**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;

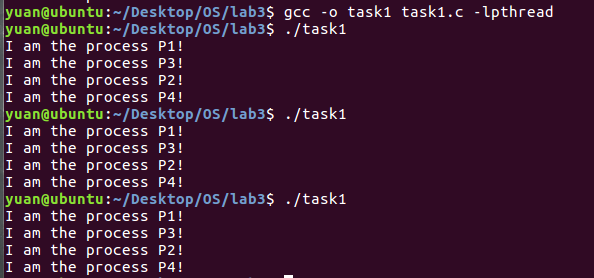
}

**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。

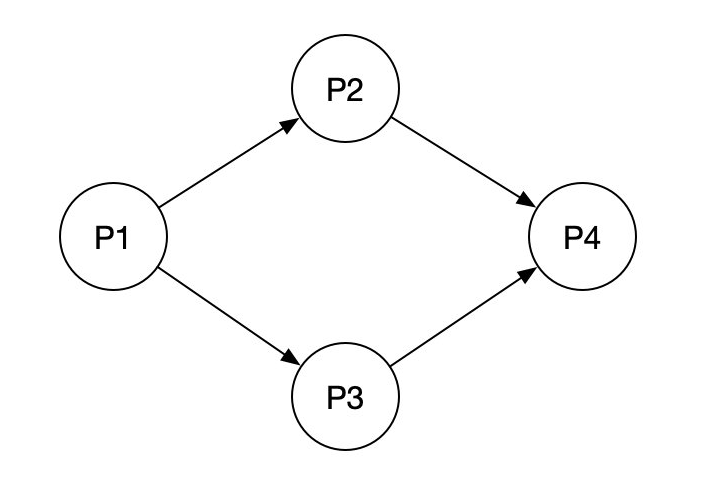
**实验结果：**



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：

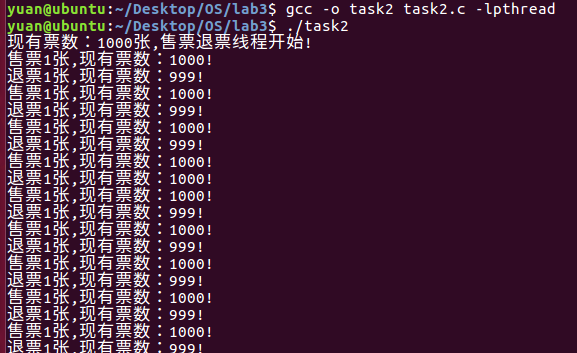
带有“//同步操作”的程序代码行为实现同步机制所添加，在不想要同步前应注释掉；

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。

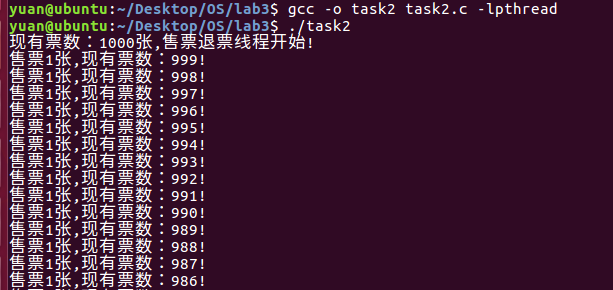
通过设置信号量mySem用来实现同步机制，即不能同时对余票量ticketCount进行写操作。

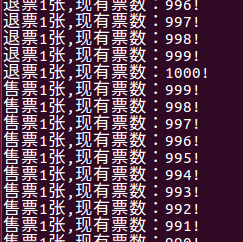
**实验结果：**

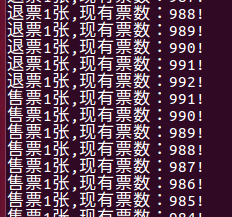
**添加同步前：**



**添加同步后：**







**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;

}

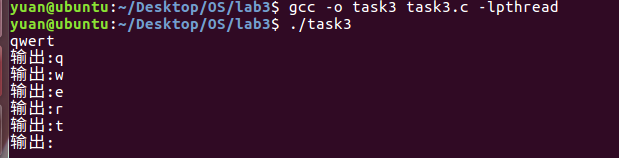
**代码解析：**

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；

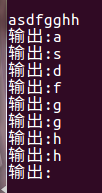
需注意：在没有数据以前，不允许读出，worker2()函数阻塞等待；在数据存满以后，不允许读入，worker1()函数阻塞等待。

**实验结果：**

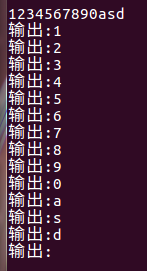
第一次测试：



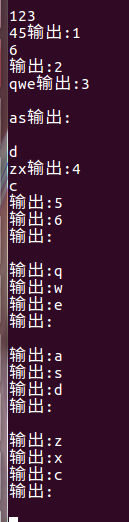
第二次测试：



第三次测试（输入数据大于10）：



第四次测试（连续输入）：



**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：**

1. 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;

}

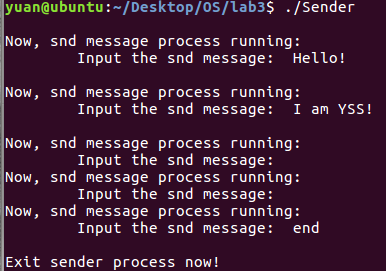
**代码执行：**

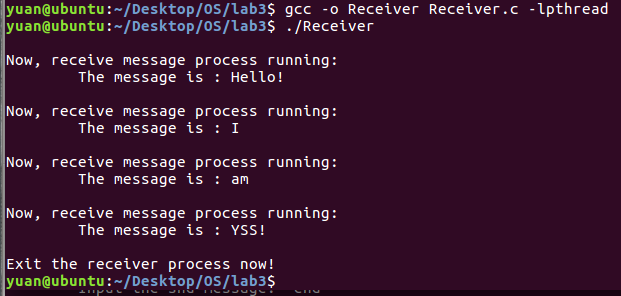
打开两个终端，分别执行编译命令：

gcc -o Sender Sender.c -lpthread

gcc -o Receiver Receiver.c -lpthread

**实验结果：**



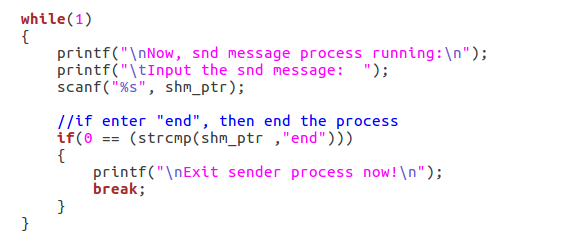


**实验结果解析：**

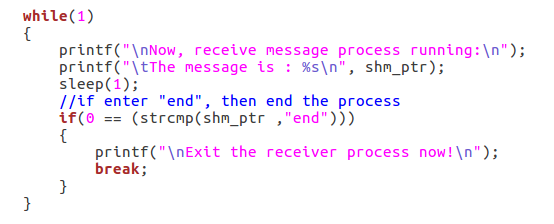
可见Sender进程发出的消息Receiver进程准确无误的接收；但Sender读入信息时用的是scanf("%s", shm\_ptr)机制，遇到空格自动结束，则出现上述截图效果。

**删除互斥的代码：**

代码变化处仅以下两个地方：

**Sender.c**

**Receiver.c**



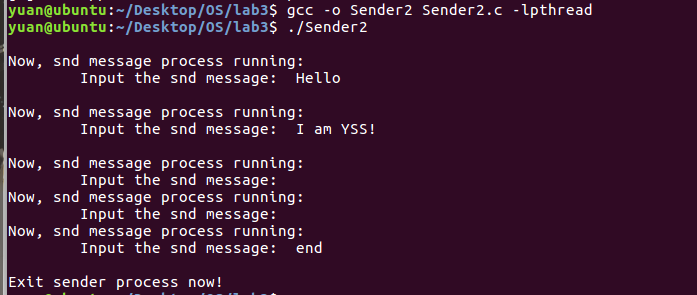
**代码执行：**

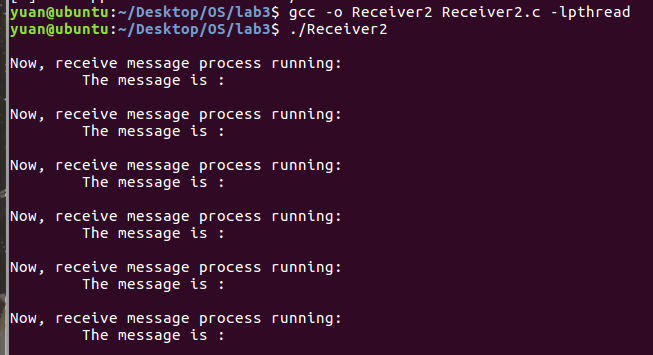
打开两个终端，分别执行编译命令：

gcc -o Sender2 Sender2.c -lpthread

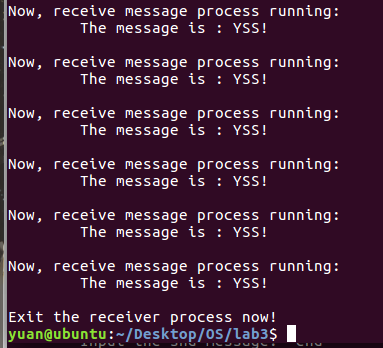
gcc -o Receiver2 Receiver2.c -lpthread

**实验结果：**









**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：**

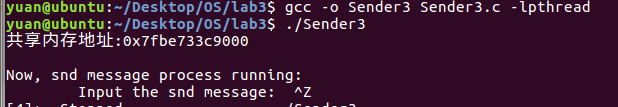
**代码变动：**

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：**





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";

//`. 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));

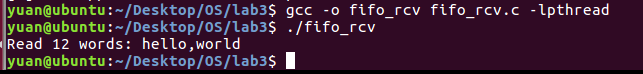
printf("Read %d words: %s\n", num, buf);

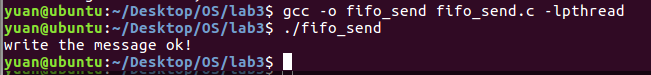
close(fifo\_fd);

return 0;

}

**实验结果：**





**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));

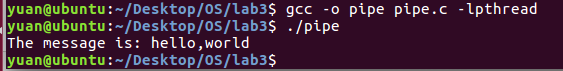
read(fd[0], buf, 12);

printf("The message is: %s\n", buf);

return 0;

}

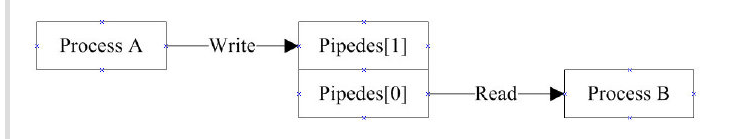
**实验结果：**



**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 strcut (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 msgId;

printf("THe process(%s),pid=%d started~\n", argv[0], getpid());

key = ftok(".", 0xFF);

msgId = msgget(key, IPC\_CREAT|0644);

if(-1 == msgId)

{

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 messge 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 mesg: ");

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 strcut (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 msgId;

key = ftok(".", 0xFF);

msgId = msgget(key, IPC\_CREAT|0644);

if(-1 == msgId)

{

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(msgId, &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(msgId, &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。

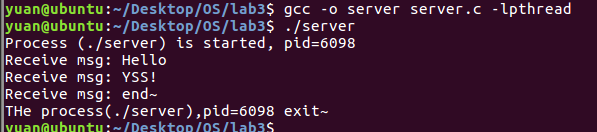
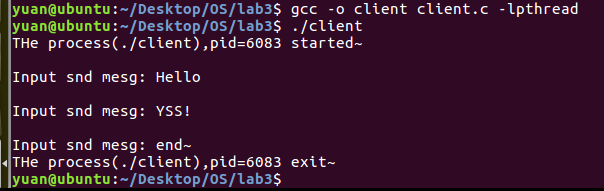
**client.c:**

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：**



**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\_t3 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棧上，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。

再来看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 may11 have overflowed its stack. Each thread has less than 4 kB12 of stack, so a few big automatic arrays or moderate13 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 is10 somewhere in the middle, this locates the curent 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 bool3 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棧的指针复制到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(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 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， 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 else12 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 in3 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 and10 #### restore the registers. As part of switching stacks we record the11 #### current stack pointer in CUR's thread structure.12 13 .globl switch\_threads14 .func switch\_threads15 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. See20 # [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记录线程和棧之间的间隙， 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棧指针， 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)就是被切换的线程棧指针。

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的一个棧指针指向了当前线程棧的尾部， 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， 最后是这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 NDEBUG22 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palloc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 lock 3 {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 semaphore 3 {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palloc\_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 bool 4 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palloc\_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 bool4 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 bool3 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 to10 BIT\_IDX is turned on. \*/11 static inline elem\_type12 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 else10 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. See10 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 void16 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 it22 is guaranteed to be atomic on a uniprocessor machine. See23 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>