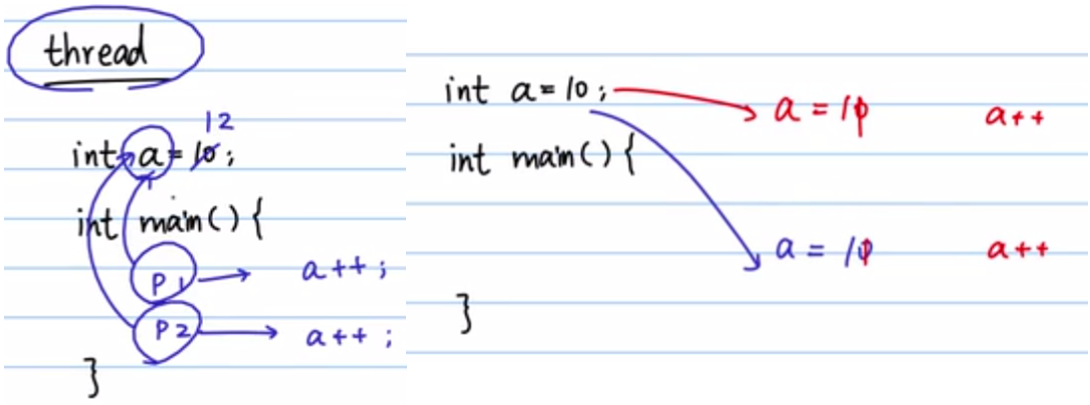
- thread & process 都可做并行运算，区别是thread有共享内存而process没有共享内存



- 需要 `#include<pthread.h>`

  声明 `pthread_t th;`

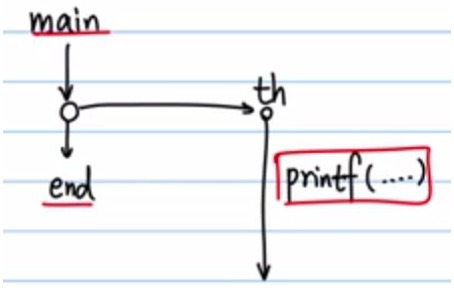
  函数原型 `pthread_create(pthread_t *thread, const pthread_attr_t *attr, void* (*start_routine)(void *), void *arq);`

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

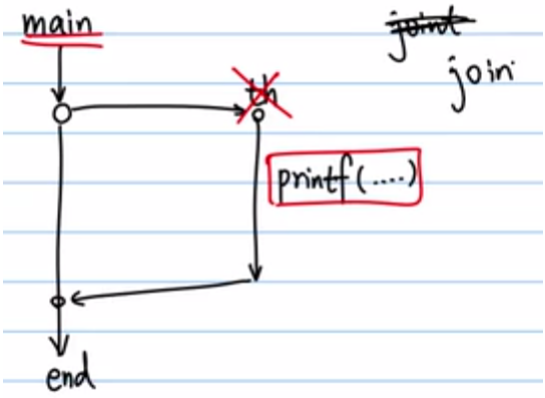
  例如：

```c
void* myfunc(void* args) {
    printf("Hello World\n");
    return NULL;
}

int main() {
    pthread_t th;
    pthread_create(&th, NULL, myfunc, NULL);

    return 0;
}
```

  运行没有结果，解释：



  主线程先结束，子线程没来得及 `printf` 就结束了

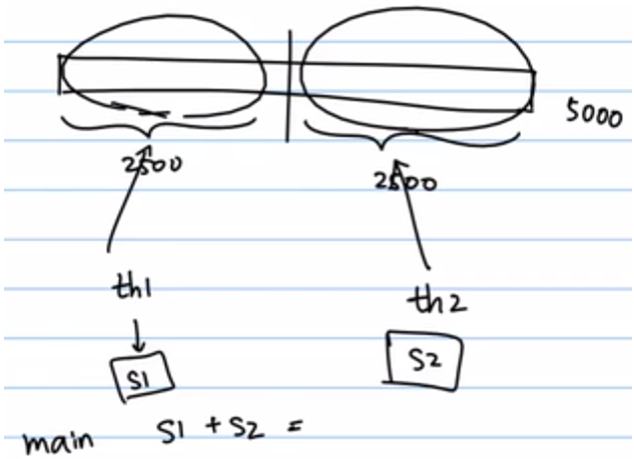


在`return 0;`前面加一个`pthread_join(th, NULL);`，等待子线程th的结束再`return 0;`

- 两条线程：

```
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;
}

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;
}
```

- 定义一个数组`arr[5000]`



最简单做法`myfunc1`加前2500个，`myfunc2`加后2500个

```
pthread_create(&th1, NULL, myfunc1, NULL);
pthread_create(&th1, NULL, myfunc2, NULL);
```

优化代码（`myfunc1`和`myfunc2`代码重复）：

定义一个结构体，内容包括`int first`和`int last`，再把`pthread_create`最后一个参数改成结构体，在`myfunc`里面再强制转换一下类型即可

```
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];
    }
    printf("s = %d\n", s);
    return NULL;
}
```

这样只是在`myfunc`里面输出，并不能返回`s`，也不能在`main`函数里输出，解决办法是在结构体里面加一个`int result`，在`myfunc`函数里把`s`赋值给`result`就可以了

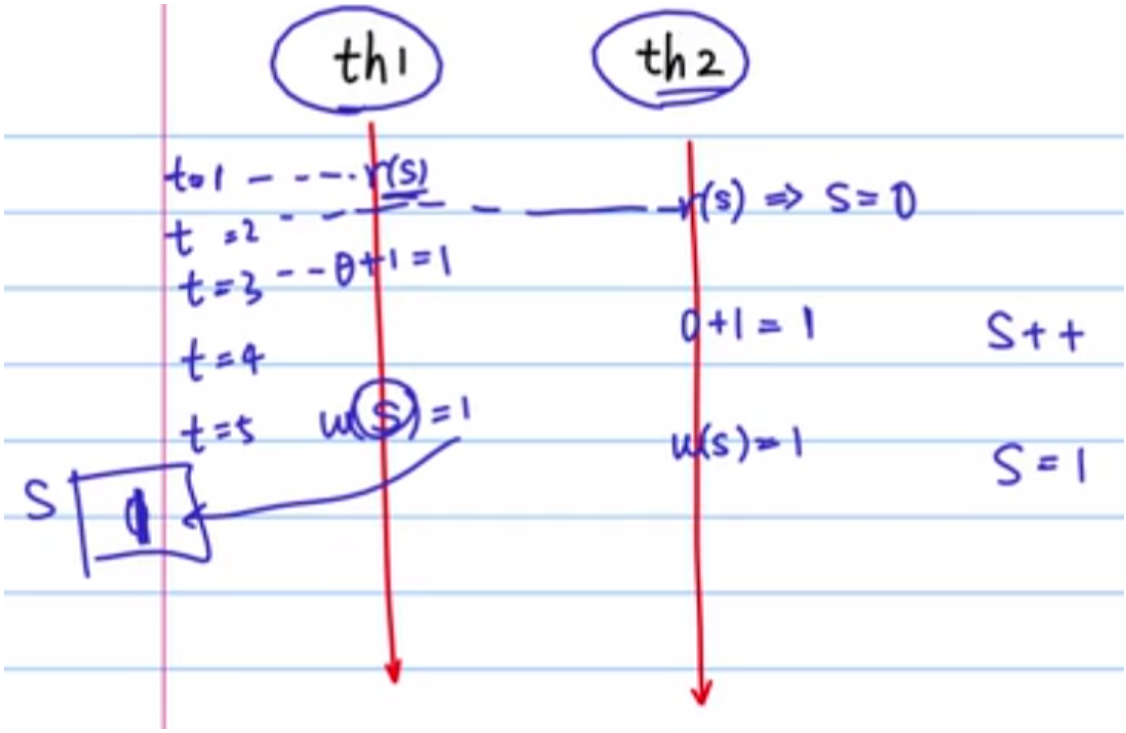
- 一个问题：

```
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;
}
```

这样输出`s`并不等于2000000，why？而且每次运行结果不稳定，丢失了很多次`s++;`解释：

① `s++;`是一个复合语句，先读入s，再s+1，然后把新的s写回内存

②若th1读入s，在th1对s加1之前，th2也读入s，这样就少加了一个1：



这种情况叫做race condition

- race condition的解决方法：加锁

  `pthread_mutex_t lock;`

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

  使用锁：`s++;` —> `pthread_mutex_lock(&lock); s++; pthread_mutex_unlock(&lock);`

  th1和th2用同一个锁，二者一起抢这个锁

- 算法性能（锁的位置关系到代码的效率）：

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

这样一次调用`myfunc`做了100000次加锁和解锁

```
void* myfunc(void* args) {

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

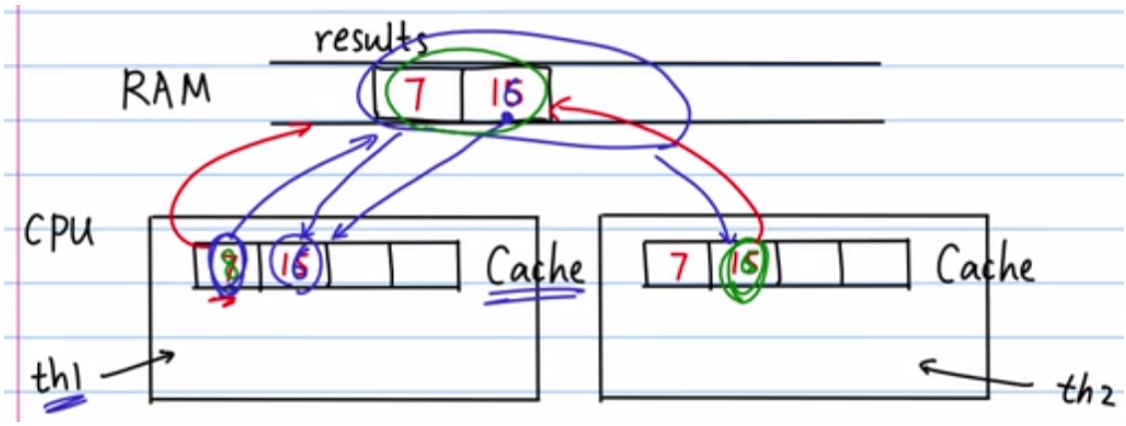
这样一次调用`myfunc`只做1次加锁和解锁，但是这样的话就是th1(或者th2)先走100000次循环，再th2走100000次循环，这样还不如直接在一个线程里写两个for循环，所以回到上面，使用结构体最好

- ①结构体加一个`result`

  ②定义两个全局变量`result[0]`和`result[1]`

  性能比较：前者效率更高，why?

  False sharing假共享，讲解见part4，与CPU和储存有关



提高②的效率，扩大`result[]`，本来是`result[2]`，扩大成`result[101]`，th1结果存在`result[0]`，th2结果存在`result[100]`