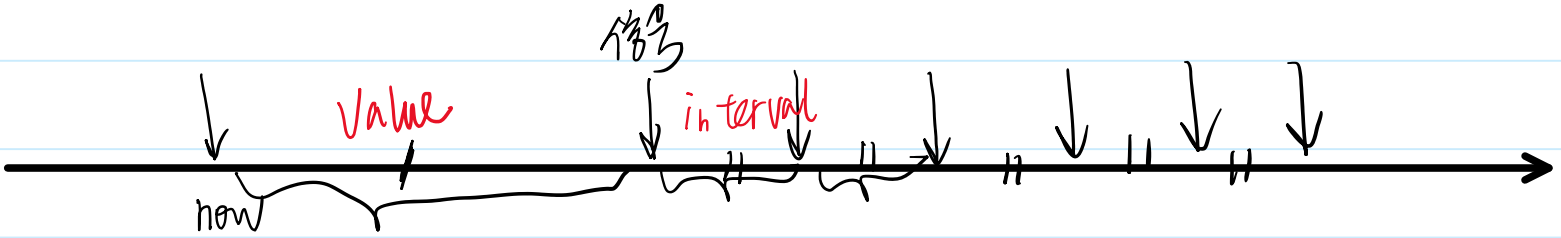
间隔定时器

2023年8月15日 9:30

```
int setitimer(int which, const struct itimerval *new value,
              struct itimerval *old value);
```




```
struct itimerval {
    struct timeval it_interval; /* Interval for periodic timer */
    struct timeval it_value;    /* Time until next expiration */
};
```

ITIMER_REAL 墙上时间

ITIMER_VIRTUAL 只有占用用户cpu才计时.

ITIMER_PROF 实用时钟. 用户cpu+内核cpu

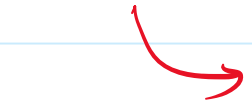
profile

写代码的要求

2023年8月15日 10:07

正确 > 可读性 > 性能

过早优化是万恶之源.



① 先写好优雅的代码.

② 压力测试. profile

③ 找到瓶颈.

2023年8月15日 10:27

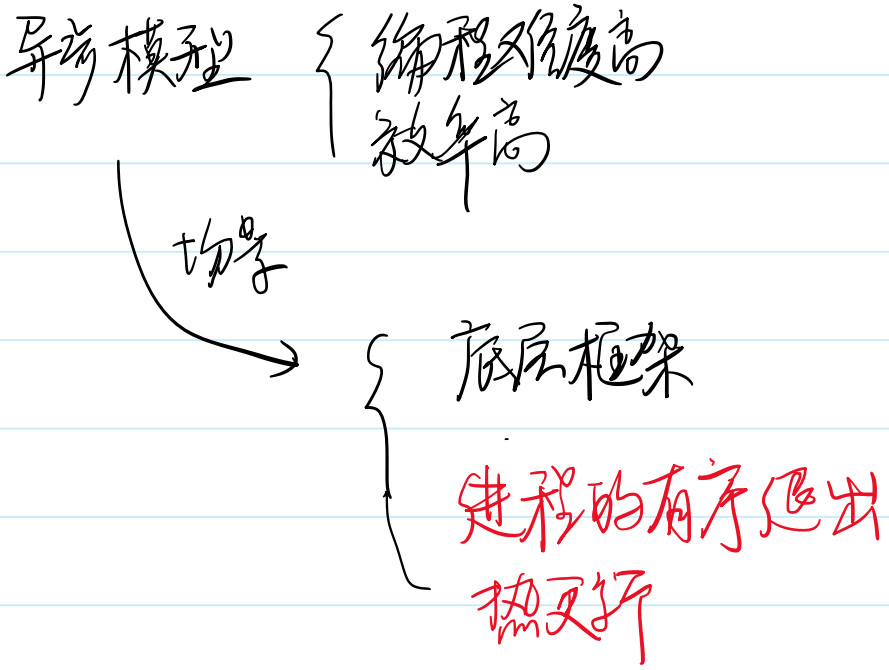
```c
#include <52func.h>
void handler(int signum){
    printf("signum = %d\n", signum);
    time_t now = time(NULL);
    printf("curtime = %s\n", ctime(&now));
}
int main(int argc, char *argv[])
{
    struct itimerval itimer;
    itimer.it_value.tv_sec = 3;
    itimer.it_value.tv_usec = 0;
    itimer.it_interval.tv_sec = 2;
    itimer.it_interval.tv_usec = 0;

    //setitimer(ITIMER_REAL,&itimer,NULL);
    setitimer(ITIMER_PROF,&itimer,NULL);
    //signal(SIGALRM,handler);
    signal(SIGPROF,handler);
    handler(0);
    while(1){

    }
    return 0;
}
```

信号

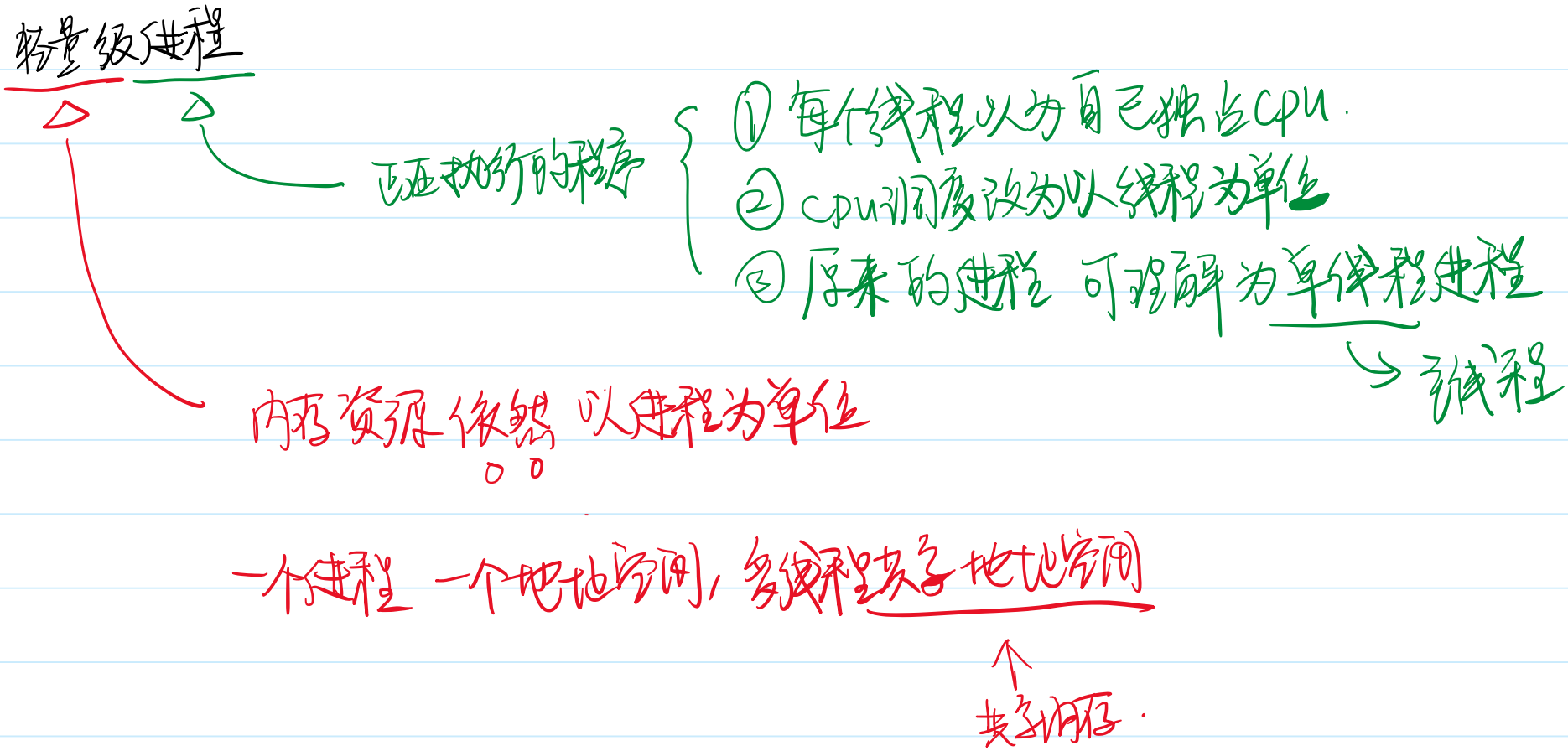
2023年8月15日 10:27



去多次使用协程. 排行榜.

线程

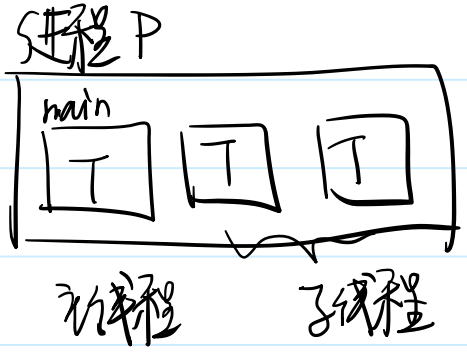
2023年8月15日 11:01



进程和线程之间的关系

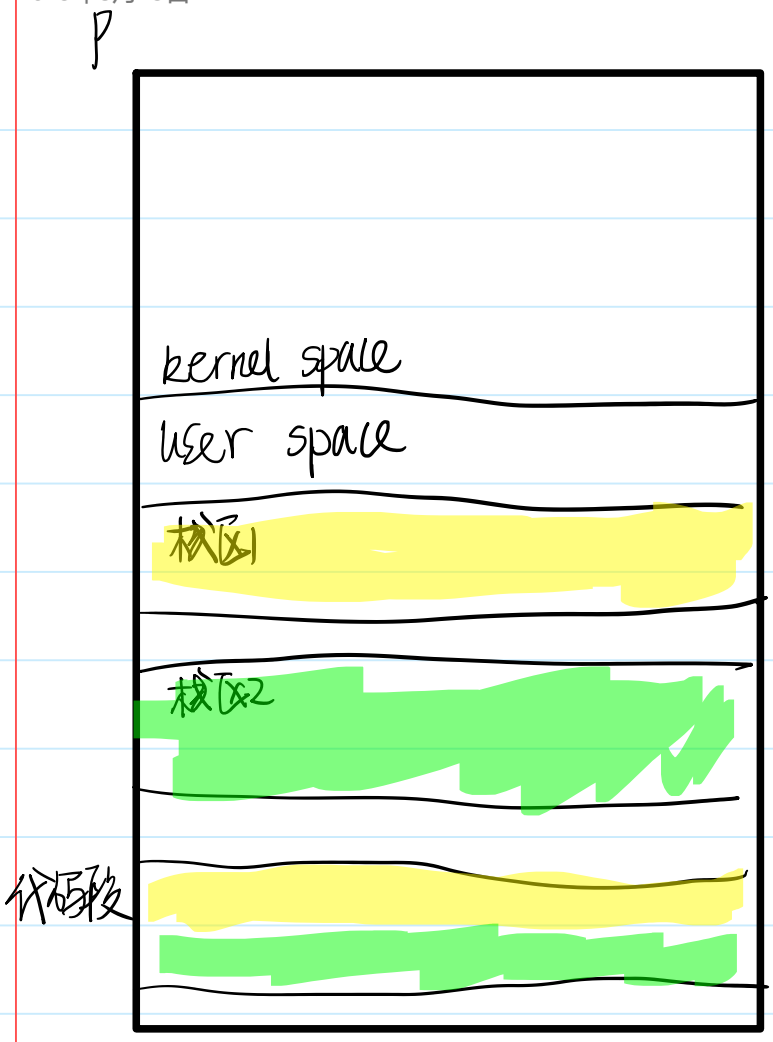
2023年8月15日 11:11

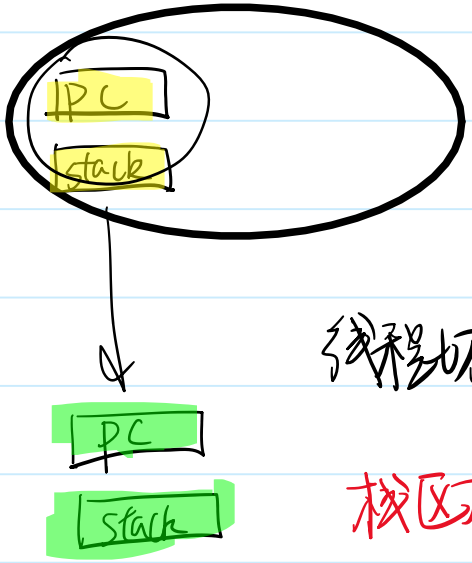
在Linux中，线程不能脱离进程单独存在.



多线程下的内存模型

2023年8月15日 11:13





线程切换效率高

栈区相互独立

① 压栈弹栈独立

② 共享地址空间.

/proc

2023年8月15日 11:25

```
/proc
liao:proc$ l
1/      11073/  11227/  11447/  131/   207/  229/   248/   266/   3/    5/     728/  81/   97/             iomem         pressure/
10/     11079/  11228/  11450/  132/   209/  230/   249/   26648/ 30/   531/   743/  813/  98/             ioports       schedstat
100/    11086/  11230/  1147/   133/   21/   231/   25/    267/   31/   538/   745/  82/   99/             irq/          scsi/
101/    111/    11232/  115/    134/   210/  232/   250/   27/    317/  5954/  771/  83/   acpi/           kallsyms      self@
102/    11100/  11236/  11587/  136/   211/  233/   251/   27075/ 318/  6/     772/  838/  asound/         kcore         slabinfo
1028/   11118/  11243/  116/    137/   212/  23341/ 252/   27150/ 32/   6056/  775/  85/   bootconfig      keys          softirqs
103/    11122/  11244/  117/    138/   213/  234/   253/   27151/ 33/   6198/  776/  86/   buddyinfo       key-users     stat
104/    11123/  11248/  118/    139/   214/  23422/ 254/   27161/ 34/   6202/  777/  87/   bus/            kmsg          swaps
105/    11127/  11250/  119/    14/    215/  23423/ 255/   27214/ 357/  6215/  779/  88/   cgroups         kpagecgroup   sys/
106/    11130/  11252/  12/     15/    216/  235/   256/   27215/ 385/  6217/  781/  885/  cmdline         kpagecount    sysrq-trigger
107/    11139/  11258/  120/    151/   217/  236/   257/   27225/ 4/    6241/  7864/ 887/  consoles        kpageflags    sysvipc/
1071/   11143/  11262/  121/    154/   218/  237/   258/   27278/ 400/  6267/  787/  89/   cpuinfo         loadavg       thread-self@
108/    11145/  11275/  1210/   155/   219/  238/   259/   27279/ 401/  6288/  788/  90/   crypto          locks         timer_list
109/    11154/  11286/  122/    16/    22/   239/   26/    27319/ 402/  6310/  790/  908/  devices         mdstat        tty/
10953/  11163/  11295/  123/    160/   220/  24/    260/   27320/ 403/  6342/  792/  91/   diskstats       meminfo       uptime
10967/  11179/  11298/  124/    1645/  221/  240/   261/   27423/ 404/  6350/  7939/ 914/  dma             misc          version
10972/  11182/  113/    125/    1703/  222/  241/   26184/ 27427/ 405/  6367/  794/  93/   driver/         modules       version_signature
10975/  112/    11302/  126/    18/    223/  242/   26185/ 27456/ 406/  6378/  7940/ 94/   dynamic_debug/  mounts@       vmallocinfo
10984/  11203/  11306/  127/    19/    224/  243/   262/   27458/ 407/  6391/  795/  943/  execdomains     mpt/          vmstat
11/     11210/  11313/  128/    2/     225/  244/   263/   27466/ 408/  6409/  796/  951/  fb              mtrr          zoneinfo
110/    11224/  114/    1286/   20/    226/  245/   264/   29/    409/  6448/  798/  953/  filesystems     net@
11051/  11225/  11415/  129/    204/   227/  246/   265/   294/   410/  6456/  799/  955/  fs/             pagetypeinfo
11068/  11226/  11423/  13/     205/   228/  247/   26582/ 295/   418/  727/   8/     96/   interrupts      partitions
```

2023年8月15日 11:28

```
-r--r--r--   1 mysql mysql 0 Aug 10 09:56 wchan
liao:1028$ cd task/
liao:task$ ll
total 0
dr-xr-xr-x 40 mysql mysql 0 Aug 15 11:28 ./
dr-xr-xr-x  9 mysql mysql 0 Aug 10 09:30 ../
dr-xr-xr-x  7 mysql mysql 0 Aug 15 11:28 1028/
dr-xr-xr-x  7 mysql mysql 0 Aug 15 11:28 1588/
dr-xr-xr-x  7 mysql mysql 0 Aug 15 11:28 1589/
dr-xr-xr-x  7 mysql mysql 0 Aug 15 11:28 1590/
dr-xr-xr-x  7 mysql mysql 0 Aug 15 11:28 1591/
```

man 7 pthreads

2023年8月15日 11:29

POSIX

Linux implementations of POSIX threads

Over time, two threading implementations have been provided by the GNU C library on Linux:

LinuxThreads

This is the original Pthreads implementation. Since glibc 2.4, this implementation is no longer supported.

NPTL (Native POSIX Threads Library)

This is the modern Pthreads implementation. By comparison with LinuxThreads, NPTL provides closer conformance to the requirements of the POSIX.1 specification and better performance when creating large numbers of threads. NPTL is available since glibc 2.3.2, and requires features that are present in the Linux 2.6 kernel.

Both of these are so-called 1:1 implementations, meaning that each thread maps to a kernel scheduling entity. Both threading implementations employ the Linux **clone**(2) system call. In NPTL, thread synchronization primitives (mutexes, thread joining, and so on) are implemented using the Linux **futex**(2) system call.

用户级线程 用户实现线程(PC, stack..). OS调度进程, 进程内的代码切换线程.

内核级线程 OS调单位是线程

协程、纤程、虚拟线程

坏处: 多核支持不好

好处: 切换无systemcall

创建子线程

2023年8月15日 11:45

线程：进程从main开始启动

```
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);

       Compile and link with -pthread.
```

pthread_t 线程ID类型

线程标识

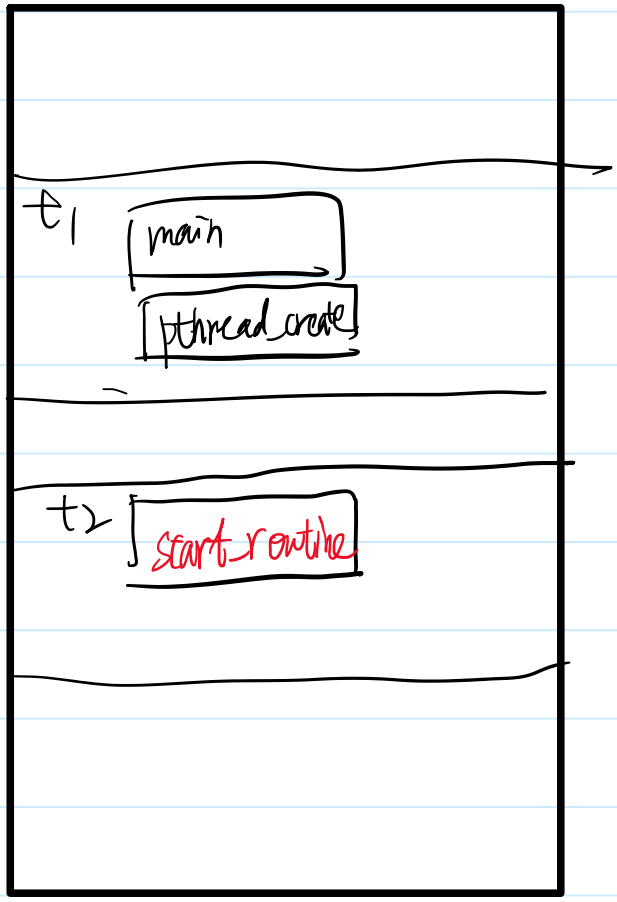
NULL 默认属性

start_routine 是一个函数指针

arg 是给 start_routine 传递的参数

start_routine

2023年8月15日 11:49



① start_routine不是main调用的.

② start_routine之于子线程 → 线程入口函数.

main 之于主线程

链接选项

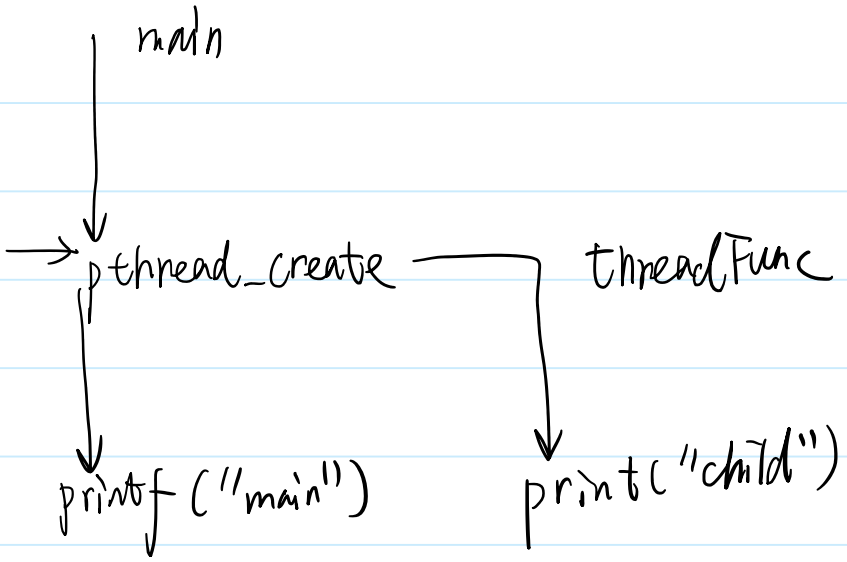
2023年8月15日 11:58

```
liao:LinuxDay17$ make
gcc 2_pthread_create.c -o 2_pthread_create -g
/usr/bin/ld: /tmp/ccdftu0S.o: in function `main':
/home/liao/cpp52/2_Linux/LinuxDay17/2_pthread_create.c:8: undefined ref
erence to `pthread_create'
collect2: error: ld returned 1 exit status
make: *** [Makefile:5: 2_pthread_create] Error 1
liao:LinuxDay17$ gcc 2_pthread_create.c -o 2_pthread_create -g -lpthread
liao:LinuxDay17$ gcc 2_pthread_create.c -o 2_pthread_create -g -pthread
```

多线程

2023年8月15日 14:30

```c
void * threadFunc(void *arg){
    printf("I am child!\n");
}
int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    printf("I am main!\n");
    sleep(1);
    return 0;
}
```



性质

2023年8月15日 14:34

① 主线程终止 → 进程终止 ⟶ 其他线程也终止

② 没有父子关系，main 线程 其他子线程

线程的并发性

2023年8月15日 14:37

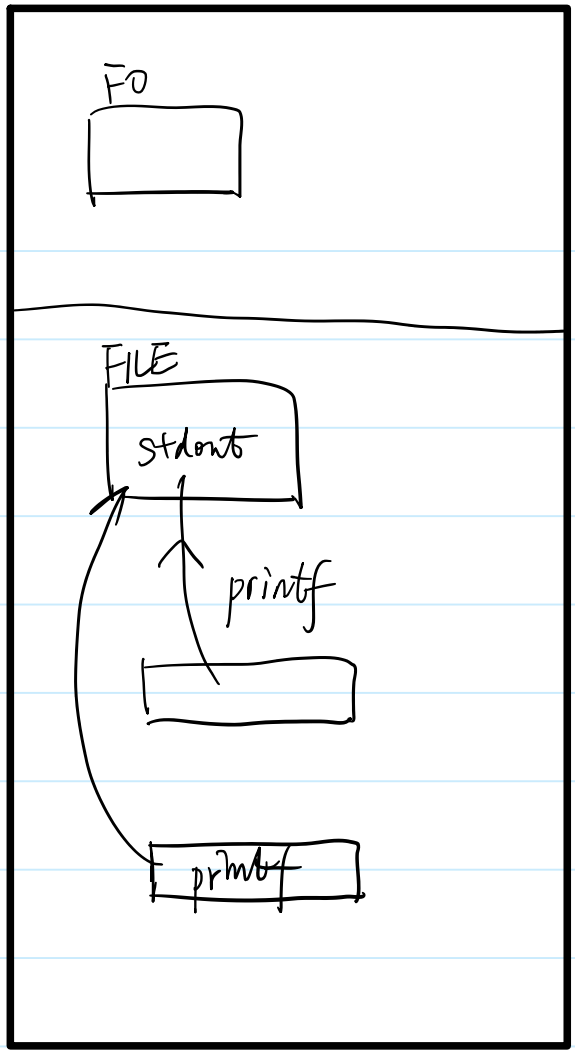
```
liao:LinuxDay17$ ./2_pthread_create
I am main!
liao:LinuxDay17$ ./2_pthread_create
I am main!
I am child!
liao:LinuxDay17$ ./2_pthread_create
I am main!
I am child!
liao:LinuxDay17$ ./2_pthread_create
I am main!
I am child!
I am child!
```

a.

b.

d

c



线程终止时，子线程的printf执行到什么程度

a → 常量 → stdout

b → stdout → FD

c → 清空stdout

d →

多线程的报错处理

2023年8月15日 14:53

多线程访问全局变量 ——→ 竞争条件.

之前的系统调用，先看返回值，知道是否有错.

知道具体什么错误，errno

pthread系列函数 { 返回值为0 正常
返回值非0. 报错原因.

```
char *strerror(int errnum);
```

```
#define THREAD_ERROR_CHECK(ret,msg) {if(ret != 0){fprintf(stderr,"%s:%s\n",msg,strerror(ret));}}
```

2023年8月15日 15:10

3_pthread_many.c buf

```c
#include <52func.h>
void *threadFunc(void *arg){
    while(1){
        sleep(1);
    }
}
int main(int argc, char *argv[])
{
    int cnt = 0;
    while(1){
        ++cnt;
        pthread_t tid;
        int ret = pthread_create(&tid,NULL,threadFunc,NULL);
        THREAD_ERROR_CHECK(ret,"pthread_create");
        if(ret != 0){
            printf("cnt = %d\n",cnt);
            break;
        }
    }
    return 0;
}
```

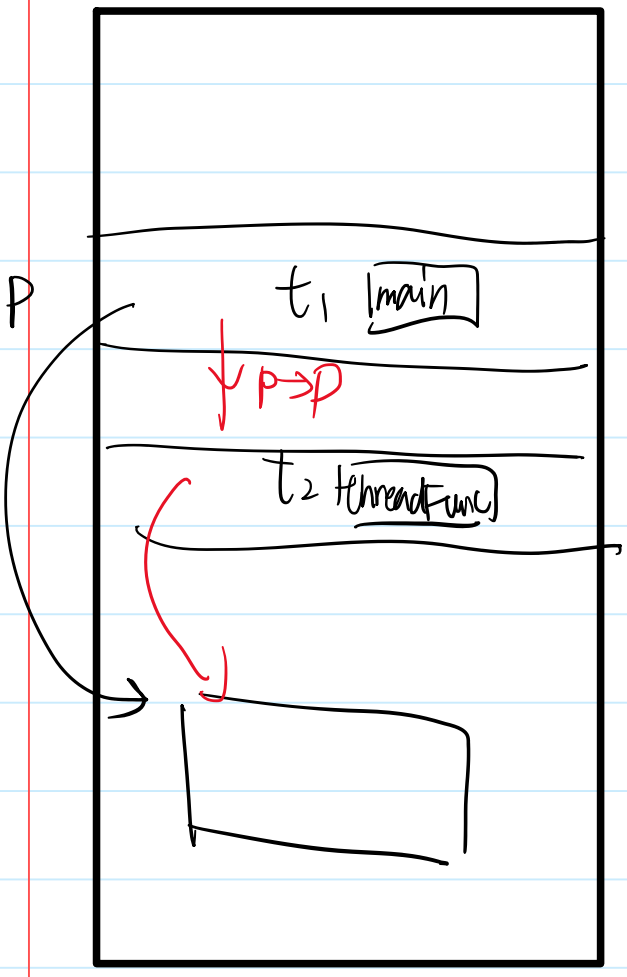
两个线程共享全局变量

2023年8月15日 15:12

```
int global = 1000;
void *threadFunc(void *arg){
    printf("child global = %d\n", global);
    ++global;
}
int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    sleep(1);
    printf("main global = %d\n", global);
    return 0;
}
```

两个线程共享堆空间

2023年8月15日 15:16



创建子线程

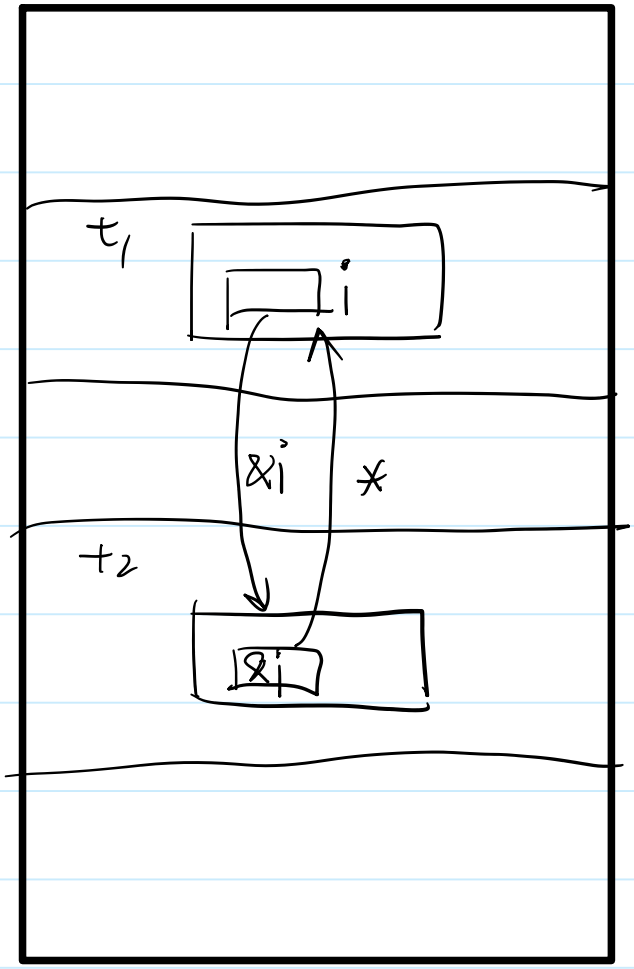
把指针传递给子线程

```c
void *threadFunc(void *arg){
    char * pHeap = (char *)arg;
    printf("child pHeap = %s\n", pHeap);
    strcpy(pHeap,"howoldareyou");
}
int main(int argc, char *argv[])
{
    // 先执行的线程 申请资源
    char * pHeap = (char *)malloc(4096);
    // 在pthread create之前初始化内容
    strcpy(pHeap,"howareyou");

    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,pHeap);
    // void *万能指针
    sleep(1);
    printf("main pHeap = %s\n", pHeap);
    return 0;
}
```

两个线程去共享栈空间

2023年8月15日 15:54



主线程：①.申请栈上数据

②.传递地址.

子线程 解引用(*)

```c
void *threadFunc(void *arg){
    int *pi = (int *)arg; // void * --> int *
    printf("child *pi = %d\n", *pi);
    ++*pi;
}
int main(int argc, char *argv[])
{
    int i = 1000; // 申请好栈上面的数据
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&i); // int * --> void *
    sleep(1);
    printf("main i = %d\n", i);
    return 0;
}
```

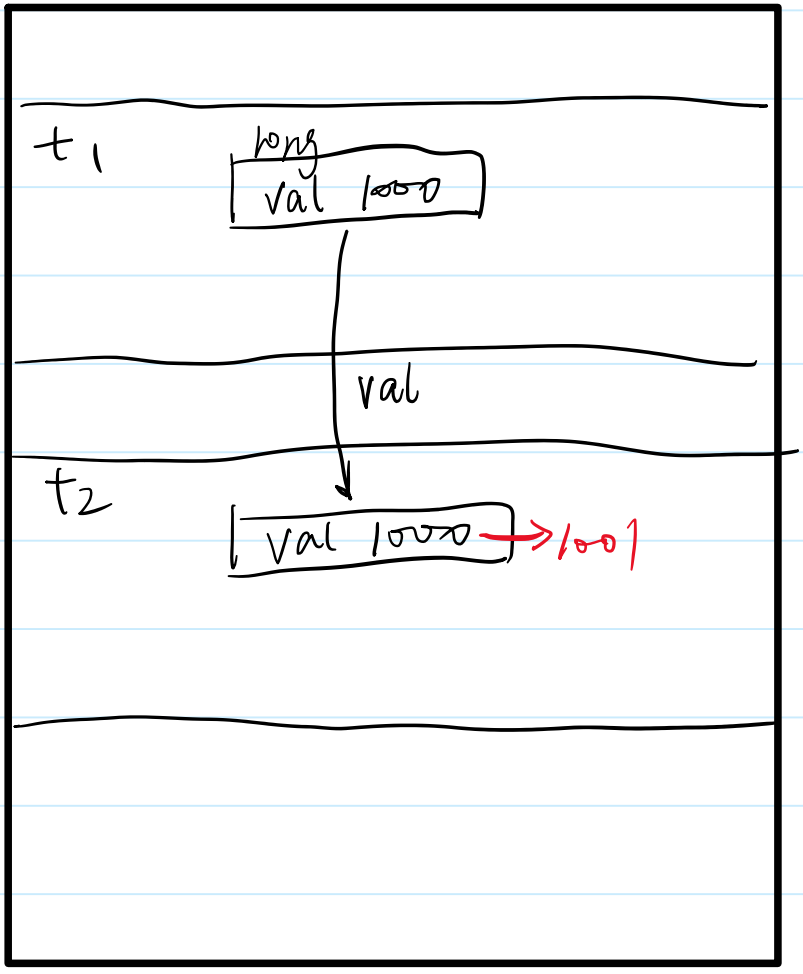
threadFunc的arg是void *

2023年8月15日 16:01

传统的做法，把void *看成地址

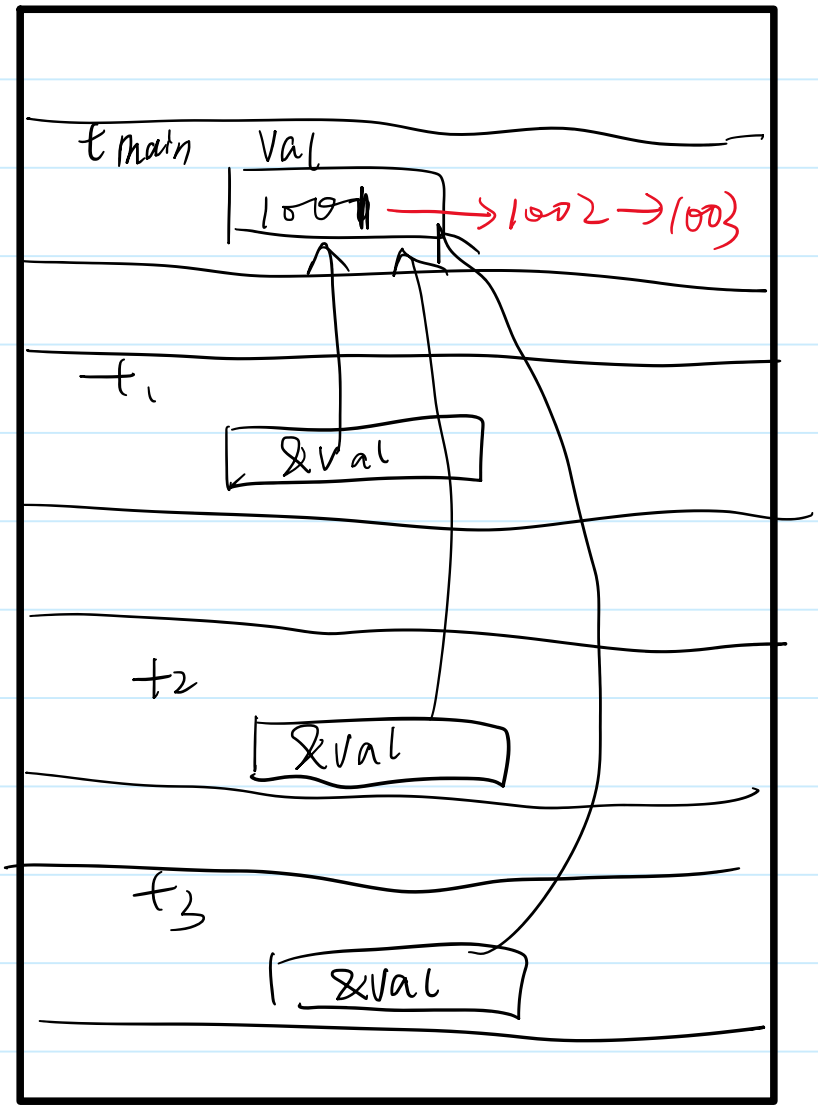
void *也可以看一个8字节的数据.

```c
void *threadFunc(void *arg){
    long val = (long )arg; // void * --> long
    printf("child val = %ld\n", val);
    ++val;
}
int main(int argc, char *argv[])
{
    long val = 1000;
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,(void *)val);
    // long --> void *
    sleep(1);
    printf("main val = %ld\n", val);
    return 0;
}
```



2023年8月15日 16:16

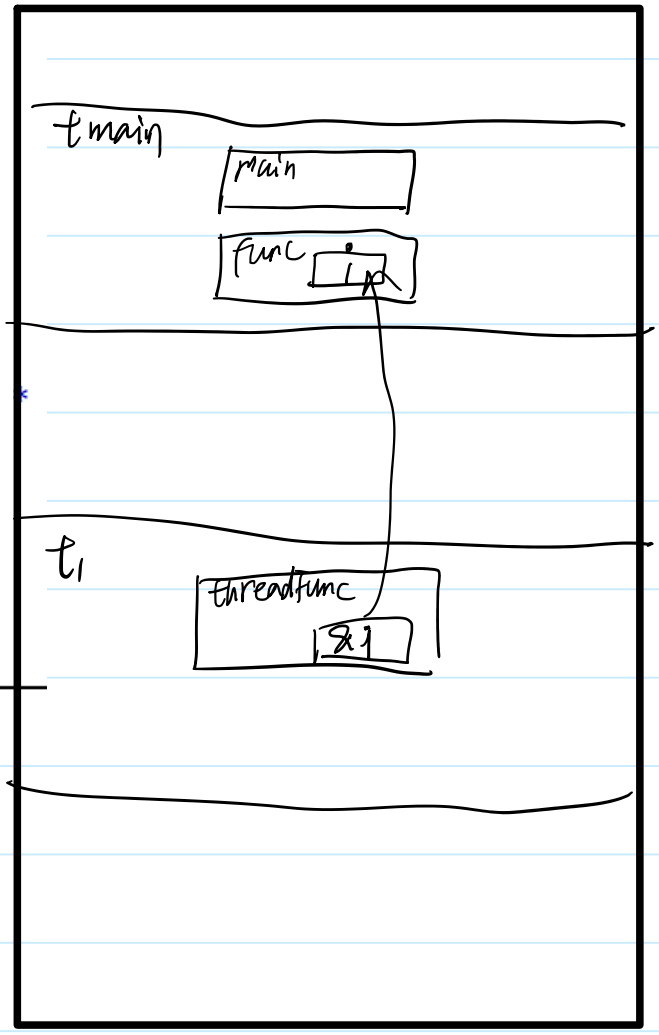
```c
void *threadFunc(void *arg){
    long * pval = (long *)arg;
    printf("val = %ld\n", *pval);
}
int main(int argc, char *argv[])
{
    long val = 1001;
    pthread_t tid1,tid2,tid3;
    pthread_create(&tid1,NULL,threadFunc,&val);
    ++val;
    pthread_create(&tid2,NULL,threadFunc,&val);
    ++val;
    pthread_create(&tid3,NULL,threadFunc,&val);
    sleep(1);
    return 0;
}
```



共享的数据的生存期

2023年8月15日 16:30

```c
void *threadFunc(void *arg){
    int *pi = (int *)arg; // void * --> int *
    printf("child *pi = %d\n", *pi);
    ++*pi;
}
void func(){
    int i = 1000; // 申请好栈上面的数据
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&i); // int * --> void *
}
int main(int argc, char *argv[])
{
    func();
    sleep(1);
    return 0;
}
```



变量i在func的栈帧上.
func返回，i内存回收，此时再解引用，是非法访问

```
liao:LinuxDay17$ ./9_pthread_create_stack_error
child *pi = -240156680
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

线程如何终止

2023年8月15日 16:42

1. 在线程入口函数执行 return. (void *)

```
0 S liao       50809    7940   50809  0    2  80   0 - 19095 hrtime 16:46 pts/2    00:00:00 ./10_pthread_return
1 S liao       50809    7940   50810  0    2  80   0 - 19095 hrtime 16:46 pts/2    00:00:00 ./10_pthread_return
0 R liao       50811   23423   50811  0    1  80   0 -  5032 -      16:47 pts/0    00:00:00 ps -elLf
liao:LinuxDay17$ ps -elLf
```

2. 任何位置 pthread_exit.

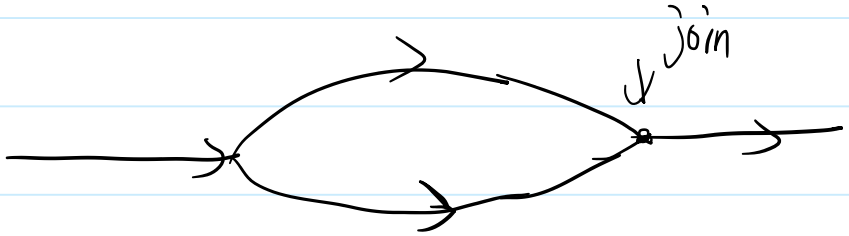
```
void pthread_exit(void *retval);
```

```c
void func(){
    //return;
    pthread_exit(NULL);
}
void *threadFunc(void *arg){
    printf("Hello\n");
    func();
    printf("World\n");
}
int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    while(1){
        sleep(1);
    }
    return 0;
}
```

pthread_join

2023年8月15日 17:11

等待某一个子线程的终止，若已终止，则回收 task_struct.




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

thread：子线程的 id（无指针）

retval：指向线程返回值的指针的指针

① 在被调函数中修改 retval 指向的指针变量的指向

② retval 中有一个元素为指针的数组.

线程的返回值是 void *，

retval：void * retval —&retval→ pthread_join

2023年8月15日 17:23

```c
void *threadFunc(void *arg){
    printf("Hello\n");
    return (void *)1;
    printf("World\n");
}
int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    void *retval;
    pthread_join(tid,&retval);
    printf("retval = %ld\n", (long )retval);
    return 0;
}
```

线程的取消

2023年8月15日 17:24

进程 被动终止 信号

多线程不要和信号混合在一起

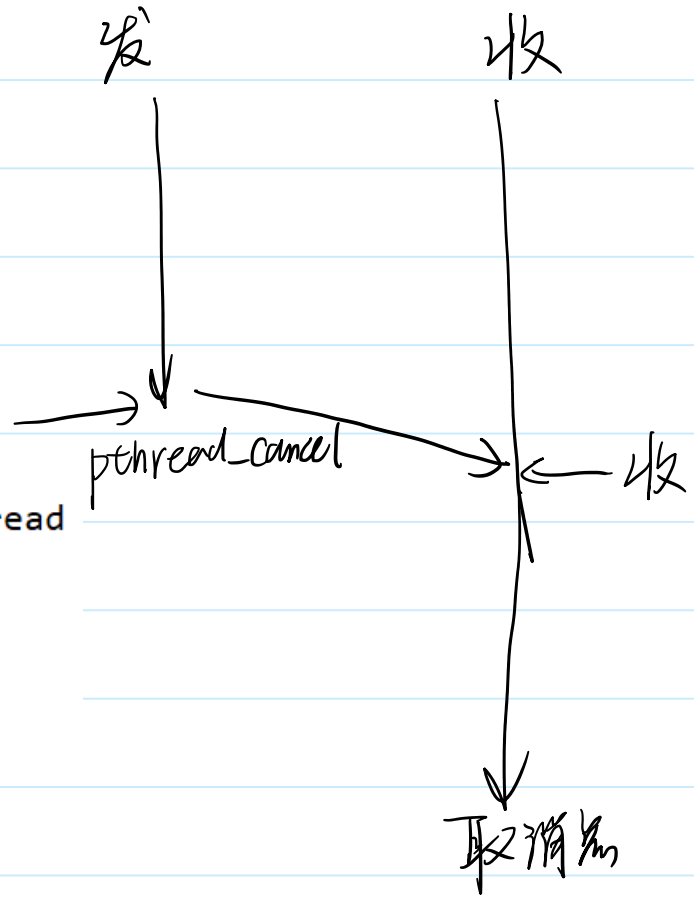
线程的取消机制

pthread_cancel - send a cancellation request to a thread

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

取消点：很多函数



有哪些取消点

man 7 pthreads

2023年8月15日 17:30

open/close/read/write. ⟹ 文件操作

select/system/wait/waitpid/... ⟹ 引发阻塞

- printf

2023年8月15日 17:39

```c
void *threadFunc(void *arg){
    while(1){}
}
int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    sleep(1);
    pthread_cancel(tid);

    void *retval;
    pthread_join(tid,&retval);
    printf("retval = %ld\n", (long) retval);
    return 0;
}
```

```c
void *threadFunc(void *arg){
    while(1){
        printf("I still alive!\n");
    }
}
int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    sleep(1);
    pthread_cancel(tid);

    void *retval;
    pthread_join(tid,&retval);
    printf("retval = %ld\n", (long) retval);
    return 0;
}
```

手动增加取消点

2023年8月15日 17:41

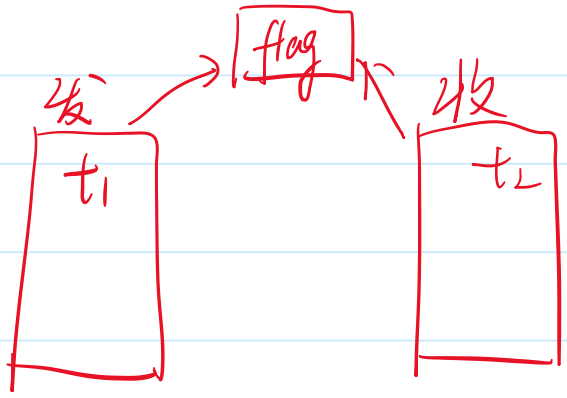
```
void pthread_testcancel(void);


void *threadFunc(void *arg){
    while(1){
        //printf("I still alive!\n");
        pthread_testcancel();
    }
}
```

让其他线程退出

2023年8月15日 17:45

不好用.



设置个共享的flag.

发线程 更改flag

收线程 自行选择合适的时机检查flag

资源清理的故事

2023年8月15日 17:51

pthread_cancel