# 0110线程控制

# 线程的缺点

- 性能损失
  - 一个很少被外部事件阻塞的计算密集型线程往往无法与共它线程共享同一个处理器。如果计算密集型线程的数量比可用的处理器多，那么可能会有较大的性能损失，这里的性能损失指的是增加了额外的同步和调度开销，而可用的资源不变。
- 健壮性降低
  - 编写多线程需要更全面更深入的考虑，在一个多线程程序里，因时间分配上的细微偏差或者因共享了不该共享的变量而造成不良影响的可能性是很大的，换句话说线程之间是缺乏保护的。
- 缺乏访问控制
  - 进程是访问控制的基本粒度，在一个线程中调用某些OS函数会对整个进程造成影响。
- 编程难度提高
  - 编写与调试一个多线程程序比单线程程序困难得多

```
void* threadRoutine(void* args)
{
    while(true)
    {
        cout << "new thread: " << (char*)args << " running ... " << endl;
        sleep(1);
        int a = 100;
        a /= 0; //让新线程异常
    }
}
```

**通过这个我们可以得出结论：**

1. 线程谁先运行与调度器相关
2. 线程一旦异常，都可能导致整个进程整体退出
3. 线程在创建并执行的时候，线程也是需要等待的，如果只进程不等待，会引起类似于进程的僵尸问题，导致内存泄漏

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
new thread: thread 1 running ...
main thread running ...
Floating point exception
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

```
int main()
{
    pthread_t tid;
    pthread_create(&tid, nullptr, threadRoutine, (void *)"thread 1");

    pthread_join(tid, nullptr); //默认会阻塞等待
    cout << "main thread wait done ... main quit ... " << endl;
#if 0
#endif

    return 0;
}
```

```
void *threadRoutine(void *args)
{
    int i = 0;
    while (true)
    {
        cout << "new thread: " << (char *)args << " running ... " << endl;
        sleep(1);
        if (i++ == 5)
            break;
    }
    //如果我想返回一个数字呢?
    return (void*)10; //问题是: 这个10返回给谁呢? 一般是给main thread
}
```

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

NAME
       pthread_join - join with a terminated thread

SYNOPSIS
       #include <pthread.h>                          返回给它了!

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

       Compile and link with -pthread.

DESCRIPTION
```

```
18      }
19      //如果我想返回一个数字呢?
20      return (void*)10; //问题是: 这个10返回给谁呢? 一般是给main thread
21  }
22
23  int main()
24  {
25      pthread_t tid;
26      pthread_create(&tid, nullptr, threadRoutine, (void *)"thread 1");
27
28      void* ret = nullptr;
29      pthread_join(tid, &ret); //默认会阻塞等待
30      printf("%d\n",ret); // 这样就把10取出来了!
31      cout << "main thread wait done ... main quit ... " << endl;
32  #if 0…
39  #endif
40
```

DEBUG CONSOLE 2 | TERMINAL | PORTS

TERMINAL

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
new thread: thread 1 running ...
new thread: thread 1 running ...
new thread: thread 1 running ...
new thread: thread 1 running ...
new thread: thread 1 running ...
new thread: thread 1 running ...
10
main thread wait done ... main quit ...
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

上面的问题解决之后，就是线程退出问题了

首先，不能在线程的threadRoutine里面直接exit()，**exit直接进程退出了！**

## pthread_exit

```cpp
void *threadRoutine(void *args)
{
    int i = 0;
    while (true)
    {
        cout << "new thread: " << (char *)args << " running ... " << endl;
        sleep(1);
        if (i++ == 5)
            break;
        pthread_exit((void*)13);//终止线程
    }
    //如果我想返回一个数字呢?
    return (void*)10; //问题是：这个10返回给谁呢? 一般是给main thread
}
```

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
new thread: thread 1 running ...
13
main thread wait done ... main quit ...
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
new thread: thread 1 running ...
new thread: thread 1 running ...
new thread: thread 1 running ...
pthread cancel: 139773778888448
-1
main thread wait done ... main quit ...
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

1. join退出结果是-1，这个-1其实就是PTHREAD_CANCELED

可以用新线程取消主线程吗？
别这么干，如果好奇可以试一试

## pthread_cancel
## 线程一直不退出，主线程cancel新线程

```cpp
void *threadRoutine(void *args)
{
    int i = 0;
    while (true)
    {
        cout << "new thread: " << (char *)args << " running ... " << endl;
        sleep(1);
        // if (i++ == 5)
        //     break;
        // pthread_exit((void*)13);//终止线程
    }
    //如果我想返回一个数字呢?
    return (void*)10; //问题是：这个10返回给谁呢? 一般是给main thread
}
```

线程一直死循环不退出

```cpp
int main()
{
    pthread_t tid;
    pthread_create(&tid, nullptr, threadRoutine, (void *)"thread 1");

    int cnt = 3;
    while(cnt--)
    {
        sleep(1);
    }

    pthread_cancel(tid);
    cout << "pthread cancel: " << tid <<endl;

    void* ret = nullptr;
    pthread_join(tid, &ret); //默认会阻塞等待
    printf("%d\n",ret); // 这样就把10取出来了!
    cout << "main thread wait done ... main quit ... " << endl;
#if 0
```

想看取消后的等待结果

# 线程id是什么

```
24
25  int main()
26  {
27      pthread_t tid;
28      pthread_create(&tid, nullptr, threadRoutine, (void *)"thread 1");
29      printf("%lu, %p\n",tid,tid);
30
```

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
140238807496448, 0x7f8be44df700
new thread: thread 1 running ...
new thread: thread 1 running ...
new thread: thread 1 running ...
new thread: thread 1 running ...
10
main thread wait done ... main quit ...
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

**pthread_t 本质是一个地址！**

由于目前我们目前不是用Linux自带的创建线程的接口，我们用的是pthread库中的接口！

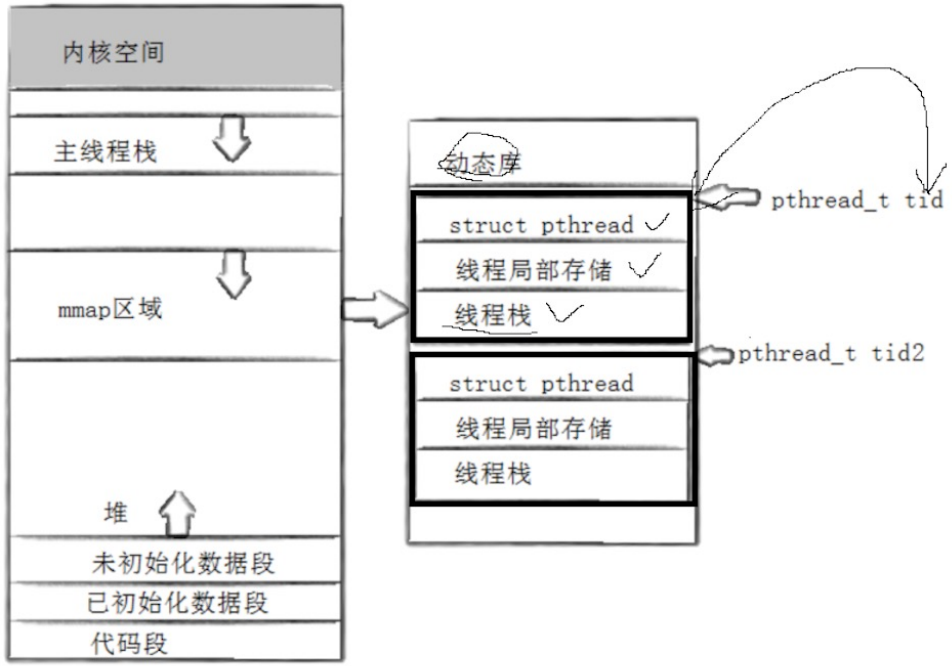
---

如何保证栈区是每一个线程独占的呢？
用户层提供

所以pthread库里面也要先描述再组织

我们ps axj看到的LWP不是线程id

tid是库层面上的



```
CLONE(2)                                Linux Programmer's Manual

NAME
       clone, __clone2 - create a child process

SYNOPSIS
       /* Prototype for the glibc wrapper function */

       #include <sched.h>                    所以操作系统是可以让我们指定栈区的

       int clone(int (*fn)(void *), void *child_stack,
                 int flags, void *arg, ...
                 /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );

       /* Prototype for the raw system call */

       long clone(unsigned long flags, void *child_stack,
                  void *ptid, void *ctid,
                  struct pt_regs *regs);

   Feature Test Macro Requirements for glibc wrapper function (see feature_test_macros(7)):
```

有了线程id，那是不是可以自己cancel自己？
不要这么干

```
void *threadRoutine(void *args)
{
    int i = 0;
    while (true)
    {                                    获取自身线程id
        cout << "new thread: " << (char *)args << " running ... "
             << "mytreadID: " << pthread_self() << endl;
        sleep(1);
        if (i++ == 3)
            break;
        // pthread_exit((void*)13);//终止线程
    }
    // 如果我想返回一个数字呢?
    return (void *)10; // 问题是: 这个10返回给谁呢? 一般是给main thread
}
```

```
TERMINAL
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ make
g++ -o mythread mythread.cc -std=c++11 -lpthread
.(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
139709758891776, 0x7f10b68ce700
new thread: thread 1 running ... mytreadID: 139709758891776
new thread: thread 1 running ... mytreadID: 139709758891776
new thread: thread 1 running ... mytreadID: 139709758891776
new thread: thread 1 running ... mytreadID: 139709758891776
10
main thread wait done ... main continue ...
main thread running ... mytreadID: 139709776271168
main thread running ... mytreadID: 139709776271168
main thread running ... mytreadID: 139709776271168
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

# 下面我们来证明，全局变量被所有线程共享

```cpp
void *threadRoutine(void *args)
{
    while (true)
    {
        cout << (char *)args << " : " << g_val << " &g_val: " << &g_val << endl;
        g_val++;
        sleep(1);
    }
}

int main()
{
    pthread_t tid;
    pthread_create(&tid, nullptr, threadRoutine, (void *)"thread 1");
    printf("%lu, %p\n", tid, tid);

    while (true)
    {
        cout << "main thread: "
             << " : " << g_val << " &g_val: " << &g_val << endl;
        sleep(1);
    }

    return 0;
}
```

新线程给全局变量做修改

主线程也能看到！

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
139942414628608, 0x7f46e1e9b700
    thread 1: 0 &g_val: 0x6011b4
main thread: 1 &g_val: 0x6011b4
    thread 1: 1 &g_val: 0x6011b4
main thread: 2 &g_val: 0x6011b4
    thread 1: 2 &g_val: 0x6011b4
main thread: 3 &g_val: 0x6011b4
    thread 1: 3 &g_val: 0x6011b4
main thread: 4 &g_val: 0x6011b4
    thread 1: 4 &g_val: 0x6011b4
^C
```

```cpp
//__thread: 修饰全局变量，带来的结果就是让每一个线程独自拥有一个全局的变量 --- 线程的局部存储!
__thread int g_val = 0;

void *threadRoutine(void *args)
{
    while (true)
    {
        cout << "    " << (char *)args << ": " << g_val << " &g_val: " << &g_val << endl;
        g_val++;
        sleep(1);
    }
}
```

__thread修饰：线程自己占有一个全局变量！

DEBUG CONSOLE TERMINAL PORTS

TERMINAL

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
140145471362816, 0x7f7629099700
    thread 1: 0 &g_val: 0x7f76290996fc
main thread: 0 &g_val: 0x7f762a12c77c
    thread 1: 1 &g_val: 0x7f76290996fc
main thread: 0 &g_val: 0x7f762a12c77c
    thread 1: 2 &g_val: 0x7f76290996fc
main thread: 0 &g_val: 0x7f762a12c77c
    thread 1: 3 &g_val: 0x7f76290996fc
main thread: 0 &g_val: 0x7f762a12c77c
    thread 1: 4 &g_val: 0x7f76290996fc
^C
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

如果在新线程里面去进行程序替换呢？

```
void *threadRoutine(void *args)
{
    sleep(5);
    execl("/bin/ls", nullptr);
    while (true)
    {
        cout << "    " << (char *)args << ": " << g_val << " &g_val: " << &g_val << endl;
        g_val++;
        sleep(1);
    }
}

int main()
```

一旦进行替换
可以这样理解：

1. 先把所有除了主线程之外的其他线程终止掉
2. 然后把主线程替换

---

下面这个场景：
如果我创建了一个线程，但是我创建完就不想管了，我也不想阻塞等待
我觉得这个线程是一个负担，怎么办

在学习进程的时候这个处理方法是：非阻塞等待/SIGCHLD信号忽略掉的方式

但是线程等待是不能非阻塞的，所以只能是下面这个方法：

##4. 分离线程

> - 默认情况下，新创建的线程是joinable的，线程退出后，需要对其进行pthread_join操作，否则无法释放资源，从而造成系统泄漏。
> - 如果不关心线程的返回值，join是一种负担，这个时候，我们可以告诉系统，当线程退出时，自动释放线程资源。

```
int pthread_detach(pthread_t thread);
```

可以是线程组内其他线程对目标线程进行分离，也可以是线程自己分离:

```
pthread_detach(pthread_self());
```

```
void *threadRoutine(void *args)
{
    pthread_detach(pthread_self());
    while (true)
    {
        cout << (char*)args << " id: " << pthread_self() << endl;
        sleep(1);
    }
}
```

detach之后如果还去join是不行的

**什么场景才回去detach呢？**
**主线程长时间不退出的场景：主线程是服务器，派发任务给新线程去做等，这些场景才会去使用detach**

# C++也给我们提供了线程库

```cpp
#include <iostream>
#include <algorithm>
#include <unistd.h>
#include <thread>

void fun()
{
    while(true)
    {
        std::cout << "hello C++ new thread" << std::endl;
        sleep(2);
    }
}
int main()
{
    std::thread t(fun);
    while(true)
    {
        std::cout << "hello C++ main thread" << std::endl;
        sleep(1);
    }
    t.join();
    return 0;
}
```

Makefile | mythread.cc

Files ▸ BitCodeField ▸ 0110 ▸ Makefile

```
mythread:mythread.cc
    g++ -o $@ $^ -std=c++11 # -lpthread
.PHONY:clean
clean:
    rm -f mythread
```

如果把它去掉

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$ ./mythread
terminate called after throwing an instance of 'std::system_error'
  what():  Enable multithreading to use std::thread: Operation not permitted
Aborted
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

其实是跑不动的

thread底层封装了 pthread.h

# 线程互斥和同步

##5. Linux线程互斥

###进程线程间的互斥相关背景概念

> - 临界资源：多线程执行流共享的资源就叫做临界资源
> - 临界区：每个线程内部，访问临界资源的代码，就叫做临界区
> - 互斥：任何时刻，互斥保证有且只有一个执行流进入临界区，访问临界资源，通常对临界资源起保护作用
> - 原子性（后面讨论如何实现）：不会被任何调度机制打断的操作，该操作只有两态，要么完成，要么未完成

###互斥量mutex

> - 大部分情况，线程使用的数据都是局部变量，变量的地址空间在线程栈空间内，这种情况，变量归属单个线程，其他线程无法获得这种变量。
> - 但有时候，很多变量都需要在线程间共享，这样的变量称为共享变量，可以通过数据的共享，完成线程之间的交互。
> - 多个线程并发的操作共享变量，会带来一些问题。

```c
// 如果多线程访问同一个全局变量，并对它进行数据计算，多线程会互相影响吗?
int tickets = 10000; // 这里的10000就是临界资源

void *getTickets(void *args)
{
    (void)args;
    while (true)
    {
        if (tickets > 0)
        {
            usleep(1000);
            printf("%p: %d\n", pthread_self(), tickets);
            tickets--;
        }
        else
        {
            // 没有票了
            break;
        }
    }
}
int main()
{
    pthread_t t1, t2, t3;
    // 多线程抢票的逻辑
    pthread_create(&t1, nullptr, getTickets, nullptr);
    pthread_create(&t2, nullptr, getTickets, nullptr);
    pthread_create(&t3, nullptr, getTickets, nullptr);
    pthread_join(t1, nullptr);
    pthread_join(t2, nullptr);
    pthread_join(t3, nullptr);

    return 0;
}
```

实际上这一份代码是有问题的

```
0x7f71db69f700: 10
0x7f71dbea0700: 9
0x7f71dae9e700: 8
0x7f71db69f700: 7
0x7f71dbea0700: 6
0x7f71dae9e700: 5
0x7f71db69f700: 4
0x7f71dbea0700: 3
0x7f71dae9e700: 2
0x7f71db69f700: 1
0x7f71dbea0700: 0
0x7f71dae9e700: -1
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0110]$
```

为什么会出现 -1？
这种情况肯定是不能出现的

所以这个代码不加保护
肯定是会出问题的，这个就是并发的时序问题

在并发访问的时候，由于sleep，导致数据不一致的问题
这个挺好理解的，静下心想清楚