Linux线程的创建与同步

1. 线程概念

1. 概念

线程：进程内部的一条执行路径或者序列

进程：一个正在执行的程序，是动态运作的

2. 多个线程、进程之间

多个线程共享进程的资源

多个进程各自拥有各自的资源

3. 例子

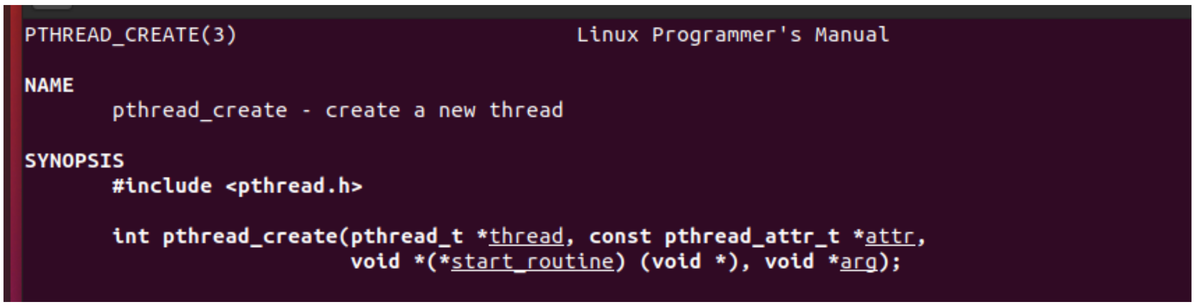
从main函数的第一行执行到最后一行，这就叫做一个线程

一个程序段里面两个函数，main函数执行和thread_fun函数就是一个进程的两个执行序列，即一个进程里面的两个线程

进程与线程是相辅相成的，不能完全将其分隔开，二者分不开，从调度执行看的是线程角度，从资源分隔方面看的是进程

可以形象的比做一场羽毛球，1班和2班去比赛就是进程之间，1班的张三和2班的李四进行比赛就是线程

4. 线程的创建



```
PTHREAD_CREATE(3)                          Linux Programmer's Manual

NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);
```

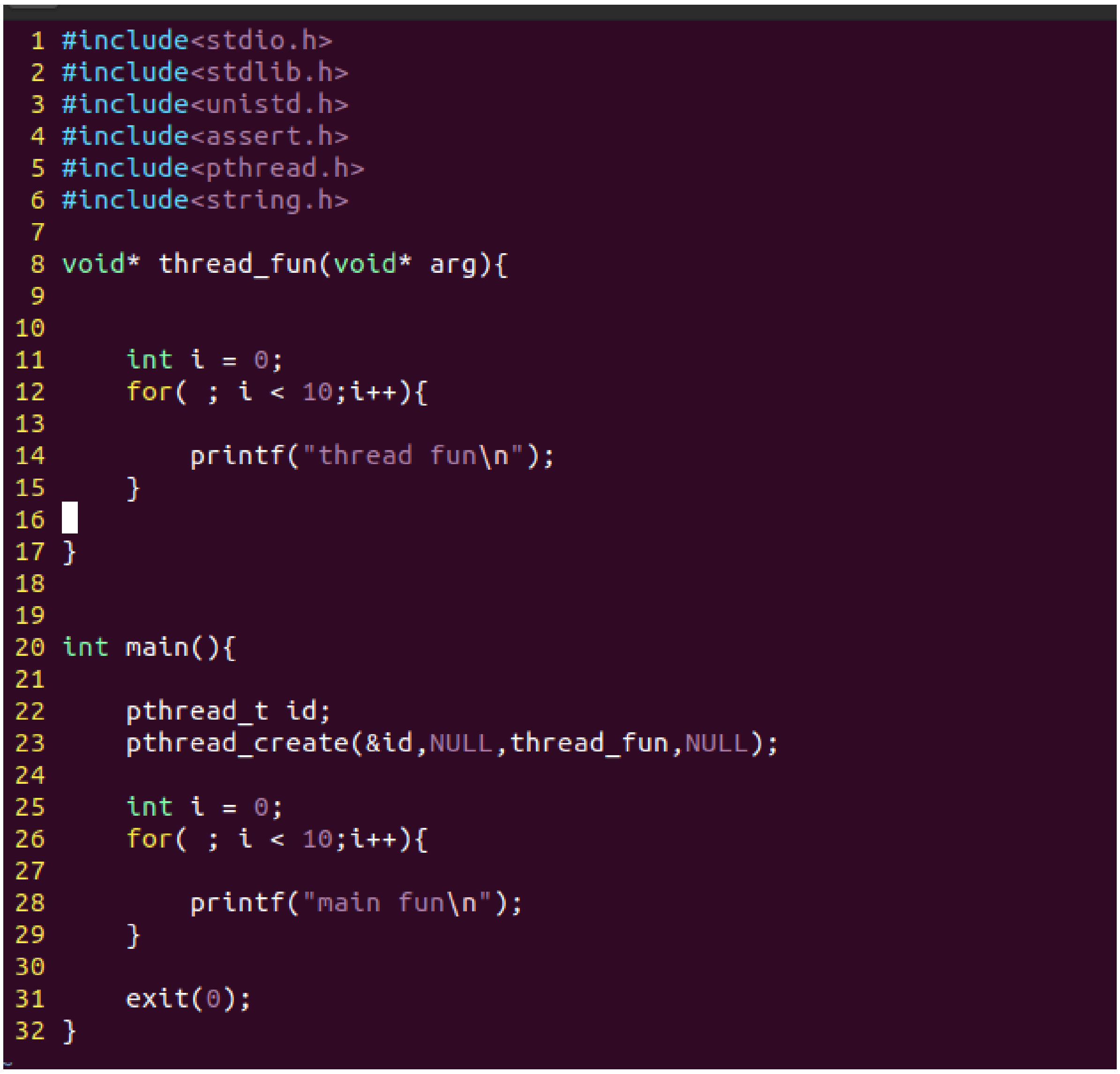
第一个参数指的是线程id，可以传入线程的id

第二个参数指的是线程的相关属性

第三个参数是一个函数指针

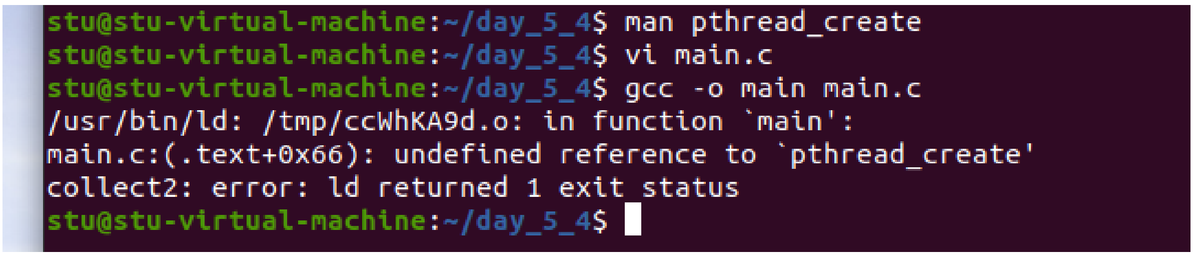
第四个参数是对第三个参数中的函数指针的参数

5. 代码示例



```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>

void* thread_fun(void* arg){


    int i = 0;
    for( ; i < 10;i++){

        printf("thread fun\n");
    }

}


int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    int i = 0;
    for( ; i < 10;i++){

        printf("main fun\n");
    }

    exit(0);
}
```

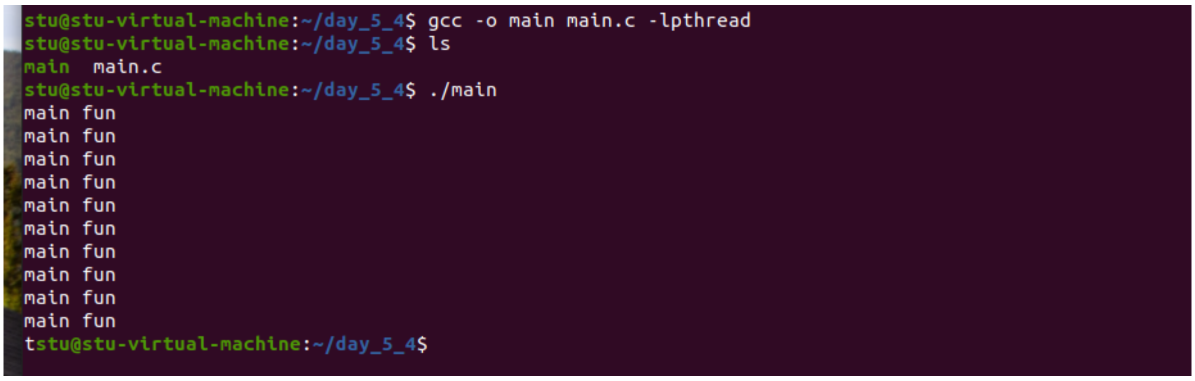
这地方如果直接编译链接就会报错，因为编译器默认从标准库里面找函数，但是这个库是在pthread库里面，需要在编译链接的时候指定库名



```
stu@stu-virtual-machine:~/day_5_4$ man pthread_create
stu@stu-virtual-machine:~/day_5_4$ vi main.c
stu@stu-virtual-machine:~/day_5_4$ gcc -o main main.c
/usr/bin/ld: /tmp/ccWhKA9d.o: in function `main':
main.c:(.text+0x66): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
stu@stu-virtual-machine:~/day_5_4$
```

就像上面这样，报错是链接阶段错误

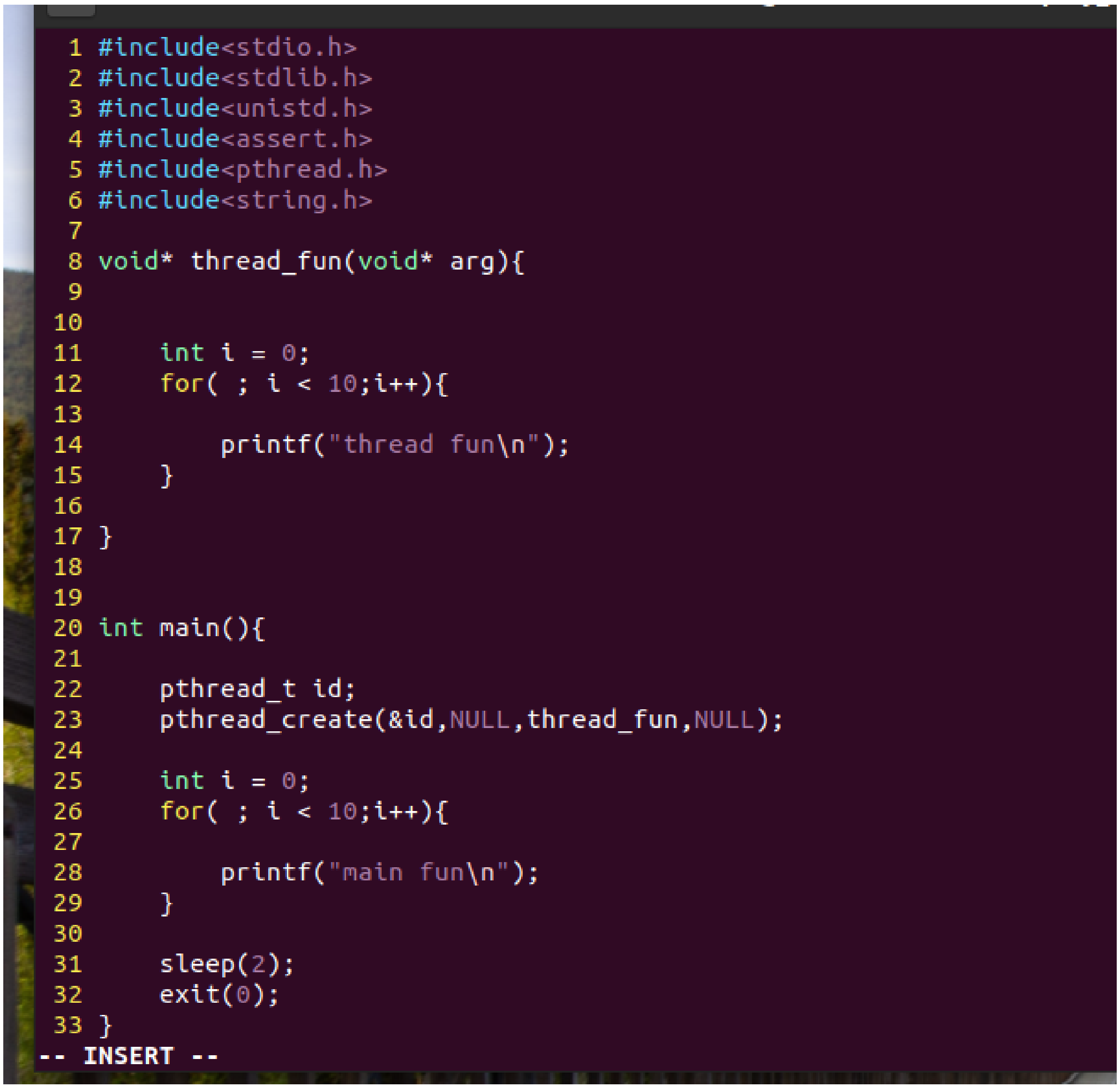
需要指定库名，如下面这样：



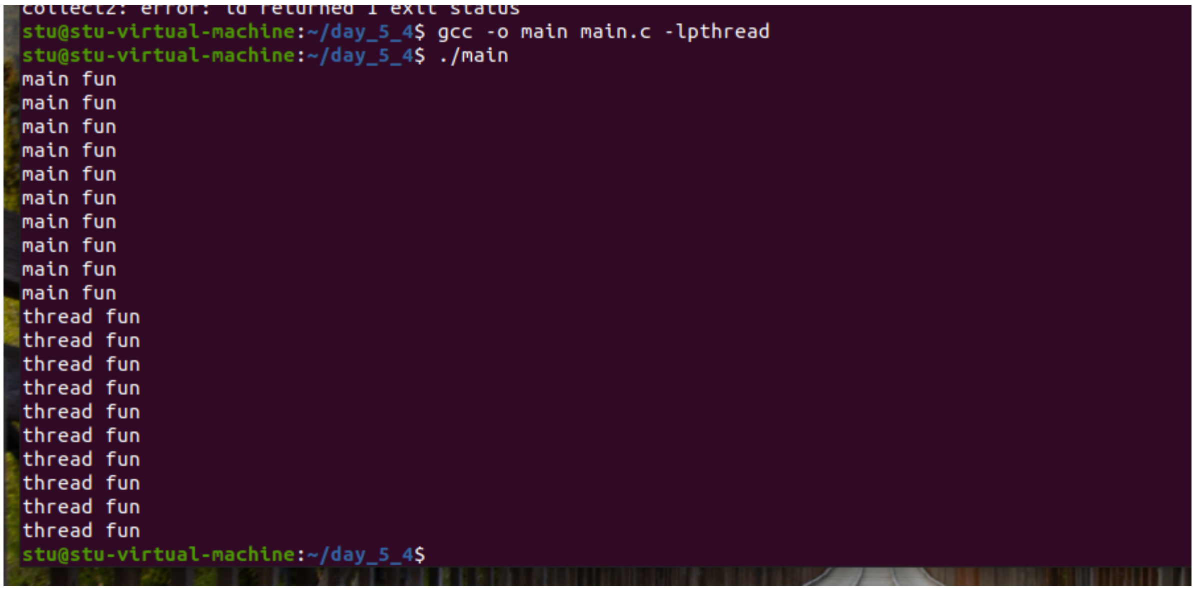
```
stu@stu-virtual-machine:~/day_5_4$ gcc -o main main.c -lpthread
stu@stu-virtual-machine:~/day_5_4$ ls
main  main.c
stu@stu-virtual-machine:~/day_5_4$ ./main
main fun
main fun
main fun
main fun
main fun
main fun
main fun
main fun
main fun
main fun
tstu@stu-virtual-machine:~/day_5_4$
```

为什么只看见了main?

因为创建线程需要一定时间，但是主线程运行时间很短，可能直接结束之后没有等到fun执行，就结束了整个进程，这个时候不管存不存在线程，都会全部销毁，为了看到两个都执行的效果，我们可以让主线程输出代码部分下面睡眠几秒，等待一下thread_fun线程

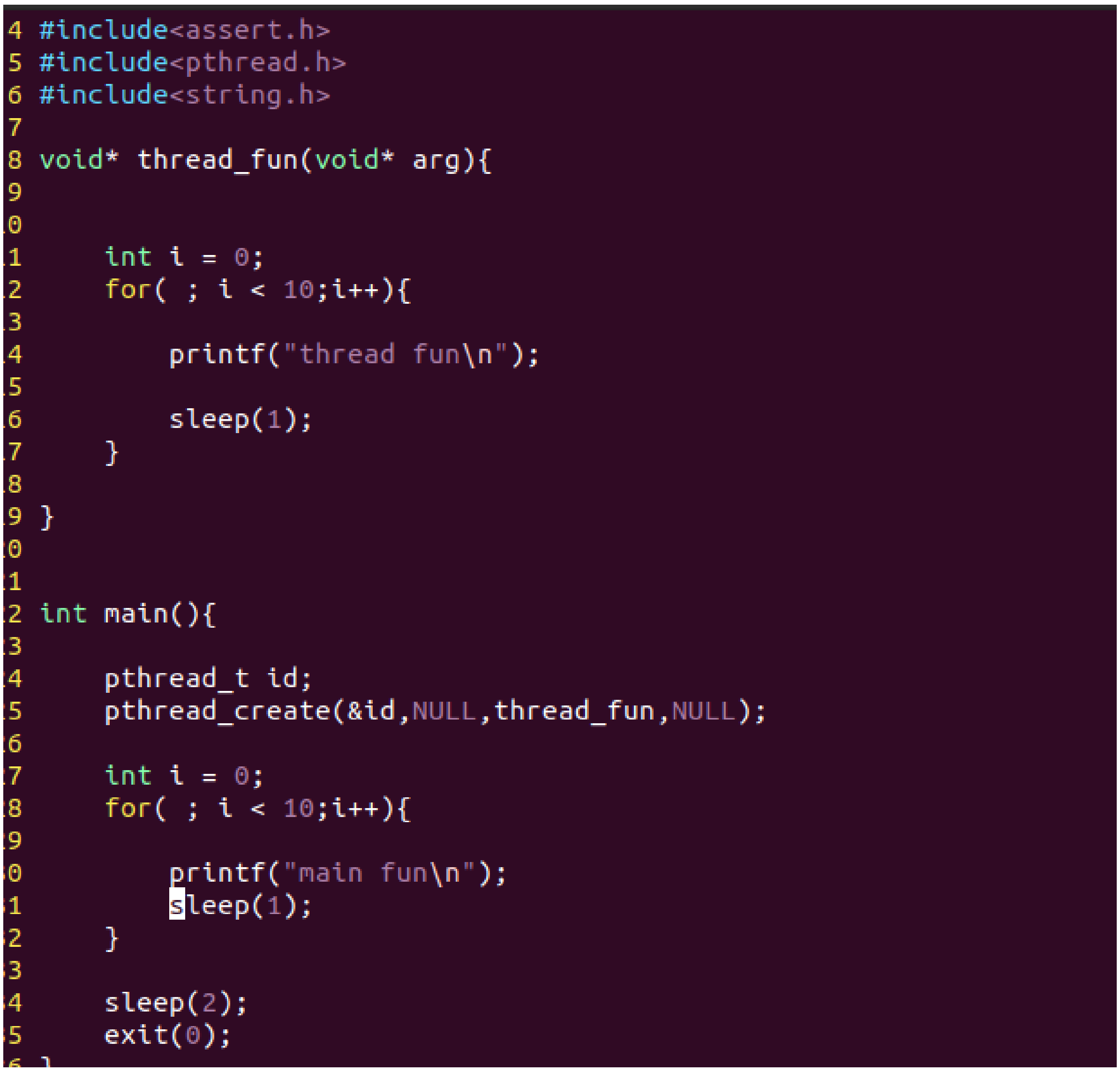


```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>

void* thread_fun(void* arg){


    int i = 0;
    for( ; i < 10;i++){

        printf("thread fun\n");
    }

}


int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    int i = 0;
    for( ; i < 10;i++){

        printf("main fun\n");
    }

    sleep(2);
    exit(0);
}
-- INSERT --
```

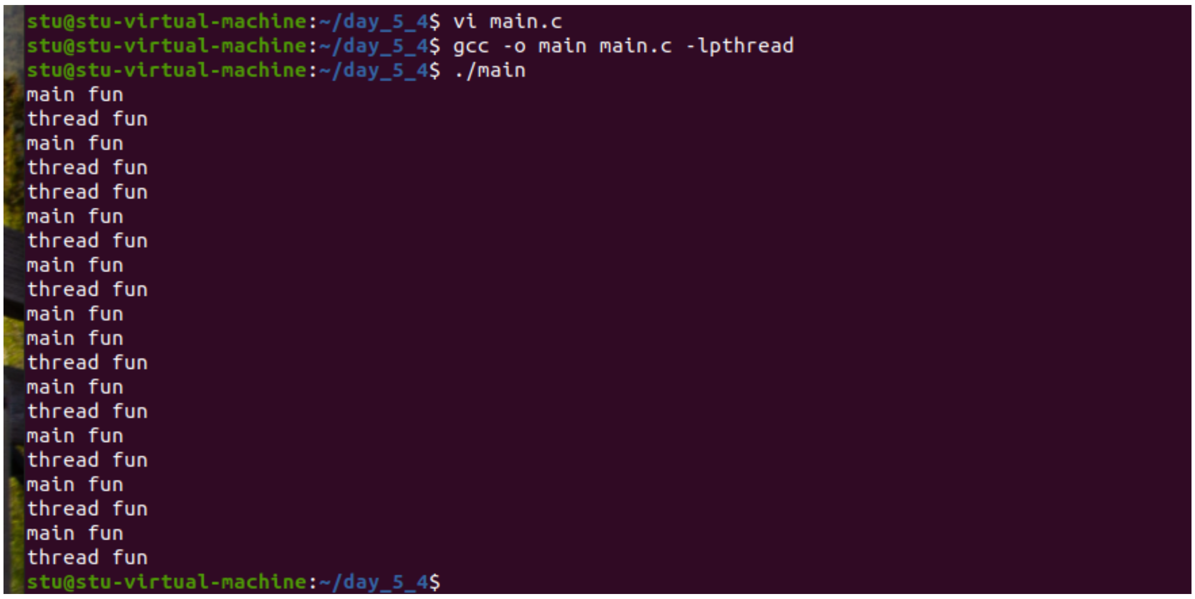


```
collect2: error: ld returned 1 exit status
stu@stu-virtual-machine:~/day_5_4$ gcc -o main main.c -lpthread
stu@stu-virtual-machine:~/day_5_4$ ./main
main fun
main fun
main fun
main fun
main fun
main fun
main fun
main fun
main fun
main fun
thread fun
thread fun
thread fun
thread fun
thread fun
thread fun
thread fun
thread fun
thread fun
thread fun
stu@stu-virtual-machine:~/day_5_4$
```

先打印一个main fun，再打印一个thread fun，让两个都等待，处理器交替运行，两个线程并发运行



```c
#include<assert.h>
#include<pthread.h>
#include<string.h>

void* thread_fun(void* arg){


    int i = 0;
    for( ; i < 10;i++){

        printf("thread fun\n");

        sleep(1);
    }

}


int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    int i = 0;
    for( ; i < 10;i++){

        printf("main fun\n");
        sleep(1);
    }

    sleep(2);
    exit(0);
```



```
stu@stu-virtual-machine:~/day_5_4$ vi main.c
stu@stu-virtual-machine:~/day_5_4$ gcc -o main main.c -lpthread
stu@stu-virtual-machine:~/day_5_4$ ./main
main fun
thread fun
main fun
thread fun
thread fun
main fun
thread fun
main fun
thread fun
main fun
main fun
thread fun
main fun
thread fun
main fun
thread fun
main fun
thread fun
main fun
thread fun
stu@stu-virtual-machine:~/day_5_4$
```

2. 线程的并发

1. 并发与并行

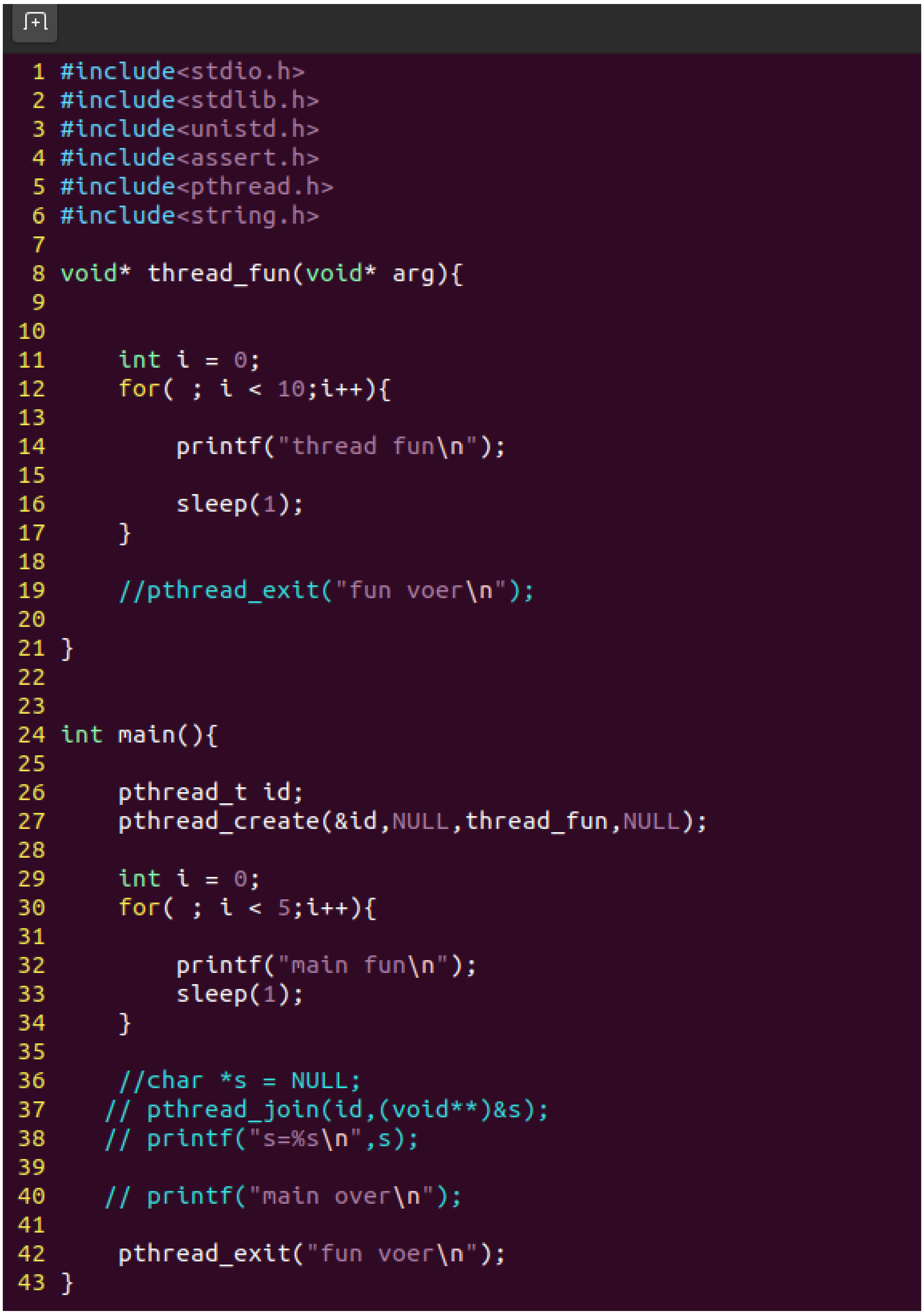
并发是某一段时间a与b两个进程交替运行

并行指的是某一段时间内，a与b两个进程同时运行，但是其实现需要硬件支持

2. 线程退出与等待

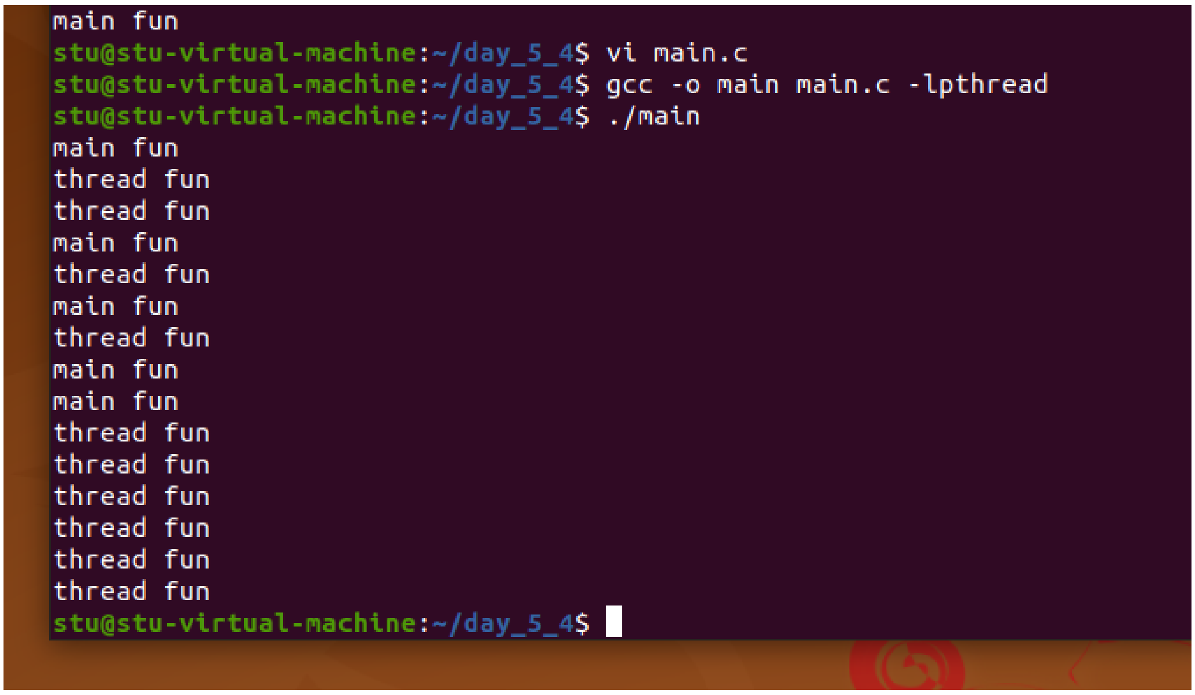
主线程执行结束之后，子线程不能继续执行了

所以这里面可以在主线程执行完毕之后，执行退出线程的操作，使得主线程结束，但是这个时候进程还未退出，子线程还可以执行



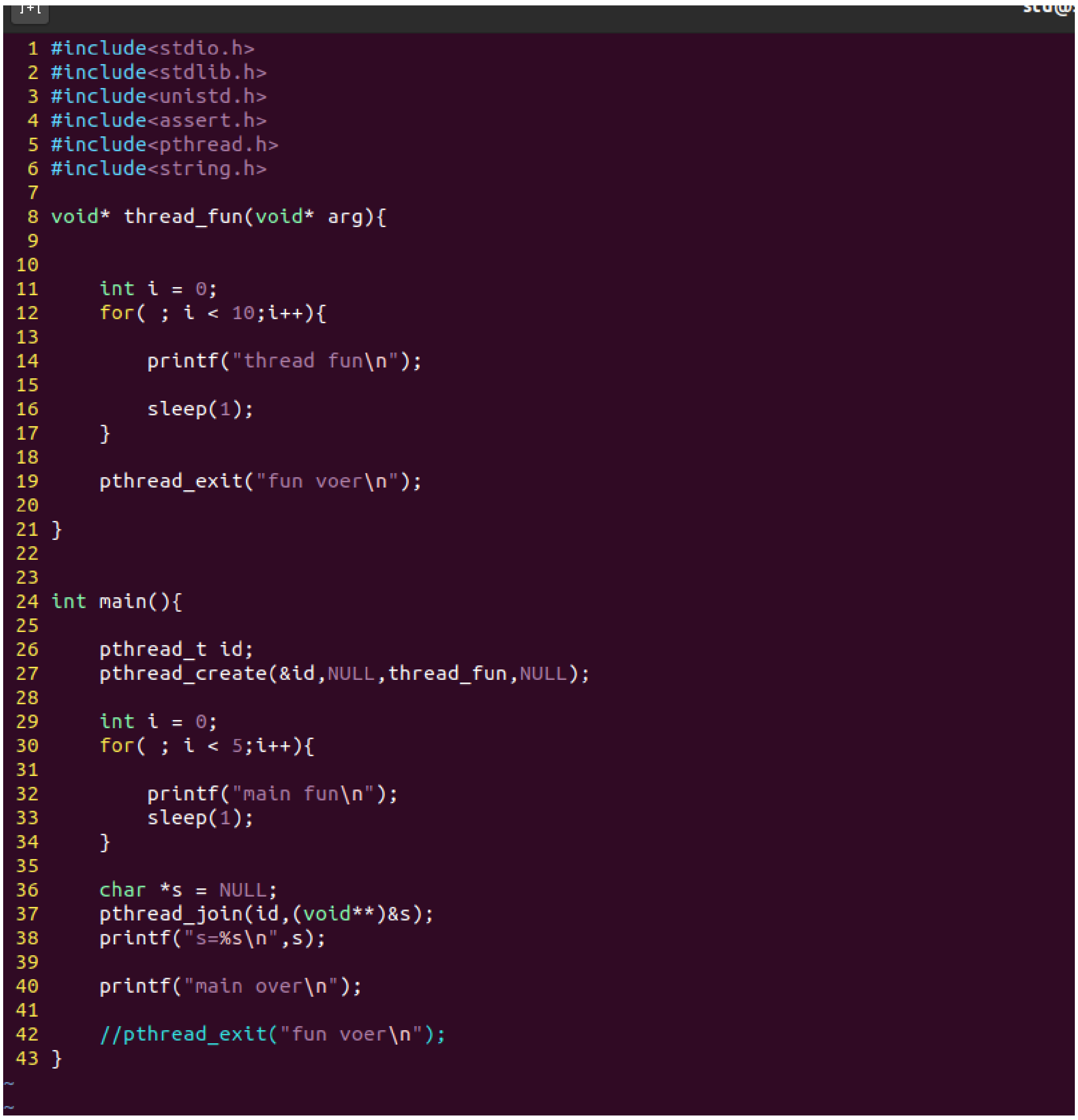
```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>

void* thread_fun(void* arg){


    int i = 0;
    for( ; i < 10;i++){

        printf("thread fun\n");

        sleep(1);
    }

    //pthread_exit("fun voer\n");

}


int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    int i = 0;
    for( ; i < 5;i++){

        printf("main fun\n");
        sleep(1);
    }

    //char *s = NULL;
   // pthread_join(id,(void**)&s);
   // printf("s=%s\n",s);

   // printf("main over\n");

    pthread_exit("fun voer\n");
}
```

执行之后，会有下面的效果



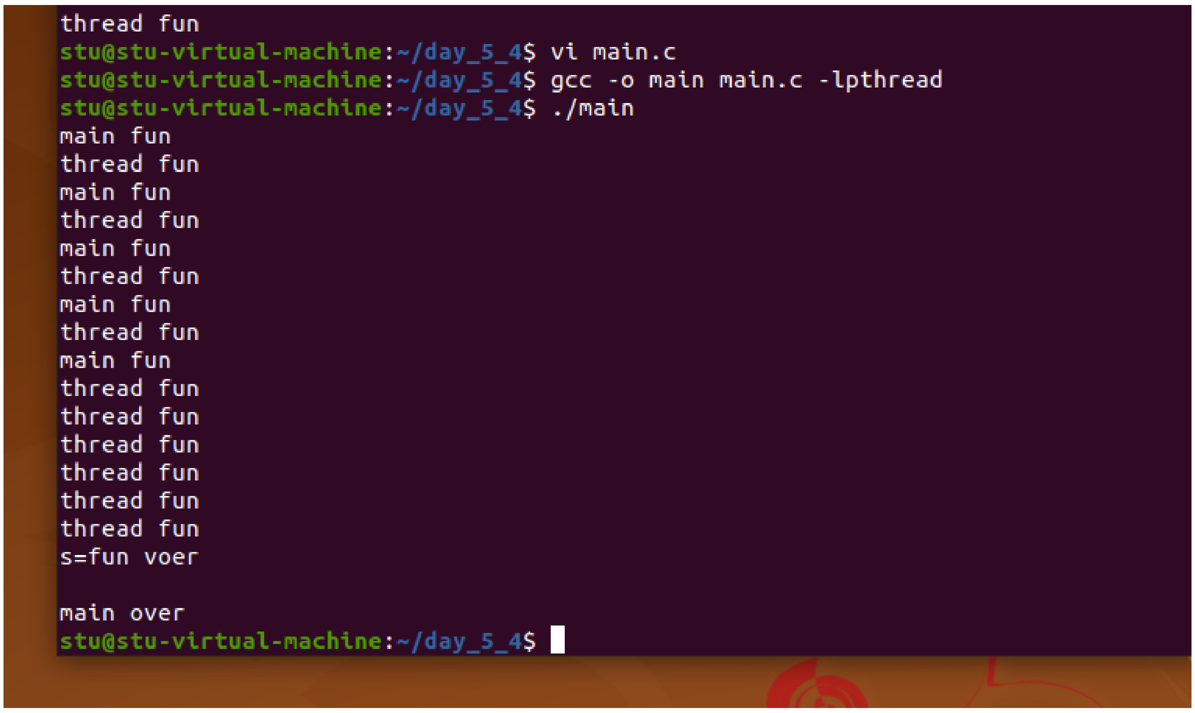
```
main fun
stu@stu-virtual-machine:~/day_5_4$ vi main.c
stu@stu-virtual-machine:~/day_5_4$ gcc -o main main.c -lpthread
stu@stu-virtual-machine:~/day_5_4$ ./main
main fun
thread fun
thread fun
main fun
thread fun
main fun
thread fun
main fun
main fun
thread fun
thread fun
thread fun
thread fun
thread fun
thread fun
stu@stu-virtual-machine:~/day_5_4$ 
```

但是我们一般的逻辑是所有子线程结束之后，主线程才结束，这里面调用线程等待函数，如下面所示：



```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>

void* thread_fun(void* arg){


    int i = 0;
    for( ; i < 10;i++){

        printf("thread fun\n");

        sleep(1);
    }

    pthread_exit("fun voer\n");

}


int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    int i = 0;
    for( ; i < 5;i++){

        printf("main fun\n");
        sleep(1);
    }

    char *s = NULL;
    pthread_join(id,(void**)&s);
    printf("s=%s\n",s);

    printf("main over\n");

    //pthread_exit("fun voer\n");
}
```

运行结果：

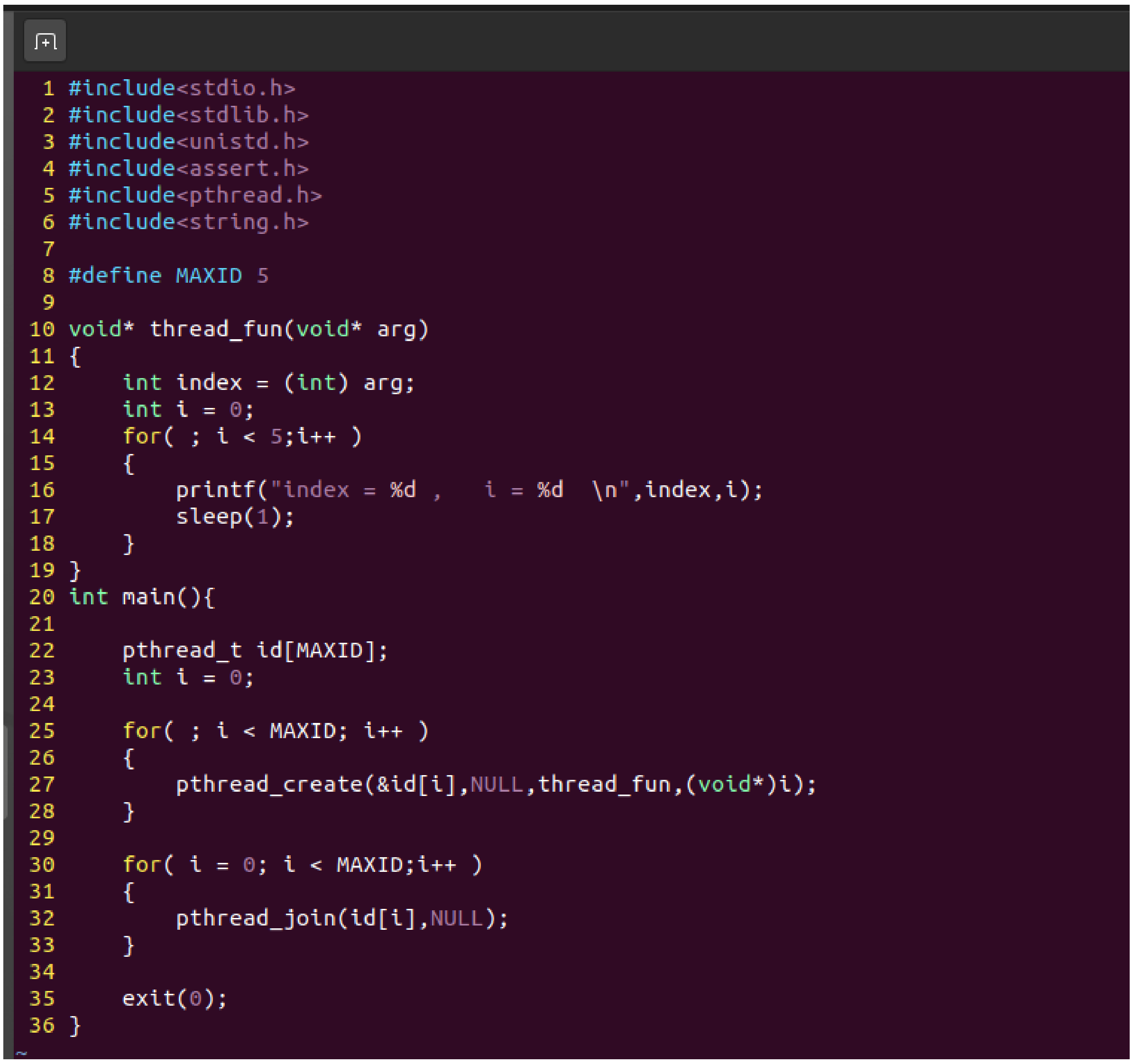


```
thread fun
stu@stu-virtual-machine:~/day_5_4$ vi main.c
stu@stu-virtual-machine:~/day_5_4$ gcc -o main main.c -lpthread
stu@stu-virtual-machine:~/day_5_4$ ./main
main fun
thread fun
main fun
thread fun
main fun
thread fun
main fun
thread fun
main fun
thread fun
thread fun
thread fun
thread fun
thread fun
thread fun
s=fun voer

main over
stu@stu-virtual-machine:~/day_5_4$ 
```

3. 函数参数举例

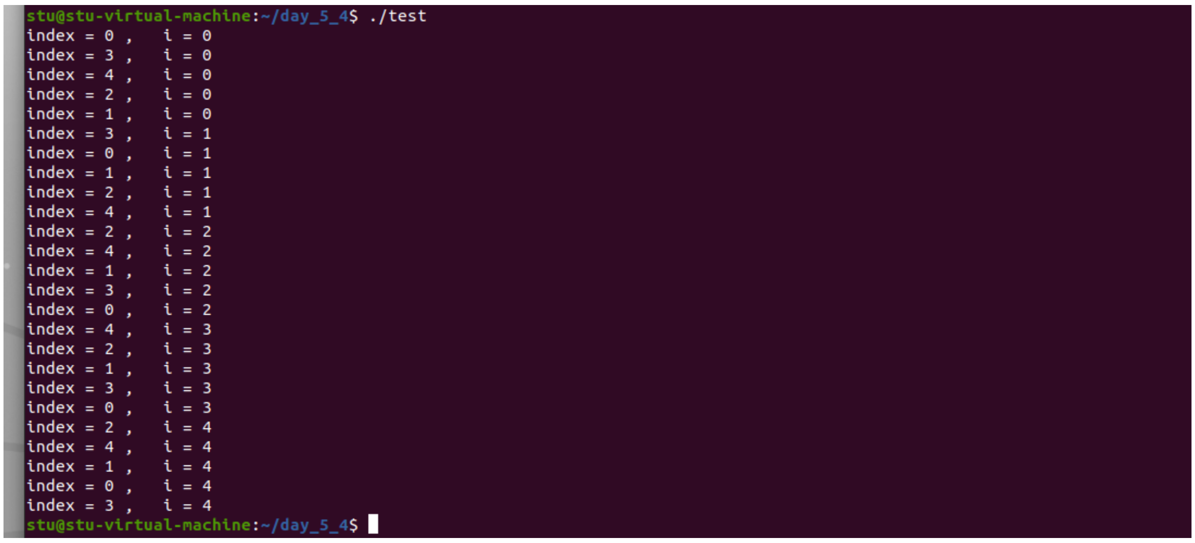
3.1 五个进程并发运行

创建五个线程，理解参数该如何赋值



```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>

#define MAXID 5

void* thread_fun(void* arg)
{
    int index = (int) arg;
    int i = 0;
    for( ; i < 5;i++ )
    {
        printf("index = %d ,    i = %d  \n",index,i);
        sleep(1);
    }
}
int main(){

    pthread_t id[MAXID];
    int i = 0;

    for( ; i < MAXID; i++ )
    {
        pthread_create(&id[i],NULL,thread_fun,(void*)i);
    }

    for( i = 0; i < MAXID;i++ )
    {
        pthread_join(id[i],NULL);
    }

    exit(0);
}
```

运行上面的代码我们可以得到下面的效果

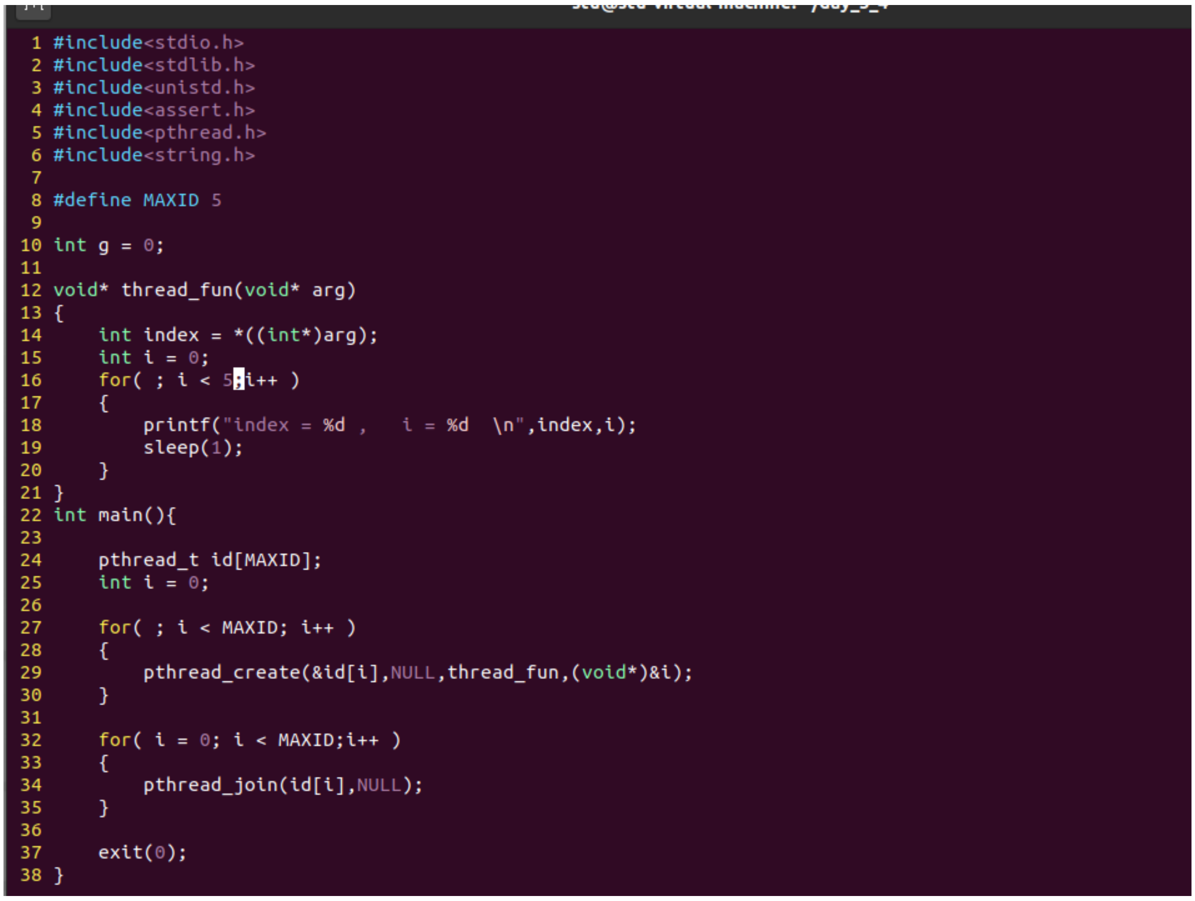


```
stu@stu-virtual-machine:~/day_5_4$ ./test
index = 0 ,    i = 0
index = 3 ,    i = 0
index = 4 ,    i = 0
index = 2 ,    i = 0
index = 1 ,    i = 0
index = 3 ,    i = 1
index = 0 ,    i = 1
index = 1 ,    i = 1
index = 2 ,    i = 1
index = 4 ,    i = 1
index = 2 ,    i = 2
index = 4 ,    i = 2
index = 1 ,    i = 2
index = 3 ,    i = 2
index = 0 ,    i = 2
index = 4 ,    i = 3
index = 2 ,    i = 3
index = 1 ,    i = 3
index = 3 ,    i = 3
index = 0 ,    i = 3
index = 2 ,    i = 4
index = 4 ,    i = 4
index = 1 ,    i = 4
index = 0 ,    i = 4
index = 3 ,    i = 4
stu@stu-virtual-machine:~/day_5_4$ 
```

3.2 线程修改临界资源

一共是五组在一段时间里面同时执行，即在1秒之内五个线程各执行一次，i代表第几次执行，说明五个线程并发运行

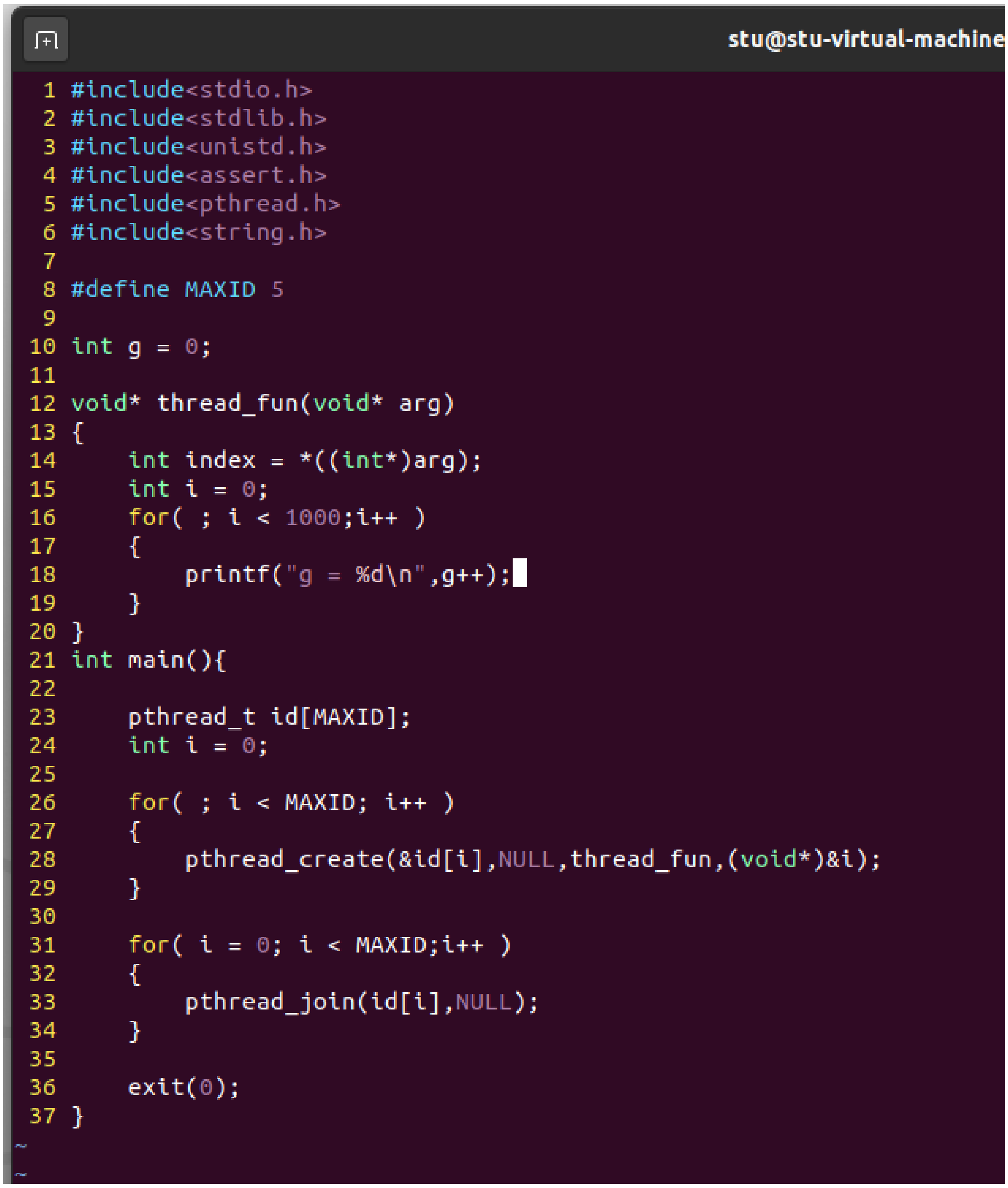
但是将出传入的参数改变，改变为取i的地址



```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>

#define MAXID 5

int g = 0;

void* thread_fun(void* arg)
{
    int index = *((int*)arg);
    int i = 0;
    for( ; i < 5;i++ )
    {
        printf("index = %d ,   i = %d  \n",index,i);
        sleep(1);
    }
}
int main(){

    pthread_t id[MAXID];
    int i = 0;

    for( ; i < MAXID; i++ )
    {
        pthread_create(&id[i],NULL,thread_fun,(void*)&i);
    }

    for( i = 0; i < MAXID;i++ )
    {
        pthread_join(id[i],NULL);
    }

    exit(0);
}
```

这里面拿最好的那个例子来解释吧，就是五个进程的index都是0的时候，由于创建线程太慢了，程序已经跑到了下面这个for循环里面了，程序获取的index值为0，而且这个时候每一个线程都执行等待，也就导致了后面一直index都是0，但是这种情况是不一定的，每次的值都会不一样

为了更好的理解并发，创建一个全局变量，创建五个进程，对这个全局变量进行改变


stu@stu-virtual-machine

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>

#define MAXID 5

int g = 0;

void* thread_fun(void* arg)
{
    int index = *((int*)arg);
    int i = 0;
    for( ; i < 1000;i++ )
    {
        printf("g = %d\n",g++);
    }
}
int main(){

    pthread_t id[MAXID];
    int i = 0;

    for( ; i < MAXID; i++ )
    {
        pthread_create(&id[i],NULL,thread_fun,(void*)&i);
    }

    for( i = 0; i < MAXID;i++ )
    {
        pthread_join(id[i],NULL);
    }

    exit(0);
}
~
~
```

那么正常的理解是最后g的值会累加到4999，因为这里是g++，不是++g，但是如果是一个处理器，并且是单核的处理器的话，就会出现少数据的情况，这种原因就是，在当前线程修改这个全局变量的值的时候，由于g++不是原子操作，所以其操作会被其他线程打断，这个时候会使得这两个线程只++了一次这个全局变量，我虚拟机暂时设置不了单处理器单核，所以演示不了这种情况，大概就是下面这种情况：

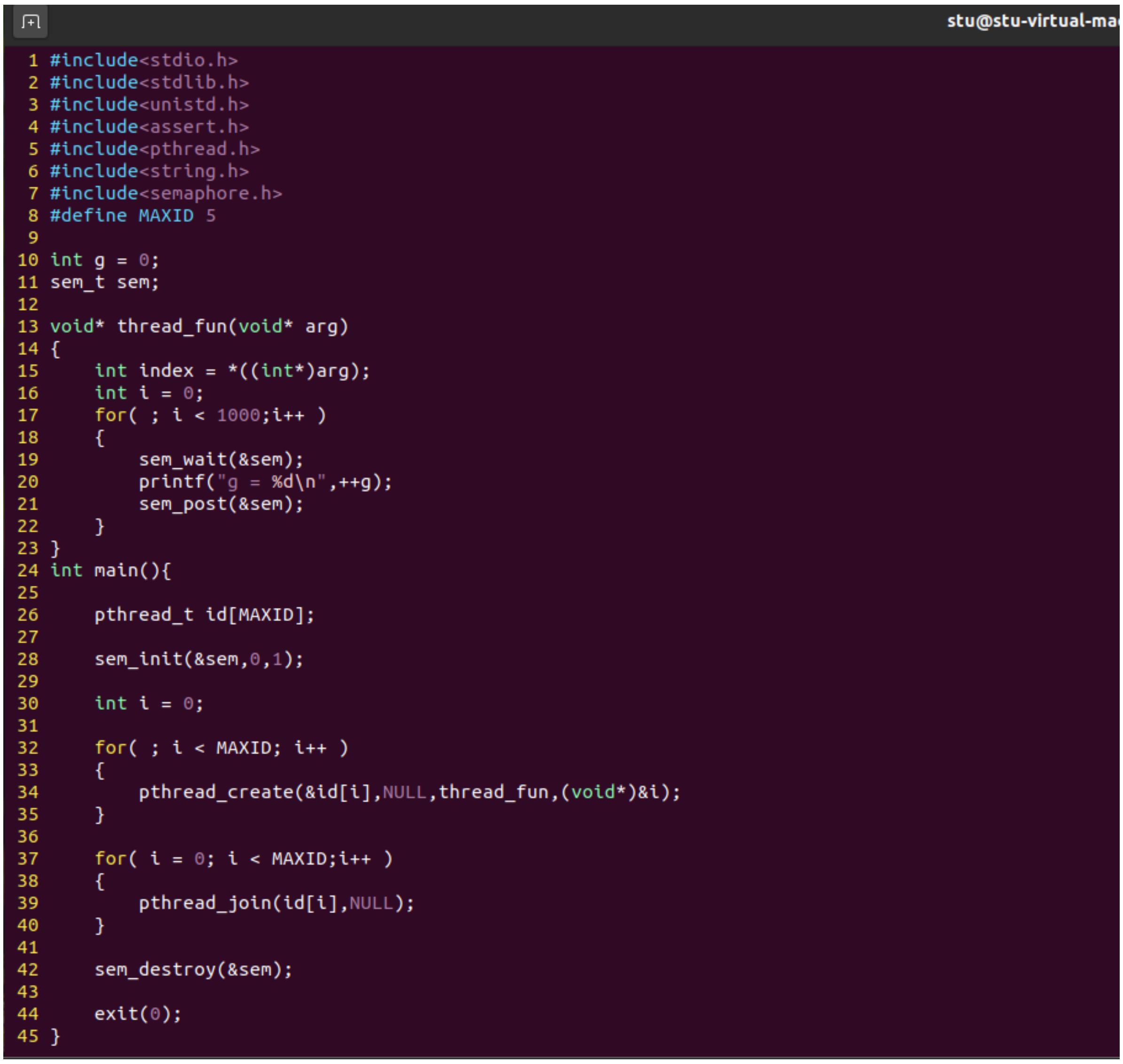
> 0
>
> ...
>
> n

这里面n是小于4999的正整数

3.3 线程修改临界资源加信号量控制

那么如何解决这种情况，还是一个就是加pv操作，对临界资源进行控制

所以需要设置信号量

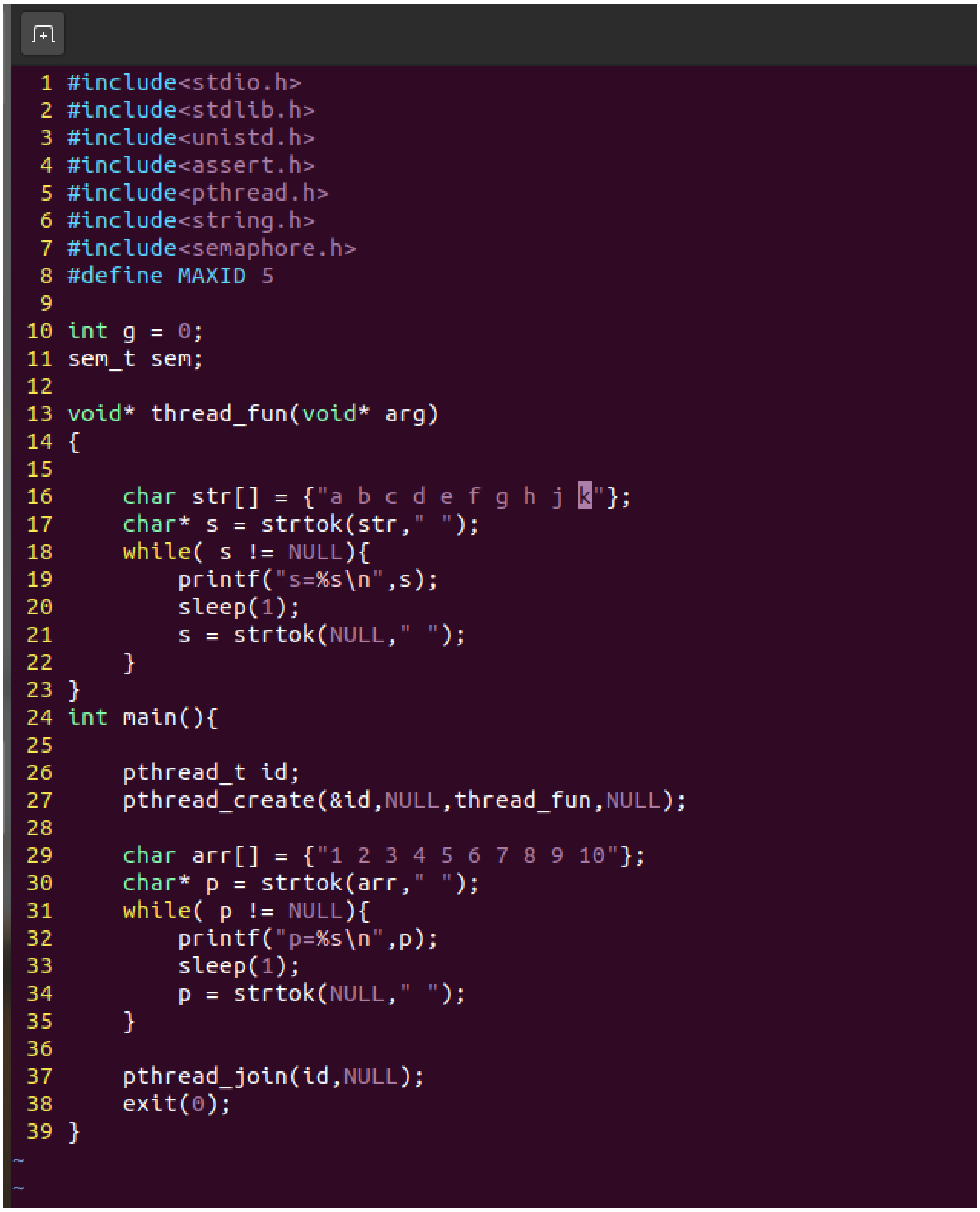

stu@stu-virtual-ma

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>
#include<semaphore.h>
#define MAXID 5

int g = 0;
sem_t sem;

void* thread_fun(void* arg)
{
    int index = *((int*)arg);
    int i = 0;
    for( ; i < 1000;i++ )
    {
        sem_wait(&sem);
        printf("g = %d\n",++g);
        sem_post(&sem);
    }
}
int main(){

    pthread_t id[MAXID];

    sem_init(&sem,0,1);

    int i = 0;

    for( ; i < MAXID; i++ )
    {
        pthread_create(&id[i],NULL,thread_fun,(void*)&i);
    }

    for( i = 0; i < MAXID;i++ )
    {
        pthread_join(id[i],NULL);
    }

    sem_destroy(&sem);

    exit(0);
}
```

那么现在执行这个程序，可以累加到5000，这里面将后置++改变为前置++

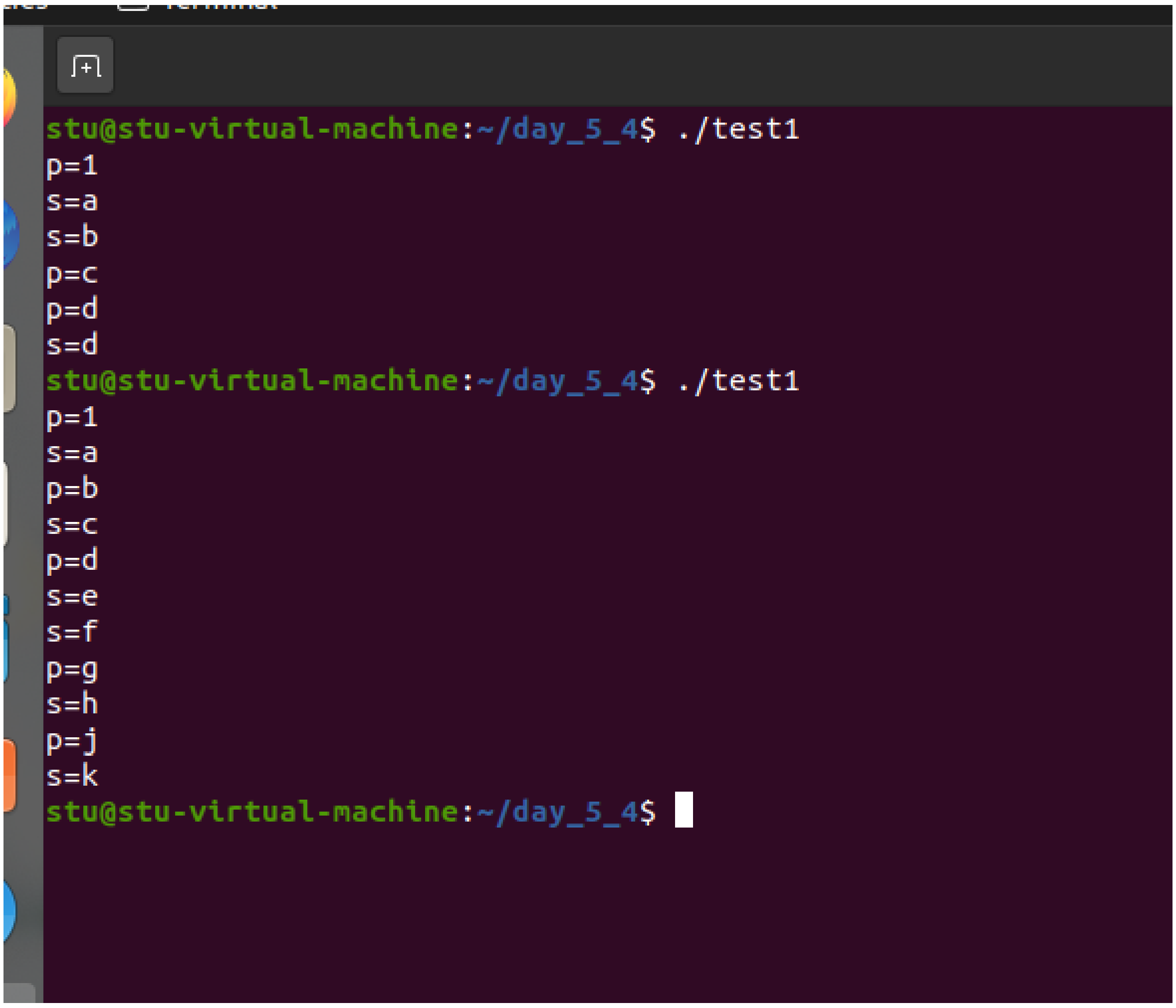
如下图：

3.4 思考的作业



```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>
#include<semaphore.h>
#define MAXID 5

int g = 0;
sem_t sem;

void* thread_fun(void* arg)
{

    char str[] = {"a b c d e f g h j k"};
    char* s = strtok(str," ");
    while( s != NULL){
        printf("s=%s\n",s);
        sleep(1);
        s = strtok(NULL," ");
    }
}
int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    char arr[] = {"1 2 3 4 5 6 7 8 9 10"};
    char* p = strtok(arr," ");
    while( p != NULL){
        printf("p=%s\n",p);
        sleep(1);
        p = strtok(NULL," ");
    }

    pthread_join(id,NULL);
    exit(0);
}
~
~
```

结果:



```
stu@stu-virtual-machine:~/day_5_4$ ./test1
p=1
s=a
s=b
p=c
p=d
s=d
stu@stu-virtual-machine:~/day_5_4$ ./test1
p=1
s=a
p=b
s=c
p=d
s=e
s=f
p=g
s=h
p=j
s=k
stu@stu-virtual-machine:~/day_5_4$
```

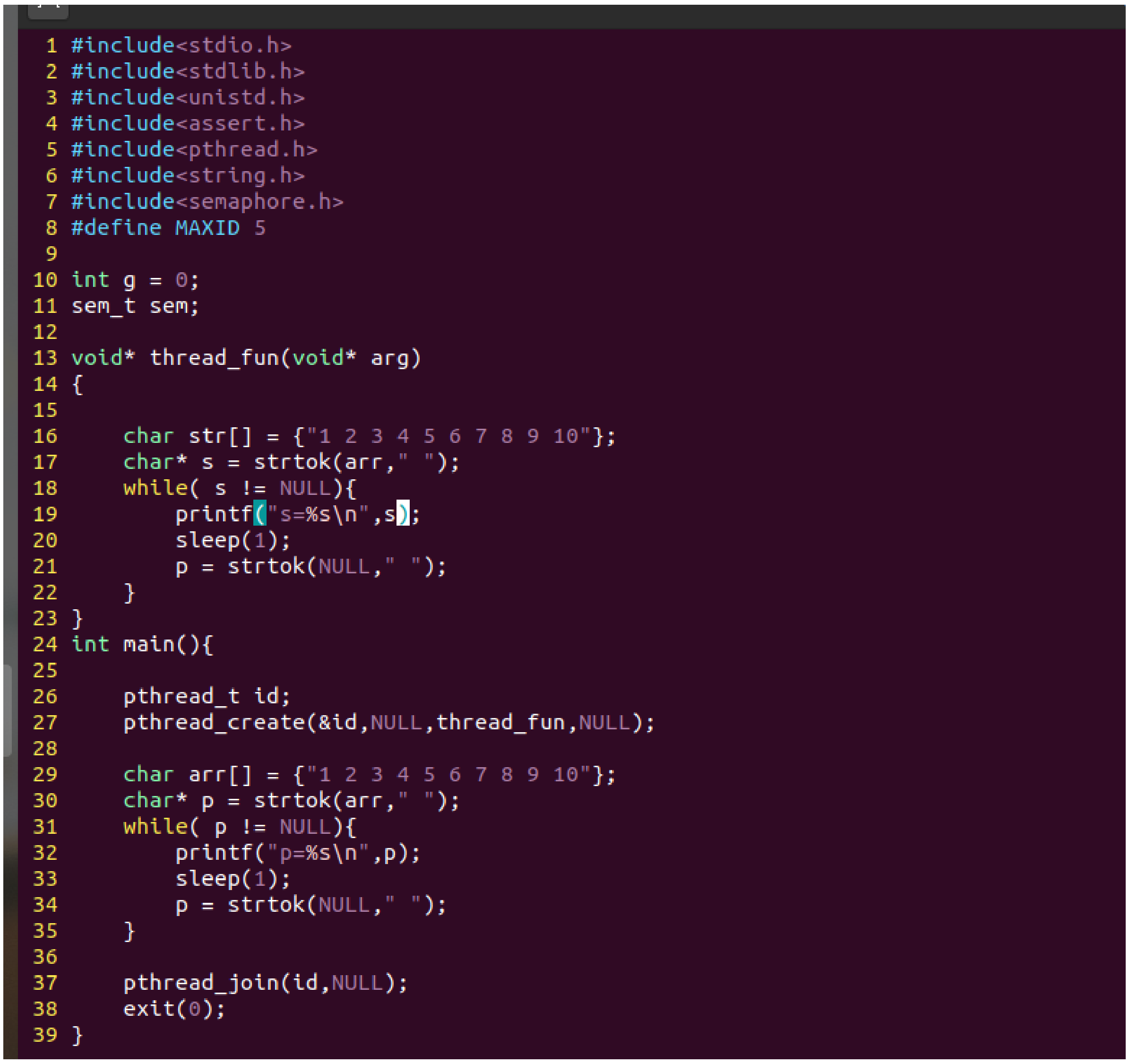
3. 为什么要采用多线程?

1. 同时做多件事情
2. 利用多处理器的资源，一个处理器运行一个线程

4. 线程同步

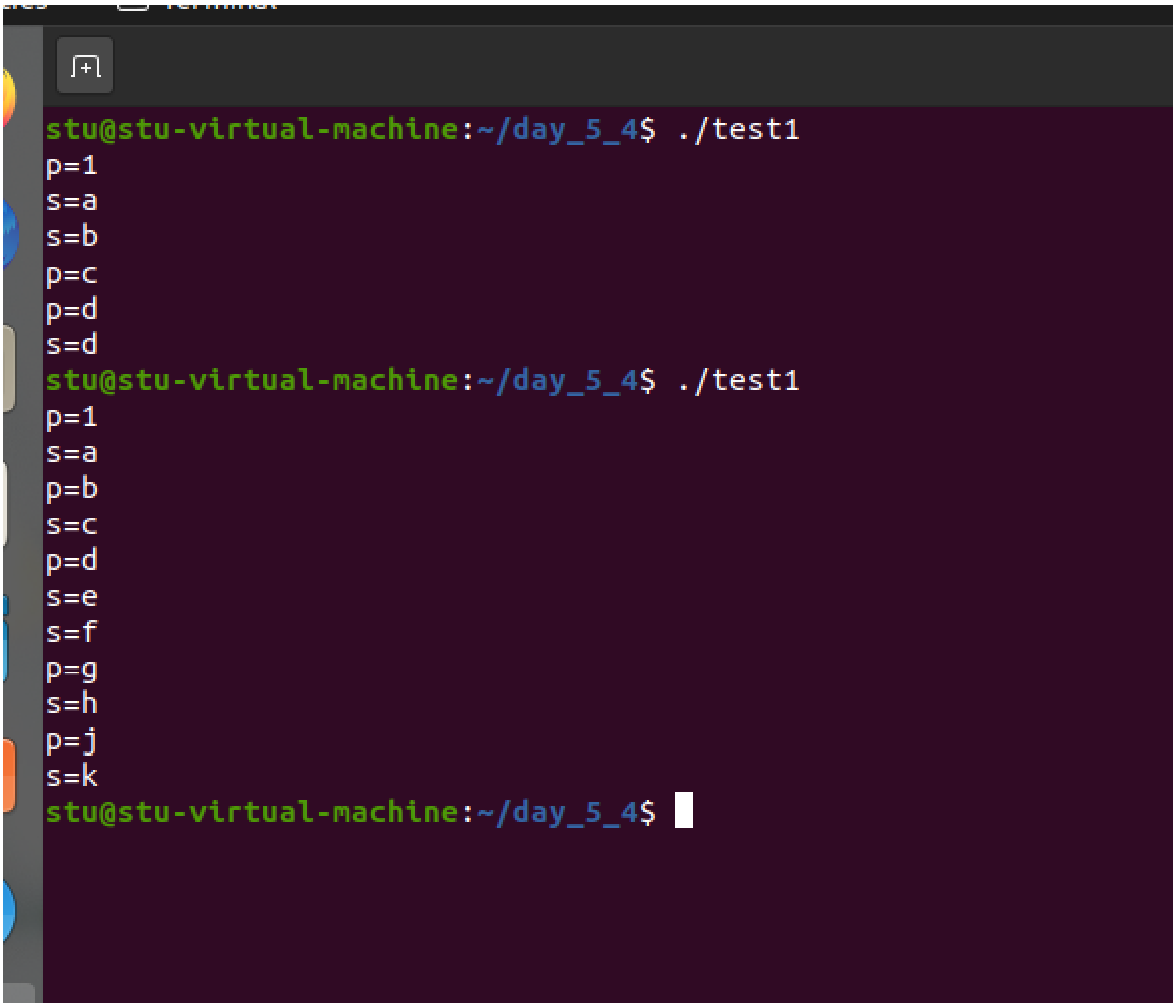
4.1 线程安全的函数

解决上面的作业



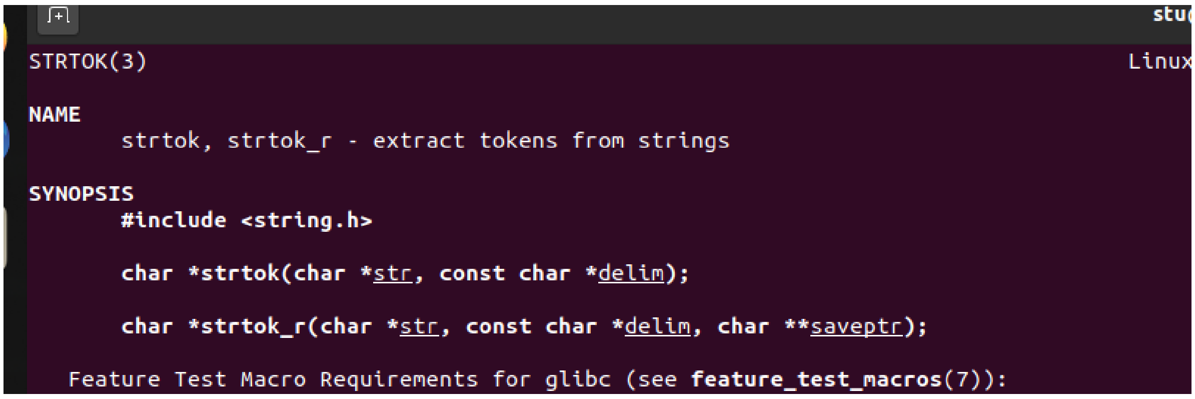
```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>
#include<semaphore.h>
#define MAXID 5

int g = 0;
sem_t sem;

void* thread_fun(void* arg)
{

    char str[] = {"1 2 3 4 5 6 7 8 9 10"};
    char* s = strtok(arr," ");
    while( s != NULL){
        printf("s=%s\n",s);
        sleep(1);
        p = strtok(NULL," ");
    }
}
int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    char arr[] = {"1 2 3 4 5 6 7 8 9 10"};
    char* p = strtok(arr," ");
    while( p != NULL){
        printf("p=%s\n",p);
        sleep(1);
        p = strtok(NULL," ");
    }

    pthread_join(id,NULL);
    exit(0);
}
```

结果:



```
stu@stu-virtual-machine:~/day_5_4$ ./test1
p=1
s=a
s=b
p=c
p=d
s=d
stu@stu-virtual-machine:~/day_5_4$ ./test1
p=1
s=a
p=b
s=c
p=d
s=e
s=f
p=g
s=h
p=j
s=k
stu@stu-virtual-machine:~/day_5_4$ 
```

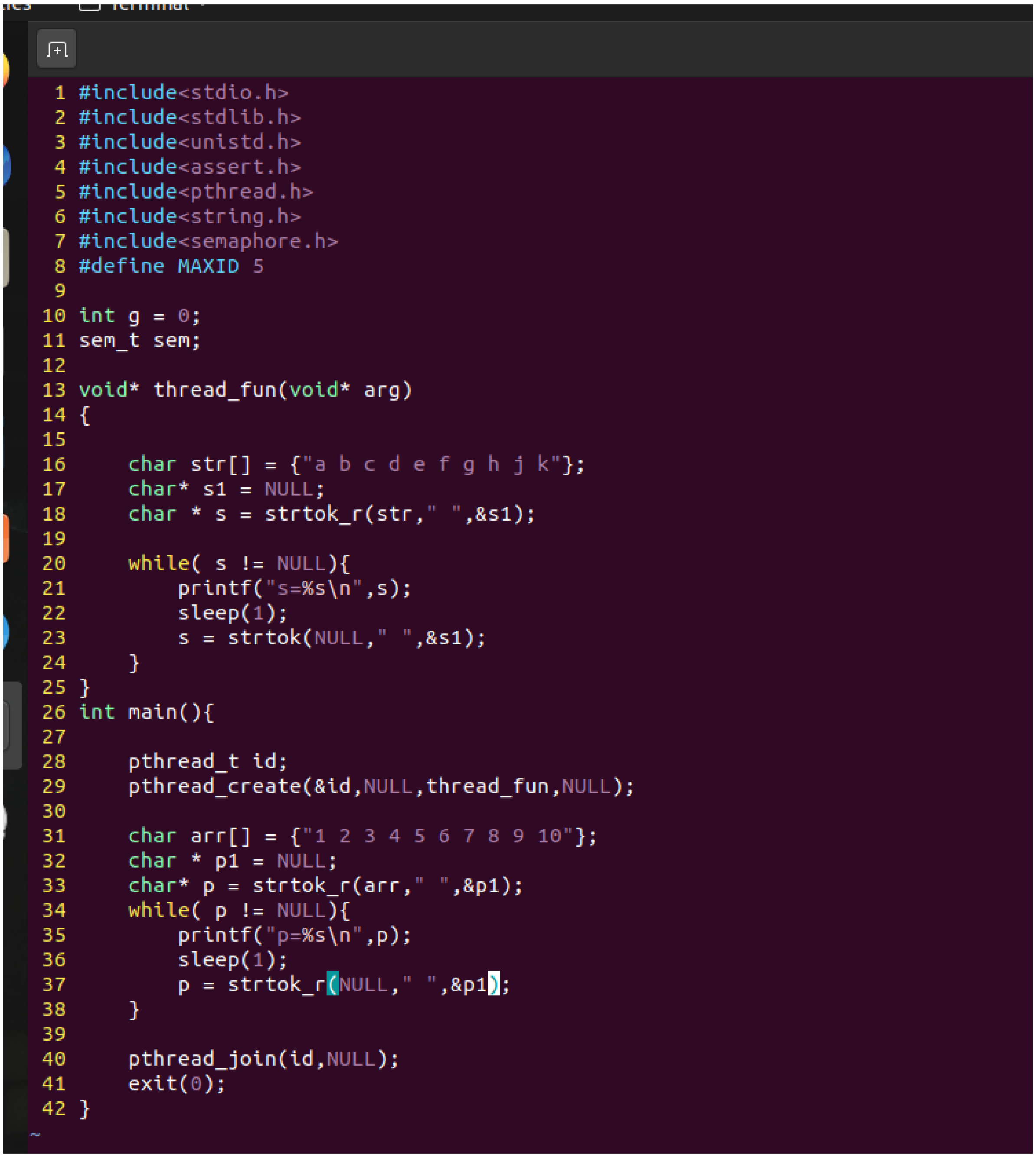
上面这个函数是线程不安全的，因为在多个线程在进行strtok的时候，线程之间在没执行完之前会来回strtok分割，而这个时候就会有覆盖掉前面值的风险，所以是线程不安全的，所以要使用线程安全的函数



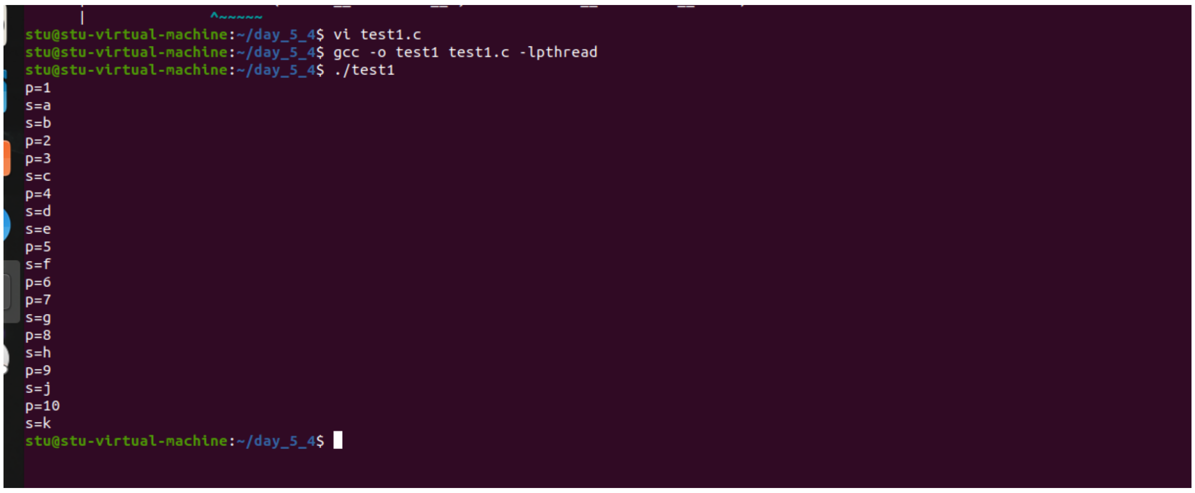
```
STRTOK(3)                                                          Linux

NAME
       strtok, strtok_r - extract tokens from strings

SYNOPSIS
       #include <string.h>

       char *strtok(char *str, const char *delim);

       char *strtok_r(char *str, const char *delim, char **saveptr);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
```

带_r表示线程安全，这里面的第三个参数表示一存放当前分割字符串分别指向字符串的哪个位置

我们修改之后

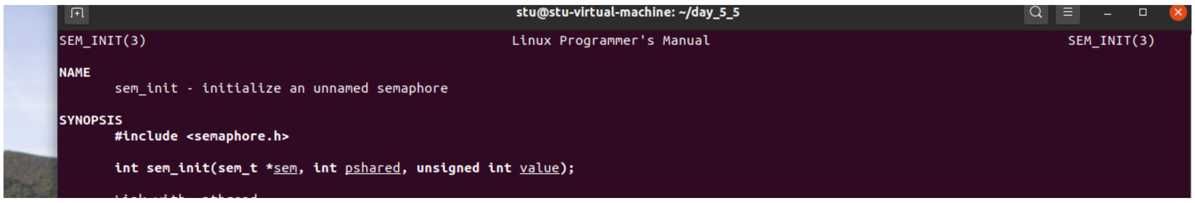


```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>
#include<semaphore.h>
#define MAXID 5

int g = 0;
sem_t sem;

void* thread_fun(void* arg)
{

    char str[] = {"a b c d e f g h j k"};
    char* s1 = NULL;
    char * s = strtok_r(str," ",&s1);

    while( s != NULL){
        printf("s=%s\n",s);
        sleep(1);
        s = strtok(NULL," ",&s1);
    }
}
int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    char arr[] = {"1 2 3 4 5 6 7 8 9 10"};
    char * p1 = NULL;
    char* p = strtok_r(arr," ",&p1);
    while( p != NULL){
        printf("p=%s\n",p);
        sleep(1);
        p = strtok_r(NULL," ",&p1);
    }

    pthread_join(id,NULL);
    exit(0);
}
~
```

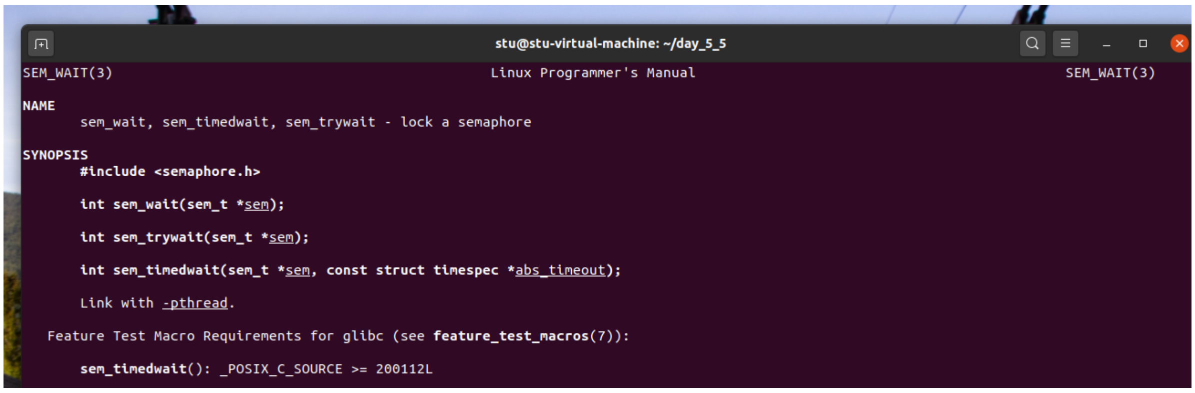


```
stu@stu-virtual-machine:~/day_5_4$ vi test1.c
stu@stu-virtual-machine:~/day_5_4$ gcc -o test1 test1.c -lpthread
stu@stu-virtual-machine:~/day_5_4$ ./test1
p=1
s=a
s=b
p=2
p=3
s=c
p=4
s=d
s=e
p=5
s=f
p=6
p=7
s=g
p=8
s=h
p=9
s=j
p=10
s=k
stu@stu-virtual-machine:~/day_5_4$ 
```

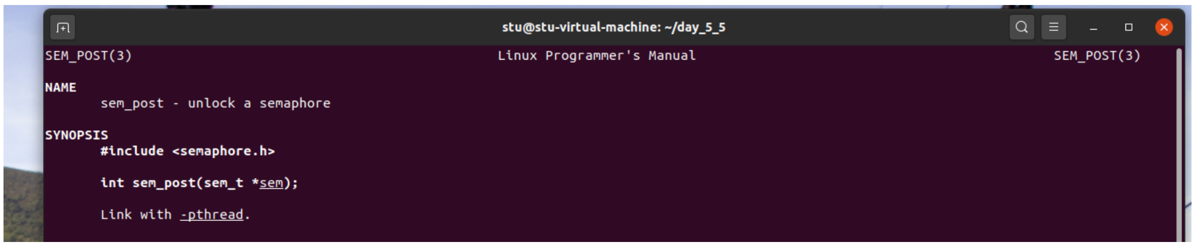
4.2 信号量机制实现同步



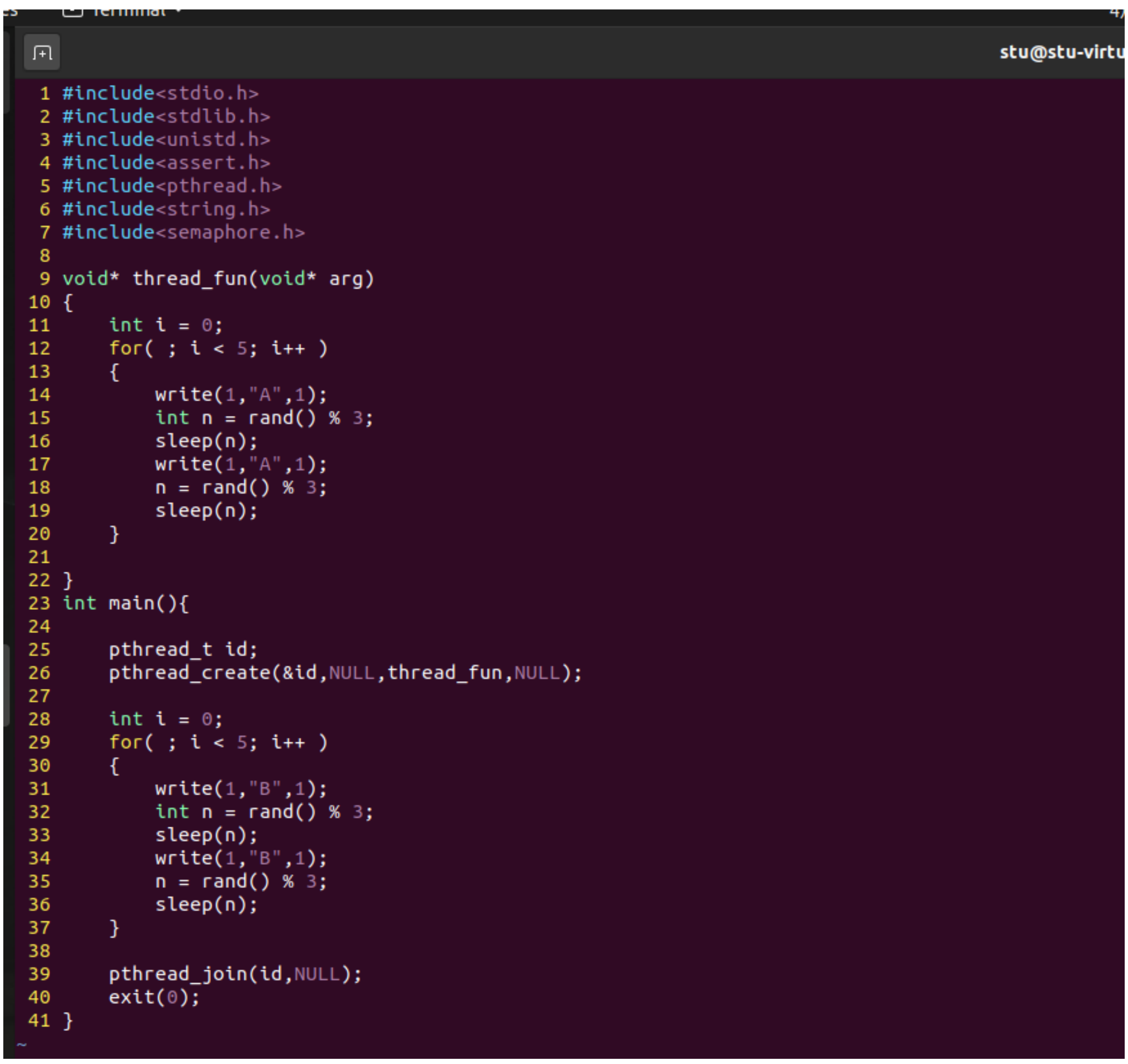
```
SEM_INIT(3)                    Linux Programmer's Manual                    SEM_INIT(3)

NAME
       sem_init - initialize an unnamed semaphore

SYNOPSIS
       #include <semaphore.h>

       int sem_init(sem_t *sem, int pshared, unsigned int value);
```



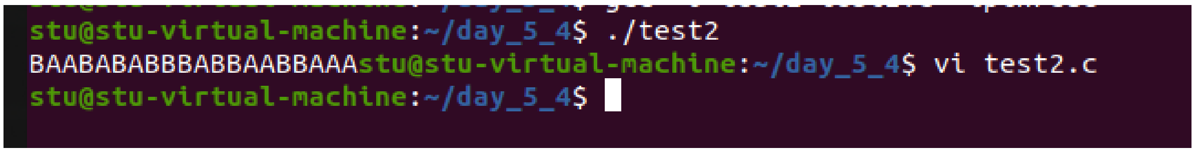
```
SEM_WAIT(3)                    Linux Programmer's Manual                    SEM_WAIT(3)

NAME
       sem_wait, sem_timedwait, sem_trywait - lock a semaphore

SYNOPSIS
       #include <semaphore.h>

       int sem_wait(sem_t *sem);

       int sem_trywait(sem_t *sem);

       int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

       Link with -pthread.

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       sem_timedwait(): _POSIX_C_SOURCE >= 200112L
```



```
SEM_POST(3)                    Linux Programmer's Manual                    SEM_POST(3)

NAME
       sem_post - unlock a semaphore

SYNOPSIS
       #include <semaphore.h>

       int sem_post(sem_t *sem);

       Link with -pthread.
```

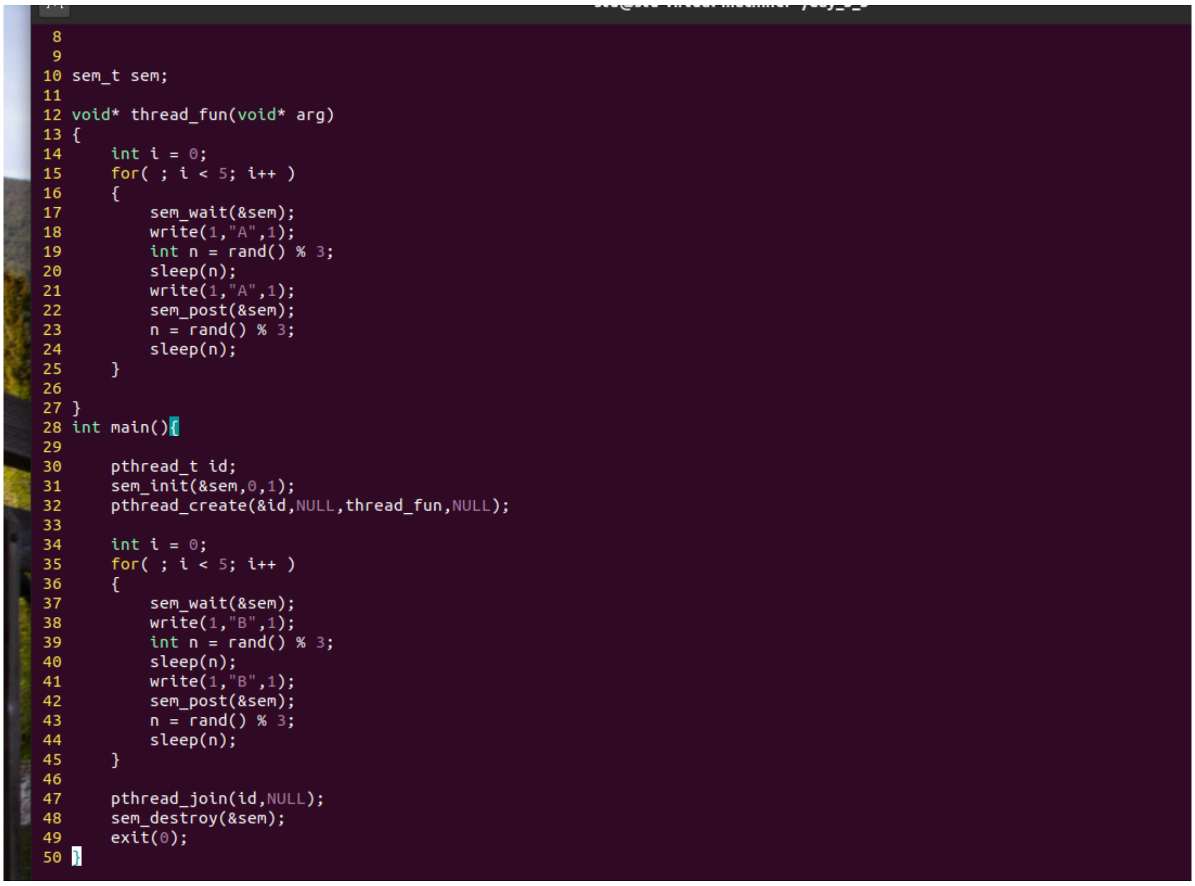


```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>
#include<semaphore.h>

void* thread_fun(void* arg)
{
    int i = 0;
    for( ; i < 5; i++ )
    {
        write(1,"A",1);
        int n = rand() % 3;
        sleep(n);
        write(1,"A",1);
        n = rand() % 3;
        sleep(n);
    }

}
int main(){

    pthread_t id;
    pthread_create(&id,NULL,thread_fun,NULL);

    int i = 0;
    for( ; i < 5; i++ )
    {
        write(1,"B",1);
        int n = rand() % 3;
        sleep(n);
        write(1,"B",1);
        n = rand() % 3;
        sleep(n);
    }

    pthread_join(id,NULL);
    exit(0);
}
```

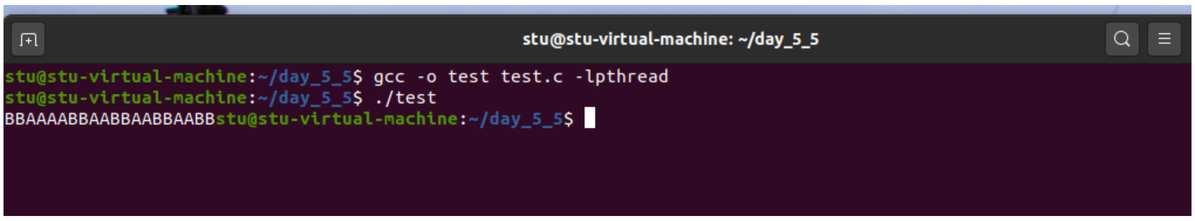


```
stu@stu-virtual-machine:~/day_5_4$ ./test2
BAABABABBBABBAABBAAAstu@stu-virtual-machine:~/day_5_4$ vi test2.c
stu@stu-virtual-machine:~/day_5_4$ 
```

采用信号量机制实现两个线程输出不被其他线程打断



```c


sem_t sem;

void* thread_fun(void* arg)
{
    int i = 0;
    for( ; i < 5; i++ )
    {
        sem_wait(&sem);
        write(1,"A",1);
        int n = rand() % 3;
        sleep(n);
        write(1,"A",1);
        sem_post(&sem);
        n = rand() % 3;
        sleep(n);
    }

}
int main(){

    pthread_t id;
    sem_init(&sem,0,1);
    pthread_create(&id,NULL,thread_fun,NULL);

    int i = 0;
    for( ; i < 5; i++ )
    {
        sem_wait(&sem);
        write(1,"B",1);
        int n = rand() % 3;
        sleep(n);
        write(1,"B",1);
        sem_post(&sem);
        n = rand() % 3;
        sleep(n);
    }

    pthread_join(id,NULL);
    sem_destroy(&sem);
    exit(0);
}
```



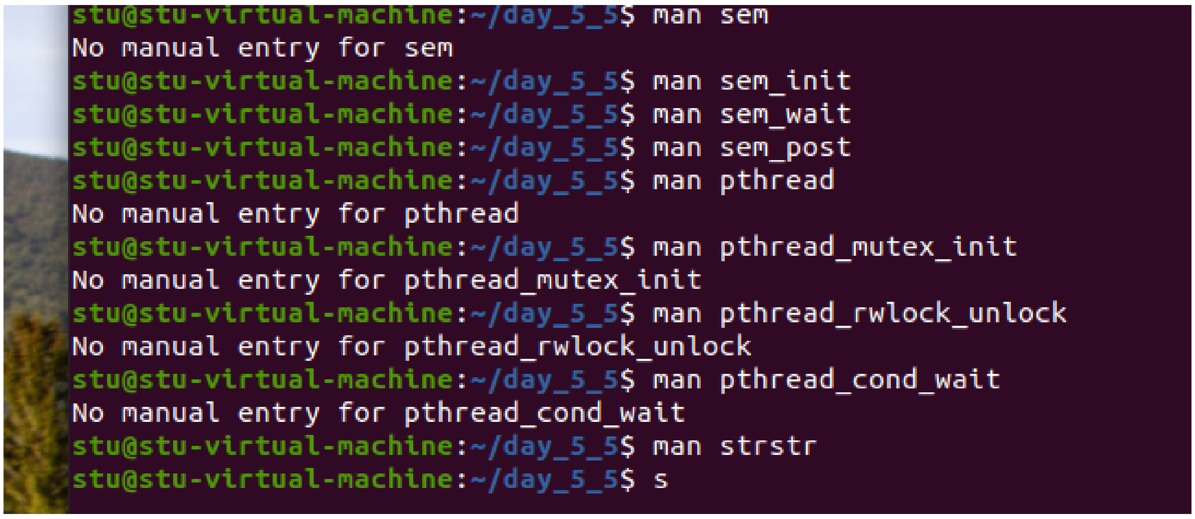
```
stu@stu-virtual-machine: ~/day_5_5
stu@stu-virtual-machine:~/day_5_5$ gcc -o test test.c -lpthread
stu@stu-virtual-machine:~/day_5_5$ ./test
BBAAAABBAABBAABBAABBstu@stu-virtual-machine:~/day_5_5$ 
```

4.4 互斥锁实现同步

```
1.  #include <pthread.h>
2.
3.  int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
4.
5.  int pthread_mutex_lock(pthread_mutex_t *mutex);
6.
```

```c
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

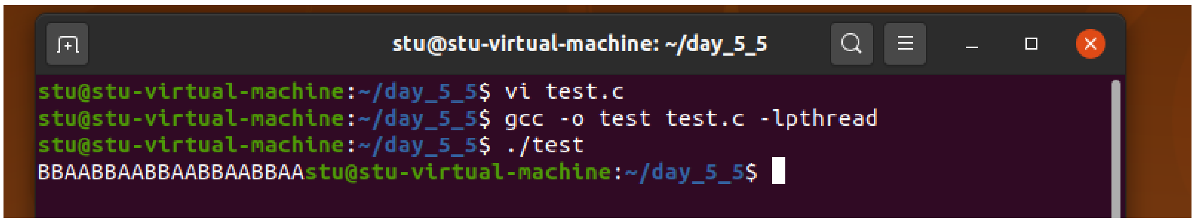
为什么没有？？？



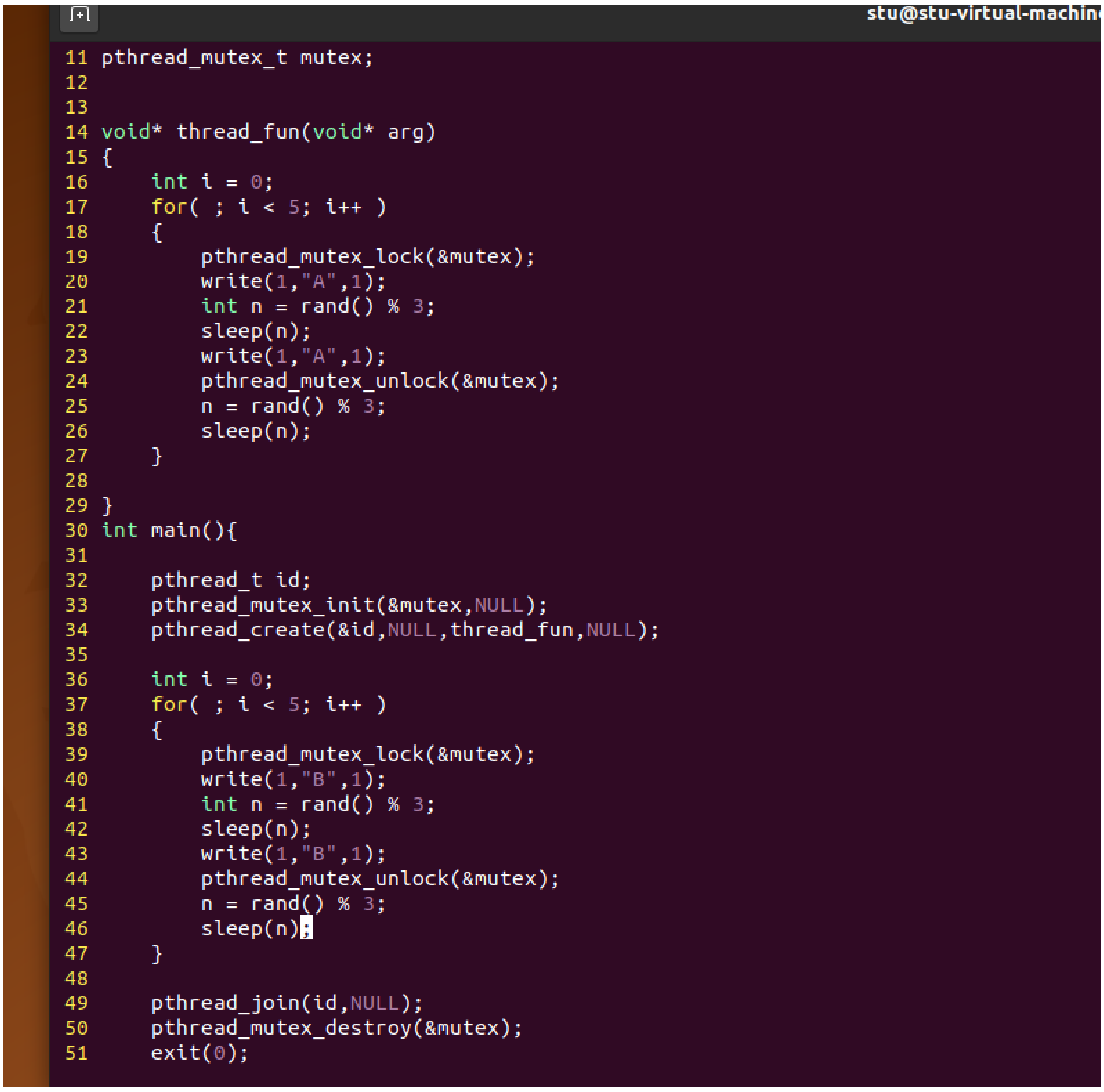
```
stu@stu-virtual-machine:~/day_5_5$ man sem
No manual entry for sem
stu@stu-virtual-machine:~/day_5_5$ man sem_init
stu@stu-virtual-machine:~/day_5_5$ man sem_wait
stu@stu-virtual-machine:~/day_5_5$ man sem_post
stu@stu-virtual-machine:~/day_5_5$ man pthread
No manual entry for pthread
stu@stu-virtual-machine:~/day_5_5$ man pthread_mutex_init
No manual entry for pthread_mutex_init
stu@stu-virtual-machine:~/day_5_5$ man pthread_rwlock_unlock
No manual entry for pthread_rwlock_unlock
stu@stu-virtual-machine:~/day_5_5$ man pthread_cond_wait
No manual entry for pthread_cond_wait
stu@stu-virtual-machine:~/day_5_5$ man strstr
stu@stu-virtual-machine:~/day_5_5$ s
```

安装manpages可以查看：

```
sudo apt-get install manpages-posix-dev
```



```
stu@stu-virtual-machine:~/day_5_5$ vi test.c
stu@stu-virtual-machine:~/day_5_5$ gcc -o test test.c -lpthread
stu@stu-virtual-machine:~/day_5_5$ ./test
BBAABBAABBAABBAABBAAstu@stu-virtual-machine:~/day_5_5$ 
```

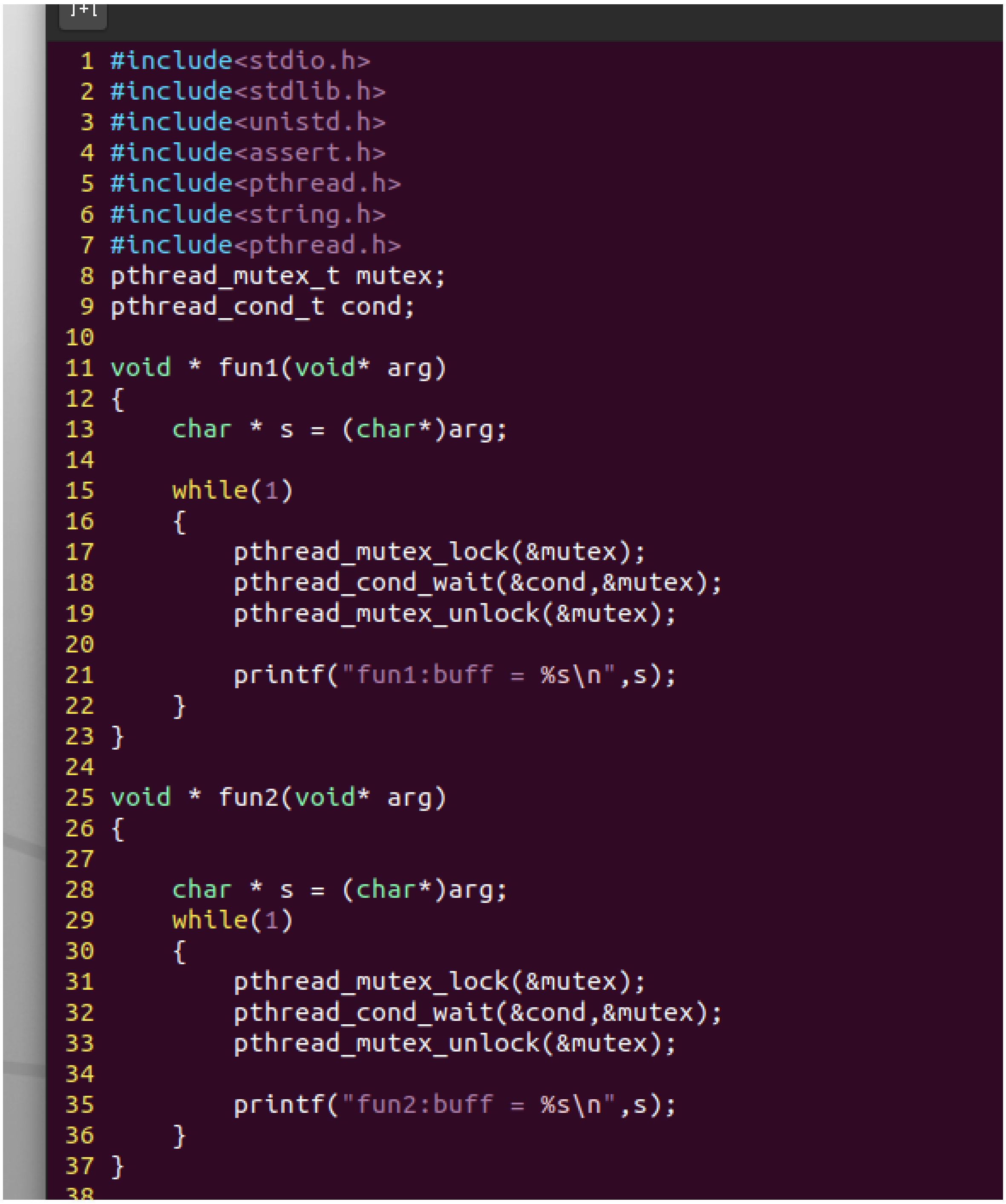


```
pthread_mutex_t mutex;


void* thread_fun(void* arg)
{
    int i = 0;
    for( ; i < 5; i++ )
    {
        pthread_mutex_lock(&mutex);
        write(1,"A",1);
        int n = rand() % 3;
        sleep(n);
        write(1,"A",1);
        pthread_mutex_unlock(&mutex);
        n = rand() % 3;
        sleep(n);
    }

}
int main(){

    pthread_t id;
    pthread_mutex_init(&mutex,NULL);
    pthread_create(&id,NULL,thread_fun,NULL);

    int i = 0;
    for( ; i < 5; i++ )
    {
        pthread_mutex_lock(&mutex);
        write(1,"B",1);
        int n = rand() % 3;
        sleep(n);
        write(1,"B",1);
        pthread_mutex_unlock(&mutex);
        n = rand() % 3;
        sleep(n);
    }

    pthread_join(id,NULL);
    pthread_mutex_destroy(&mutex);
    exit(0);
```

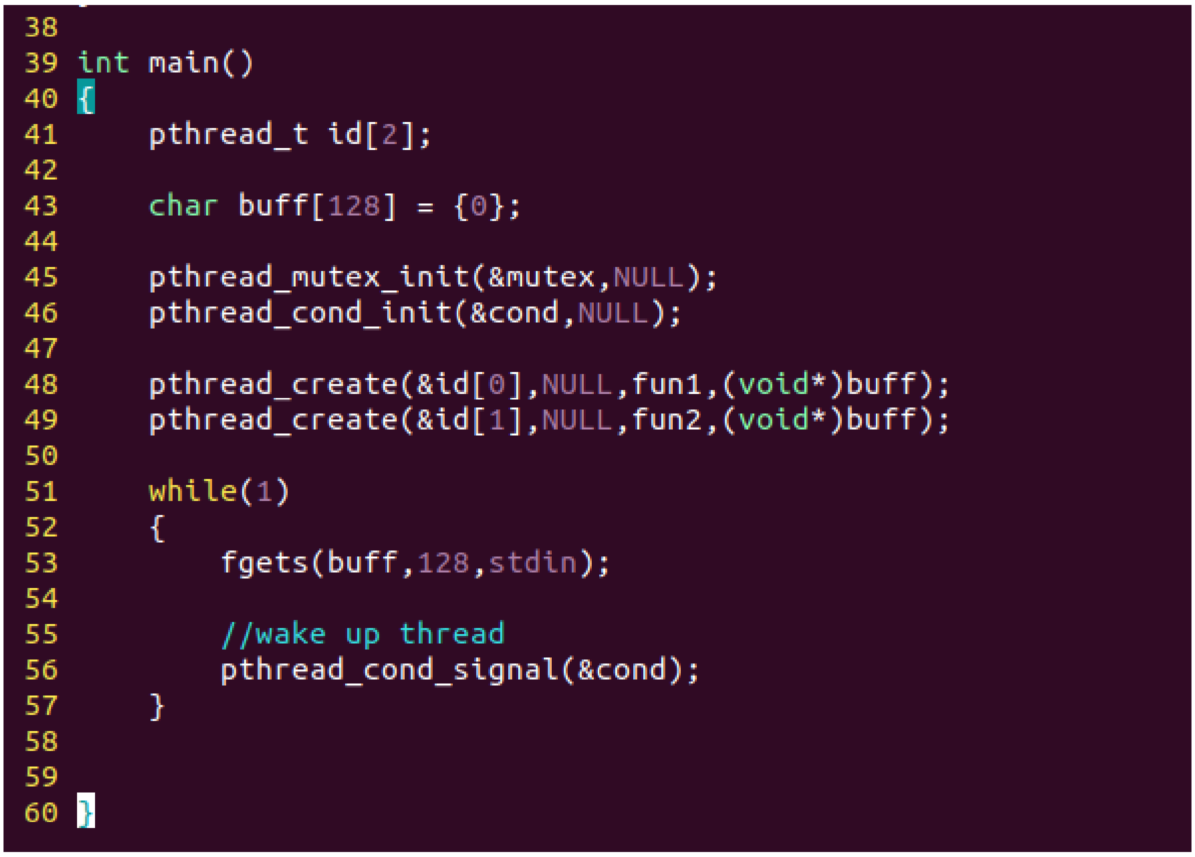
4.5 条件变量

条件变量提供了一种线程间的通知机制：当某个共享数据达到某个值的时候，唤醒等待这个共享数据的所有线程。

```c
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);         //唤醒单个线程

int pthread_cond_broadcast(pthread_cond_t *cond);  //唤醒所有等待的线程

int pthread_cond_destroy(pthread_cond_t *cond);
```



```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<assert.h>
#include<pthread.h>
#include<string.h>
#include<pthread.h>
pthread_mutex_t mutex;
pthread_cond_t cond;

void * fun1(void* arg)
{
    char * s = (char*)arg;

    while(1)
    {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&cond,&mutex);
        pthread_mutex_unlock(&mutex);

        printf("fun1:buff = %s\n",s);
    }
}

void * fun2(void* arg)
{

    char * s = (char*)arg;
    while(1)
    {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&cond,&mutex);
        pthread_mutex_unlock(&mutex);

        printf("fun2:buff = %s\n",s);
    }
}
```

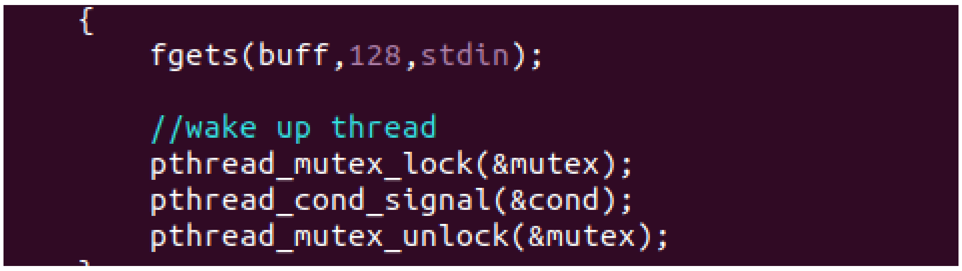


```
int main()
{
    pthread_t id[2];

    char buff[128] = {0};

    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);

    pthread_create(&id[0],NULL,fun1,(void*)buff);
    pthread_create(&id[1],NULL,fun2,(void*)buff);

    while(1)
    {
        fgets(buff,128,stdin);

        //wake up thread
        pthread_cond_signal(&cond);
    }


}
```

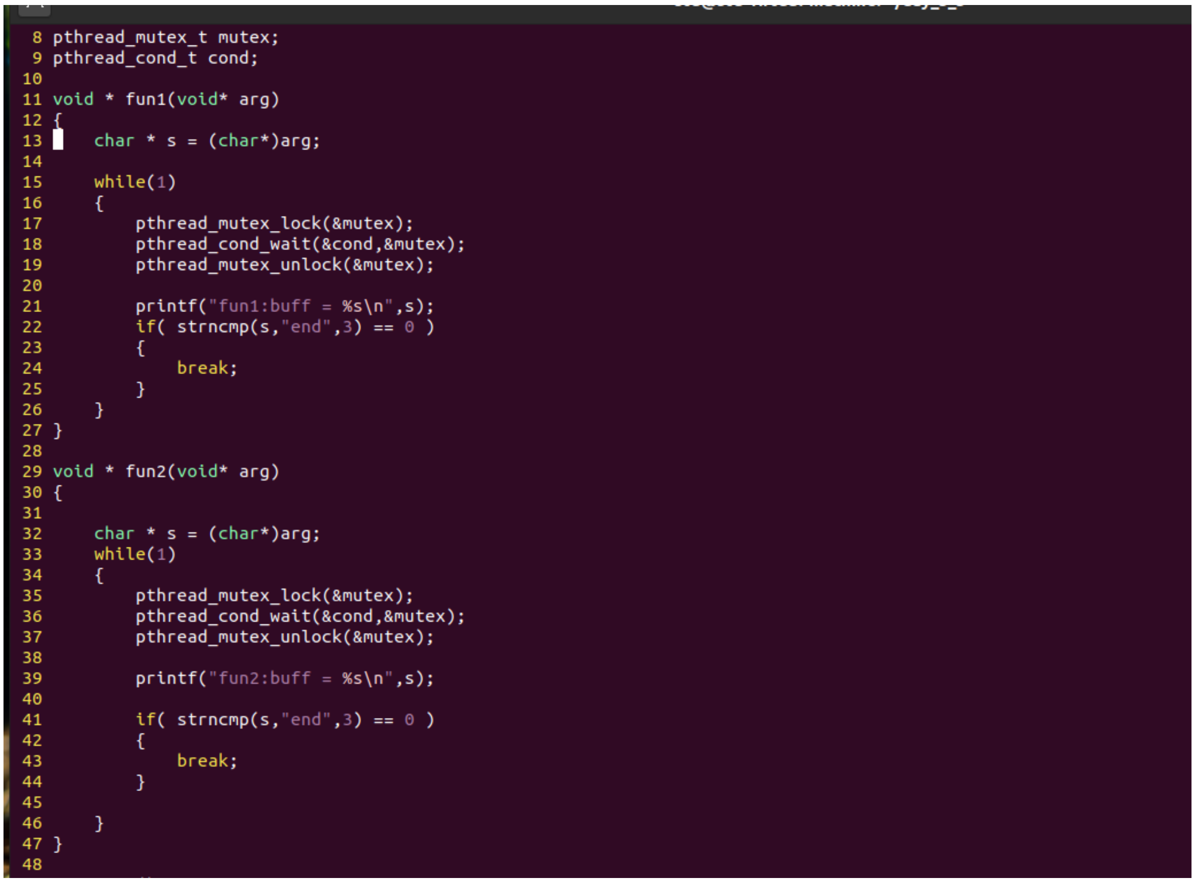
运行结果

在pthread_cond_wait之前加锁，后面解锁就是为了防止多个线程在等到信号量满足条件之后执行紊乱的情况出现，加上锁之后就不会出现这种情况，若正在唤醒线程的时候，这个时候某个线程正在进行操作，那么我们在唤醒线程前后也要加锁：

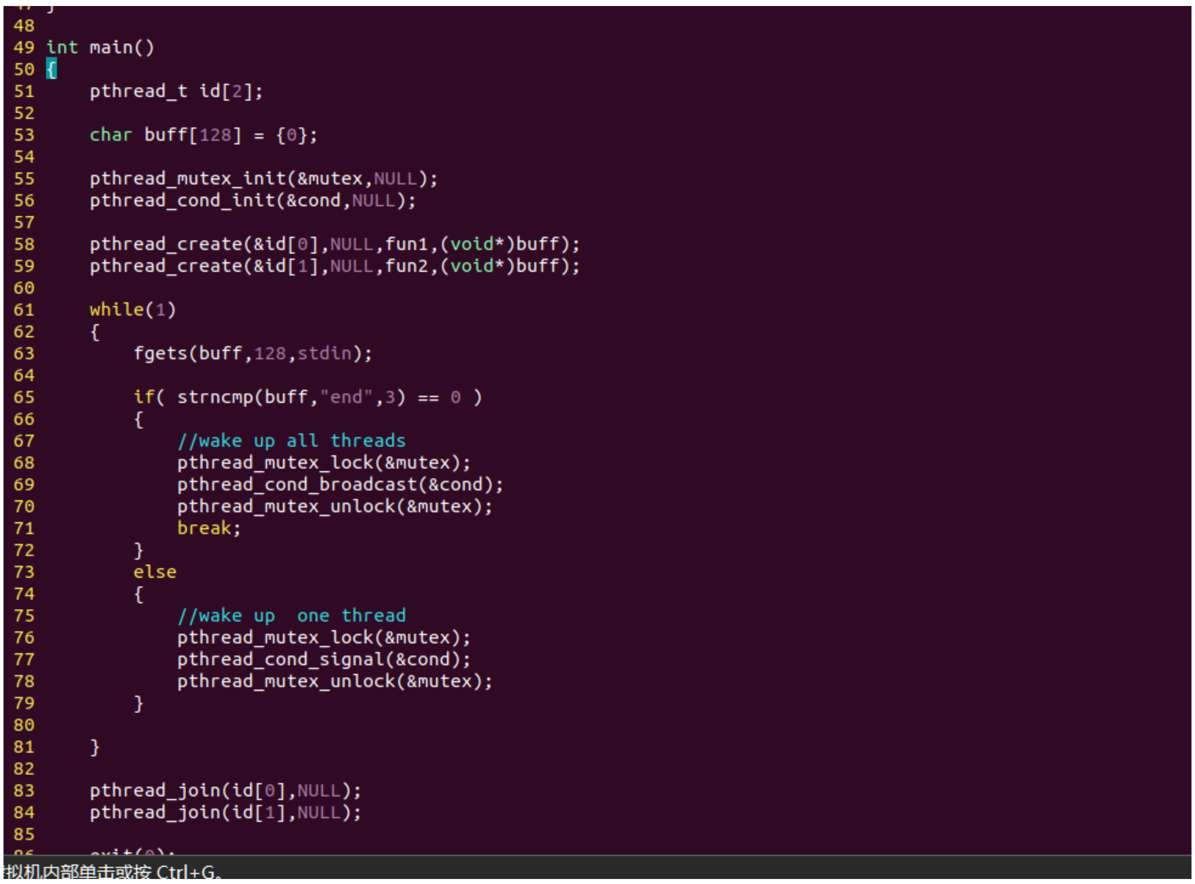
改进之后：



```
    {
        fgets(buff,128,stdin);

        //wake up thread
        pthread_mutex_lock(&mutex);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
```



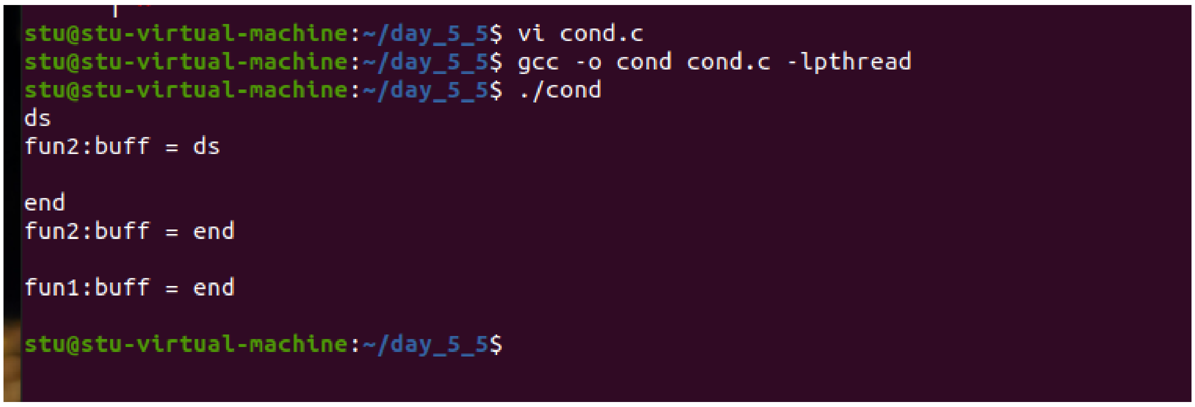
```c
pthread_mutex_t mutex;
pthread_cond_t cond;

void * fun1(void* arg)
{
    char * s = (char*)arg;

    while(1)
    {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&cond,&mutex);
        pthread_mutex_unlock(&mutex);

        printf("fun1:buff = %s\n",s);
        if( strncmp(s,"end",3) == 0 )
        {
            break;
        }
    }
}

void * fun2(void* arg)
{

    char * s = (char*)arg;
    while(1)
    {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&cond,&mutex);
        pthread_mutex_unlock(&mutex);

        printf("fun2:buff = %s\n",s);

        if( strncmp(s,"end",3) == 0 )
        {
            break;
        }

    }
}

```



```c

int main()
{
    pthread_t id[2];

    char buff[128] = {0};

    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);

    pthread_create(&id[0],NULL,fun1,(void*)buff);
    pthread_create(&id[1],NULL,fun2,(void*)buff);

    while(1)
    {
        fgets(buff,128,stdin);

        if( strncmp(buff,"end",3) == 0 )
        {
            //wake up all threads
            pthread_mutex_lock(&mutex);
            pthread_cond_broadcast(&cond);
            pthread_mutex_unlock(&mutex);
            break;
        }
        else
        {
            //wake up  one thread
            pthread_mutex_lock(&mutex);
            pthread_cond_signal(&cond);
            pthread_mutex_unlock(&mutex);
        }

    }

    pthread_join(id[0],NULL);
    pthread_join(id[1],NULL);

    exit(0);
```

拟机内部单击或按 Ctrl+G。

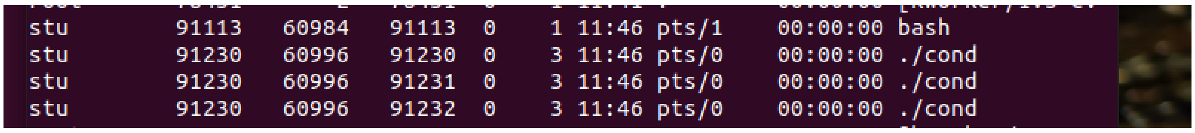
运行程序



```
stu@stu-virtual-machine:~/day_5_5$ vi cond.c
stu@stu-virtual-machine:~/day_5_5$ gcc -o cond cond.c -lpthread
stu@stu-virtual-machine:~/day_5_5$ ./cond
ds
fun2:buff = ds

end
fun2:buff = end

fun1:buff = end

stu@stu-virtual-machine:~/day_5_5$
```

如何看三个线程

```
ps -eLf  //中间的L显示线程id
```



```
stu        91113   60984   91113  0    1 11:46 pts/1    00:00:00 bash
stu        91230   60996   91230  0    3 11:46 pts/0    00:00:00 ./cond
stu        91230   60996   91231  0    3 11:46 pts/0    00:00:00 ./cond
stu        91230   60996   91232  0    3 11:46 pts/0    00:00:00 ./cond
```

线程的实现：在Linux中是通过进程方式实现的

strace跟踪系统调用

ltrace跟踪库函数

```
strace ./cond1
```