多线程共享全局变量

2023年5月2日 9:23

```c
#include <49func.h>
int global = 0;
//extern int global;
void *threadFunc(void *arg){
    printf("I am child thread, global = %d\n", global);
    ++global;
}
int main()
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    sleep(1);
    printf("I am main thread, global = %d\n", global);
    return 0;
}
```

多线程共享堆数据

2023年5月2日 10:06

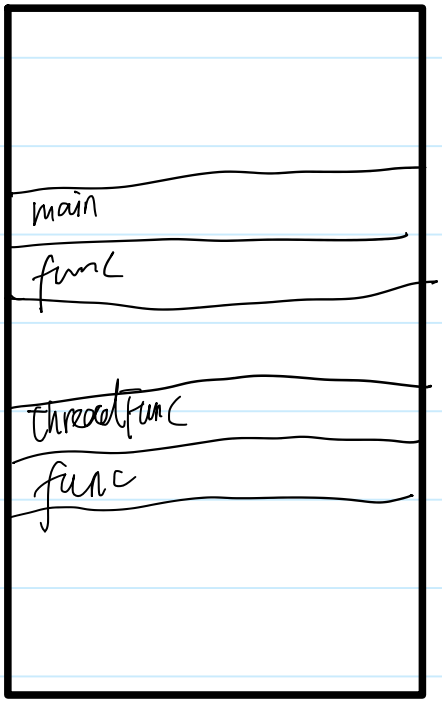
```c
#include <49func.h>
void *threadFunc(void *arg){
    // arg接收了传入的参数（pHeap）
    // void * 传入是什么类型，就恢复成什么类型
    int *pHeap = (int *)arg;
    printf("I am child, *pHeap = %d\n", *pHeap);
    ++*pHeap;
}
int main()
{
    int *pHeap = (int *)malloc(sizeof(int));
    *pHeap = 1;
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,pHeap);
    sleep(1);
    printf("I am main, *pHeap = %d\n", *pHeap);
    return 0;
}
```

void *p = xxx* addr

多线程共享栈数据

2023年5月2日 10:25

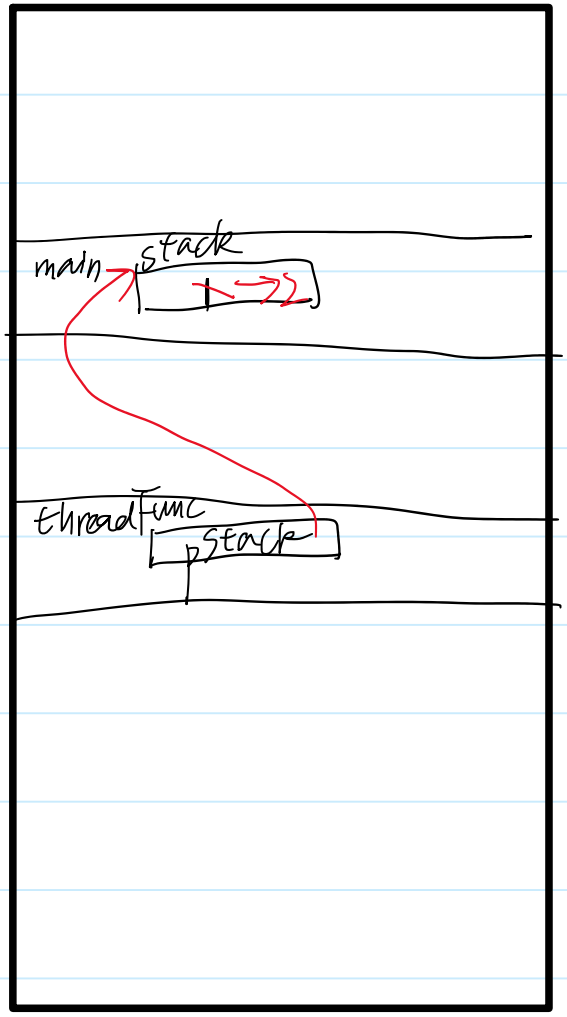
多线程中每个线程拥有一个相对独立的栈区



在同一个地址空间内

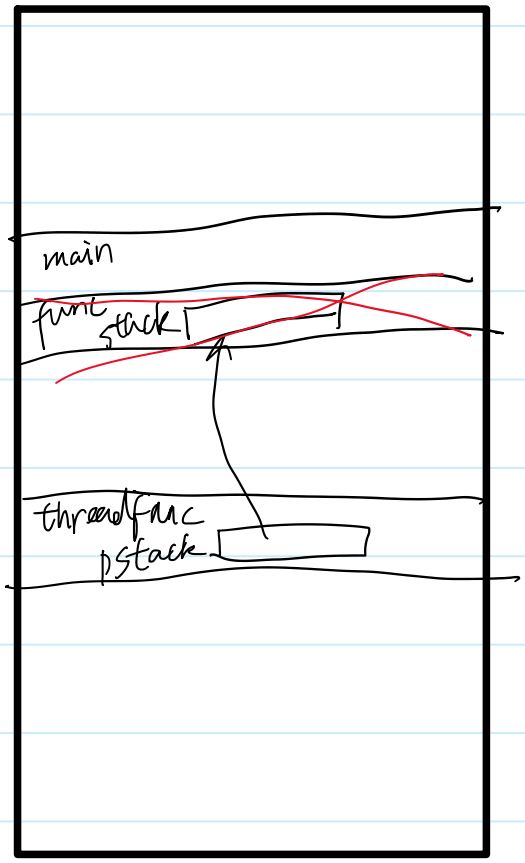
2023年5月2日 10:35

```c
#include <49func.h>
void *threadFunc(void *arg){
    int * pStack = (int *)arg;
    printf("I am child, stack = %d\n", *pStack);
    ++*pStack;
}
void func(){
    int stack = 1;
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&stack);
}
int main()
{
    int stack = 1;
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&stack);
    // func();
    sleep(1);
    printf("I am main, stack = %d\n", stack);
    return 0;
}
```



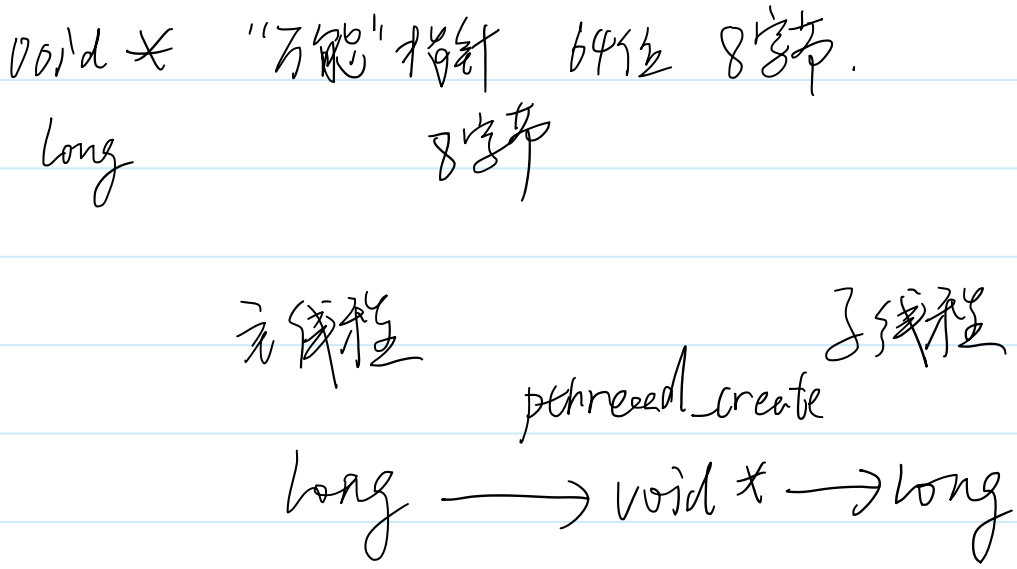
共享栈数据，一定要注意内存的释放

2023年5月2日 10:37



传递一个long类型

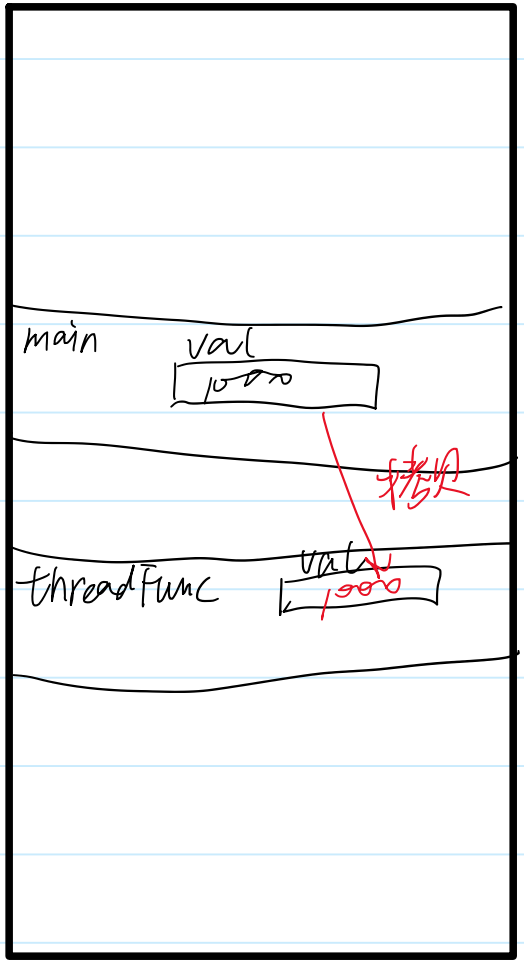
2023年5月2日 11:03




```c
void * threadFunc(void *arg){
    long val = (long )arg;
    printf("I am child, val = %ld\n", val);
    ++val;
}
int main()
{
    long val = 1000;
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,(void *)val);
    sleep(1);
    printf("val = %ld\n", val);
    return 0;
}
```

传递整数，采用拷贝的形式

2023年5月2日 11:11



共享 拷贝

2023年5月2日 11:21

4_create.c

```c
#include <49func.h>
void * threadFunc(void *arg){
    //long val = (long )arg;
    //printf("I am child, val = %ld\n", val);
    long *pVal = (long *)arg;
    printf("I am child, val = %ld\n", *pVal);
}
int main()
{
    pthread_t tid1,tid2,tid3;
   // long val1 = 1001;
   // long val2 = 1002;
   // long val3 = 1003;
   // pthread_create(&tid1,NULL,threadFunc,(void *)val1);
   // pthread_create(&tid2,NULL,threadFunc,(void *)val2);
   // pthread_create(&tid3,NULL,threadFunc,(void *)val3);
    long val = 1001;
    pthread_create(&tid1,NULL,threadFunc,&val);
    ++val;
    pthread_create(&tid2,NULL,threadFunc,&val);
    ++val;
    pthread_create(&tid3,NULL,threadFunc,&val);
    sleep(1);
    return 0;
}
```

线程的主动退出

2023年5月2日 11:26

1.在线程入口函数 执行 return.

```
ps -elLf
```

2.在何位置.

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

```c
void *func(){
    printf("A\n");
    //return (void *)0;
    pthread_exit((void *)0);
}
void * threadFunc(void *arg){
    //sleep(10);
    //return (void *)0;
    func();
    printf("B\n");
}
int main()
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    while(1);
    return 0;
}
```

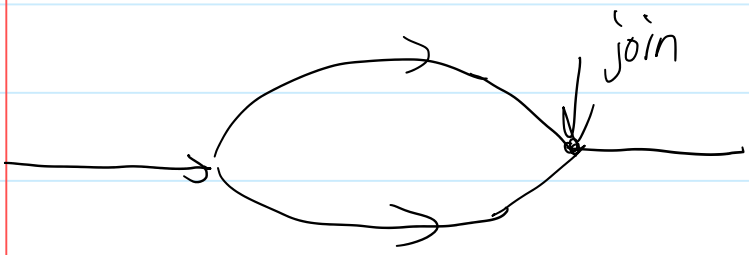
获取另一个线程的退出状态

2023年5月2日 11:43

不需要父子关系

tid 没有数无指针

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



线程为申请一个void *.
pthread_join中会设置其内容

使用指针的场景

① 在被调函数修改线程的数据
线程为申请内存，取地址传入.
被调方使用时按引用（*，->，[]）

② 数组类型的数据，（数组名退化）.

二级指针 ① 线程的一级指针变量 被被调方修改
② 线程方存在一个元素为一级指针的数组

代码

2023年5月2日 14:42

```c
void *threadFunc(void *arg){
    //sleep(5);
    printf("I am child thread!\n");
    //return (void *)123;
    pthread_exit((void *)246);
}
int main()
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    //void ** pret;
    //pthread_join(tid,pret);

    //void * ret;
    //pthread_join(tid,&ret);
    //printf("ret = %ld\n", (long) ret);

    pthread_join(tid,NULL);
    printf("I am main thread!\n");
    return 0;
}
```

线程的取消

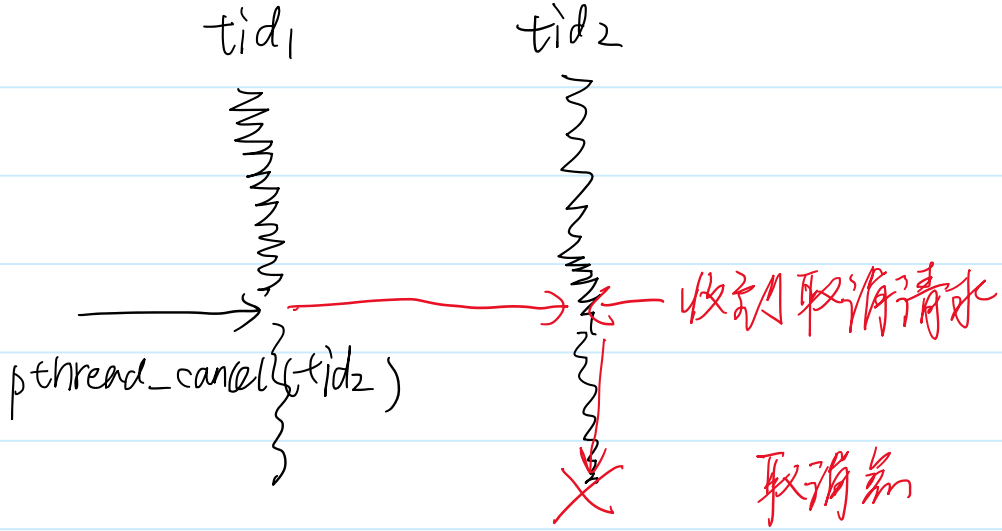
2023年5月2日 14:46

多线程不提倡与信号一起使用

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

SYNOPSIS
       #include <pthread.h>

       int pthread_cancel(pthread_t thread);
```



有哪些函数是取消点

2023年5月2日 14:53

```
$ man 7 pthreads
```

总结：①libc手册所有会引发阻塞的函数 select sleep wait...

②涉及文件函数. open close read write

fopen fclose printf...

2023年5月2日 15:03

```c
#include <49func.h>
void *threadFunc(void *arg){
    while(1){
        printf("I still alive!\n");
    }
}
int main()
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    sleep(1);
    printf("sleep over!\n");
    pthread_cancel(tid);
    void *ret;
    pthread_join(tid,&ret);
    printf("You die, ret = %ld\n", (long ) ret);
    return 0;
}
```

```
I still alive!
I still alive!
I still alive!
I still alive!
I still alive!
I still alive!
sleep over!
I still alive!
You die, ret = -1
```

手动添加取消点

2023年5月2日 15:07

```
void pthread_testcancel(void);
```

```c
void *threadFunc(void *arg){
    while(1){
        //printf("I still alive!\n");
        pthread_testcancel();
    }
}
int main()
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    sleep(1);
    printf("sleep over!\n");
    pthread_cancel(tid);
    void *ret;
    pthread_join(tid,&ret);
    printf("You die, ret = %ld\n", (long ) ret);
    return 0;
}
```

资源清理函数栈

2023年5月2日 15:14

p1 → free(p1)　　fd3 → close(fd3)

```
void *threadFunc(void *arg){
    void *p1 = malloc(4);
    void *p2 = malloc(4);
    int fd3 = open("file1",O_RDWR);
    // ...
    close(fd3);
    free(p2);
    free(p1);
}
```

1 → 无
2 → p1
3 → p1,p2
4 → p1,p2,fd3
5 → p1,p2
6 → p1
7 → 无

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

清理函数

清理 cleanup-push

```
void free3(void *arg){
    close(...)(一真正清理)
}
```

2023年5月2日 16:04

```c
void *threadFunc(void *arg){
    pthread_exit(NULL);
    void *p1 = malloc(4);//申请资源之后，马上压栈
    pthread_cleanup_push(cleanup1,p1);
    void *p2 = malloc(4);
    pthread_cleanup_push(cleanup2,p2);
    int fd3 = open("file1",O_RDWR);
    pthread_cleanup_push(cleanup3,&fd3);
    // ...
    //close(fd3);
    pthread_cleanup_pop(1);//把原来释放资源的行为，替换成弹栈
    //free(p2);
    pthread_cleanup_pop(1);
    //free(p1);
    pthread_cleanup_pop(1);
}
int main()
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    void *ret;
    pthread_join(tid,&ret);
    printf("ret = %ld\n", (long )ret);
    return 0;
}
```

```c
void cleanup1(void *arg){
    // arg 直接传入p1
    printf("cleanup1\n");
    // free p1
    free(arg);
}
void cleanup2(void *arg){
    // arg 直接传入p2
    printf("cleanup2\n");
    // free p2
    free(arg);
}
void cleanup3(void *arg){
    // arg 传入fd3的地址
    int *pFd3 = (int *)arg;
    printf("cleanup3\n");
    // close fd3
    close(*pFd3);
}
```

资源清理函数的原理

2023年5月2日 16:17

pthread_cleanup_push

RAII

栈：

栈弹出的场景：

① pthread_cleanup_pop弹栈 弹出栈顶 { 参数为0，不调用清理
参数>0，调用清理

② pthread_exit / 被cancel且运行到取消点
依次弹栈，每弹出一个就调用清理函数。

③ 在线程入口函数中 return.

push和pop必须成对出现

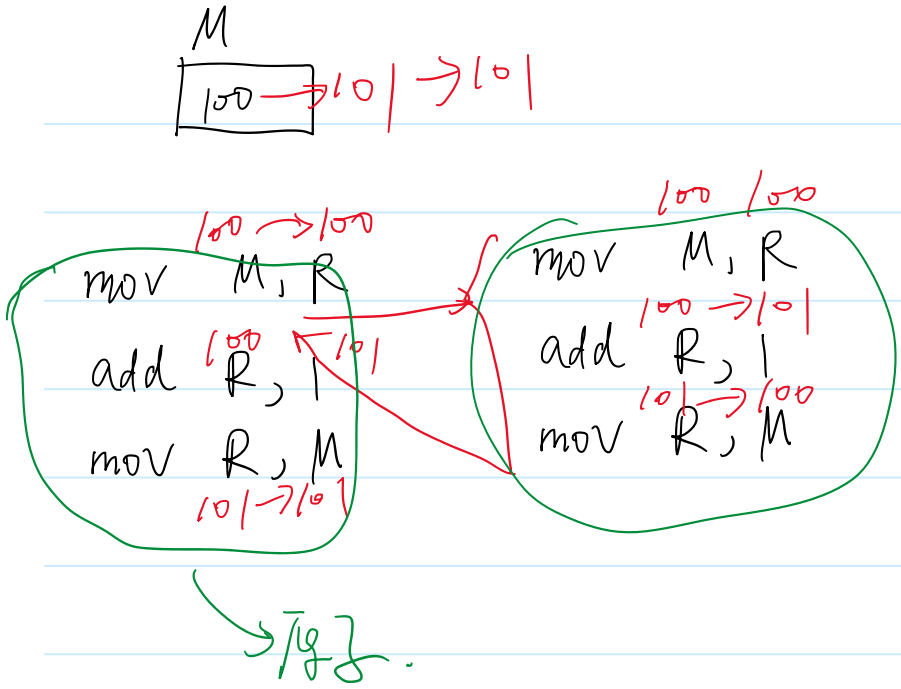
2023年5月2日 16:27

```
1 #  define pthread_cleanup_push(routine, arg) \
2   do {                                        \
3     __pthread_cleanup_class __clframe (routine, arg)
4
5 /* Remove a cleanup handler installed by the matching pthread_cleanup_push.
6    If EXECUTE is non-zero, the handler function is called. */
7 #  define pthread_cleanup_pop(execute) \
8     __clframe.__setdoit (execute);                \
9   } while (0)
```

竞争条件

2023年5月2日 16:39

```c
#define NUM 10000000
void *threadFunc(void *arg){
    int *pNum = (int *)arg;
    for(int i = 0; i < NUM; ++i){
        ++*pNum;
    }
    pthread_exit(NULL);
}
int main()
{
    int num = 0;
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&num);
    for(int i = 0; i < NUM; ++i){
        ++num;
    }
    pthread_join(tid,NULL);
    printf("num = %d\n",num);
    return 0;
}
```



互斥锁

2023年5月2日 17:12

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
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

→ 静态初始化

mutex → "mutal exclude"

一个线程持有资源时，另一个线程不允许访问

锁 未锁 / 已锁

原子的.

加锁 { 测试对锁的状态，若未锁，则加锁，然后正常运行
若已锁，则阻塞.

解锁 将锁处于未锁状态.

t1

```
加锁
++num
解锁.
```

代码段 临界区

t2

```
加锁
++num
解锁
```

加锁和解锁

2023年5月2日 17:31

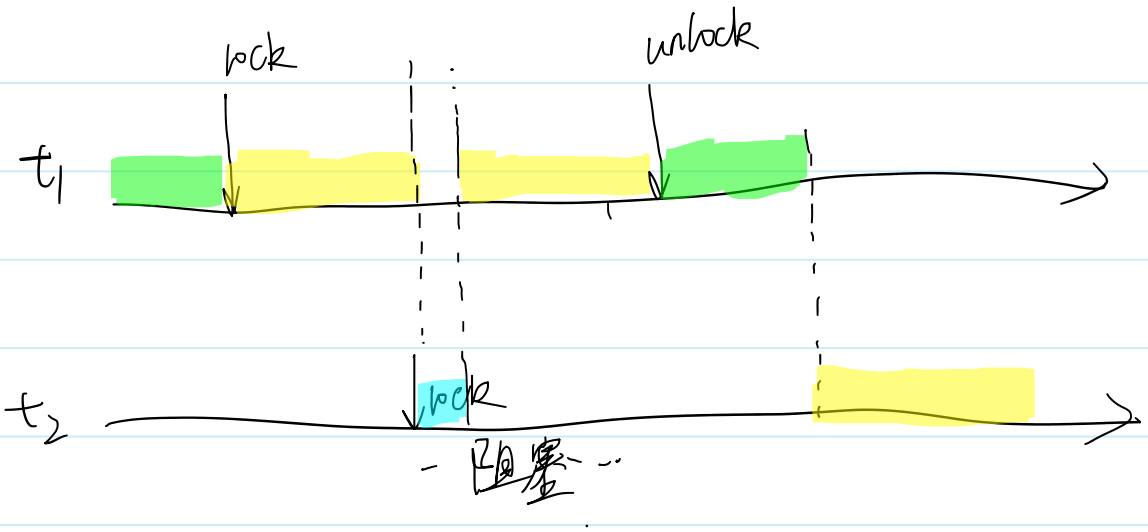
```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
typedef struct shareRes_s {
    int num;
    pthread_mutex_t mutex;
} shareRes_t;
void *threadFunc(void *arg){
    shareRes_t * pShareRes = (shareRes_t *)arg;
    for(int i = 0; i < NUM; ++i){
        pthread_mutex_lock(&pShareRes->mutex);
        ++pShareRes->num;
        pthread_mutex_unlock(&pShareRes->mutex);
    }
    pthread_exit(NULL);
}
int main()
{
    shareRes_t shareRes;
    shareRes.num = 0;
    pthread_mutex_init(&shareRes.mutex,NULL);//初始化一个锁资源，第二参数是NULL表示锁是默认属性
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&shareRes);
    for(int i = 0; i < NUM; ++i){
        pthread_mutex_lock(&shareRes.mutex);
        ++shareRes.num;
        pthread_mutex_unlock(&shareRes.mutex);
    }
    pthread_join(tid,NULL);
    printf("num = %d\n",shareRes.num);
```

```
[liao@ubuntu Linuxday_16]$ ./9_add
num = 20000000
```

某个时间段只能有一个线程访问共享资源

2023年5月2日 17:45



为什么需要临界区越小越好

2023年5月2日 17:51

循环不应该放入加锁/解锁内部.

饥饿.