

用户线程库设计 教程

作者：扫地僧

目录

一、设计任务、要求、目的	4
1.1 设计任务	4
1.2 设计目的和要求	4
二、开发环境	4
三、相关原理及算法	5
3.1 不得不知道的知识——c 语言的函数参数传递机制	5
3.1.1 从汇编的角度理解 C 语言函数传参的方式	5
3.1.2 函数调用过程的堆栈变化	6
3.2 神奇的并发执行——线程切换原理	10
3.2.1 栈溢出攻击与函数的跳转	10
3.2.2 从生活的场景来理解线程的切换原理——上下文切换	12
3.2.3 线程切换的实现方式	13
3.3 不让一个人挨饿——时间片轮转调度算法	14
3.3.1 时间片轮转调度算法的基本介绍	14
3.3.2 时间片轮转调度算法的具体分析——该用多长的时间片？	14
四、系统结构和主要的算法设计思路	18
4.1 需要做的工作	18
⑤实现多种线程状态：	18
⑥实现多线程的状态转换：	19
⑦实现多线程的切换	20
⑧实现线程的调度算法：	20
4.2 三思而行——关于算法设计的一些细节	20
1) 如何设计线程的栈空间？	20
2) 在线程切换的过程中怎么保存线程的现场？	20
3) 如何保留 CPU 当前执行指令的下一条指令地址？	20
五、程序实现---主要数据结构	25
5.1 存在的证明，线程的个人档案——线程控制块 TCB	25
5.2 文明的社会都需要排队——线程队列	27
六、程序实现---主要程序清单	28
6.1 线程的创建	28
6.2 线程调度	30
6.3 线程上下文切换	33
6.4 thread_join——阻塞式线程启动	34
6.5 detach——分离式线程启动	35
6.6 等待子线程执行结束而阻塞父线程	35
七、运行的主要界面和结果截图	37
八、总结和感想体会	37
参考文献	39

一、设计任务、要求、目的

1.1 设计任务

- ① 建立用户级线程库；
- ② 至少包括线程的创建、撤销、状态转换等，以及线程切换；
- ③ 库写好后，构建一个程序，演示线程库的使用；

1.2 设计目的和要求

系统应该包含两个部分，一个部分是按内核代码原则设计用户态的线程库，由一系列的函数和以线程控制块 TCB 为核心的一系列数据结构组成；另一个部分是演示系统，调用线程库创建多线程的程序，使其并发执行，以展示系统的运行状态，显示系统的关键数据结构的内容。

二、开发环境

表 1 开发环境配置

环境项目	配置
操作系统	ubuntu 14.04
操作系统位数	32 位
开发工具	VSCODE 32 位
开发语言	c 语言、32 位汇编语言
指令集架构	x86
cpu	intel core i3 8 th GEN

三、相关原理及算法

3.1 不得不知道的知识——c 语言的函数参数传递机制

3.1.1 从汇编的角度理解 C 语言函数传参的方式

C 语言是一门面向过程的、抽象化的通用程序设计语言，广泛应用于底层开发。C 语言能以简易的方式编译、处理低级存储器。C 语言是仅产生少量的机器语言以及不需要任何运行环境支持便能运行的高效率程序设计语言。尽管 C 语言提供了许多低级处理的功能，但仍然保持着跨平台的特性，以一个标准规格写出的 C 语言程序可在包括类似嵌入式处理器以及超级计算机等作业平台的许多计算机平台上进行编译。

C 语言在执行前会先编译生成汇编语言，再由汇编语言转换为机器可理解的二进制语言。那么，C 语言是如何处理函数调用过程中的参数传递的呢？换句话说，在汇编语言层面，C 语言的函数调用过程是怎样的呢？由于我们要设计一个复杂的用户态线程库，其中涉及了很复杂函数调用和线程切换，因此我们不得不先弄清楚上面说的这个问题。

关于 C 语言是如何处理函数之间参数传递的，我总结出来了如下几点：

1) `__cdecl`

`__cdecl` 是 C Declaration 的缩写，表示 C 语言默认的函数调用方法：所有参数从右到左依次入栈，这些参数由调用者清除，称为手动清栈。被调用函数不会要求调用者传递多少参数，调用者传递过多或者过少的参数，甚至完全不同的参数都不会产生编译阶段的错误。

2) `__stdcall`

`__stdcall` 是 Standard Call 的缩写，是 C++ 的标准调用方式：所有参数从右到左依次入栈，如果是调用类成员的话，最后一个入栈的是 `this` 指针。这些

堆栈中的参数由被调用的函数在返回后清除，使用的指令是 `ret nX`，X 表示参数占用的字节数，CPU 在 `ret` 之后自动弹出 X 个字节的堆栈空间，称为自动清栈。函数在编译的时候就必须确定参数个数，并且调用者必须严格的控制参数的生成，不能多，不能少，否则返回后会出错。

3) `__pascal`

`__pascal` 是 Pascal 语言（Delphi）的函数调用方式，也可以在 C/C++ 中使用，参数压栈顺序与前两者相反。返回时的清栈方式与 `__stdcall` 相同。

4) `__fastcall`

`__fastcall` 是编译器指定的快速调用方式。由于大多数的函数参数个数很少，使用堆栈传递比较费时。因此 `__fastcall` 通常规定将前两个（或若干个）参数由寄存器传递，其余参数还是通过堆栈传递。不同编译器编译的程序规定的寄存器不同，返回方式和 `__stdcall` 相当。

5) `__thiscall`

`__thiscall` 是为了解决类成员调用中 `this` 指针传递而规定的。

`__thiscall` 要求把 `this` 指针放在特定寄存器中，该寄存器由编译器决定。VC 使用 `ecx`，Borland 的 C++ 编译器使用 `eax`。返回方式和 `__stdcall` 相当。

`__fastcall` 和 `__thiscall` 涉及的寄存器由编译器决定，因此不能用作跨编译器的接口。所以 Windows 上的 COM 对象接口都定义为 `__stdcall` 调用方式。

C 语言中不加说明默认函数为 `__cdecl` 方式（C 中也只能用这种方式），C++ 也一样，但是默认的调用方式可以在 IDE 环境中设置。

3.1.2 函数调用过程的堆栈变化

举例说明，如果 `cpu` 目前正在执行的函数为 `father()`，而 `father` 即将调用函数 `child(int a)`，在 `__cdecl` 的函数调用规范下，函数栈空间将会发生怎样的变化呢？

假设 `father` 函数如下：

```
void father(){  
  
    ...  
  
    child(120);  
  
    ...  
}
```

`child` 函数如下：

```
void child(int a){  
  
    ...  
  
    int b=a;  
  
    ...  
}
```

`father` 函数经过编译后将会产生如下汇编代码（仅关注函数调用部分）：

```
father proc:  
  
    ...  
  
    push eax  
  
    mov eax,120
```

```
push eax //此处将参数传递给了 child 函数

call child

pop eax

...

ret
```

而 `child` 函数经过汇编之后将会产生如下代码（仅关注函数如何使用参数部分）：

```
child proc:

...

push ebp

mov ebp,esp

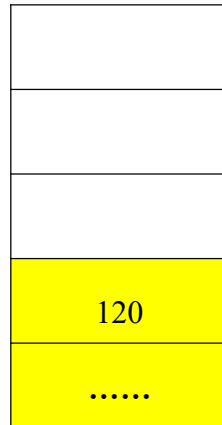
mov eax,[ebp+8] //此处便取出了参数，请思考这一步的意义

...

ret
```

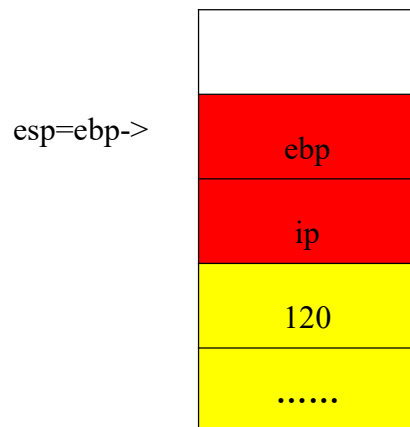
那么在这个过程中，函数的堆栈是如何变化的呢？请看下面进行详细的分解。

在 `father` 执行 `call child` 之前函数的堆栈为下图所示：



此时，120 为 father 向 child 函数传递的参数

Father 执行了 call child 之后，函数的堆栈如下图所示：

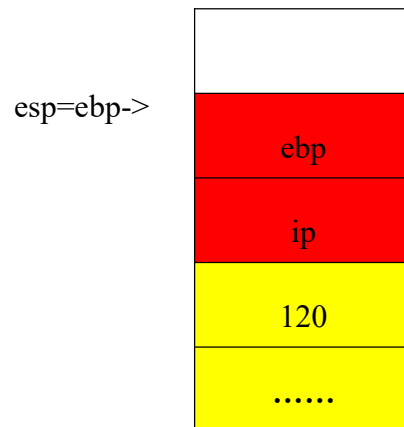


此时，call 将 child 函数的返回地址，即 father 函数下一条要执行的指令地址压入栈中。而且 esp 和 ebp 堆栈指针寄存器指向了当前的栈顶。

现在能明白为什么取参数时，汇编的操作是 mov eax,[ebp+8]了吗？ebp 指向了当前的栈顶，而参数则在返回地址后，32 位机器的字长为 4 字节，故而需要“ebp+8” ;明白了这个道理，我们就能够进行接下来的工作了。

3.2 神奇的并发执行——线程切换原理

3.2.1 栈溢出攻击与函数的跳转



上面我们已经详细讨论了 C 语言中函数是如何传递参数的。首先主调函数将函数的参数压入栈中，然后通过 `call` 指令，将主调函数下一条即将执行的指令地址压入到栈中。当函数执行 `return` 指令的时候，将会把返回地址取出，并且跳转到该地址继续执行指令。

于是我们便可以思考一个问题——如果我们修改了这个返回地址 `ip` 会发生什么呢？

我们来看下面这个例子：

```
void fun() {  
  
    while(1) {  
  
        printf("Hello, I'm fun!\n");  
  
        sleep(1);  
  
    }  
}
```

```

}

int main() {

    int a[5] = { 0 };

    // 传说中的溢出攻击

    a[5] = (int)fun;

    a[6] = (int)fun;

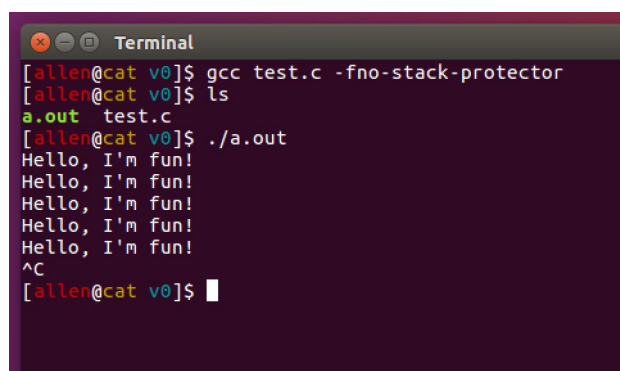
    a[7] = (int)fun;

    return 0;

}

```

很显然上面给出了这个例子，一眼就能看出来，它存在一个函数数组下标越界的问题。不过我们不用关心这个问题，因为我们可以通过编译器指令取消编译器对数组下标越界的检查使它能够正常编译通过。这就成了大名鼎鼎的栈溢出攻击。这一段代码执行起来会有怎么样的结果呢？我们看看下图：



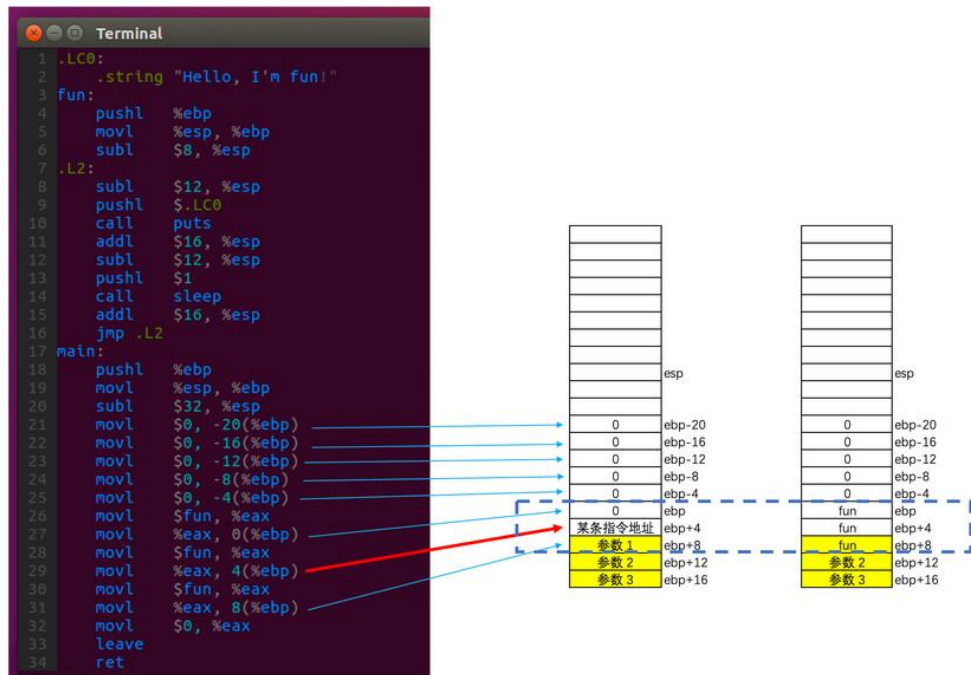
```

Terminal
[allen@cat v0]$ gcc test.c -fno-stack-protector
[allen@cat v0]$ ls
a.out  test.c
[allen@cat v0]$ ./a.out
Hello, I'm fun!
Hello, I'm fun!
Hello, I'm fun!
Hello, I'm fun!
Hello, I'm fun!
^C
[allen@cat v0]$

```

看到这个结果，我们不禁想问，为什么会开始跳转执行函数 `fun` 呢？我们先

来看看其中堆栈的变化情况。如下图所示：



现在看明白了吗？没错，就是函数原本的返回地址变成了另外一个函数的地址，因此当 `main` 函数返回的时候，它将会跳转执行我们预期的 `fun` 函数。

3.2.2 从生活的场景来理解线程的切换原理——上下文切换

至此，我们已经讨论完了函数的跳转问题，下面我们将开始讨论线程到底是如何切换的。在正式研究线程的切换方法之前，我们先来思考一个生活中常见的场景。

小绿是一个作家，他写的小说非常的精彩。小红约小绿下午一起出去吃饭，小绿愉快的答应了。然而小绿并不知道小红什么时候会来约她出去吃饭，他不能明确的知道是几点几分几秒。可是小绿也不可能一直等着小红约他吃饭，他决定利用这个等待的时间，继续写最近构思的新小说。不知不觉，时间慢慢流逝，小红给小绿打了一个电话，告诉小绿吃饭的时间到了。然而此时的小绿并没有完成自己全部的工作，小说还没有写完。于是小绿在出发前先把自己已经写了的部分保存了起来，并且在笔记本上记录下来了接下来该写的内容，以免回来的时候忘

了写到哪里了。在愉快的晚餐之后，小绿回到了自己的工作室，打开吃饭前保存的小说文档，看了看之前笔记本上记录的内容——关于接下来要写的内容，于是继续写起了小说。

解析：

在上面的例子中，我们可以把小绿当做一个 CPU，这个 CPU 当前需要处理两个线程，一个是写小说，一个是吃饭。CPU 一开始正在执行的线程是写小说。小红打电话叫小绿出去吃饭则代表着线程调度的发生。小绿在出去吃饭之前。把已经写了的部分保存了起来，象征着线程切换之前保留现场，而最后记录下来接下来该写的内容代表着线程切换前记录了下一条该执行的指令，以便这个现场重新得到资源之后能够继续之前的工作，顺着往下执行。

3.2.3 线程切换的实现方式

现在我们明白了线程切换的两个要点。第 1 个要点是保存现场，第 2 个要点则是记录接下来需要执行的指令。我们需要细化的考虑一下——现场指的是什么，接下来需要执行的指令指的是什么，以及如何保存现场现场该保存在哪里？

显而易见，“保存现场”则是保存 CPU 内部的寄存器状态。保存接下来需要执行的指令就是需要保存接下来要执行的指令的地址。这些内容该保存到哪里呢？当然是线程的栈里面。

具体来说，线程的切换有以下几个要点：

- 1) 我们需要为每一个线程设立一个独立的，互相不干扰的栈空间。
- 2) 当线程发生切换的时候，当前线程被切换之前，需要把自己的现场进行完好的保留，同时记录下下一条需要执行指令的指令地址。
- 3) 把 CPU 的栈顶指针寄存器 `esp` 切换到即将被调入的线程的堆栈的栈顶地址，完成了线程栈空间的切换。

经过上述这几个步骤，我们便完成了线程的切换，由于上面的步骤需要直接

访问 CPU 的寄存器，于是这个过程往往是采用汇编的方式来进行。

3.3 不让一个人挨饿——时间片轮转调度算法

3.3.1 时间片轮转调度算法的基本介绍

时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法。每个线程被分配一个时间段，称作它的时间片，即该线程允许运行的时间。如果在时间片结束时线程还在运行，则 CPU 将被剥夺并分配给另一个线程。如果线程在时间片结束前阻塞或结束，则 CPU 当即进行切换。调度程序所要做的就是维护一张就绪线程列表，当线程用完它的时间片后，它被移到队列的末尾。

时间片轮转调度中值得关注的一点是时间片的长度。从一个线程切换到另一个线程是需要一定时间的--保存和装入寄存器值及内存映像，更新各种表格和队列等。假如线程切换(process switch) - 有时称为上下文切换(context switch)，需要 5 毫秒，再假设时间片设为 20 毫秒，则在做完 20 毫秒有用的工作之后，CPU 将花费 5 毫秒来进行线程切换。CPU 时间的 20%被浪费在了管理开销上。

为了提高 CPU 效率，我们可以将时间片设为 500 毫秒。这时浪费的时间只有 1%。但考虑在一个分时系统中，如果有十个交互用户几乎同时按下回车键，将发生什么情况？假设所有其他线程都用足它们的时间片的话，最后一个不幸的线程不得不等待 5 秒钟才获得运行机会。多数用户无法忍受一条简短命令要 5 秒钟才能做出响应。同样的问题在一台支持多道程序的个人计算机上也会发生。

结论可以归结如下：时间片设得太短会导致过多的线程切换，降低了 CPU 效率；而设得太长又可能引起对短的交互请求的响应变差。将时间片设为 100 毫秒通常是一个比较合理的折中。

3.3.2 时间片轮转调度算法的具体分析——该用多长的时间片？

该算法中，将一个较小时间单元定义为时间量或时间片。时间片的大小通常

为 10~100ms。就绪队列作为循环队列。CPU 调度程序循环整个就绪队列，为每个线程分配不超过一个时间片的 CPU。

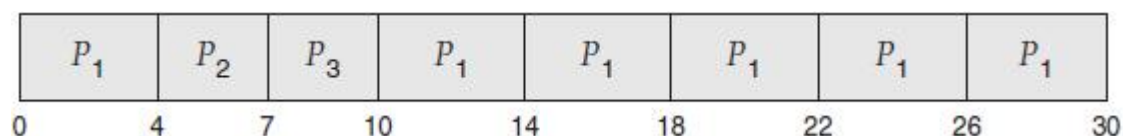
为了实现 RR 调度，我们再次将就绪队列视为线程的 FIFO 队列。新线程添加到就绪队列的尾部。CPU 调度程序从就绪队列中选择第一个线程，将定时器设置在一个时间片后中断，最后分派这个线程。

接下来，有两种情况可能发生。线程可能只需少于时间片的 CPU 执行。对于这种情况，线程本身会自动释放 CPU。调度程序接着处理就绪队列的下一个线程。否则，如果当前运行线程的 CPU 执行大于一个时间片，那么定时器会中断，进而中断操作系统。然后，进行上下文切换，再将线程加到就绪队列的尾部，接着 CPU 调度程序会选择就绪队列内的下一个线程。

不过，采用 RR 策略的平均等待时间通常较长。假设有如下一组线程，它们在时间 0 到达，其 CPU 执行以 ms 计：

线程	执行时间
P1	24
P2	3
P3	3

如果使用 4ms 的时间片，那么 P1 会执行最初的 4ms。由于它还需要 20ms，所以在第一个时间片之后它会被抢占，而 CPU 就交给队列中的下一个线程。由于 P2 不需要 4ms，所以在其时间片用完之前就会退出。CPU 接着交给下一个线程，即线程 P3。在每个线程都得到了一个时间片之后，CPU 又交给了线程 P1 以便继续执行。因此，RR 调度结果如下：



现在，我们计算这个调度的平均等待时间。P1 等待 $10-4 = 6\text{ms}$ ，P2 等待 4ms ，而 P3 等待 7ms 。因此，平均等待时间为 $17/3 = 5.66\text{ms}$ 。

在 RR 调度算法中，没有线程被连续分配超过一个时间片的 CPU（除非它是唯一可运行的线程）。如果线程的 CPU 执行超过一个时间片，那么该线程会被抢占，并被放回到就绪队列。因此，RR 调度算法是抢占的。

如果就绪队列有 n 个线程，并且时间片为 q ，那么每个线程会得到 $1/n$ 的 CPU 时间，而且每次分得的时间不超过 q 个时间单元。每个线程等待获得下一个 CPU 时间片的时间不会超过 $(n-1)q$ 个时间单元。例如，如果有 5 个线程，并且时间片为 20ms ，那么每个线程每 100ms 会得到不超过 20ms 的时间。

RR 算法的性能很大程度取决于时间片的大小。在一种极端情况下，如果时间片很大，那么 RR 算法与 FCFS 算法一样。相反，如果时间片很小（如 1ms ），那么 RR 算法可以导致大量的上下文切换。

例如，假设我们只有一个需要 10 个时间单元的线程。如果时间片为 12 个时间单元，那么线程在一个时间片不到就能完成，而且没有额外开销。如果时间片为 6 个时间单元，那么线程需要 2 个时间片，并且还有一个上下文切换。如果时间片为 1 个时间单元，那么就会有 9 个上下文切换，相应地使线程执行更慢（图 1）。

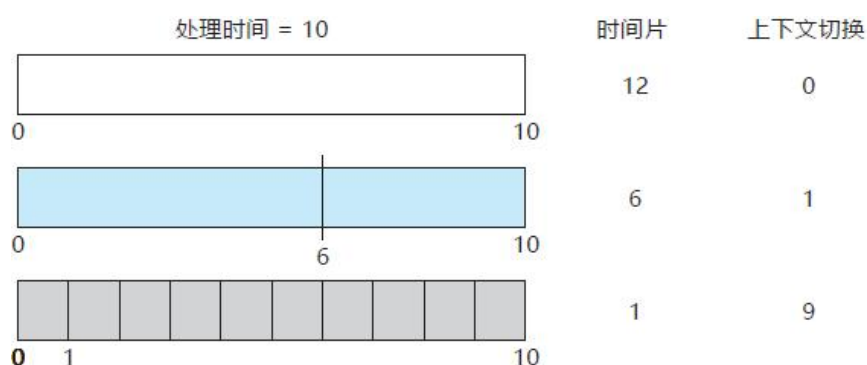


图 1 更小时间片如何增加上下文切换

因此，我们希望时间片远大于上下文切换时间。如果上下文切换时间约为时间片的 10%，那么约 10% 的 CPU 时间会浪费在上下文切换上。在实践中，大多数现代操作系统的时间片为 10~100ms，上下文切换的时间一般少于 10ms；因此，上下文切换的时间仅占时间片的一小部分。

周转时间也依赖于时间片大小。正如从图 2 中所看到的，随着时间片大小的增加，一组线程的平均周转时间不一定会改善。一般情况下，如果大多数线程能在一个时间片内完成，那么平均周转时间会改善。

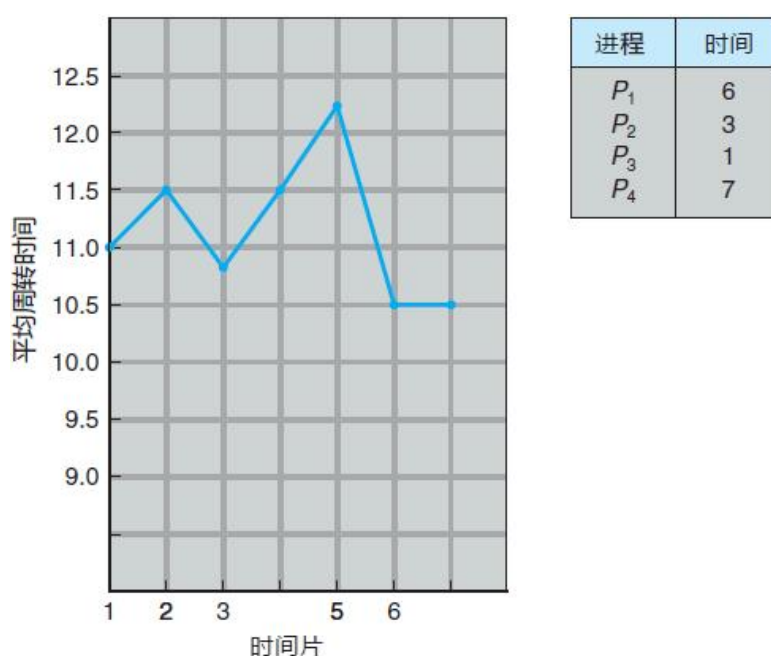


图 周转时间如何随着时间片大小而改变

例如，假设有三个线程，都需要 10 个时间单元。如果时间片为 1 个时间单元，那么平均周转时间为 29；如果时间片为 10，那么平均周转时间会降为 20；如果再考虑上下文切换时间，那么平均周转时间对于较小时时间片会增加，这是因为需要更多的上下文切换。

尽管时间片应该比上下文切换时间要大，但也不能太大。如果时间片太大，那么 RR 调度就演变成了 FCFS 调度。根据经验，80% 的 CPU 执行应该小于时间片。

四、系统结构和主要的算法设计思路

上面已经详细的讨论了线程切换的原理，以及线程调度算法的原理，下面我们需要开始考虑该如何实现这些技术已经该如何合理的构建所需要的数据结构。

4.1 需要做的工作

具体来说，我们需要做以下工作：

- ①建立描述线程的数据结构 TCB;
- ②建立描述线程队列的数据结构 tasks;
- ③通过调用 `thread_create` 函数创建线程;
- ④使用两种方式启动线程:

(a) 通过调用函数 `detach` 实现线程的分离式启动(父线程不必等待子线程执行结束，可以继续执行)

(b) 通过调用 `thread_join` 实现阻塞式启动(父调线程等待该子线程结束后才能继续执行)；

- ⑤实现多种线程状态:

线程状态表

状态名称	含义	状态特点
THREAD_READY	线程就绪	线程正在运行
THREAD_RUNNING	线程可调度	线程处于可调度队列，但暂时没有得到 cpu 资源
THREAD_SLEEP	线程睡眠	线程会在睡眠时间内不参与线程的调度，保持沉默
THREAD_BLOCK	线程阻塞	线程由于等待某个事件发生而阻塞，不接受 cpu 调度
THREAD_STOP	线程停止	线程收到调控，停止运行
THREAD_DISPOSED	线程撤销	线程的空间需要被撤销掉

⑥实现多线程的状态转换：

通过一系列的线程状态转换函数来实现线程的状态切换，这些函数如下表所示：

线程状态转换函数

函数名	参数	函数作用
resume	int tid	将标号为 tid 的线程状态设置为 THREAD_RUNNING
wait	int tid	将标号为 tid 的线程状态设置为 THREAD_BLOCK
mysleep	int tid	将标号为 tid 的线程状态设置为 THREAD_SLEEP
dispose	int tid	将标号为 tid 的线程状态设置为 THREAD_DISPOSED
remove_th	int tid	将标号为 tid 的线程状态设置为 THREAD_STOP

⑦实现多线程的切换

线程的切换需要通过汇编来完成。因此需要编写汇编代码完成线程上下文的切换。线程切换的方式有两种，第一种是主动切换，调用 `schedule` 完成切换到指定线程的任务；另外一种则是根据线程调度来完成切换，当线程需要调度时，自动完成线程切换；

⑧实现线程的调度算法：

本次设计的线程库采用了时间片轮转调度算法，该算法根据线程优先级为每个线程设置了时间片，使得每一个线程都能有相对公平的机会得到系统资源。

4.2 三思而行——关于算法设计的一些细节

我们需要提几个问题，反反复复考虑相关的实现细节之后再进行编码实现，这样会极大的提高我们编码的效率。

1) 如何设计线程的栈空间？

我们可以通过动态分配一个连续的地址空间来作为线程的栈空间。

2) 在线程切换的过程中怎么保存线程的现场？

通过汇编指令来完成对现场的保留，主要是对各个寄存器的压栈操作。

3) 如何保留 CPU 当前执行指令的下一条指令地址？

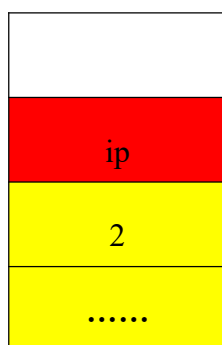
通过 `call` 指令来完成对下一条指令地址的保存，这里是对 `call` 指令一个非常巧妙的使用方法，需要非常丰富的编程经验以及对计算机汇编语言的升入理解。下面将详细讲解这一点。

前面已经分析过了，我们需要设计两种线程切换的方式，即主动切换和时钟中断切换。主动切换需要当前线程调用调度函数 `schedule` 来完成，而时

钟中断切换则是通过设置时钟中断，中断后执行 `schedule` 函数来完成。我们分别考虑这两种情况下该如何保留下一条指令的地址。提前说明：`switch(tid)` 为一个汇编函数，作用为切换到 `tid` 线程。

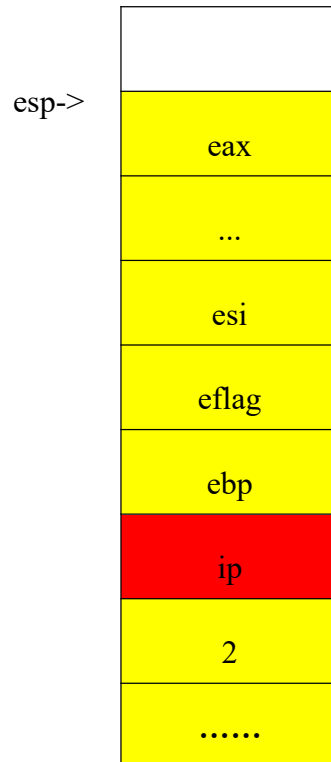
①主动调度

当线程主动调度的时候，线程 1 调用 `switch` 函数切换到线程 2，当线程 1 执行了 `call switch` 指令后，线程 1 的栈空间如下图所示：



`ip` 则保留了 `switch` 函数执行完成后需要执行的指令地址，也即为一条指令的地址。

当线程 1 完成了现场的保存后，线程 1 的栈空间如下图所示：



最后，将 esp 保存到 TCB 中即可。

当线程 1 重新被调入执行的时候，只需要将 esp 切换为 tcb 中保存的 esp，这样便完成了线程栈空间的切换，然后 switch 函数将会执行回复线程的操作，将线程 1 栈中的状态一一弹出，最后 esp 指向了 ip，然后 ret 操作之后，线程 1 便恢复了原来的执行状态。

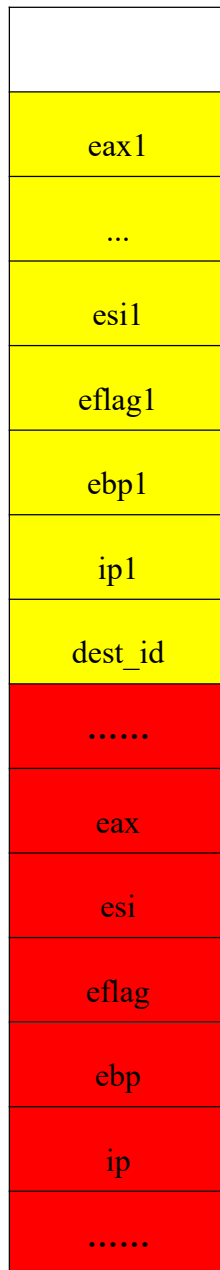
②时间片结束中断调度

线程 1 在执行过程中时间片执行结束，产生中断，执行中断处理函数 schedule，schedule 函数会从线程队列 tasks 中挑选一个最合适的线程，然后将其调入 cpu 执行。因此，当中断产生后，线程 1 的栈变成如下状态，首先是保存线程状态（中断基操）。



注意，此处的 `ip` 为中断前线程下一条指令的地址。

接下来，线程 1 响应中断，`call schedule`（实际响应中段并不是 `call`，此处为简化描述暂记为 `call`）之后线程 1 的栈为：



注：上图中红色部分为中断前保留的线程 1 现场，黄色部分为现场切换前保留的现场，`dest_id` 为 `schedule` 选出的需要被调入执行的目标线程 id，而 `ip1` 则为中断服务程序调用 `schedule` 并执行玩成后需要执行的下一条指令的地址。故由上可知，当线程 1 重新被调入时，线程切换部分（黄色部分）的现场保留与恢复上文已经说过，不再赘述，而黄色部分的现场被恢复后，紧接着就是中断现场的恢复了——红色部分的状态被恢复，也就是恢复到了线程 1 中断前的状态，使得线程 1 最终恢复到了原来的状态。

至此，我们已经完成了对线程调度主要算法的分析与设计。

五、程序实现---主要数据结构

5.1 存在的证明，线程的个人档案——线程控制块 TCB

线程控制块（Thread Control Block，TCB）是与进程的控制块（PCB）相似的子控制块，只是 TCB 中所保存的线程状态比 PCB 中保存少而已。

在用户态线程库中 TCB 是线程存在的唯一证明，通过控制 TCB 中的数据，我们可以对线程的状态等一系列的事务进行操作。因此 TCB 是线程的核心数据结构。据此，我们设计出了一下数据结构作为 TCB。注意，这个结构设计得非常巧妙紧凑，每一个数据项都与用户态线程库点实现密切相关，读者可以仔细思考并加以体会。如下所示是 TCB 的数据结构：

```
struct task_struct {  
  
    int id; //线程的标识符  
  
    void (*th_fn)(); //指向线程函数的函数指针  
  
    int esp; //用来在发生线程切换是保存线程的栈顶地址  
  
    unsigned int wakeuptime; // 线程唤醒时间  
  
    int status; // 线程状态  
  
    int counter; // 时间片  
  
    int priority; // 线程优先级  
  
    int stack[STACK_SIZE]; //现场的栈空间  
  
};
```

task_struct 即为 TCB，此处只是命名不同而已。TCB 中各个数据的详细作用

已经标注在了上图中，但是，仍然有一些地方需要我们特别注意。

`th_fn` 为指向函数的指针，我们传入该参数的时候，需要把函数的地址以整数的形式传入。`esp` 则记录了栈顶，一开始初始化的时候，`stack` 栈中预先保存好了现场的初始状态以便线程进行调度，而 `esp` 初始时则指向了 `stack` 的栈顶，初始 `stack` 如下图所示：

```
stack[STACK_SIZE-11] = 0; // eflags
stack[STACK_SIZE-10] = 0; // eax
stack[STACK_SIZE-9] = 0; // edx
stack[STACK_SIZE-8] = 0; // ecx
stack[STACK_SIZE-7] = 0; // ebx
stack[STACK_SIZE-6] = 0; // esi
stack[STACK_SIZE-5] = 0; // edi
stack[STACK_SIZE-4] = 0; // old ebp
stack[STACK_SIZE-3] = (int)start; // ret to start 线程第一次被调度时会在此启动
// start 函数栈帧，刚进入 start 函数的样子
stack[STACK_SIZE-2] = 100; // ret to unknown, 如果 start 执行结束，表明线程结束
stack[STACK_SIZE-1] = (int)tsk; // start 的参数
```

由上可知，在初始化线程的时候，我们需要在 `stack` 中设置好线程上下文的初始环境，同时传入 `start` 函数的地址作为启动函数的地址。

`wakeupime` 是做什么用的呢？`wakeupime` 指定的线程从睡眠状态并唤醒的时间点。当线程调用 `sleep` 函数之后，`Wakeup ime` 便被设置为当前的时间加上线程需要休眠的时间，同时线程的状态被设置为 `thread_sleep`。当线程发生调度的时候，调度函数会检查每一个处于睡眠状态的线程，如果当前的时间大于

wakeuptime 则将其状态设置为 `THREAD_RUNNING`，重新参与现场的调度。

另外要注意的是，`counter` 是如何发挥作用的呢？我们为每一个线程设定了一定数量的时间片，`counter` 记录了线程当前还有多少时间片，每一个时间片都是一个单位的时间，比如说 10 毫秒。每一个时间片结束都会发生一次调度中断，这个中断的中断服务子程序会检查当前线程的 `counter` 是否小于 0，如果小于零则代表当前线程的时间片用完了而需要进行线程的调度，调度算法会从线程队列中寻找另外一个可调度的而且 `counter` 大于 0 且 `counter` 最大的线程进行调度，否则的话直接结束中断。

`priority` 代表着线程的优先级，当所有的县城时间片都已经用完的时候，需要重新为每一个线程分配时间片。每一个行程卡分配多少的时间变得这个就由 `priority` 来决定。优先级高的线程能够分配到更多的时间片，而优先级低的线程分配到时间片就相对的少，这样便实现了线程之间的优先级，让优先级高的线程能够得到更多的 CPU 处理时间，而线程优先级低的线程 CPU 处理时间则相对较少。

5.2 文明的社会都需要排队——线程队列

在 5.1 节已经详述了现成的核心数据结构——TCB 的设计。然而该结构仅仅是针对于每一个线程来进行设计的，我们还需要一个数据结构来将所有的线程集合起来，让调度算法可以对其进行统一的操作。这个数据结构便是线程队列。

队列有以下特点：

- 1、队列的容量一旦在构造时指定，后续不能改变；
- 2、插入元素时，在队尾进行；删除元素时，在队首进行；
- 3、队列满时，插入元素会阻塞线程；队列空时，删除元素也会阻塞线程；
- 4、支持公平/不公平策略，默认为不公平策略（5.1 节讲述了更具 `counter` 值的大小进行调度）。

同时，队列应该是多队列，包括阻塞队列，就绪队列，睡眠队列等，然而本

设计只设计一个队列就完成了所有队列的功能，其巧妙之处就在于，在线程的 TCB 中我们设置了一个线程状态标志，第 2 度算法根据这个标识就能够进行合理的调度了。

具体，该线程的队列设计如下：

```
static struct task_struct init_task = {0, NULL, 0, THREAD_RUNNING, 0, 15, 15, {0}};
};

struct task_struct *current = &init_task;

struct task_struct *task[NR_TASKS] = {&init_task,};
```

init_task 代表的是当前的主线程,其线程 id 为 0;

至此，我们已经把用户态线程库的主要的设计思想，设计原理，算法，以及其核心数据结构详细的分析清楚了。下面我们再来看一看一些重要的算法以及函数的实现源代码。

六、程序实现---主要程序清单

6.1 线程的创建

```
int thread_create(int *tid, void (*start_routine)()) {

    int id = -1;

    struct task_struct *tsk = (struct task_struct*)malloc(sizeof(struct task_struct));

    while(++id < NR_TASKS && task[id]);

    if (id == NR_TASKS) return -1;
```

```

task[id] = tsk;

if (tid) *tid = id; //返回值

tsk->id = id;

tsk->th_fn = start_routine;

int *stack = tsk->stack; // 栈顶界限

tsk->esp = (int)(stack+STACK_SIZE-11);

tsk->wakeup_time = 0;

tsk->status = THREAD_STOP;

tsk->counter = 15;

tsk->priority = 15;

// 初始 switch_to 函数栈帧

stack[STACK_SIZE-11] = 0; // eflags

stack[STACK_SIZE-10] = 0; // eax

stack[STACK_SIZE-9] = 0; // edx

stack[STACK_SIZE-8] = 0; // ecx

stack[STACK_SIZE-7] = 0; // ebx

stack[STACK_SIZE-6] = 0; // esi

stack[STACK_SIZE-5] = 0; // edi

stack[STACK_SIZE-4] = 0; // old ebp

stack[STACK_SIZE-3] = (int)start; // ret to start 线程第一次被调度时会在此启动

// start 函数栈帧，刚进入 start 函数的样子

```

```

    stack[STACK_SIZE-2] = 100; // ret to unknown, 如果 start 执行结束, 表明线程结束

    stack[STACK_SIZE-1] = (int)tsk; // start 的参数

    /*
    汇编函数调用,c 风格参数传递

    传入参数分别是 IP,c1,c2

    */

    return 0;
}

```

6.2 线程调度

```

void schedule() {

    //线程的调度函数

    struct task_struct *next = pick();

    if (next) {

        switch_to(next); //线程的上下文切换

    }

}

static struct task_struct *pick() {

```

```

/*找到时间片最大的线程进行调度*/

int i, next, c;

for (i = 0; i < NR_TASKS; ++i) {

    if(!task[i])continue;

    if( task[i]->status == THREAD_EXIT){

        if(task[i]!=current)

            remove_th(i);

        continue;

    }

    if (task[i]->status == THREAD_DISPOSED)

    {

        if(task[i]!=current)

            remove_th(i);

        continue;

    }

    if (task[i]->status != THREAD_STOP&& task[i]->status != THREAD_BLOCK

        && getmstime() > task[i]->wakeuptime) {

        task[i]->status = THREAD_RUNNING;

    }

}

//上面的作用是唤醒睡眠的线程,使其可以接受调度

```

```

while(1) {

    c = -1;

    next = 0;

    for (i = 0; i < NR_TASKS; ++i) {

        if (!task[i]) continue;

        if (task[i]->status == THREAD_RUNNING && task[i]->counter > c) {

            c = task[i]->counter;

            next = i;

        }

    }

    if (c) break;

    // 如果所有任务时间片都是 0，重新调整时间片的值

    if (c == 0) {

        for (i = 0; i < NR_TASKS; ++i) {

            if(task[i]) {

                task[i]->counter = task[i]->priority + (task[i]->counter >> 1);

            }

        }

    }

}

return task[next];

```



```
}
```

6.3 线程上下文切换

```
.section .text

.global switch_to

switch_to:

    call closealarm /* 模拟关中断 */

    push %ebp

    mov %esp, %ebp /* 更改栈帧，以便寻参 */

    /* 保存现场 */

    push %edi

    push %esi

    push %ebx

    push %edx

    push %ecx

    push %eax

    pushfl

    /* 准备切换栈 */

    mov current, %eax /* 取 current 基址放到 eax */

    mov %esp, 8(%eax) /* 保存当前 esp 到线程结构体 */
```

```
mov 8(%ebp), %eax /* 8(%ebp)即为 c 语言的传入参数 next 取下一个线程结构体基址*/
```

```
mov %eax, current /* 更新 current */
```

```
mov 8(%eax), %esp /* 切换到下一个线程的栈 */
```

```
/* 恢复现场, 到这里, 已经进入另一个线程环境了, 本质是 esp 改变 */
```

```
popfl
```

```
popl %eax
```

```
popl %ecx
```

```
popl %edx
```

```
popl %ebx
```

```
popl %esi
```

```
popl %edi
```

```
popl %ebp
```

```
call openalarm /* 模拟开中断 */
```

```
ret
```

6.4 thread_join——阻塞式线程启动

```
int thread_join(int tid) {  
  
    while(task[tid]&&task[tid]->status != THREAD_EXIT) {  
  
        if(task[tid]->status==THREAD_STOP){  
  
            task[tid]->status=THREAD_RUNNING;
```

```

    }

    schedule();

}

}

```

6.5 detach——分离式线程启动

```

void detach(int tid){

    if(task[tid]!=NULL && task[tid]->status==THREAD_STOP&& task[tid]->status!=
THREAD_EXIT){

        task[tid]->status=THREAD_RUNNING;

        schedule();

    }

}

```

6.6 等待子线程执行结束而阻塞父线程

```

void wait_all(){

int i=0;

int remain=0;

while(1){

    remain=0;

```

```

for(i=1;i<NR_TASKS;i++){

if(task[i]&&task[i]->status!=THREAD_EXIT){

    remain=1;

    schedule();

    break;

    continue;

}

}

if(!remain){

    break;

}

}

}

void wait_thread(int tid){

while (task[tid]&&task[tid]->status != THREAD_EXIT)

{

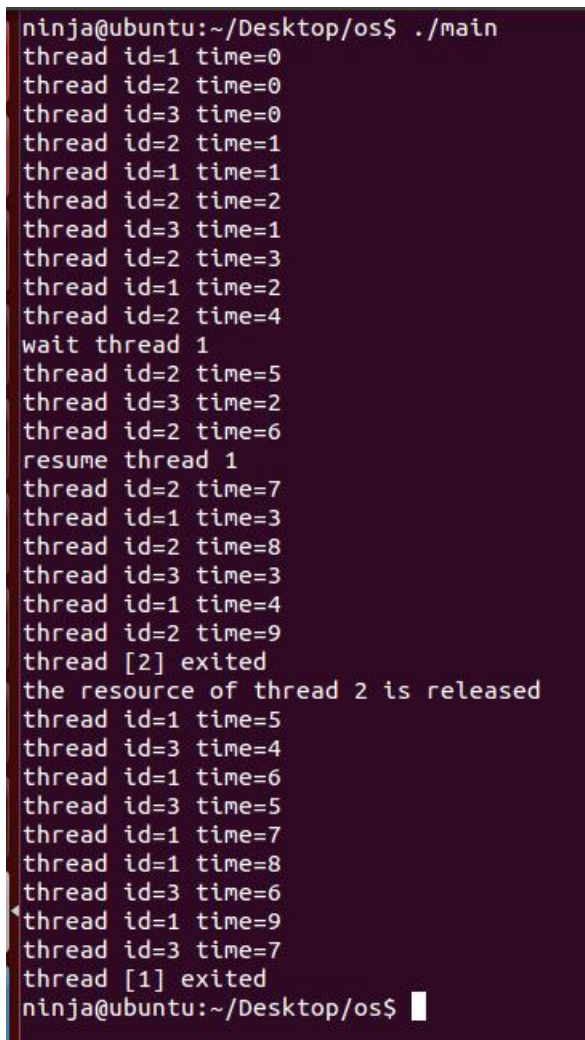
    schedule();

}

}

```

七、运行的主要界面和结果截图



```
ninja@ubuntu:~/Desktop/os$ ./main
thread id=1 time=0
thread id=2 time=0
thread id=3 time=0
thread id=2 time=1
thread id=1 time=1
thread id=2 time=2
thread id=3 time=1
thread id=2 time=3
thread id=1 time=2
thread id=2 time=4
wait thread 1
thread id=2 time=5
thread id=3 time=2
thread id=2 time=6
resume thread 1
thread id=2 time=7
thread id=1 time=3
thread id=2 time=8
thread id=3 time=3
thread id=1 time=4
thread id=2 time=9
thread [2] exited
the resource of thread 2 is released
thread id=1 time=5
thread id=3 time=4
thread id=1 time=6
thread id=3 time=5
thread id=1 time=7
thread id=1 time=8
thread id=3 time=6
thread id=1 time=9
thread id=3 time=7
thread [1] exited
ninja@ubuntu:~/Desktop/os$
```

如图所示，我们创建了三个线程，其中线程一在中途会被挂起，然后再执行。如上图所示，三个线程都能够正常的并发执行，并且在线程结束的时候，能够正确的回收已结束线程所有的资源，避免造成计算机资源的浪费。

八、总结和感想体会

这是我第 1 次手动开发用户态的线程库，也是第 1 次参与实现操作系统内核的代码开发。这次开发的过程中我遇到了种种的困难，也遇到了很多的挫折。其中最让我困惑的是函数底层的调用规则以及在汇编语言角度下的上下文切换。很

长一段时间，我无法彻底搞清楚函数的参数到底是如何传递的，以及函数是如何返回的。但是当我真正的解决了这些问题之后，我发现我的前面变得豁然开朗。我有一种非常好的开阔感，通过自己的努力，我感觉我走到了一个新的高度。

同时在本次开发中，我仔细的阅读了 Linux 操作系统的内核代码。另外一个系统以及稳定性开源而流行于这个世界，被广泛的用作服务器操作系统。当我读过他的源码之后，我不由得感叹，设计源码的人真的是个天才。许多非常巧妙的操作，看了之后让人佩服的五体投地。从代码里面我也能够看到 linus 本人，在设计 linux 操作系统的时候那种朴素却又严谨，充满智慧的设计思想。

最后我要感谢我的指导老师，田卫东老师。他在我最困难的时候给予了我非常大的帮助。通过电话他给给了我非常大的鼓励，也给我指明了前进的方向，没有他我很难顺利的完成这个用户及线程库的开发。同时也感谢田卫东老师平时的课堂，他的课堂总让我感叹操作系统的美妙，让我获益匪浅。

参考文献

- [1] 《linux 内核设计与实现 》 (美) Robert Love 著，陈莉君 康华 译
机械工业出版社
- [2] 《windows 核心编程 》 (美) Jeffrey Richter (法) Christophe Nasarre 著，
葛子昂 周靖 廖敏 译，清华大学出版社
- [3] 《汇编语言 第三版》 王爽 著 ， 清华大学出版社
- [4] 《计算机操作系统》（第四版） 汤小丹 梁红兵 哲凤屏 汤子瀛 著 ， 西安
电子科技大学出版社
- [5] Hansen Per Brinch. Operating System principles. prentice-Hall,1973
- [6] Pramod Chandra P.Bhatt An Introduction to Operating Systems Concepts and
Practice(Thrid Edition),PHI 2010