第一性原理

2024年5月24日 9:28

程序

||

指令 + 数据

数据：类型，值，类，对象，

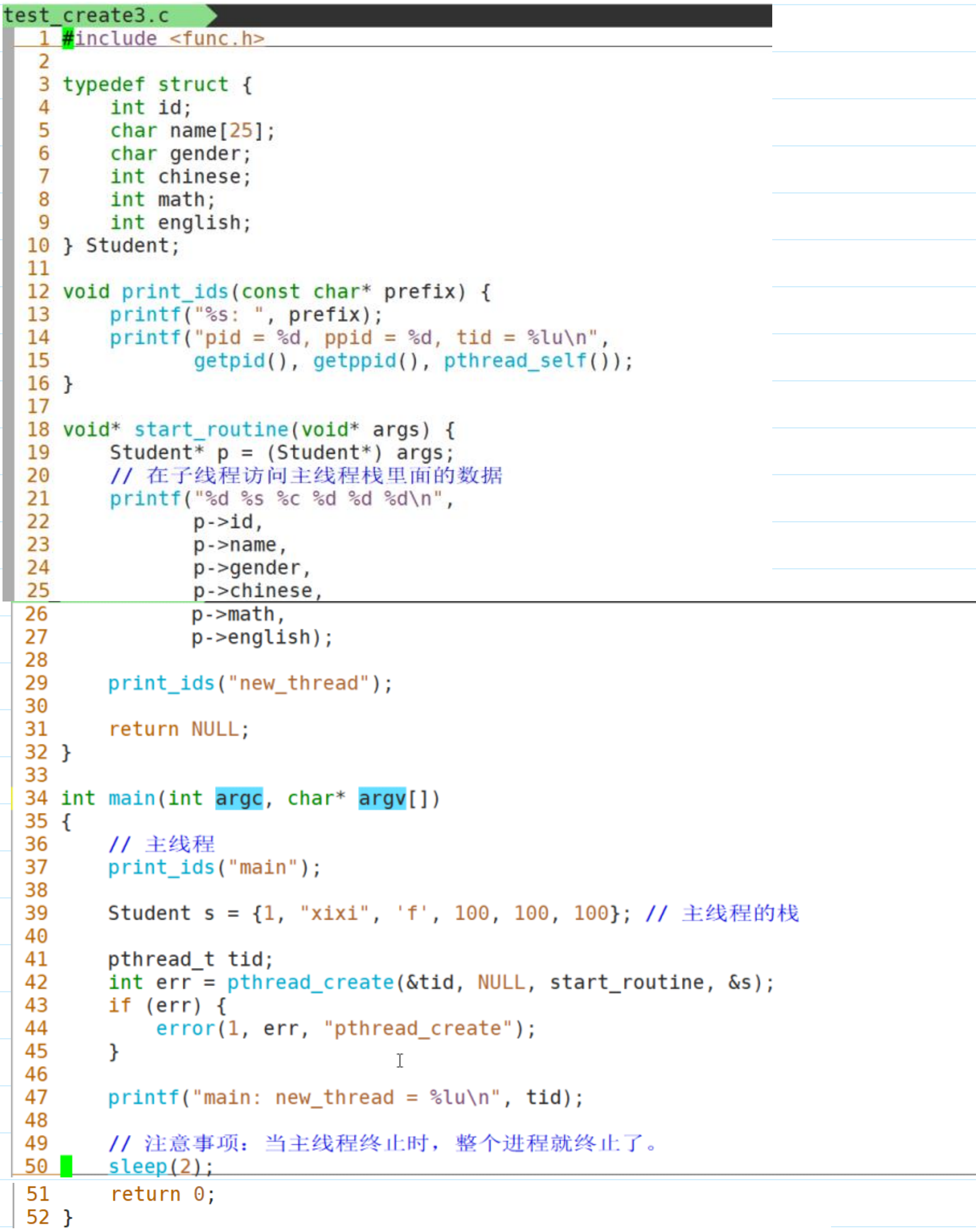
指令：运算符，语句，函数，方法，闭包，...

程序的运行方式：

- 同步／异步
- 并发／并行

创建线程

2024年5月24日 10:14



test_create3.c

```c
#include <func.h>

typedef struct {
    int id;
    char name[25];
    char gender;
    int chinese;
    int math;
    int english;
} Student;

void print_ids(const char* prefix) {
    printf("%s: ", prefix);
    printf("pid = %d, ppid = %d, tid = %lu\n",
           getpid(), getppid(), pthread_self());
}

void* start_routine(void* args) {
    Student* p = (Student*) args;
    // 在子线程访问主线程栈里面的数据
    printf("%d %s %c %d %d %d\n",
           p->id,
           p->name,
           p->gender,
           p->chinese,
           p->math,
           p->english);

    print_ids("new_thread");

    return NULL;
}

int main(int argc, char* argv[])
{
    // 主线程
    print_ids("main");

    Student s = {1, "xixi", 'f', 100, 100, 100}; // 主线程的栈

    pthread_t tid;
    int err = pthread_create(&tid, NULL, start_routine, &s);
    if (err) {
        error(1, err, "pthread_create");
    }

    printf("main: new_thread = %lu\n", tid);

    // 注意事项：当主线程终止时，整个进程就终止了。
    sleep(2);
    return 0;
}
```

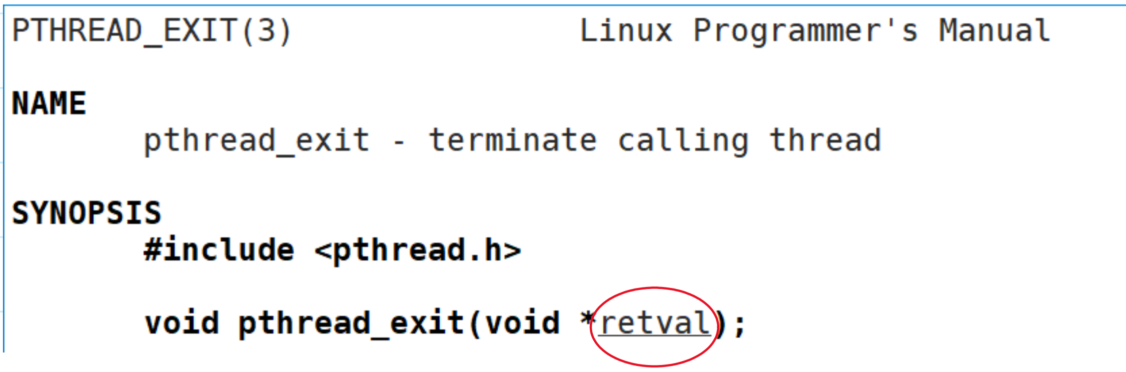
终止线程

2024年5月24日 10:50

进程	线程
从 main 返回	从 start_routine 返回
exit()	pthread_exit()
收到信号	pthread_cancel()

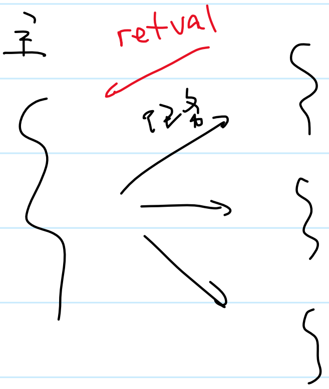
pthread_exit()

2024年5月24日 10:55



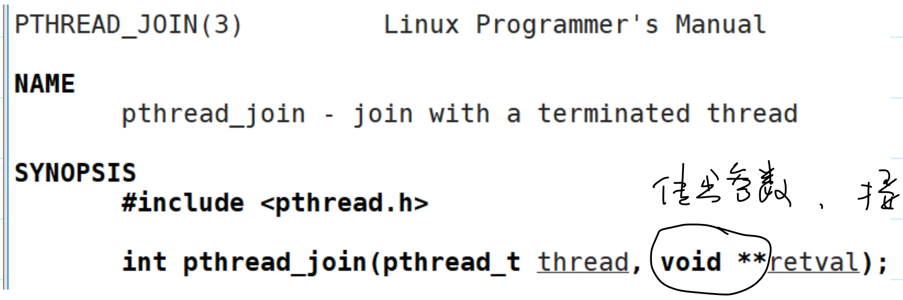
```
PTHREAD_EXIT(3)                 Linux Programmer's Manual

NAME
       pthread_exit - terminate calling thread

SYNOPSIS
       #include <pthread.h>

       void pthread_exit(void *retval);
```

void* retval：返回任意值给主线程。



pthread_join()

2024年5月24日 11:01



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

NAME
       pthread_join - join with a terminated thread

SYNOPSIS
       #include <pthread.h>

       int pthread_join(pthread_t thread, void **retval);
```

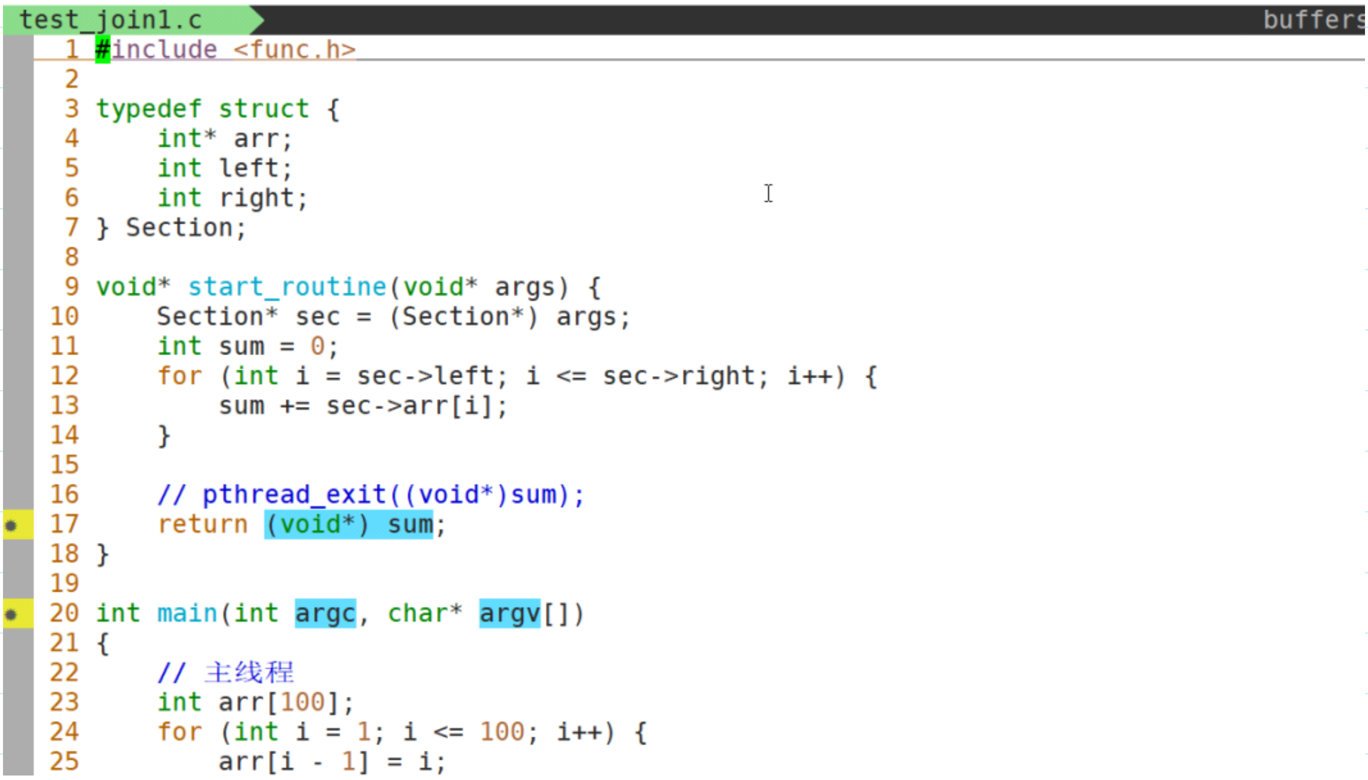
int**

传出参数，接收 void* 类型的值。

等待子线程结束，并接收子线程返回值。

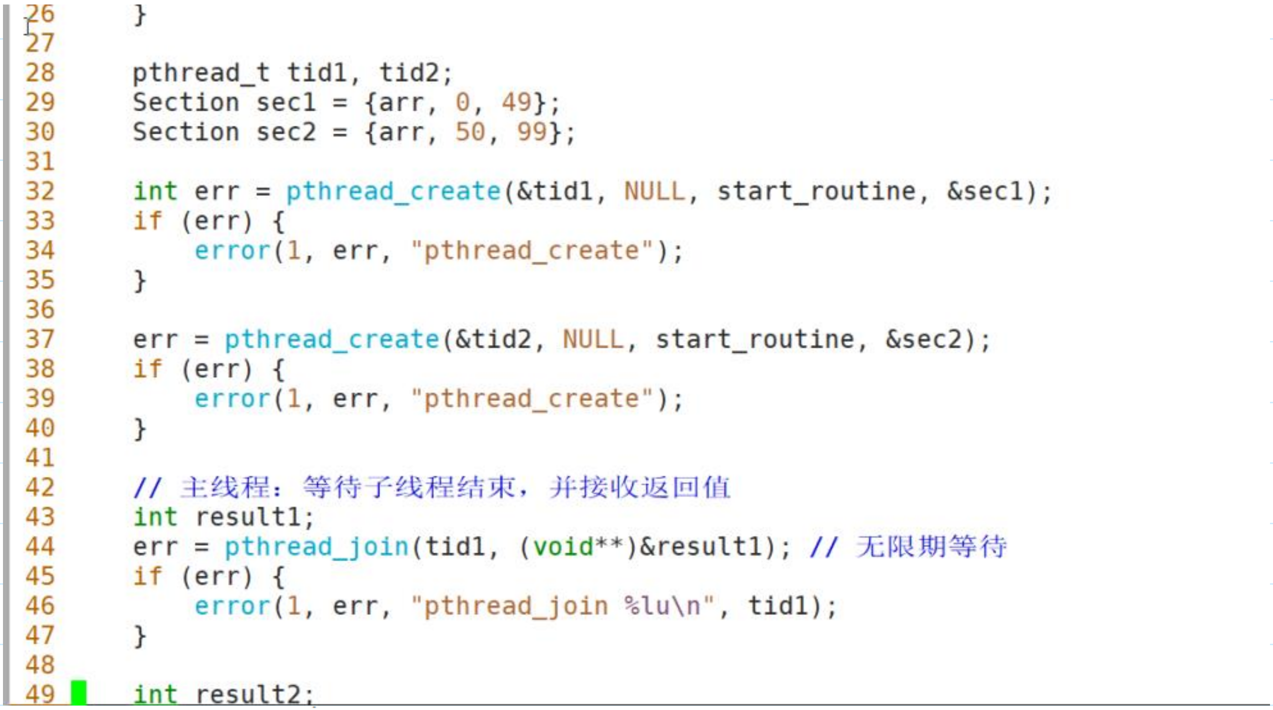
thread，等待哪个子线程结束

void** retval：接收子线程的返回值。

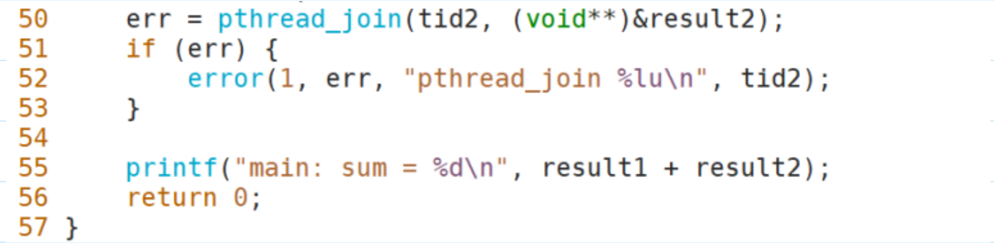


test_join1.c

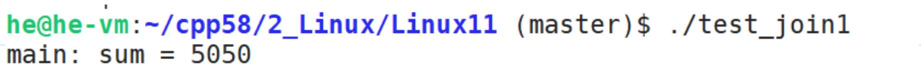
```
#include <func.h>

typedef struct {
    int* arr;
    int left;
    int right;
} Section;

void* start_routine(void* args) {
    Section* sec = (Section*) args;
    int sum = 0;
    for (int i = sec->left; i <= sec->right; i++) {
        sum += sec->arr[i];
    }

    // pthread_exit((void*)sum);
    return (void*) sum;
}

int main(int argc, char* argv[])
{
    // 主线程
    int arr[100];
    for (int i = 1; i <= 100; i++) {
        arr[i - 1] = i;
```



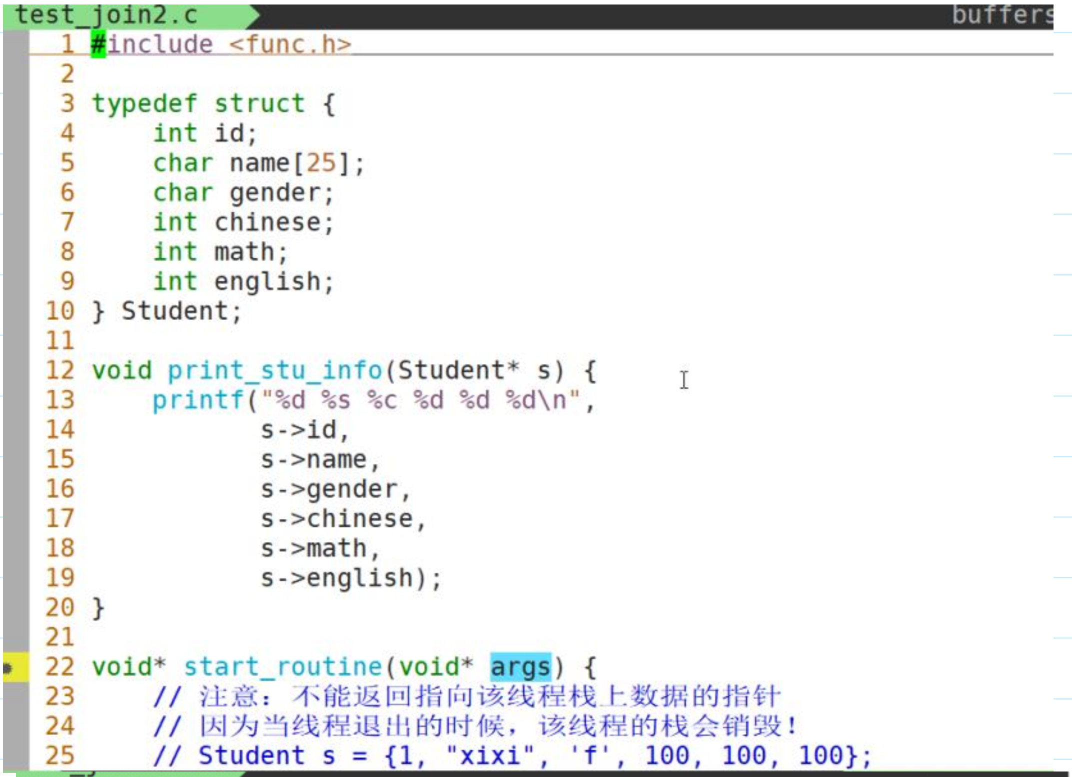
```c
    }

    pthread_t tid1, tid2;
    Section sec1 = {arr, 0, 49};
    Section sec2 = {arr, 50, 99};

    int err = pthread_create(&tid1, NULL, start_routine, &sec1);
    if (err) {
        error(1, err, "pthread_create");
    }

    err = pthread_create(&tid2, NULL, start_routine, &sec2);
    if (err) {
        error(1, err, "pthread_create");
    }

    // 主线程：等待子线程结束，并接收返回值
    int result1;
    err = pthread_join(tid1, (void**)&result1); // 无限期等待
    if (err) {
        error(1, err, "pthread_join %lu\n", tid1);
    }

    int result2;
```



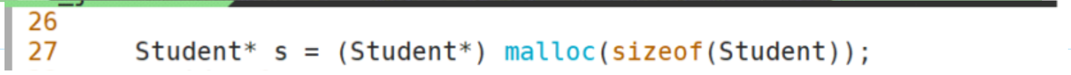
```c
    err = pthread_join(tid2, (void**)&result2);
    if (err) {
        error(1, err, "pthread_join %lu\n", tid2);
    }

    printf("main: sum = %d\n", result1 + result2);
    return 0;
}
```



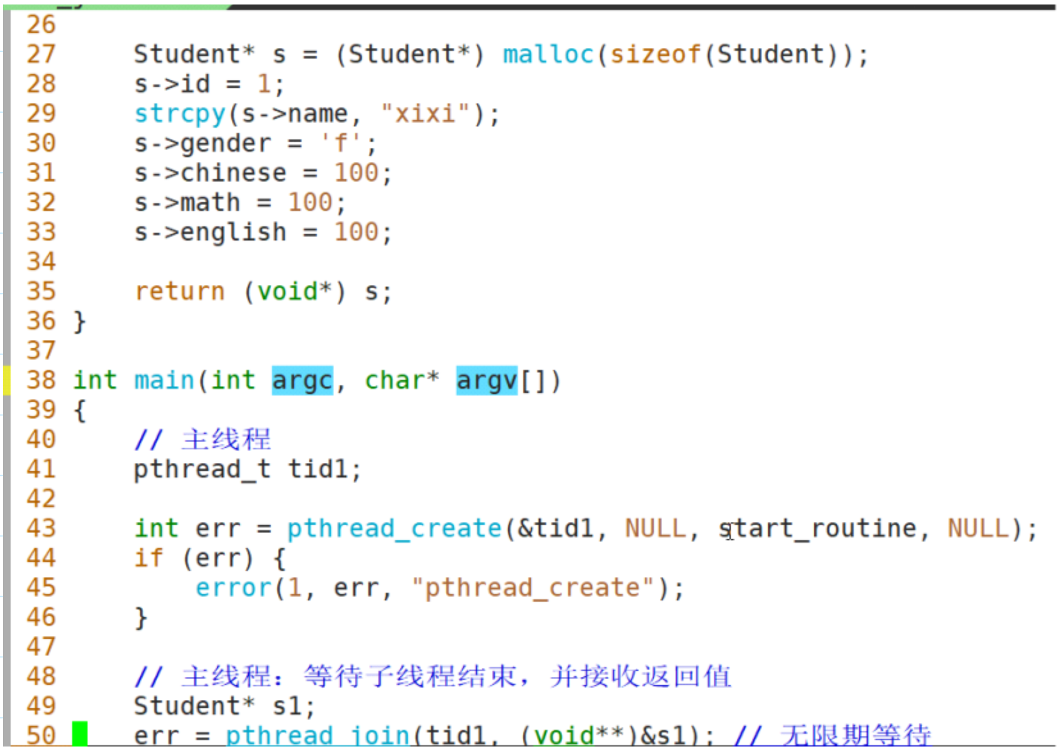
```
he@he-vm:~/cpp58/2_Linux/Linux11 (master)$ ./test_join1
main: sum = 5050
```



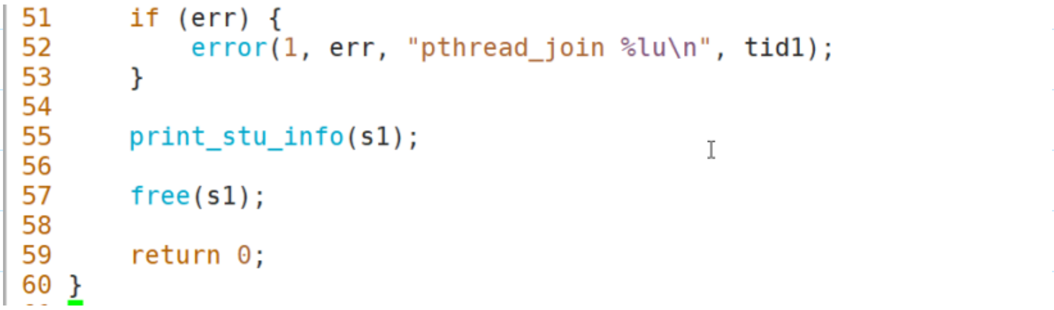
```c
#include <func.h>

typedef struct {
    int id;
    char name[25];
    char gender;
    int chinese;
    int math;
    int english;
} Student;

void print_stu_info(Student* s) {
    printf("%d %s %c %d %d %d\n",
           s->id,
           s->name,
           s->gender,
           s->chinese,
           s->math,
           s->english);
}

void* start_routine(void* args) {
    // 注意：不能返回指向该线程栈上数据的指针
    // 因为当线程退出的时候，该线程的栈会销毁！
    // Student s = {1, "xixi", 'f', 100, 100, 100};
```



```c

    Student* s = (Student*) malloc(sizeof(Student));
```



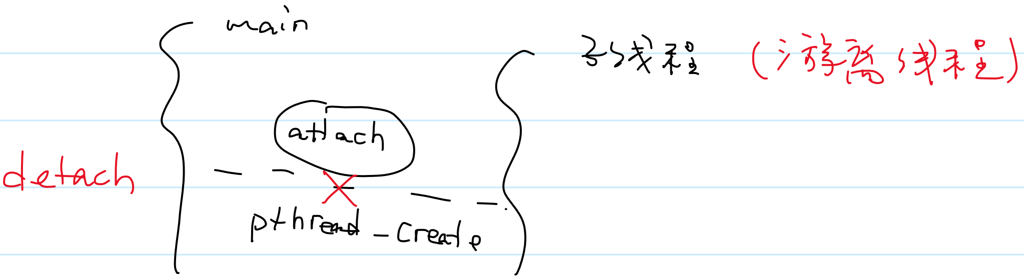
```
    Student* s = (Student*) malloc(sizeof(Student));
    s->id = 1;
    strcpy(s->name, "xixi");
    s->gender = 'f';
    s->chinese = 100;
    s->math = 100;
    s->english = 100;

    return (void*) s;
}

int main(int argc, char* argv[])
{
    // 主线程
    pthread_t tid1;

    int err = pthread_create(&tid1, NULL, start_routine, NULL);
    if (err) {
        error(1, err, "pthread_create");
    }

    // 主线程: 等待子线程结束, 并接收返回值
    Student* s1;
    err = pthread_join(tid1, (void**)&s1); // 无限期等待
```



```
    if (err) {
        error(1, err, "pthread_join %lu\n", tid1);
    }

    print_stu_info(s1);

    free(s1);

    return 0;
}
```

pthread_detach

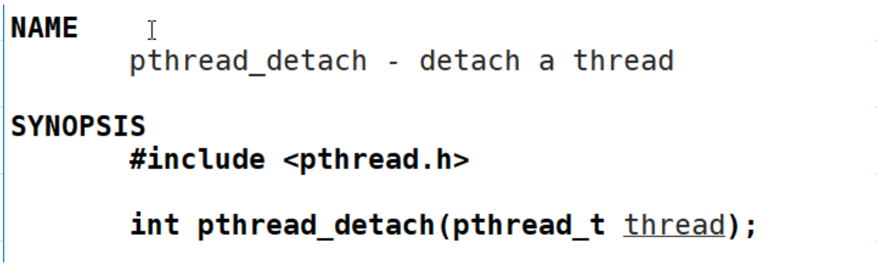
2024年5月24日 11:36



主线程才能接收子线程的返回值.

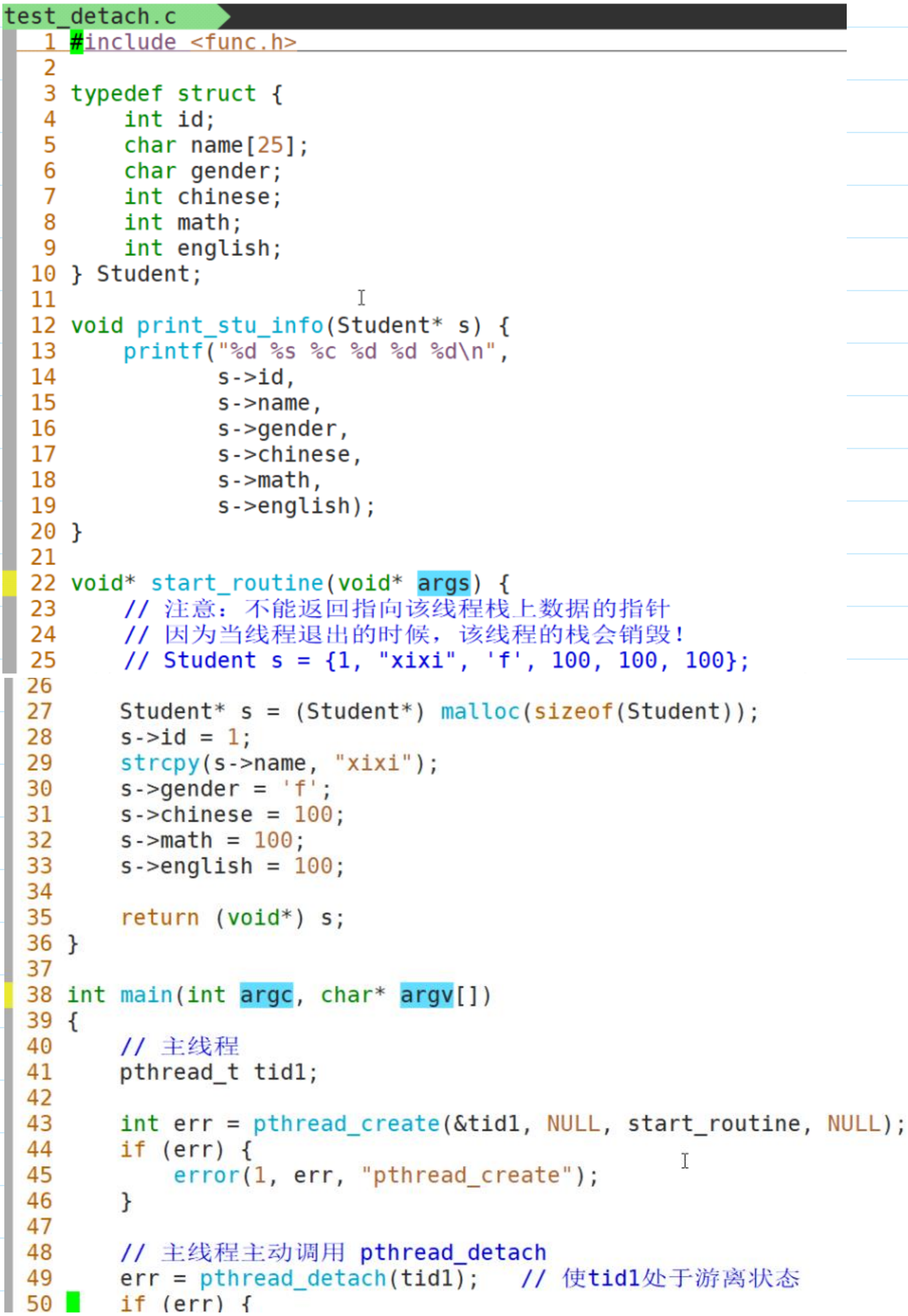
父进程 wait 子进程

主线程 pthread_join 子线程



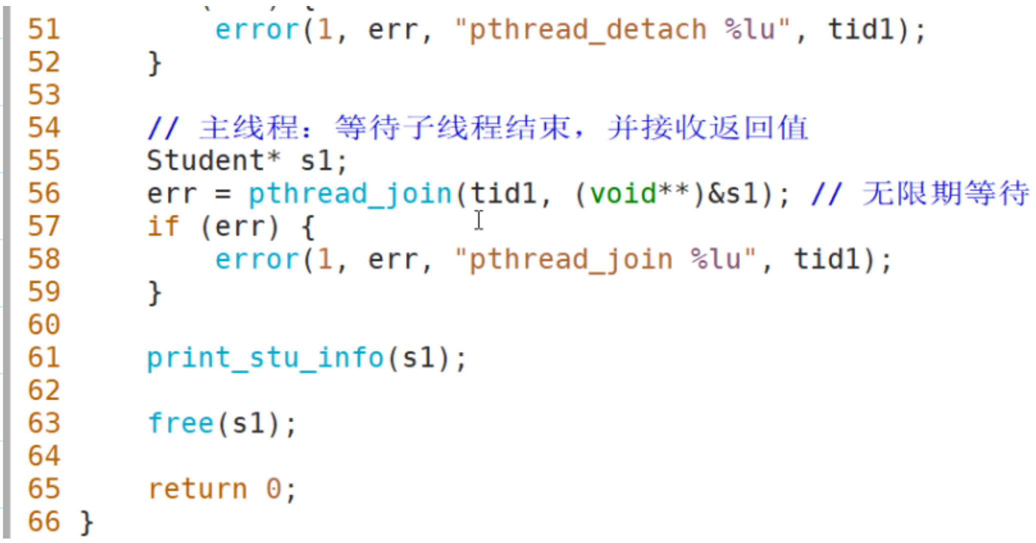
```
NAME
       pthread_detach - detach a thread

SYNOPSIS
       #include <pthread.h>

       int pthread_detach(pthread_t thread);
```

thread: 要游离的子线程.

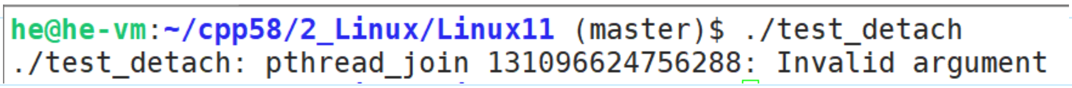


test_detach.c

```c
#include <func.h>

typedef struct {
    int id;
    char name[25];
    char gender;
    int chinese;
    int math;
    int english;
} Student;

void print_stu_info(Student* s) {
    printf("%d %s %c %d %d %d\n",
           s->id,
           s->name,
           s->gender,
           s->chinese,
           s->math,
           s->english);
}

void* start_routine(void* args) {
    // 注意：不能返回指向该线程栈上数据的指针
    // 因为当线程退出的时候，该线程的栈会销毁！
    // Student s = {1, "xixi", 'f', 100, 100, 100};

    Student* s = (Student*) malloc(sizeof(Student));
    s->id = 1;
    strcpy(s->name, "xixi");
    s->gender = 'f';
    s->chinese = 100;
    s->math = 100;
    s->english = 100;

    return (void*) s;
}

int main(int argc, char* argv[])
{
    // 主线程
    pthread_t tid1;

    int err = pthread_create(&tid1, NULL, start_routine, NULL);
    if (err) {
        error(1, err, "pthread_create");
    }

    // 主线程主动调用 pthread_detach
    err = pthread_detach(tid1);   // 使tid1处于游离状态
    if (err) {
```



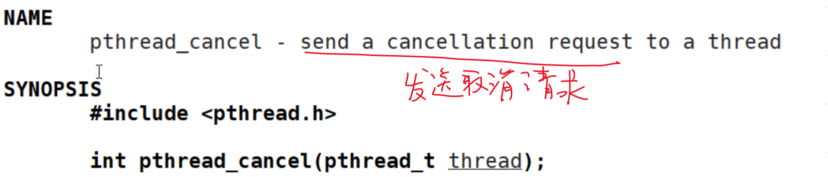
```c
        error(1, err, "pthread_detach %lu", tid1);
    }

    // 主线程：等待子线程结束，并接收返回值
    Student* s1;
    err = pthread_join(tid1, (void**)&s1); // 无限期等待
    if (err) {
        error(1, err, "pthread_join %lu", tid1);
    }

    print_stu_info(s1);

    free(s1);

    return 0;
}
```



```
he@he-vm:~/cpp58/2_Linux/Linux11 (master)$ ./test_detach
./test_detach: pthread_join 131096624756288: Invalid argument
```

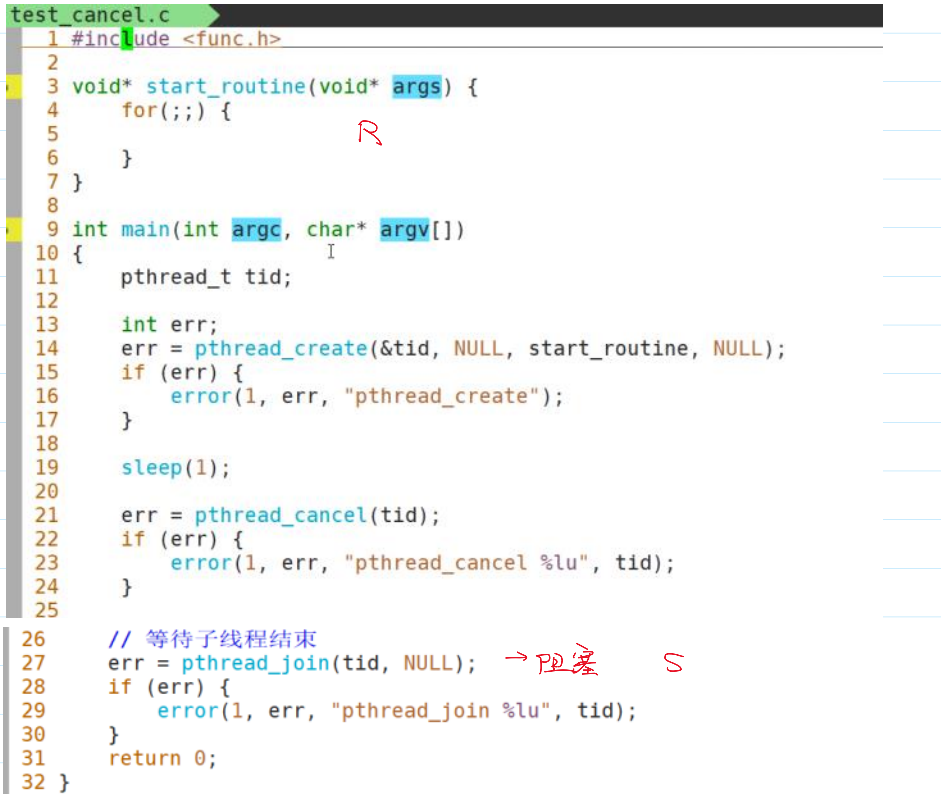
pthread_cancel (了解)

2024年5月24日 14:28



```
NAME
       pthread_cancel - send a cancellation request to a thread

SYNOPSIS
       #include <pthread.h>

       int pthread_cancel(pthread_t thread);
```

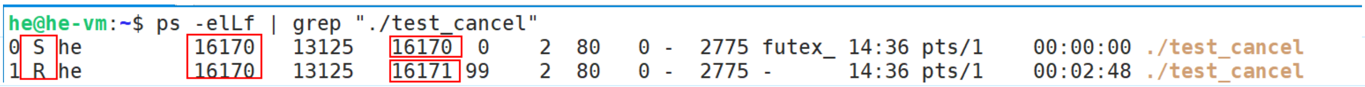
发送取消请求



```c
#include <func.h>

void* start_routine(void* args) {
    for(;;) {

    }
}

int main(int argc, char* argv[])
{
    pthread_t tid;

    int err;
    err = pthread_create(&tid, NULL, start_routine, NULL);
    if (err) {
        error(1, err, "pthread_create");
    }

    sleep(1);

    err = pthread_cancel(tid);
    if (err) {
        error(1, err, "pthread_cancel %lu", tid);
    }

    // 等待子线程结束
    err = pthread_join(tid, NULL);
    if (err) {
        error(1, err, "pthread_join %lu", tid);
    }
    return 0;
}
```

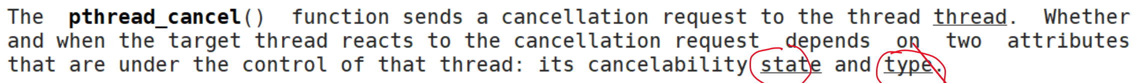
R

→阻塞 S



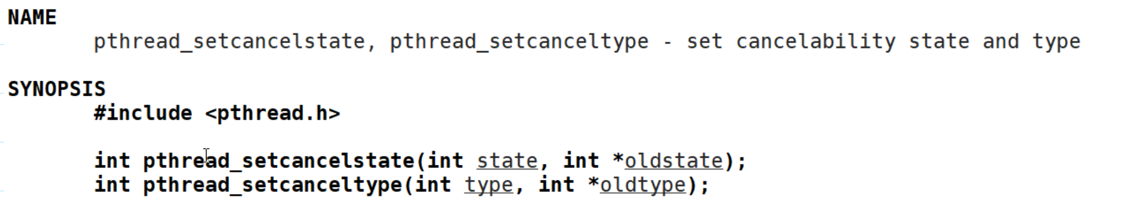
```
he@he-vm:~$ ps -elLf | grep "./test_cancel"
0 S he        16170  13125  16170  0    2  80    0 -  2775 futex_ 14:36 pts/1    00:00:00 ./test_cancel
1 R he        16170  13125  16171 99    2  80    0 -  2775 -      14:36 pts/1    00:02:48 ./test_cancel
```

会不会响应，以及何时响应，取决于线程属性.



```
The  pthread_cancel()  function sends a cancellation request to the thread thread.  Whether
and when the target thread reacts to the cancellation request  depends  on  two  attributes
that are under the control of that thread: its cancelability state and type.
```

取消状态 取消类型



```
NAME
       pthread_setcancelstate, pthread_setcanceltype - set cancelability state and type

SYNOPSIS
       #include <pthread.h>

       int pthread_setcancelstate(int state, int *oldstate);
       int pthread_setcanceltype(int type, int *oldtype);
```

线程属性.

CANCEL_STATE: 是否响应

PTHREAD_CANCEL_ENABLE （能够，默认值）

PTHREAD_CANCEL_DISABLE （不能够）

CANCEL_TYPE： 何时响应。

PTHREAD_CANCEL_DEFERRED （延迟响应，延迟到取消点才响应） 默认值

PTHREAD_CANCEL_ASYNCHRONOUS （在任何时刻都可以响应）

取消点：（可能会陷入长时间阻塞）

```
accept()
aio_suspend()
clock_nanosleep()
close()
connect()
creat()
fcntl() F_SETLKW
fdatasync()
fsync()
getmsg()
getpmsg()
lockf() F_LOCK
mq_receive()
mq_send()
mq_timedreceive()
mq_timedsend()
msgrcv()
msgsnd()
msync()
nanosleep()
open()
openat() [Added in POSIX.1-2008]
pause()
poll()
pread()
pselect()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_join()
pthread_testcancel()
putmsg()
putpmsg()
pwrite()
read()
readv()
recv()
recvfrom()
recvmsg()
select()
sem_timedwait()
sem_wait()
send()
sendmsg()
sendto()
sigpause() [POSIX.1-2001 only (moves to "may" list in POSIX.1-2008)]
sigsuspend()
sigtimedwait()
sigwait()
sigwaitinfo()
sleep()
system()
tcdrain()
usleep() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
wait()
waitid()
waitpid()
write()
writev()
```

检查是否有响应请求，如果有，就立刻响应。

线程清理函数

2024年5月24日 15:07

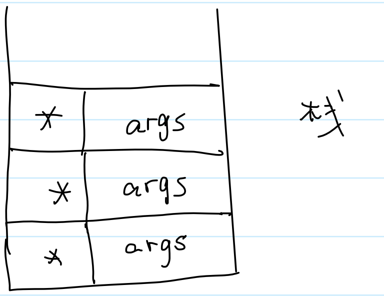
进程终止	
从main返回	(✓)
exit()	(✓)
响应信号	(✗)

atexit() 注册进程退出函数

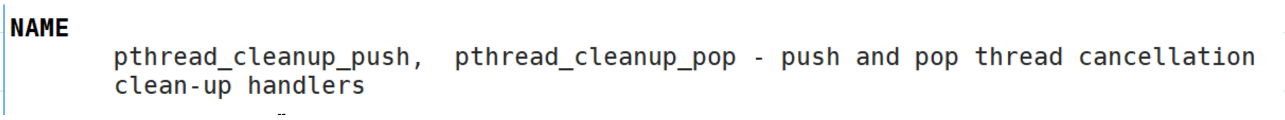
线程终止	
从 start_routine 返回	(✗)
pthread_exit()	(✓)
响应 pthread_cancel 请求	(✓)

pthread_cleanup_push()

pthread_cleanup_pop()

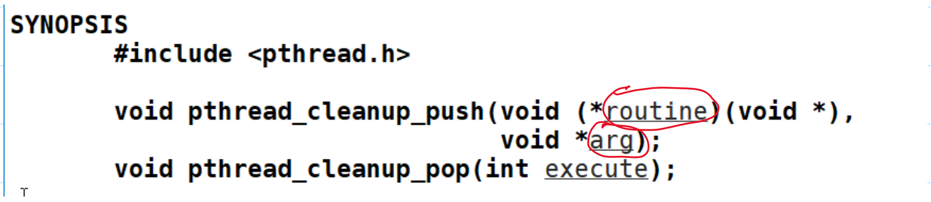


执行顺序与注册顺序相反



NAME

pthread_cleanup_push, pthread_cleanup_pop - push and pop thread cancellation clean-up handlers



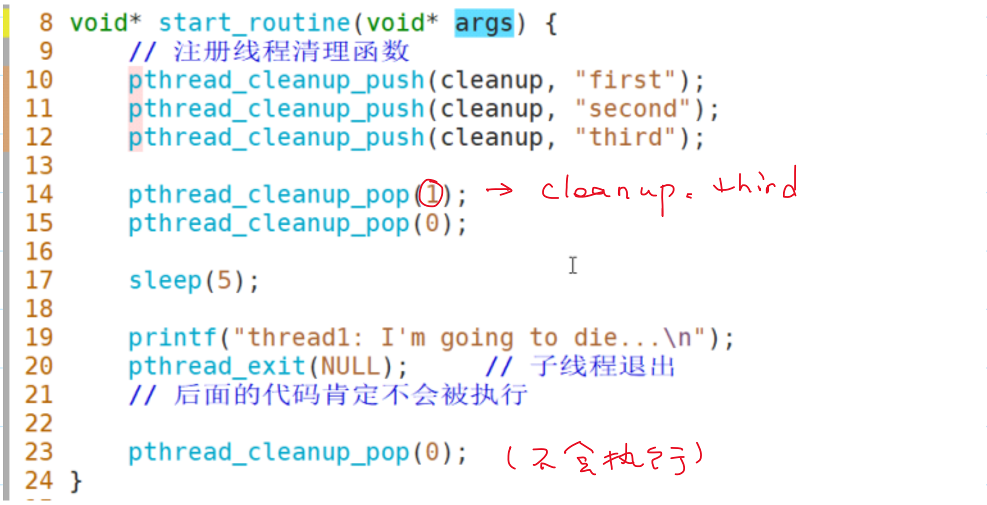
SYNOPSIS

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *),
                          void *arg);
void pthread_cleanup_pop(int execute);
```

execute：

0：不执行

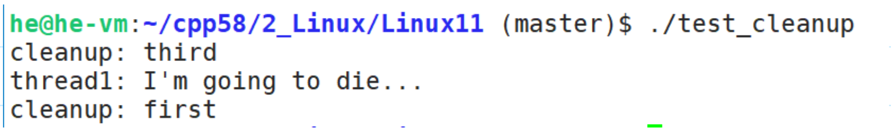
非0：执行



```c
void* start_routine(void* args) {
    // 注册线程清理函数
    pthread_cleanup_push(cleanup, "first");
    pthread_cleanup_push(cleanup, "second");
    pthread_cleanup_push(cleanup, "third");

    pthread_cleanup_pop(1);
    pthread_cleanup_pop(0);

    sleep(5);

    printf("thread1: I'm going to die...\n");
    pthread_exit(NULL);     // 子线程退出
    // 后面的代码肯定不会被执行

    pthread_cleanup_pop(0);
}
```

→ cleanup: third

(不会执行)



```
he@he-vm:~/cpp58/2_Linux/Linux11 (master)$ ./test_cleanup
cleanup: third
thread1: I'm going to die...
cleanup: first
```

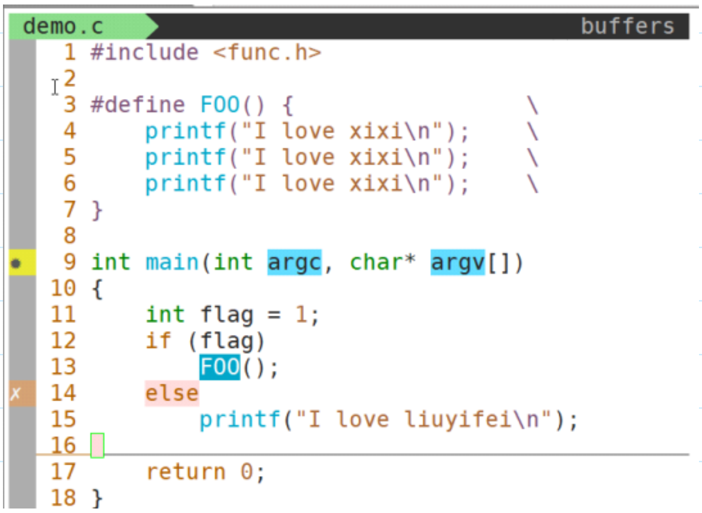
注意事项: ① 从 start_routine 返回, 不会执行线程清理函数.

② pthread_cleanup_push 和 pthread_cleanup_pop 必须成对出现

解释：宏函数

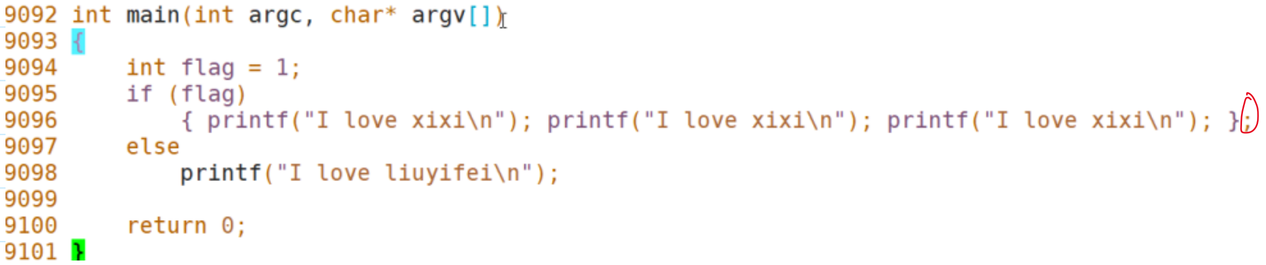
2024年5月24日 16:01

pthread_cleanup_push() 和 pthread_cleanup_pop 必须成对出现

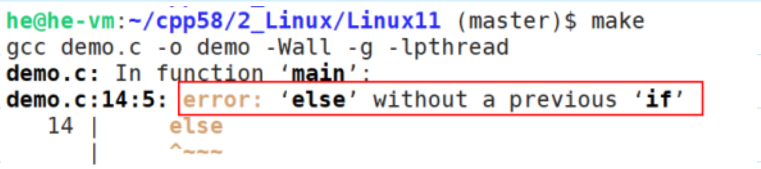


```
demo.c                                          buffers
 1 #include <func.h>
 2
 3 #define FOO() {                      \
 4     printf("I love xixi\n");         \
 5     printf("I love xixi\n");         \
 6     printf("I love xixi\n");         \
 7 }
 8
 9 int main(int argc, char* argv[])
10 {
11     int flag = 1;
12     if (flag)
13         FOO();
14     else
15         printf("I love liuyifei\n");
16
17     return 0;
18 }
```

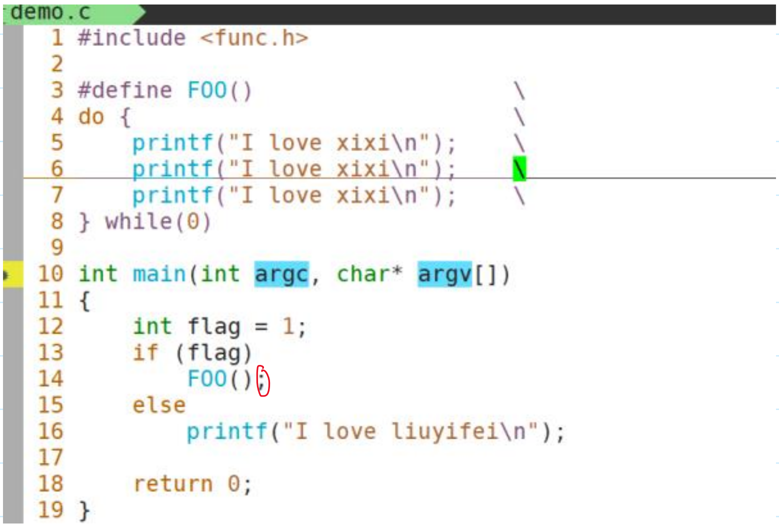
if (flag) {
} ;
else



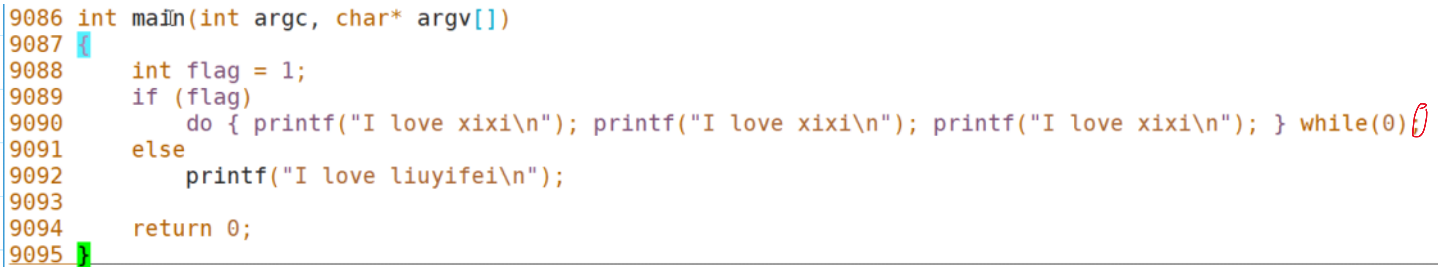
```
int main(int argc, char* argv[])
{
    int flag = 1;
    if (flag)
        { printf("I love xixi\n"); printf("I love xixi\n"); printf("I love xixi\n"); };
    else
        printf("I love liuyifei\n");

    return 0;
}
```



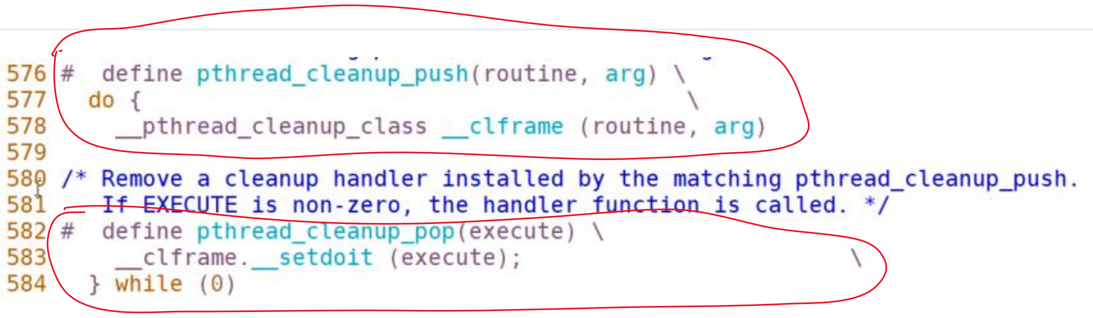
```
he@he-vm:~/cpp58/2_Linux/Linux11 (master)$ make
gcc demo.c -o demo -Wall -g -lpthread
demo.c: In function ‘main’:
demo.c:14:5: error: ‘else’ without a previous ‘if’
   14 |     else
      |     ^~~~
```



```
demo.c
#include <func.h>

#define FOO()                         \
do {                                  \
    printf("I love xixi\n");          \
    printf("I love xixi\n");          \
    printf("I love xixi\n");          \
} while(0)

int main(int argc, char* argv[])
{
    int flag = 1;
    if (flag)
        FOO();
    else
        printf("I love liuyifei\n");

    return 0;
}
```



```c
int main(int argc, char* argv[])
{
    int flag = 1;
    if (flag)
        do { printf("I love xixi\n"); printf("I love xixi\n"); printf("I love xixi\n"); } while(0);
    else
        printf("I love liuyifei\n");

    return 0;
}
```

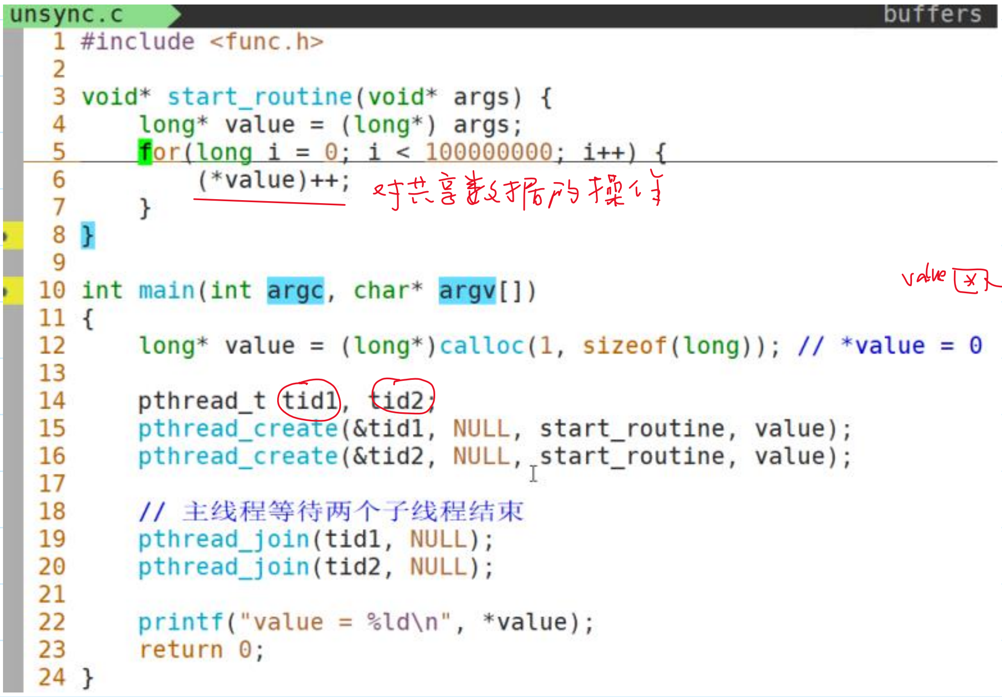


```c
#  define pthread_cleanup_push(routine, arg) \
  do {                                       \
    __pthread_cleanup_class __clframe (routine, arg)

/* Remove a cleanup handler installed by the matching pthread_cleanup_push.
   If EXECUTE is non-zero, the handler function is called. */
#  define pthread_cleanup_pop(execute) \
    __clframe.__setdoit (execute);                      \
  } while (0)
```

必须成对出现

线程的同步

2024年5月24日 14:21

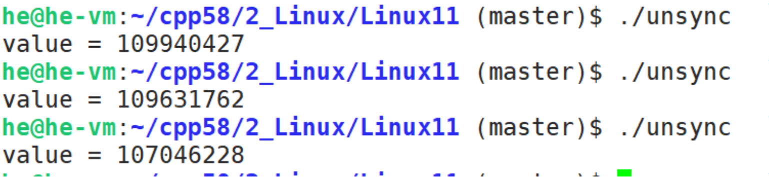


unsync.c　　buffers

```c
#include <func.h>

void* start_routine(void* args) {
    long* value = (long*) args;
    for(long i = 0; i < 100000000; i++) {
        (*value)++;
    }
}

int main(int argc, char* argv[])
{
    long* value = (long*)calloc(1, sizeof(long)); // *value = 0

    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, start_routine, value);
    pthread_create(&tid2, NULL, start_routine, value);

    // 主线程等待两个子线程结束
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("value = %ld\n", *value);
    return 0;
}
```

对共享数据的操作

value [*] → 共享数据



```
he@he-vm:~/cpp58/2_Linux/Linux11 (master)$ ./unsync
value = 109940427
he@he-vm:~/cpp58/2_Linux/Linux11 (master)$ ./unsync
value = 109631762
he@he-vm:~/cpp58/2_Linux/Linux11 (master)$ ./unsync
value = 107046228
```

思考: ① 为什么结果不正确?　　++操作不是原子性

② 为什么结果每一次都不一样?　　调度是不确定的

原子性.　　cpu 指令是原子的

instruction 1
→ 切换
instruction 2
→ 切换
instruction 3
→ 切换

i++:

```
Load   $r , i
ADD    $r , 1
Store  i , $r
```

i = 1

```
           tid1                                         tid2

                 0
 |     Load   $r ,  i  → 切换
 |                                                         0
 |                                             Load   $r ,  i
 |                                             ADD    $r ,  1
 t                                      切换 ← store  i  ,  $r 1
 |
 |              0
 |     ADD    $r ,  1
 |     store  i  ,  $r 1
 v
```

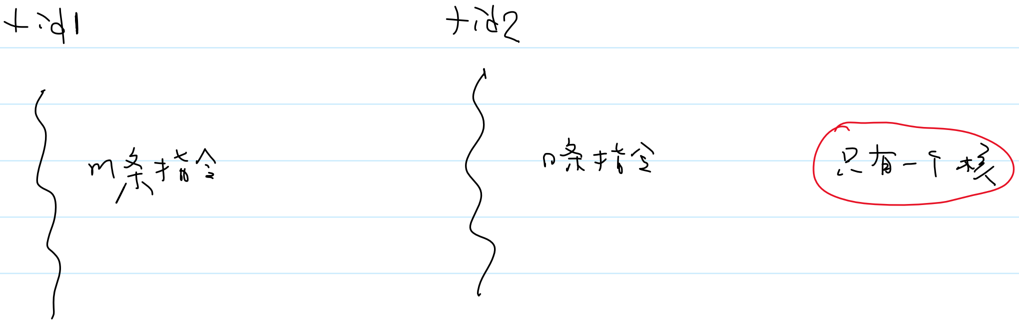
#2. 术语

程序并发执行时，

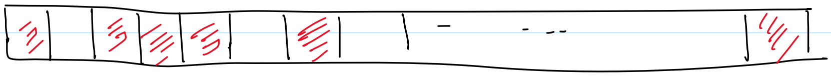
竞态条件：(race condition)

1. 多个执行流程，
2. 共享资源
3. 程序的结果（状态）取决于执行流程调度的情况，

↓ 为什么并发问题比较困难



调度：C_{m+n}^{m} 种情况



异步和同步.

异步：任何调度情况都可能出现 （速度快）

（两个执行流程不作任何交流）

同步：让一些调度不可能出现 （同步会有一些开销）

（两个执行流程不能同时运行）

同步：让一些调度不可能出现。（同步会有一些开销）

互斥锁
条件变量

并发：一种现象：在一个时间段中，执行流程可以交替执行

一个CPU，一个核

线程1	线程2
i1	
	i1
	i2
i2	
i3	
	i3

并行：一种技术。同一时刻，可以执行多个执行流程。（并行是并发一种）

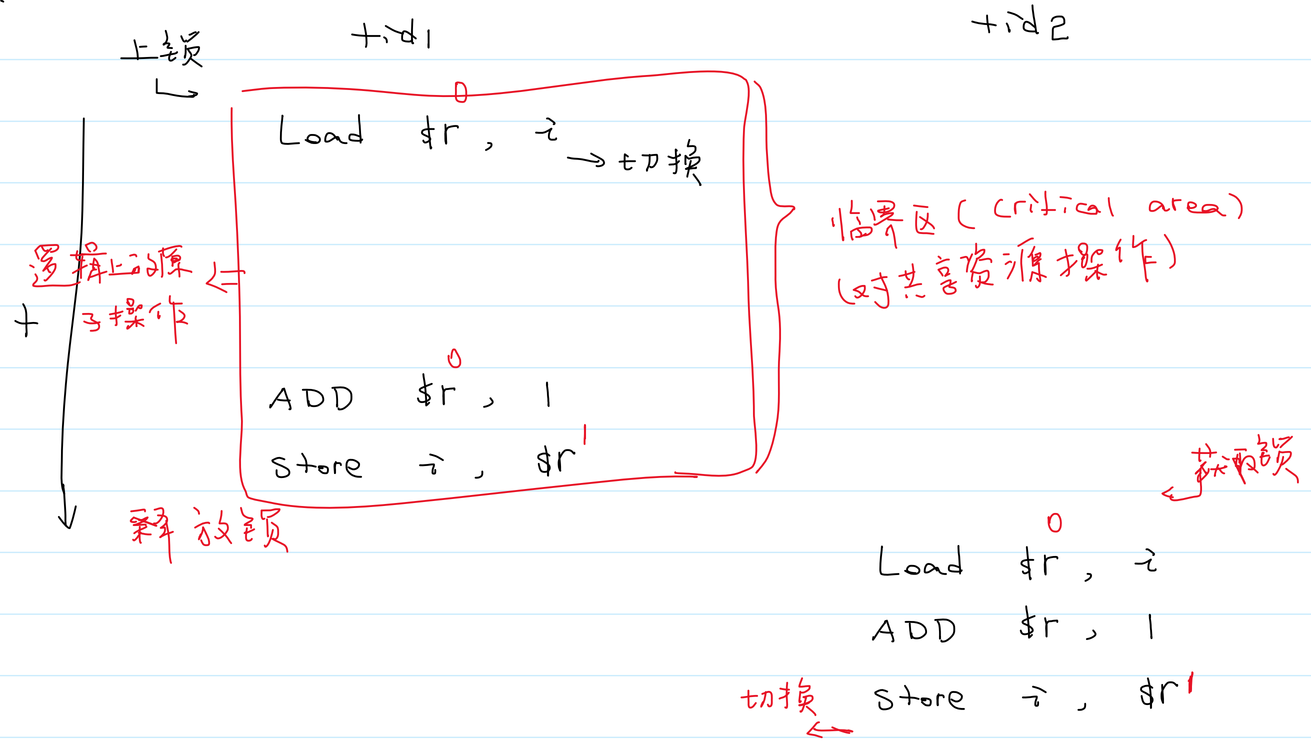
线程1	线程2
i1	i1
i2	i2
i3	i3

同步

2024年5月24日 17:22

① 互斥地访问资源. ② 等待某个条件成立.

i++, ⇐ 操作共享资源



tid1:

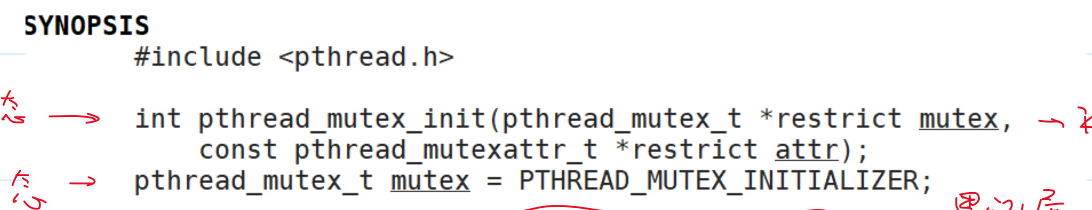
```
Load  $r , i  → 切换
ADD   $r , 1
Store i  , $r
```

tid2:

```
      Load  $r , i
      ADD   $r , 1
切换 ← Store i  , $r
```

互斥锁

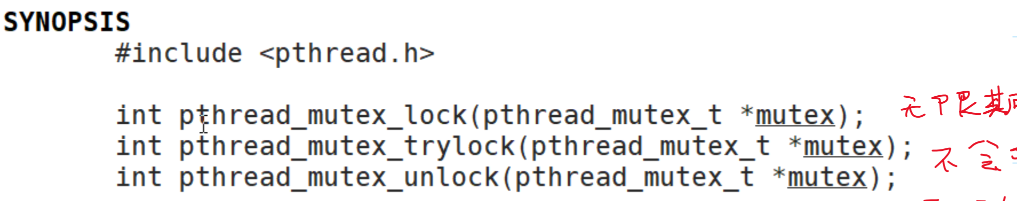
2024年5月24日 16:38



```
SYNOPSIS
       #include <pthread.h>

       int pthread_mutex_init(pthread_mutex_t *restrict mutex,
           const pthread_mutexattr_t *restrict attr);
       pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

动态 → 初始化

静态 → 默认属性

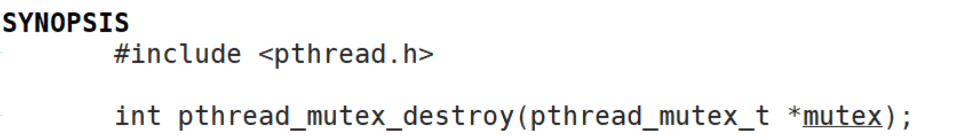


```
SYNOPSIS
       #include <pthread.h>

       int pthread_mutex_lock(pthread_mutex_t *mutex);
       int pthread_mutex_trylock(pthread_mutex_t *mutex);
       int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

无限期阻塞

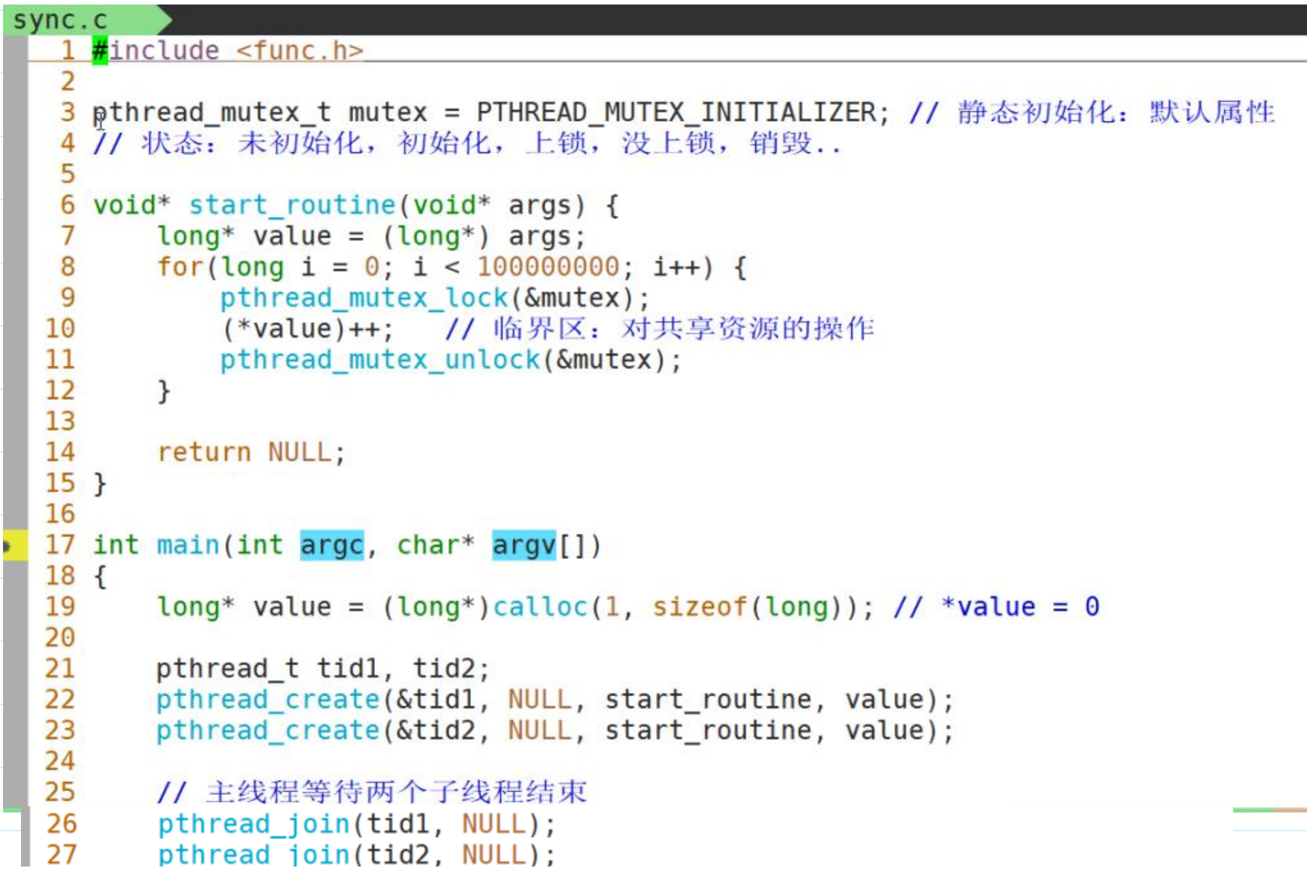
不会阻塞

释放锁



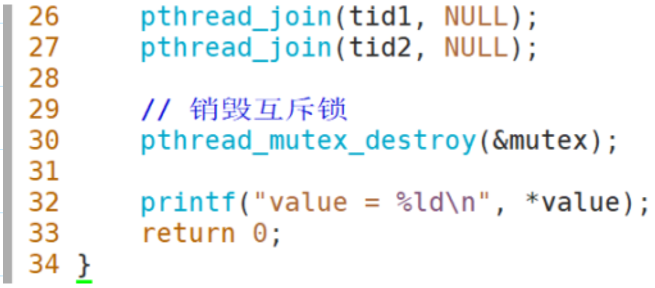
```
SYNOPSIS
       #include <pthread.h>

       int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

销毁

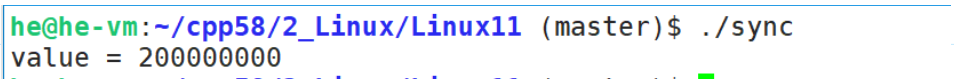


sync.c

```c
#include <func.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 静态初始化：默认属性
// 状态：未初始化，初始化，上锁，没上锁，销毁..

void* start_routine(void* args) {
    long* value = (long*) args;
    for(long i = 0; i < 100000000; i++) {
        pthread_mutex_lock(&mutex);
        (*value)++;    // 临界区：对共享资源的操作
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

int main(int argc, char* argv[])
{
    long* value = (long*)calloc(1, sizeof(long)); // *value = 0

    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, start_routine, value);
    pthread_create(&tid2, NULL, start_routine, value);

    // 主线程等待两个子线程结束
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
```



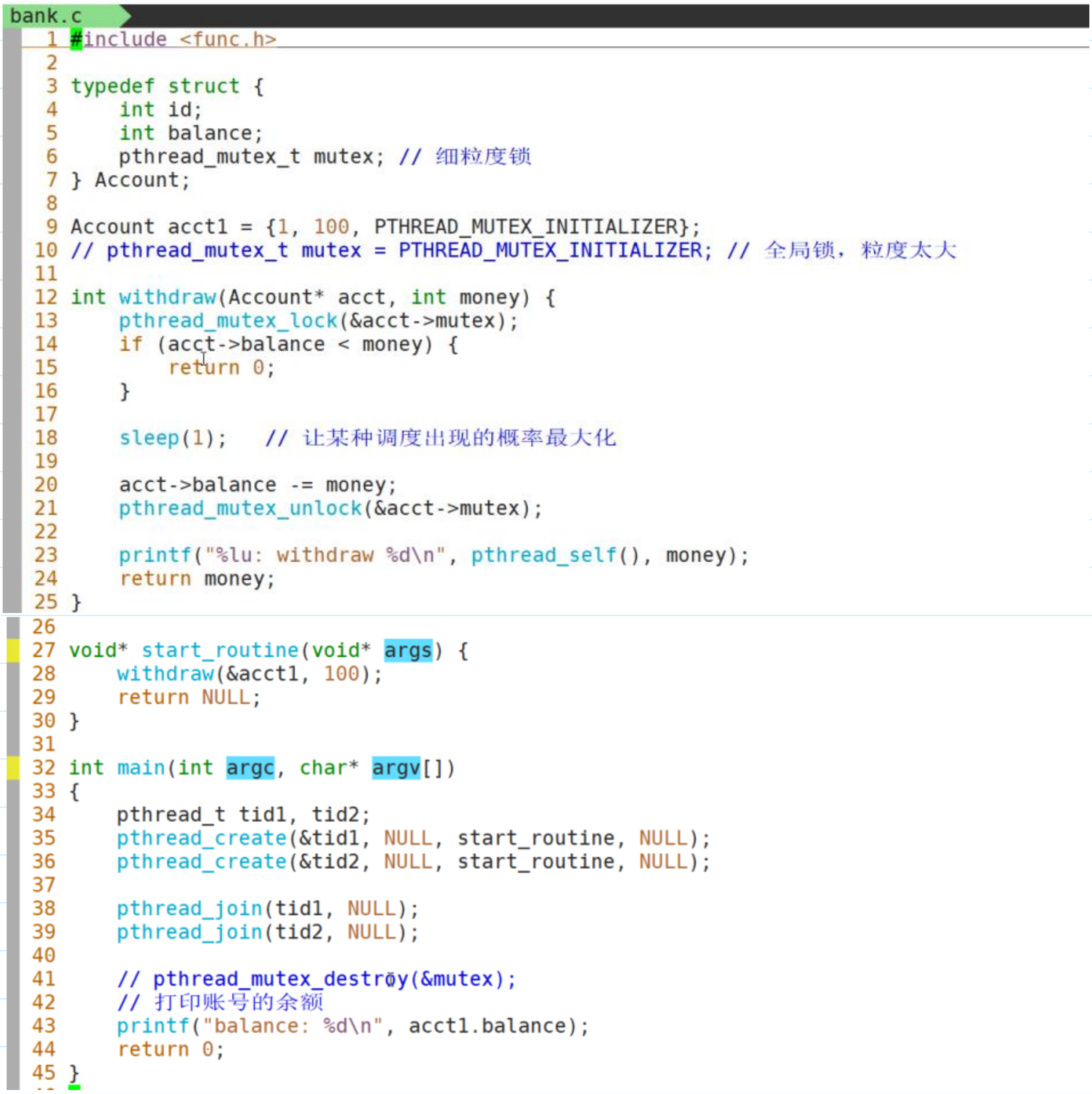
```c
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    // 销毁互斥锁
    pthread_mutex_destroy(&mutex);

    printf("value = %ld\n", *value);
    return 0;
}
```



```
he@he-vm:~/cpp58/2_Linux/Linux11 (master)$ ./sync
value = 200000000
```

银行例子

2024年5月24日 17:44



bank.c

```c
#include <func.h>

typedef struct {
    int id;
    int balance;
    pthread_mutex_t mutex; // 细粒度锁
} Account;

Account acct1 = {1, 100, PTHREAD_MUTEX_INITIALIZER};
// pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 全局锁，粒度太大

int withdraw(Account* acct, int money) {
    pthread_mutex_lock(&acct->mutex);
    if (acct->balance < money) {
        return 0;
    }

    sleep(1);   // 让某种调度出现的概率最大化

    acct->balance -= money;
    pthread_mutex_unlock(&acct->mutex);

    printf("%lu: withdraw %d\n", pthread_self(), money);
    return money;
}

void* start_routine(void* args) {
    withdraw(&acct1, 100);
    return NULL;
}

int main(int argc, char* argv[])
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, start_routine, NULL);
    pthread_create(&tid2, NULL, start_routine, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    // pthread_mutex_destroy(&mutex);
    // 打印账号的余额
    printf("balance: %d\n", acct1.balance);
    return 0;
}
```