

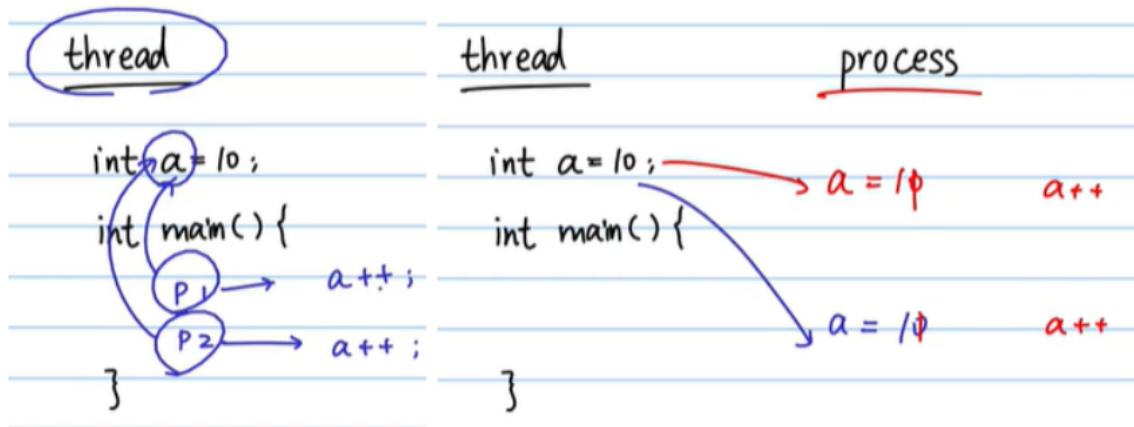
# 多线程程序学习

注：本次学习资源来源于<https://www.bilibili.com/video/BV1kt411z7ND?from=search&seid=9129914462868456444>，版权所有：b站up主——正月点灯笼。

## 第一讲

### 一条线程

thread（线程）和 process（进程）都可以做并行运算，但是前者有共享内存，后者无共享内存。



- 需要用到 `pthread.h` 头文件

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int main(){
    pthread_t th;
    pthread_create();

    return 0;
}
```

函数 `pthread_create()` 的原型：

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

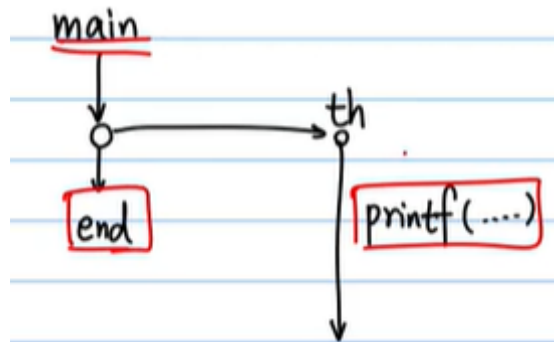
调用函数 `pthread_create(&th, NULL, 函数指针, 函数参数);`

```

void* myfunc(void* args) {
    printf("Hello World\n");
    return NULL;
}
int main() {
    pthread_t th;
    pthread_create(&th, NULL, myfunc, NULL);
    //pthread_join(th,NULL);
    return 0;
}

```

上述代码运行过程:



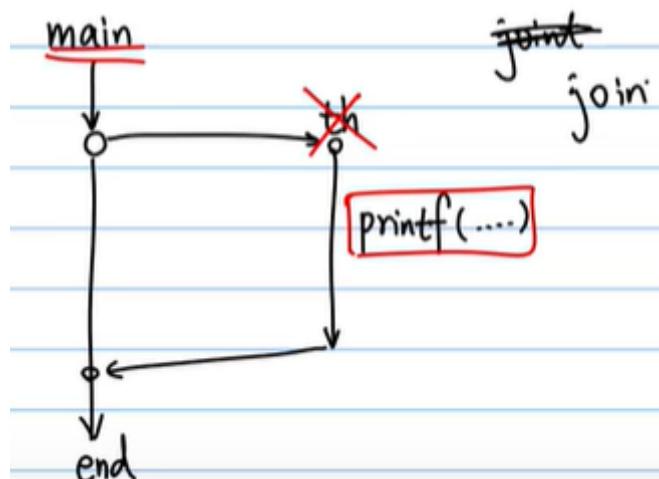
主线程在创建子线程后直接结束了, `th` 还没有运行 `printf`, 此时主线程不会等待子线程运行结束, 导致 `printf` 没有运行。

改进:

```

void* myfunc(void* args) {
    printf("Hello World\n");
    return NULL;
}
int main() {
    pthread_t th;
    pthread_create(&th, NULL, myfunc, NULL);
    pthread_join(th,NULL);
    return 0;
}

```



加入了一个 `join` 函数，等待子线程结束后再 `return 0`;

## 两条线程

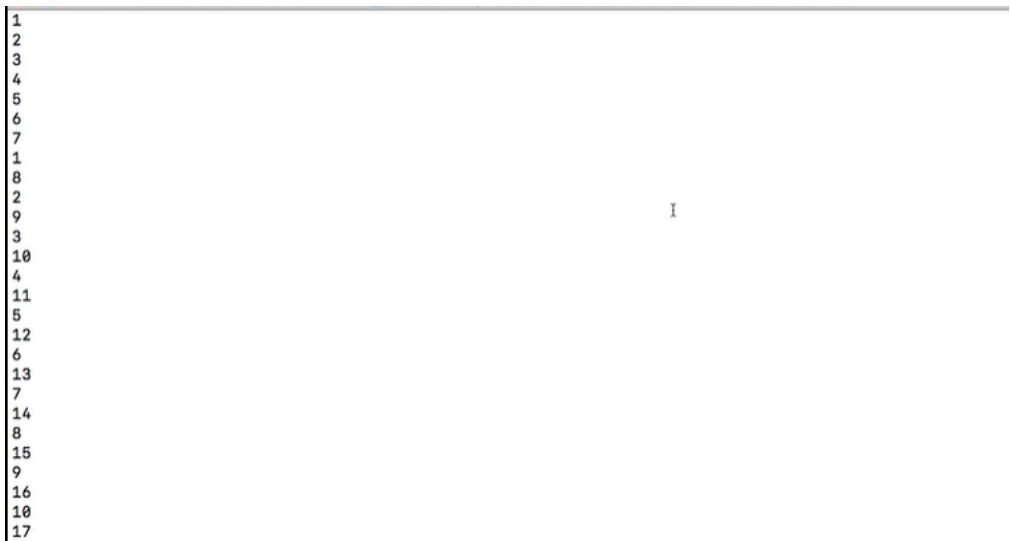
```
void* myfunc(void* args) {
    int i;
    for( i=1; i<50, i++) {
        printf("%d\n",i);
    }
    return NULL;
}

int main() {
    pthread_t th1;
    pthread_t th2;

    pthread_create(&th1, NULL, myfunc, "th1");//th1传给myfunc的args参数
    pthread_create(&th2, NULL, myfunc, "th2");//th2传给myfunc的args参数

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```



从运行结果可以发现两条线程的运行速度是不一样的，并且说明这两条线程是同时在运行的。

## 第二讲

从上一讲最后写的程序的运行结果来看，无法显示最后打印出的数字是由哪一条线程运行得到的。

```
void* myfunc(void* args) {
    int i;
    char* name=(char*)args;//强制转换成字符串
    for( i=1; i<50, i++) {
        printf("%s: %d\n", name, i);
    }
    return NULL;
}
```

```

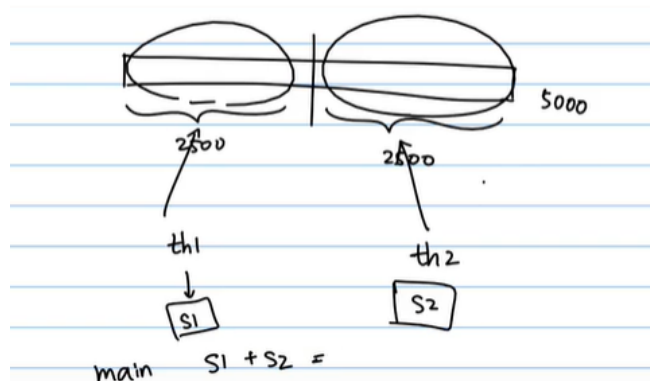
Terminal Shell Edit View Window Help
multithread — haojie@area51: ~ — -bash - xournal — 104x27

th1: 16
th2: 16
th1: 17
th2: 17
th1: 18
th2: 18
th1: 19
th2: 19
th1: 20
th2: 20
th1: 21
th2: 21
th1: 22
th2: 22
th1: 23
th2: 23
th1: 24
th2: 24
th1: 25
th2: 25
th1: 26
th2: 26
th1: 27
th2: 27
th1: 28
th2: 28
th1: 29

```

## 数组求和简易写法

定义一个数组 `arr[5000]`



```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int arr[5000];
int s1=0;
int s2=0;

void* myfunc1(void* args) {
    int i;
    for( i=0; i<2500; i++) {
        s1=s1+arr[i];
    }
    return NULL;
}

void* myfunc2(void* args) {
    int i;
    for( i=2500; i<5000; i++) {
        s2=s2+arr[i];
    }
    return NULL;
}

```

```

int main() {
    int i;
    for(i=0;i<5000;i++){
        arr[i]=rand()%50;
    }
    pthread_t th1;
    pthread_t th2;
    pthread_create(&th1, NULL, myfunc1, NULL);
    pthread_create(&th2, NULL, myfunc2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("%s1 = %d\n",s1);
    printf("%s2 = %d\n",s2);
    printf("%s1 + s2 = %d\n",s1+s2);
    return 0;
}

```

```

arr[4996] = 14
arr[4997] = 19
arr[4998] = 7
arr[4999] = 27
users-MBP:multithread user$ vim example2.c
users-MBP:multithread user$ vim example2.c
users-MBP:multithread user$ vim example2.c
users-MBP:multithread user$ gcc example2.c -lpthread -o example2
users-MBP:multithread user$ ./example2
s1 = 60241
s2 = 61214
s1 + s1 = 121455 I
users-MBP:multithread user$ █

```

## 改进写法

发现 `myfunc1` 和 `myfunc2` 代码重复，于是进一步改进代码。

```

typedef struct{
    int first;
    int last;
} MY_ARGS;

MY_ARGS args1 = {0,2500};
MY_ARGS args2 = {2500,5000};

pthread_create(&th1, NULL, myfunc, &args1);
pthread_create(&th2, NULL, myfunc, &args2);

```

只需保留一个 `myfunc`，并对该函数进行改进：

```

void* myfunc(void* args) {
    int i;
    MY_ARGS* my_args = (MY_ARGS*) args;
    int first = my_args->first;
    int last = my_args->last;

    for( i=first; i<last, i++) {
        s=s+arr[i];
    }
    printf("s = %d\n",s);
    return NULL;
}

```

```

users-MBP:multithread user$ ./example2
s1 = 60241
s2 = 61214
s1 + s2 = 121455
users-MBP:multithread user$ vim example2.c
users-MBP:multithread user$ gcc example2.c -lpthread -o example2
users-MBP:multithread user$ ./example2
s = 60241
s = 61214
users-MBP:multithread user$

```

但这样运行后并没有返回 `s`，无法将两次运算的 `s` 相加得到最终求和的结果。

- 再次改进

```

typedef struct{
    int first;
    int last;
    int result;
} MY_ARGS;

MY_ARGS args1 = {0,2500,0};
MY_ARGS args2 = {2500,5000,0};

```

```

void* myfunc(void* args) {
    int i;
    int s = 0;
    MY_ARGS* my_args = (MY_ARGS*) args;
    int first = my_args->first;
    int last = my_args->last;

    for (i=first; i<last; i++) {
        s = s + arr[i];
    }
    my_args->result = s;
    return NULL;
}

```

再加入一个 `my_args->result = s;` 即可在 `main` 函数中获得求和结果。

```

pthread_join(th1, NULL);
pthread_join(th2, NULL);

int s1 = args1.result;
int s2 = args2.result;
printf("s1 = %d\n", s1);
printf("s2 = %d\n", s2);
printf("s1 + s2 = %d\n", s1 + s2);
return 0;

```

```

s1 = 60241
s2 = 61214
s1 + s2 = 121455
users-MBP:multithread user$ vim example2.c
users-MBP:multithread user$ gcc example2.c -lpthread -o example2
users-MBP:multithread user$ ./example2
s = 60241
s = 61214
users-MBP:multithread user$ vim example2.c
users-MBP:multithread user$ gcc example2.c -lpthread -o example2
users-MBP:multithread user$ ./example2
s1 = 60241
s2 = 61214
s1 + s2 = 121455
users-MBP:multithread user$

```

- 为什么每次输出的结果都是一样的？

C语言 rand 产生的随机数不是真正的随机数，称之为“伪随机数”。

## 第三讲

### race condition

在上一讲的例题中，可以直接定义一个全局变量 s。

```
for (i=first; i<last; i++) {
    s = s + arr[i];
}

users-MBP:multithread user$ gcc exam
users-MBP:multithread user$ ./exampl
s = 121455
users-MBP:multithread user$
```

但是这有可能存在问题：

```
int s = 0;

void* myfunc(void* args) {
    int i = 0;
    for (i=0; i<1000000; i++) {
        s++;
    }
}

int main() {
    pthread_t th1;
    pthread_t th2;

    pthread_create(&th1, NULL, myfunc, NULL);
    pthread_create(&th2, NULL, myfunc, NULL);

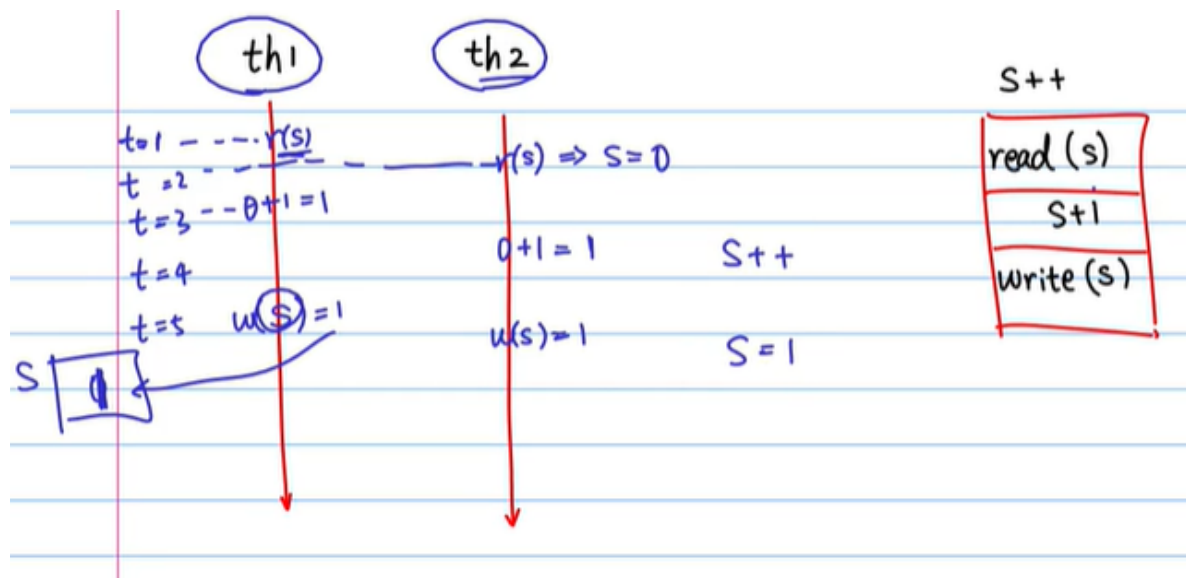
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    printf("s = %d\n", s);
    return 0;
}

users-MBP:multithread user$ gcc example4.c -lpthread -o example
users-MBP:multithread user$ ./example
s = 1089546
users-MBP:multithread user$
```

- 这个程序的结果并不是 s=2000000，而且每次运行的结果不稳定，这是为什么？

- ① s++ 是一个复合语句，先读入 s，再进行 s+1，再把新的 s 写回内存；
- ② 若 th1 读入 s，在 th1 对 s 加 1 后，th2 也读入 s，此时少加了一个 1。



这样便会丢失多次 s++ 的结果，每次的结果都会不一样。

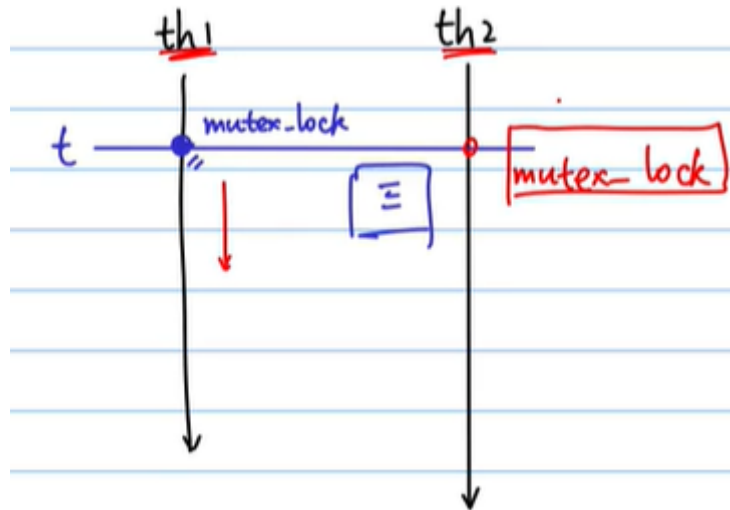
这样的情况称为 **race condition**。

### 解决方法

最常用的解决方法——加锁： `pthread_mutex_t lock;`

对锁进行初始化： `pthread_mutex_init(&lock, NULL);`

锁不是用来锁变量的，是用来锁代码的。



两条线程一起“抢”锁。

## 不同加锁位置时代码的效率

- 放在循环里面

```
void* myfunc(void* args) {  
    int i = 0;  
    for (i=0; i<100000; i++) {  
        pthread_mutex_lock(&lock);  
        s++;  
        pthread_mutex_unlock(&lock);  
    }  
    return NULL;  
}
```

```
users-MBP:multithread user$ vim example4.c  
users-MBP:multithread user$ gcc example4.c -lpthread -o example4  
users-MBP:multithread user$ ./example4  
s = 200000  
users-MBP:multithread user$ ./example4  
s = 200000  
users-MBP:multithread user$ ./example4  
s = 200000  
users-MBP:multithread user$ time ./example4  
s = 200000  
I  
real    0m0.697s  
user    0m0.204s  
sys     0m0.696s  
users-MBP:multithread user$
```

每次循环都做一次加锁和解锁，调用一次 myfunc 共进行了1000000次加锁和解锁。

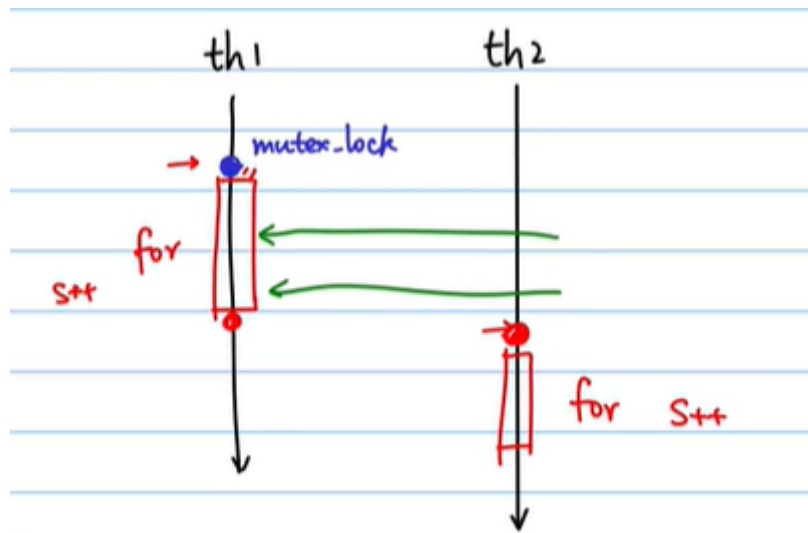
- 放在循环外面

```
void* myfunc(void* args) {  
    pthread_mutex_lock(&lock);  
    int i = 0;  
    for (i=0; i<100000; i++) {  
        s++;  
    }  
    pthread_mutex_unlock(&lock);  
    return NULL;  
}
```

```
users-MBP:multithread user$ vim example4.c  
users-MBP:multithread user$ gcc example4.c -lpthread -o example4  
users-MBP:multithread user$ ./example4  
s = 200000  
users-MBP:multithread user$ time ./example4  
s = 200000  
I  
real    0m0.004s  
user    0m0.002s  
sys     0m0.001s  
users-MBP:multithread user$
```

这说明锁的位置直接关系到代码的效率。



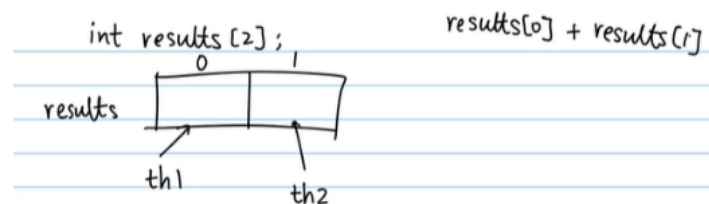


这样的话两次 for 循环是独立进行的，这样还不如在同一个线程中写两个 for 循环，省去了加锁解锁的时间，于是回到了上一个例子中使用结构体的方法。

## 第四讲

### 两种方法的效率

①定义两个全局变量 result[0] 和 result[1]。



```
int* arr;
int results[2];

void* myfunc(void* args) {
    int i;
    MY_ARGS* my_args = (MY_ARGS*) args;
    int first = my_args -> first;
    int last = my_args -> last;
    int id = my_args -> id;

    for (i=first; i<last; i++) {
        results[id] = results[id] + arr[i];
    }

    return NULL;
}
```

②结构体中定义 result

```
void* myfunc(void* args) {
    int i;
    MY_ARGS* my_args = (MY_ARGS*) args;
    int first = my_args -> first;
    int last = my_args -> last;
    int s = 0;
    for (i=first; i<last; i++) {
        s = s + arr[i];
    }
    my_args -> result = s;
    return NULL;
}
```

从本质上看，这两种方法是没有什么区别的，但是运行的下效率不同。

```

users-MBP:multithread user$ time ./example5
s1 = 50003728
s2 = 49996381
s1 + s2 = 100000109

real    0m0.661s
user    0m0.764s
sys     0m0.079s
users-MBP:multithread user$ time ./example6
s1 = 50003728
s2 = 49996381
s1 + s2 = 100000109

real    0m0.539s
user    0m0.495s
sys     0m0.078s

```

```

users-MBP:multithread user$ gcc example5.c -lpthread -o example5
users-MBP:multithread user$ time ./example5
s1 = 500012058
s2 = 499962035
s1 + s2 = 999974093

real    0m9.104s
user    0m6.154s
sys     0m5.409s
users-MBP:multithread user$ time ./example6
s1 = 500012058
s2 = 499962035
s1 + s2 = 999974093

real    0m5.133s
user    0m4.919s
sys     0m0.728s

```

多次尝试后发现第二种方法的速度总比第一种快些。

- 这是为什么呢?

## False sharing 假共享

