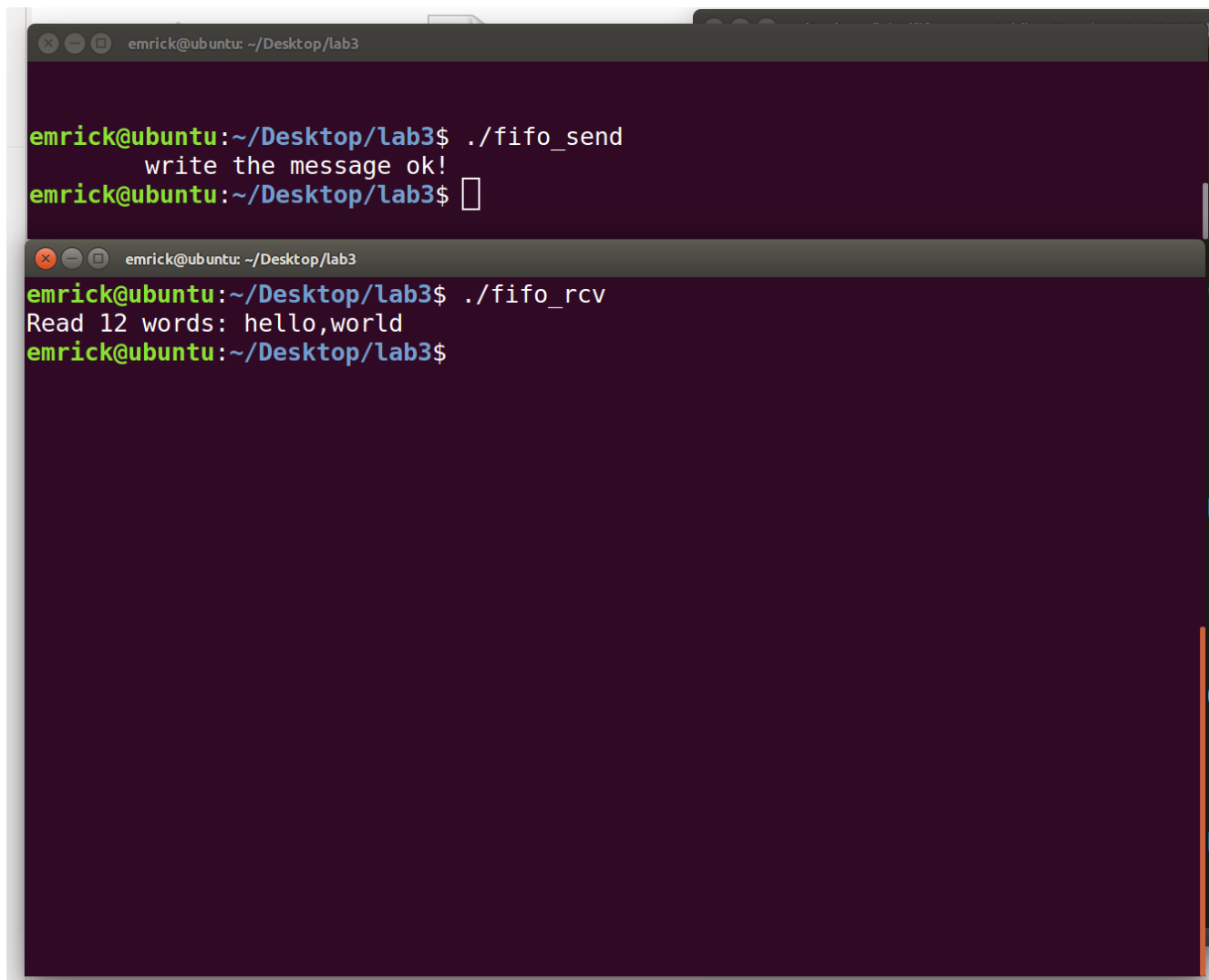
```

emrick@ubuntu: ~/Desktop/lab3$ ./pipe
parent have writen msg:helloworld
child have receive msg:helloworld
parent have writen msg:helloworld
child have receive msg:
parent have writen msg:helloworld
child have receive msg:helloworld
parent have writen msg:helloworld
parent have writen msg:helloworld
child have receive msg:
child have receive msg:
parent have writen msg:helloworld
parent have writen msg:helloworld
child have receive msg:helloworld
child have receive msg:
^Z
[1]+  Stopped                  ./pipe
emrick@ubuntu: ~/Desktop/lab3$

```

可以看到并没有实现同步机制, 读的一方并不影响写的一方, 所以某一方速度较快时就会发生数据丢失。

有名管道运行样例代码:



The image shows two terminal windows from an Ubuntu system. The top window has a title bar 'emrick@ubuntu: ~/Desktop/lab3' and shows the command `./fifo_send` being executed. The prompt is `emrick@ubuntu:~/Desktop/lab3$`. The output is `write the message ok!` followed by a blank line. The bottom window also has a title bar 'emrick@ubuntu: ~/Desktop/lab3' and shows the command `./fifo_rcv` being executed. The prompt is `emrick@ubuntu:~/Desktop/lab3$`. The output is `Read 12 words: hello,world` followed by a blank line.

```
emrick@ubuntu:~/Desktop/lab3$ ./fifo_send
write the message ok!
emrick@ubuntu:~/Desktop/lab3$

emrick@ubuntu:~/Desktop/lab3$ ./fifo_rcv
Read 12 words: hello,world
emrick@ubuntu:~/Desktop/lab3$
```

通过不断的运行可以看出，有名管道的发送和接受是同步的，发送端不发送的时候接收端不会接收数据，接收端不接受的时候发送端不能继续发送。

阻塞的情况也是如此，当发送端不发送时接收端阻塞等待，接收端不接收时发送端阻塞等待。

当单独打开其中的一个进程时进程会阻塞等待另一个进程运行。说明以上想法正确

**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？**

直接运行代码：

```
emrick@ubuntu: ~/Desktop/lab3

Now, snd message process running:
Input the snd message: ^Z
[3]+ Stopped ./sender3
emrick@ubuntu:~/Desktop/lab3$ gcc sever.c -o sever -pthreadgcc: error: seve
r.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
emrick@ubuntu:~/Desktop/lab3$ gcc server.c -o server -pthread
emrick@ubuntu:~/Desktop/lab3$ ./server
Process (./server) is started, pid=4686
Receive msg: hello
Receive msg: hei
Receive msg: haha
Receive msg: time
[]

emrick@ubuntu: ~/Desktop/lab3

emrick@ubuntu:~/Desktop/lab3$ ./Client
The process(./Client),pid=4687 started~

Input snd mesg: hello
Server said: hello

Input snd mesg: hei
Server said: hei

Input snd mesg: haha
Server said: haha

Input snd mesg: time
Server said: time

Input snd mesg: []
```

在运行过程中可以看到已经实现了同步机制。

观察实验代码中有一条代码：

```
msgrcv(msgId, &rcvBuf, BUF_SIZE, 1,0);
```

这条代码是实现同步的关键。

我们将改变语句中的参数：

```
msgrcv(msgId, &rcvBuf, BUF_SIZE, 1,IPC_NOWAIT);
```

重新运行代码：



```
emrick@ubuntu: ~/Desktop/lab3
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
testReceive msg:
^C
emrick@ubuntu:~/Desktop/lab3$
```

可以看到这个时候就不再同步了，实际上最后一个参数决定了进程是否需要阻塞等待

有新的消息送入缓冲区，如果去掉之后就不会进行阻塞等待。

通过上面的实验我们可以看出来，发送者在消息队列满的时候会发生阻塞，接受者在队列中没有合适的消息时会发生阻塞。

## 实验五: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。

自己看实在看不下去，参考了博客：

[https://www.cnblogs.com/laiy/p/pintos\\_project1\\_thread.html](https://www.cnblogs.com/laiy/p/pintos_project1_thread.html)

这里相当于做了一些摘录。

我们先来看一下devices目录下timer.c中的timer\_sleep实现：

```
/* Sleeps for approximately TICKS timer ticks. Interrupts must
   be turned on. */
void timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();
    ASSERT (intr_get_level () == INTR_ON);
    while (timer_elapsed (start) < ticks)
        thread_yield ();
}
```

下面开始逐行分析这个函数，第6行的timer\_ticks函数也在timer.c文件中，跳转到该函数中：

让我们一行一行解析：

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又做了什么鬼。

```

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 Hardware
10     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)。

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 and
2    returns the previous interrupt status. */
3 enum intr_level
4 intr_set_level (enum intr_level level)
5 {
6     return level == INTR_ON ? intr_enable () : intr_disable ();
7 }
```

这个函数描述的是: 如果允许中断的 (INTR\_ON) 则enable否则就disable.

```
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 Maskable
11    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。

再来看intr\_context

```

1 /* Returns true during processing of an external interrupt
2    and false at all other times. */
3 bool
4 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, which
2    should be a value once returned by timer_ticks(). */
3 int64_t
4 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 }

```

和

```

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 }

```

操作系统是怎么设计page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(PGSHIFT, 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 x86
4     hardware page tables. */
5
6 #define BITMASK(SHIFT, CNT) (((1ul << (CNT)) - 1) << (SHIFT))

```

一个页面12位，而struct thread是在一个页面的最开始的地方，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一个元素。

```

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 %ebx
25     pushl %ebp
26     pushl %esi
27     pushl %edi
28
29     # Get offsetof (struct thread, stack).
30 .globl thread_stack_ofs
31     mov thread_stack_ofs, %edx
32
33     # Save current stack pointer to old thread's stack, if any.
34     movl SWITCH_CUR(%esp), %eax
35     movl %esp, (%eax,%edx,1)
36
37     # Restore stack pointer from new thread's stack.
38     movl SWITCH_NEXT(%esp), %ecx
39     movl (%ecx,%edx,1), %esp
40
41     # Restore caller's register state.
42     popl %edi
43     popl %esi
44     popl %ebp
45     popl %ebx
46     ret
47 .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。现在eax(函数返回值是eax)就是被切换的线程栈指针。因此 schedule先把当前线程丢到就绪队列， 然后如果下一个线程和当前线程不一样的话切换线程。