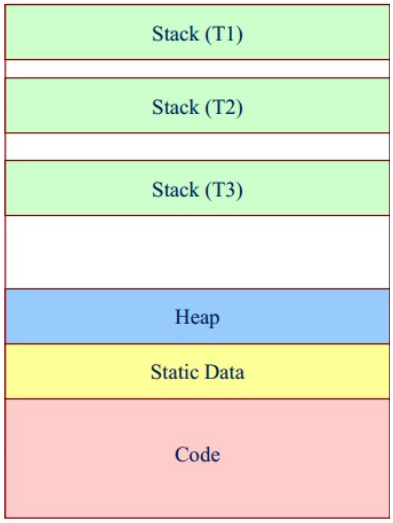
Operating System MIT 6.828 JOS Thread

Computer Science
ChenHao(1100012776)

2013 年 12 月 22 日

1 线程的基本概念

线程是同一进程中的共享同一地址空间的不同的控制流，每个进程可以拥有很多的线程，每个线程并行执行不同的任务。每个线程拥有自己独立的 Program Counter，寄存器组，以及独立的栈空间，同时他们会共享同一块地址空间，也就共享代码，很大一部分数据，以及文件等等。从另一方面来讲可以认为线程是共享地址空间的进程。



为什么我们需要线程，由于线程相对轻量级，在上下文切换的时候，线程上下文切换的开销要比进程上下文切换开销要小，其中不需要切换页目录是最直接的节约开销。另一方面线程在创建和删除所带来的开销也比进程，因此线程在很多情况下要比进程有更好的表现。另一方面进程进行交互比较困难，在 Lab4 中我们可以知道，为了实现进程的交互我们需要 IPC 机制，而且需要陷入到内核，一方面非常麻烦，另一方面开销也非常大。因此我们需要线程，当然线程也有自己需要解决的问题，那就是共享资源会导致竞争。

多线程模型：

1. 内核线程：即内核管理所有的线程，一个内核线程处于阻塞状态时不影响其他的内核线

程。

2. 用户线程：完全由用户库实现线程的创建，调度，撤销，以及调度，不需要任何内核的参与。其中一个比较大的问题是一个进程中的某一个线程因 I/O 被阻塞住时，该进程的别的线程无法被调度。

3. 复合用户线程：用户级别线程映射到内核线程。根据映射的不同分为 Many-To-One, One-To-One, Many-To-Many 的三种不同模式。

2 Linux 中的线程实现

2.1 LinuxThreads

在 Linux Kernel 2.6 以前，Linux 对于线程的实现源于 LinuxThreads 项目，当时的设计者认为同一进程中的线程上下文切换要比进程之间的上下文切换要快，减少了上下文切换的开销。因此 LinuxThreads 采用 One-To-One 的线程模型，其实际的实现方式是在操作系统看来每个线程实际上都为一个轻量级的进程，对于线程的创建和进程创建的入口一致，只是线程共享了同一个地址空间，并且稍作设置。另一个 LinuxThreads 非常著名的特性就是管理线程，管理线程负责对线程创建，线程回收，以及对于信号的处理。

在 LinuxThread 2.4 源代码中，线程的创建通过调用 clone() 的系统调用接口，clone() 通过参数传递并设置其为线程创建后传递至 do_fork() 中，这个函数同时也是 fork 系统调用的入口实现函数。其是否共享同一内存中由宏 CLONE_VM 中控制。

但是 LinuxThreads 拥有许多局限性，一方面管理线程对于线程的管理增加了开销，另外线程的管理方式以及进程号以及信号的处理都与 POSIX 规范不兼容。

2.2 Naive POSIX Thread Library

从 Linux Kernel 2.6 后，Linux 开始采用了新的线程处理，NPTL，它克服了许多 LinuxThreads 的缺点，也符合了 POSIX 的标准。NPTL 实现和 LinuxThreads 类似，也是 One-To-One 的模型，并在也是一个轻量级的进程，线程的创建和进程创建的入口一致，在 Kernel 看来线程和进程几乎没有什么区别。另一方面 NPTL 没有使用管理线程，线程的创建、回收、以及调度都是由内核来管理。管理线程的一些需求，例如向作为进程一部分的所有线程发送终止信号，是并不需要的，因为内核本身就可以实现这些功能。内核还会处理每个线程堆栈所使用的内存的回收工作。它甚至还通过在清除父线程之前进行等待，从而实现对所有线程结束的管理，这样可以避免僵尸进程的问题。最重要的 NPTL 是 POSIX 兼容的。之后 LinuxThreads 也不断改进客服了许多缺点，但是随着 NPTL 深入以及性能非常好，LinuxThreads 依旧依赖于管理线程，依旧存在许多问题。因此慢慢地 Linux 也不再积极地更新了。

3 JOS 中的线程实现

3.1 涉及代码

```
//inc:
inc\thread.h
inc\lib.h

//kern:
kern\env.c:
    void env_free(struct Env *e);   // different handling with main thread and other threads
kern\syscall.c:
    static envid_t sys_exothread(void);             // allocate a new enviroment without allocate new page direcotry
    static int sys_join(envid_t envid);             // check whether given thread has been terminated
    static int sys_env_destroy(envid_t envid);      // killing all threads before main thread exit
kern\syscall.h

//lib:
lib\mem.c           // malloc function
lib\thread.c        // user library

//for check:
user\thread_t1.c    // test threads create and terminate
user\thread_t2.c    // test share memory, and cause race
user\thread_t3.c    // test mutex
```

3.2 基本函数实现

我所实现的 JOS 线程是 One-To-One 模型的，每个线程在内核看来都是一个进程，内核不加区别地对待线程和进程。为了实现线程的回收和调度，我为 struct Env 增加了两项。

```
struct Env {
    ......
    // Thread:
    bool isthread;          // whether is a thread
    envid_t env_tgid;       // thread group id
};
```

对于 pthread_create，十分类似 exec 的实现，只需要令其共享同一个地址空间即可，并设置线程的 eip 指向线程函数的开头，esp 设置为该线程独立的栈空间。对于线程独立的栈空间这一部分我采用了 malloc 的方式，我实现了一个 malloc，利用 malloc 来为线程的创建来分配栈空间。其中调用 sys_exothread 来分配一个不需要分配页目录的 enviroment。在这里我就贴出 pthread create 中的关键代码。

```
int
pthread_create(uint32_t * t_id, void (*f)(void *), void *arg)
{
    char * t_stack = malloc(PGSIZE);            // for thread stack
    struct Trapframe child_tf;

    int childpid = sys_exothread();
    if (childpid < 0) {
        panic("fork sys_exofork error : %e\n", childpid);
    }
```

```
    int r;
    uint32_t sta_top, sta[2];
    sta_top = (uint32_t)t_stack + PGSIZE;
    sta[0] = (uint32_t)exit;                    // return address
    sta[1] = (uint32_t)arg;                     // thread arg
    sta_top -= 2 * sizeof(uint32_t);
    memcpy((void *)sta_top, (void *)sta, 2 * sizeof(uint32_t));

    child_tf = envs[ENVX(childpid)].env_tf;
    child_tf.tf_eip = (uint32_t)f;                  // set eip
    child_tf.tf_esp = sta_top;                      // set esp

    if ((r = sys_env_set_trapframe(childpid, &child_tf)) < 0) {
        cprintf("pthread create: sys_env_set_trapframe: %e\n", r);
        return r;
    }
    if ((r = sys_env_set_status(childpid, ENV_RUNNABLE)) < 0) {
        cprintf("pthread create: set thread status error : %e\n", r);
        return r;
    }

    *t_id = childpid;
    return 0;
}
```

对于函数 pthead_join(envid_t id)，我采用非常 Naive 的方法，不断通过系统调用来看某个线程是否已经结束。

```
int
pthread_join(envid_t id)
{
    int r;
    while (1) {
        r = sys_join(id);
        if (r != 0) return r;
    }
}
```

这个时候，如果任何一个线程退出，则会删除其地址空间，若此时还存在别的线程正在运行，则会导致灾难性的结果。这个是我们想要避免，因此对于子线程的退出，我们只需要删除其 enviroment，但是不回收其地址空间。如果是主线程需要退出，则我们需要删去地址空间，但是为了防止其它线程导致 page fault，因此需要先将所有线程进行回收，如果这个时候别的核正在进行该线程的调度，则标记为 ENV_DYING，等待别的 CPU 将线程调度下 CPU 的时候将其回收。因此我更改了函数 sys_env_destroy 和 env_free。

到目前为止的实验效果：

```c
#include <inc/lib.h>

int num = 0;

void mythread(void * arg) {
    cprintf("Hello from %d\n", ++num);
}

void
umain(int argc, char **argv)
{
    uint32_t id[10];
    int i;
    for (i = 0; i != 10; i++) {
        pthread_create(&id[i], mythread, NULL);
    }
    for (i = 0; i != 10; i++) {
        pthread_join(id[i]);
    }
}
```

```
Device 1 presence: 1
superblock is good
Hello from 1
Hello from 2
Hello from 3
Hello from 4
Hello from 5
Hello from 6
Hello from 7
Hello from 8
Hello from 9
Hello from 10
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

```c
#include <inc/lib.h>

pthread_mutex_t Lock;
int sum;
int k;

void mythread(void * arg) {
    int i, t, g;
    for (i = 0; i != 10000; i++) {
        cprintf("%d\n", sum);
        t = sum;
        for (g = 0; g != 10; g++) k++;
        ++t;
        for (g = 0; g != 10; g++) k++;
        sum = t;
    }
}

void
umain(int argc, char **argv)
{
    pthread_mutex_init(&Lock);
    uint32_t id[2];
    sum = 0;
    pthread_create(&id[0], mythread, NULL);
    pthread_create(&id[1], mythread, NULL);
    pthread_join(id[0]);
    pthread_join(id[1]);
    cprintf("HAHA: %d\n", sum);
}
```

```
19922
19923
19924
19925
19926
19927
HAHA: 19928
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

3.3 Mutex 实现

从上面的实现可以看出现了竞争导致了错误的结果，因此我们需要实现琐。对于琐的实现有多种方式，其中一种是通过关闭中断来实现，通过关闭中断来使得 CPU 不能产生进程切换，从而实现琐，但是在多核上并不适用。另一种方式就是通过体系结构支持的原子操作 xchg 来实现，其原理是通过锁内存总线来实现的，保证在多核上也能正常运行。因此我实现了一个很 Naive 的锁机制：

```
int
pthread_mutex_init(pthread_mutex_t * mutex)
{
    mutex->lock = 0;
    return 0;
}

int
pthread_mutex_lock(pthread_mutex_t * mutex)
{
    while (xchg(&mutex->lock, 1) == 1)
        ;
    return 0;
}

int
pthread_mutex_unlock(pthread_mutex_t * mutex)
{
    xchg(&mutex->lock, 0);
    return 0;
}
```

通过实验，发现成功达到了锁的要求。

```c
#include <inc/lib.h>

pthread_mutex_t Lock;
int sum;
int k;

void mythread(void * arg) {
    int i, t, g;
    for (i = 0; i != 10000; i++) {
        cprintf("%d\n", sum);
        pthread_mutex_lock(&Lock);
        t = sum;
        for (g = 0; g != 10; g++) k++;
        ++t;
        for (g = 0; g != 10; g++) k++;
        sum = t;
        pthread_mutex_unlock(&Lock);
    }
}

void
umain(int argc, char **argv)
{
    pthread_mutex_init(&Lock);
    uint32_t id[2];
    sum = 0;
    pthread_create(&id[0], mythread, NULL);
    pthread_create(&id[1], mythread, NULL);
    pthread_join(id[0]);
    pthread_join(id[1]);
    cprintf("HAHA: %d\n", sum);
}
```

```
19993
19994
19995
19996
19997
19998
19999
HAHA: 20000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

至此，这就实现好了一个简单的 Thread。

3.4 不足与今后的改进

1. 对于主线程退出之后忙等待所有线程调度下 CPU 非常低效.

2. 可以考虑实现线程池来减少创建进程的开销.

3. 这里只实现了简单的 mutex，没有采用内核来管理，可以加入内核参与的 PV 和 Condition Variables.

4 感想与收获

一开始查了很多关于 Linux 线程是如何实现的资料，资料少而且很多都不讲怎么实现。于是我开启了读 Linux 源码的宏大计划，读了一会，结合了一些资料，大概理解了 Linux 的线程实现方法。于是有了 Linux 的帮助下，我就尝试去实现一个 Naive 的线程库。但是实现起来也遇到了许多困难，一方面是共享内存和分离的栈空间的，为了实现分离的栈空间，我一开始思考了好久如何解决，后来发现有同学的大作业是 JOS 的 malloc，我就使用了 malloc 来分配独立的栈空间，来简单实现了共享内存和分离的栈空间。第二个是意料之外的 bug ，就是主线程退出了导致子线程缺页异常，这一部份是我写的时候没考虑周全。主要问题就是写的过程中经常没太想清楚就开始码了，然后写得乱起八糟，遇到 bug 之后又需要花很多时间来处理，不过幸好有之前 5 个 lab 的磨练，对 jos 整体也比较熟悉了，所以最后有惊无险完成了简单的线程实现。

通过这个拓展，独立实现一个线程还是非常有意思的一件事情。虽然写和看内核代码固然比较枯燥，但是发现最终能成功运行真是让人无比激动。