

---

# Linux 进程管理知识总结

2013.3 Version 1.01  
yalung929@gmail.com

## 1 基本概念

### 1.1 进程

内核使用struct task\_struct描述一个进程。在2.6.32这个内核版本上，这个结构体足足有6408字节！

与进程相比程序则简单的多；程序是一个文件，文件中主要保存了代码（机器指令）和数据（全局变量），以及一些必要的格式和管理信息。

每一个进程内核分配一个ID来唯一标识它，称作PID。

### 1.2 线程

在Linux上线程和进程在实现上没有太大区别。一个线程同样对应一个task\_struct，与它所对应的进程有相同的组织层次。一个进程有一个唯一的PID描述它，一个线程也有。线程和进程都是基本的调度单元，在调度这一层次，二者是完全等同的。所以后面很多内容，都对二者不加以区分。

每一个进程在proc建立一个以PID命名的目录，里面可以查看进程相关的信息。

```
# ls /proc/13645
attr  cmdline  environ  io  maps  mountstats  oom_score  sched  stack  syscall
auxv  coredump_filter  exe  latency  mem  net  pagemap  schedstat  stat  task
cgroup  cpuset  fd  limits  mountinfo  numa_maps  personality  sessionid  statm  wchan
clear_refs  cwd  fdinfo  loginuid  mounts  oom_adj  root  smaps  status
```

task目录内是该进程所有线程的目录节点。主线程PID就是进程的PID，主线程就是入口函数是main函数的线程。

```
# ls /proc/13645/task/
13645 13647 13648 52357 52358 52359 52367 52368 52369 52370 52371 52372
```

每一个线程目录和进程目录的布局基本一致。

```
# ls /proc/13645/task/13647
attr  clear_refs  cwd  fd  latency  maps  mounts  oom_score  root  sessionid  stat  syscall
auxv  cmdline  environ  fdinfo  limits  mem  numa_maps  pagemap  sched  smaps  statm  wchan  cgroup
```

线程和它所属的进程以及它所属进程的其他线程间共享内存，这就意味着共享相同的代

---

码段和数据段；但线程的栈是独立的，若非如此，它将无法运行。

## 1.3 系统调用

系统调用就是内核提供给应用程序的接口，应用程序如果需要内核帮助它做事情，就要而且必须调用相关接口，这点和我们为了解析 XML 文件，调用一个 XML 库的接口没有任何区别。

我们在一般的应用程序开发中，为了保证程序的可移植性，很少直接使用系统调用和内核打交道，而是使用 C 库或更高级的封装。

有人说 OS 为应用程序提供了一个**舞台**。说程序在 OS 上面运行。这些说法，十分干扰对一些基本概念的理解。

你要以 CPU 为主体来看。

CPU 只知道一条条的执行指令，这些指令可能是 OS 也可能是程序，运行中没有谁托着谁的概念，就是一条流串下来。执行 main 函数的时候，OS 的任何代码都没有运行。OS 没干活儿。后来你执行 printf，要打印屏幕。打印屏幕的代码由 OS 这个特殊的库预先提供，你是去调用它而已。

再说到虚拟化一样到道理，你以为虚拟机是跑在 OS 面的？虚拟机在 OS 上蹦跶？

非也。

CPU 要么在执行宿主机上的代码，要么在执行虚拟机上的代码。

OS、进程、虚拟机 os、虚拟机进程，大家都是 CPU 这个舞台上的演员，**轮流**蹦跶。

一个时间一个 CPU 上只有一个在蹦跶。**但是 CPU 有运行状态**，不同的演员能够蹦跶的范围和力度是不同的。

### 1.3.1 STRACE

strace，是一个强大的工具，它可以跟踪一个应用程序的系统调用，让我们看到应用程序和内核交互的细节，这就意味着：

1. 根据 strace 的输出，我们大体可以猜到一些程序的实现方式。这对我们学习一些内容会有很大的帮助。
2. 根据 strace 的输出，我们可以分析定位很多问题，因为 strace 不仅可以告诉我们程序执行过程中调用了哪些系统调用，而且还可以告诉我们调用参数和返回结果！

我们看一个简单的示例，我们使用 strace 跟踪 uname 这个命令：

```
# strace -ttT uname -a
11:47:48.081379execve("/bin/uname", ["uname", "-a"], [/* 66 vars */]) = 0 <0.000172>
.....//此处省略XXX行
```

---

```
11:47:48.084453uname({sys="Linux", node="linux34", ...}) = 0<0.000007>
11:47:48.084526 write(1,"Linux linux34 2.6.32.45-0.3-defa"..., 100Linux linux34
2.6.32.45-0.3-default #1 SMP2011-08-22 10:12:58 +0200 x86_64 x86_64 x86_64 GNU/Linux
) = 100 <0.000011>
```

输出比较多，我略去了动态库加载、环境检查等部分，重点看三个系统调用：

1. `execve`<sup>1</sup> 系统调用第一个被调用，起的作用就是加载程序执行文件到内存并执行它。
2. `uname` 系统调用就是 `uname` 命令实现的核心。它用法很简单，传入一个数据结构的地址，内核会把相关信息填充到这个数据结构内。我们看到 `strace` 命令不仅可以记录‘输入参数’和‘返回值’，就连被内核修改内容的‘输出参数’也会很好的呈现出来。这对定位分析问题的帮助太大了。

```
UNAME(2) Linux Programmer's Manual
NAME
uname - get name and information about current kernel
SYNOPSIS
#include <sys/utsname.h>
int uname(struct utsname *buf);
DESCRIPTION
uname() returns system information in the structure pointed to by buf. The
utsname struct is defined in <sys/utsname.h>:
```

3. `write` 系统调用把结果写到 1 号文件描述符（标准输出）指向的文件上，即终端。

综上，我们通过 `strace` 跟踪 `uname` 这个命令，知道了它的基本实现：

1. 它是一个应用程序（不是 `ulimit` 等内 SHELL 内置命令），有对应的执行文件，其路径是 `/bin/uname`。
2. 它通过一个系统调用函数 `uname` 从内核获取内核版本信息。
3. 它把结果输出到当前终端（1 号文件描述符，标准输出）。

### 1.3.2 系统调用过程详解

```
11:47:48.084453 uname({sys="Linux", node="linux34", ...}) = 0<0.000007>
```

如上，我们通过 `uname` 这个系统调用，来获取内核版本信息，那么当我们的程序执行到这段代码时发生了什么？其实发生的事情很简单，发生了指令“跳转”，最终跳转到下面这

---

<sup>1</sup> 我们可以通过 `man 2 execve` 查看它的详细说明文档。`strace` 输出的所有调用都可以通过 `man 2` 查到资料，除非是那种比较偏的，`man` 还没有同步更新。

---

个函数，就是这个函数返回了版本信息。这个“跳转”过程经历了用户态到内核态的切换。

```
asm_kagelong sys_uname(struct new_utsname __user * name)
{
    int err;
    down_read(&uts_sem);
    err = copy_to_user(name, &system_utsname, sizeof(*name));
    up_read(&uts_sem);
    if (personality(current->personality) == PER_LINUX32)
        err |= copy_to_user(&name->machine, "i686", 5);
    return err ? -EFAULT : 0;
}
```

切换完成之前，CPU 执行的是应用程序的代码（C 库是应用程序的一部分），切换完成之后 CPU 执行的就是内核的代码。内核要为我们服务，它的代码就必须被执行，而它的代码已经躺在内存里面了，只需要我们“跳转”过去执行它就可以了。

其本质和我们调用我们自己写的一个函数没有区别，一个函数提供一个服务，我们需要它，就调用它，我们怎么调用它？通过 `call` 指令，跳转到函数的入口地址，接下来函数的代码就会被执行。

不过，本质虽如此，实际却复杂些。为什么？因为我们自己写的函数或者 C 库的函数，所在的内存区域，应用程序是可以直接访问的。而内核函数代码所在的内存区域，我们是不可以直接访问的。这自然是防止内核被应用程序篡改，保证系统可靠。

所以这里必须要有一个有效的机制可以让应用程序执行内核代码，享受内核提供的服务，而又不破坏内核。这个机制就是 CPU 的运行级别。运行级别就是 CPU 的权限状态，高级别的权限，才可以做访问内核所在的内存区域。

拿 X86 CPU 来说，它有 0~3 四个级别，0 是最高级别，3 是最低级别，在 Linux 下应用程序运行时，CPU 的运行级别是 3，如果想要执行内核代码，必须切换到 0。切换通过一个 CPU 指令实现，就是著名的 `int 0x80`（现在的 CPU 都提供了一个更高级点指令，`syscall`）。`int 0x80` 这条指令一执行就会触发 CPU 的一个中断，进而执行内核在初始化阶段注册好的中断处理函数 `system_call`。`system_call` 就是系统调用的主处理函数，所有的系统调用都是经过它。它主要做几件事：

1. 保存应用程序执行 `int 0x80` 时的现场，主要是一些寄存器的值（`sp` 和 `ip` 等寄存器的值在执行 `int 0x80` 指令的时候由 CPU 自动保存），这样系统调用执行完，返回用户态时，CPU 可以继续执行应用程序的代码。

- 
2. 根据系统调用号<sup>2</sup>，调用指定的系统调用函数。
  3. 恢复保存的现场，调用 `iret` 指令返回到用户态。

回到 `uname` 这个系统调用，`uname` 本质是一个 C 库函数，是我们应用程序代码的一部分，C 库帮我们封装了执行 `syscall` 或 `int 0x80` 的过程，因此我们不需要知道某个系统调用对应的系统调用号。也就是说，`man 2` 看到的函数名，是在 C 库里面定义，而不是内核提供的，和内核的交互只能通过 `int 0x80` 等指令，而不能直接调用函数。

关于 C 库封装系统调用函数这点，可以通过反汇编 C 库文件确认：

```
#objdump -d /lib64/libc.so.6
0000000000a2840 <uname>:
a2840: b8 3f 00 00 00 mov $0x3f,%eax //设置系统调用号
a2845: 0f 05 syscall //进行系统调用
a2847: 48 3d 01 f0 ffff cmp $0xfffffffffff001,%rax
a284d: 73 01 jae a2850 <uname+0x10>
a284f: c3 retq
a2850: 48 8b 0d 59 57 2b00 mov 0x2b5759(%rip),%rcx # 357fb0 <_DYNAMIC+0x450>
a2857: 31 d2 xor %edx,%edx
a2859: 48 29 c2 sub %rax,%rdx
a285c: 64 89 11 mov %edx,%fs:(%rcx)
a285f: 48 83 c8 ff or $0xffffffffffff,%rax
a2863: eb ea jmp a284f <uname+0xf>
```

## 1.4 用户态和内核态

如上所述，用户态和内核态是指CPU的状态，本质就是CPU的运行级别。

一个进程占用CPU运行时，如果CPU是用户态，那么这个进程就处于用户态，如果CPU是内核态这个进程就处于内核态。

通常我们把用户态执行的代码称为用户态代码，内核态执行的代码称为内核代码。应用程序的代码是用户态的，而内核以及内核驱动的代码是内核态的。

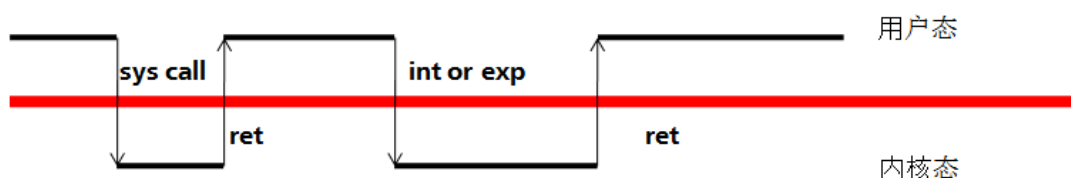
除了执行系统调用时，会发生用户态到内核态的切换。中断也会，这是一种被动切换。

---

<sup>2</sup>在调用 `int 0x80` 这个指令之前，应用程序首先要把系统调用号放在约定的寄存器（`eax`）内，这样内核根据系统调用号就知道，执行哪一个内核函数了。

---

## 用户态和内核态的转换轨迹



系统调用、中断或异常会导致进入内核态，执行内核代码。  
其中系统调用是主动的，中断或异常是被动的。

### 1.5 用户栈和内核栈

每一个进程有两个栈。执行用户态代码时使用用户栈，执行内核代码时使用内核栈。用户态切换到内核态时，栈也会切换。

使用gdb可以看用户栈：

```
(gdb) bt
#0  0x00007f8d80dc4190 in __read_nocancel () from /lib64/libc.so.6
#1  0x00007f8d80d61588 in _IO_new_file_underflow () from /lib64/libc.so.6
#2  0x00007f8d80d625de in _IO_default_uflow_internal () from /lib64/libc.so.6
#3  0x00007f8d80d5704a in _IO_getline_info_internal () from /lib64/libc.so.6
#4  0x00007f8d80d55e9b in fgets () from /lib64/libc.so.6
#5  0x000000000403d61 in ?? ()
#6  0x00007f8d80d0d3bd in __libc_start_main () from /lib64/libc.so.6
(gdb)
```

使用crash<sup>3</sup>可以看内核栈：

```
crash> bt 15140
PID: 15140 TASK: ffff880117a942c0 CPU: 1 COMMAND: "cscope"
#0 [ffff880114b8dcb8] schedule at ffffffff8153c90a
#1 [ffff880114b8dd40] pipe_wait at ffffffff81154523
#2 [ffff880114b8dd80] pipe_read at ffffffff81154c6c
#3 [ffff880114b8de10] do_sync_read at ffffffff8114b1f8
#4 [ffff880114b8df10] vfs_read at ffffffff8114ba7b
#5 [ffff880114b8df40] sys_read at ffffffff8114bb97
```

---

<sup>3</sup>参见《[CRASH工具介绍](#)》。

---

```
#6 [ffff880114b8df80] system_call_fastpath at ffffffff81546752
RIP: 00007f8d80dc4190  RSP: 00007fff02dbd730  RFLAGS: 00010206
RAX: 0000000000000000  RBX: ffffffff81546752  RCX: 00000000ffffffff
RDX: 0000000000000100  RSI: 00007f8d814d2000  RDI: 0000000000000000
RBP: 000000000000000a  R8: 0000000000000001  R9: 0000000000000000
R10: 0000000000000000  R11: 0000000000000246  R12: 00000000000000fb
R13: 00007f8d8108b200  R14: 000000000000000a  R15: 0000000000000000
ORIG_RAX: 0000000000000000  CS: 0033  SS: 002b
```

crash>

两个堆栈一起看，可以得知，这个进程调用C库函数fgets读取了一个文件，库函数调用系统调用read转给内核处理，并且可以看出这是一个管道文件。

思考一个问题：切入内核态之后，谁来切换栈？即谁来把内核栈的地址放入栈寄存器sp？这个栈的地址又保存在什么地方？

切入内核态之后，栈的切换必须由CPU自动完成，否则没有栈，代码根本没法执行。栈的地址保存在CPU的Task State Segment (TSS)内。内核每次切换至一个进程运行时，会把它的内核栈地址会保存到TSS内。这样切换到到内核态之后，CPU就可以拿到内核栈的地址，并且把用户栈的地址以及IP寄存器保存到内核栈上。等调用iret返回用户态时，可以再切换回来。

再有，进程执行系统调用切换进入内核态之后，如何快速找到这个进程的描述符？获取进程相关的一系列数据？遍历查找肯定不行。

内核把每个进程的task\_struct放在了每个进程的内核栈的上。具体来说，内核栈的下边界存放task\_struct。这样切到内核态之后，内核根据当前sp寄存器的地址，就可以直接计算出task\_struct的地址，因为栈的大小是确定的。

但是我们前面讲到，现在的内核task\_struct非常大，足有6000多字节，所以栈中已经不再直接放task\_struct了，而是放它的指针。它的指针被封装在struct thread\_info中。

thread\_info很小，只保存了一些必要信息。

```
26 struct thread_info {
27     struct task_struct *task;      /* main task structure */
28     struct exec_domain *exec_domain; /* execution domain */
29     __u32 flags;                  /* low level flags */
30     __u32 status;                 /* thread synchronous flags */
31     __u32 cpu;                    /* current CPU */
32     int preempt_count; /* 0 => preemptable,
"arch/x86/include/asm/thread_info.h" 273 lines --0%--
```

我们看到的内核代码`current`，就是一个根据`sp`快速获得当前运行进程的`task_struct`的宏。

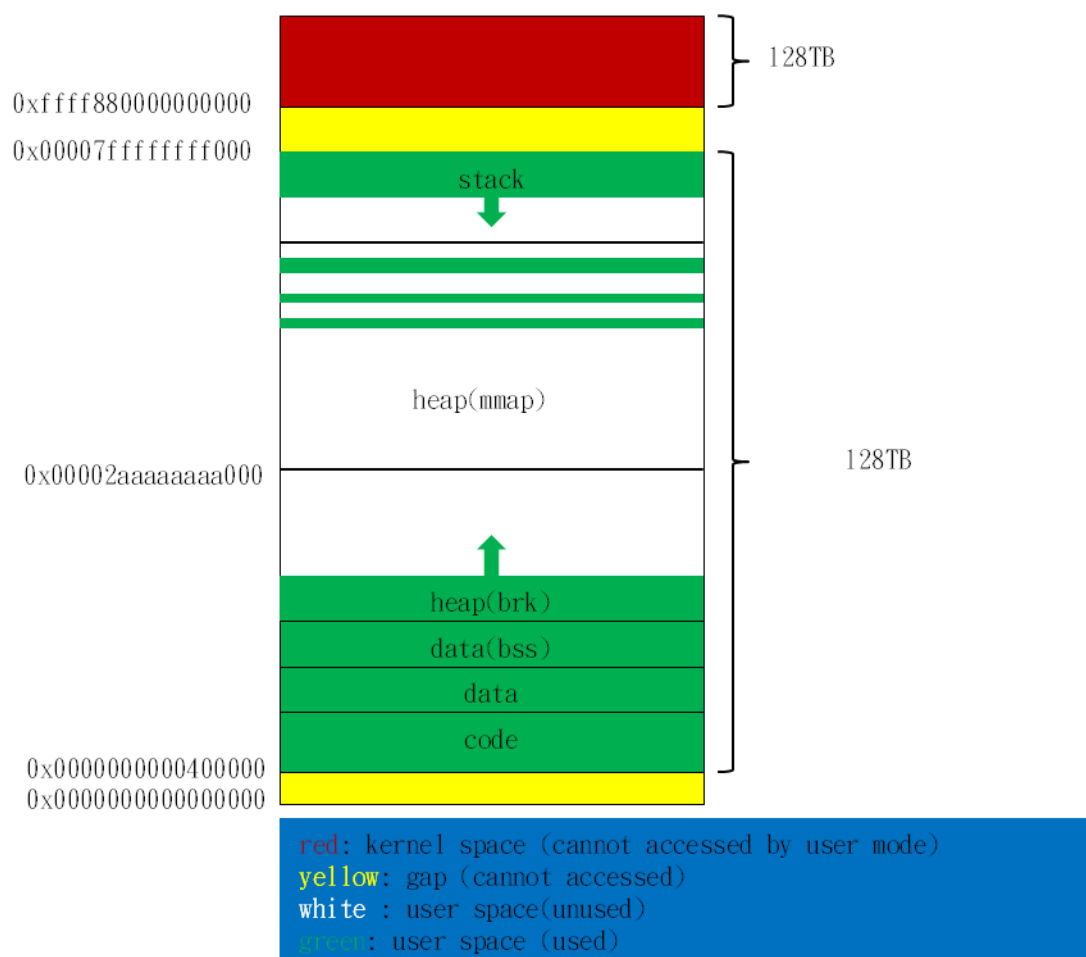
## 1.6 地址空间和地址映射

地址空间是进程运行时可以访问和操作的内存地址范围。这个地址是指**逻辑地址**。

CPU有一个MMU——内存管理单元，这个单元负责将逻辑地址转化为物理地址。

逻辑地址和物理地址的映射关系，由操作系统维护，计算却是由CPU的MMU自动完成的。

当我们的进程访问一个内存地址，这个地址是逻辑的，CPU经MMU计算<sup>4</sup>获得对应的物理地址，然后操作物理地址。



<sup>4</sup> 计算结果会被CPU的Translation Lookaside Buffers (TLB)缓存，并非每次都需要计算。



---

上图是Linux x86\_64系统下的进程地址空间分布<sup>5</sup>。内核态和用户态各有128TB的地址空间，分别称为**内核空间**和**用户空间**。实际上远远没用完64bit（ $2^{64}$ ）的地址范围。用户空间即我

们应用程序代码的运行空间，分为几个部分：

1. 代码段——存放代码，程序载入时一次分配好。
2. 数据段——存放初始化的全局变量（数据），在程序在载入时一次分配并初始化好。
3. 数据段（bss）——存放未初始化的全局变量，这些变量并不在程序文件中占用空间，

程序文件内只是记录了变量的长度，程序加载时，按照这个长度分配。bss节约的是程序文件的大小，而非内存。

4. 堆（brk）——这就是程序运行中动态申请的部分。brk是一个系统调用用于增长或收缩堆的边界。malloc会调用brk。

5. 堆（mmap）——这也是程序运行中动态申请的部分。mmap是一个系统调用，程序调

用它分配一个地址段，这个段不和brk的段连续，是独立的。在程序运行过程中这种不连续的段可能会有很多个。malloc在分配大内存或者多线程需要建立多个堆时，会调用mmap。

当然我们的程序也可以直接调用。另外动态库的代码段和数据段以及多线程情况下非主线程的栈空间，也是放在这个范围内。所以这个堆并非严格意义上的堆。其实在系统看来，堆、代码、数据等等段都是地址段，内核统一用struct vm\_area\_struct来描述。只是被程序用作不同用途罢了。使用pmap命令可以查看进程的已经分配的内存段：

（vm\_area\_struct）。

```
# pmap $$
17698: bash
START          SIZE      RSS      PSS   DIRTY    SWAP PERM MAPPING
0000000000400000    616K    520K    173K     0K      0K r-xp /bin/bash
0000000000699000      4K      4K      4K     0K      0K r--p /bin/bash
```

---

<sup>5</sup> 详见 "arch/x86/include/asm/processor.h" 和 "arch/x86/include/asm/page\_64\_types.h"

---

000000000069a000	16K	16K	16K	16K	0K rw-p /bin/bash
000000000069e000	2136K	516K	516K	272K	1500K rw-p [heap]
00007faf9f699000	44K	16K	2K	0K	0K r-xp /lib64/libnss_files-2.15.so
00007faf9f6a4000	2048K	0K	0K	0K	0K ---p /lib64/libnss_files-2.15.so
00007faf9f8a4000	4K	0K	0K	0K	4K r--p /lib64/libnss_files-2.15.so
00007faf9f8a5000	4K	0K	0K	0K	4K rw-p /lib64/libnss_files-2.15.so
00007faf9f8a6000	40K	24K	2K	0K	0K r-xp /lib64/libnss_nis-2.15.so
.....					

6. 栈——存放程序（主线程）的局部变量、函数参数和函数返回地址。Linux的栈是自顶向下增长的。

每个进程拥有相同的地址空间，但却可以共存互不干扰，这主要是因为不同进程的相同的逻辑地址，被映射到了不同的物理地址。MMU计算时有一个重要的因子，放在了CPU的一个寄存器里面，这个寄存器就是CR3。CR3里面放的是保存映射关系（表）的内存的地址。这个地址是物理地址。进程切换时，CR3里面的值被（内核调度器）改变，所以同一个逻辑地址经MMU计算出来的是不同的结果。这就是**地址映射**——说白了其核心就是一个非常简单的函数表达式：

$$z = f(x, y)$$

- 地址映射可以抽象为一个函数表达式（初中学过的）。

其中 $z$ 为物理地址， $x$ 为虚拟地址，**每个进程的 $x$ 取值范围相同**，内核为每个进程赋予不同的基地址 $y$ ，最终相同的 $x$ 得到不同的 $z$ 。 $f$ 就是CPU的内部逻辑（算法）， $y$ 由内核设置维护。

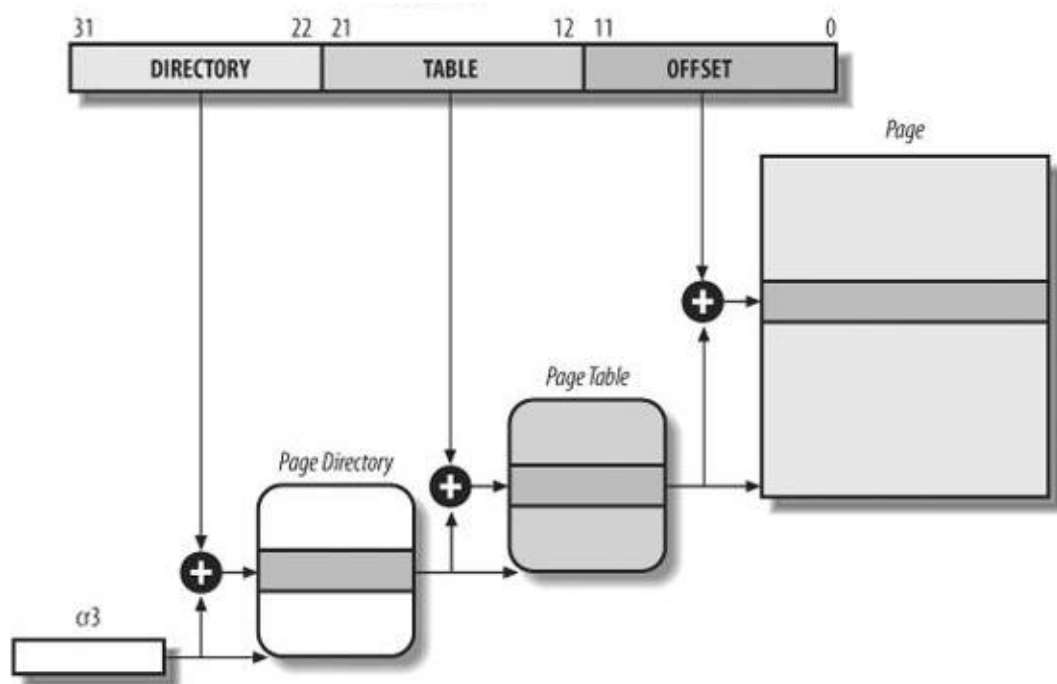
- 简单，但意义重大。

每个进程内存被彻底隔离开了。避免一颗老鼠屎坏了一锅汤。

详细来说，这个函数的计算过程。首先物理内存被划分为页，页是基本管理单位，一个页在x86上通常是4KB，如下图<sup>6</sup>描述的就是32bit CPU上的页映射计算过程：

---

<sup>6</sup> 拷贝自《Understanding the Linux Kernel, 3rd Edition》。



32bit的逻辑地址被分为三部分<sup>7</sup>， cr3内的物理地址（一级映射表——页目录首地址）加上逻辑地址的左<sup>8</sup>10位，得到一个物理地址，这个物理地址上放的是下一级映射表（页表）的首地址，再把中间10位与之相加得到页的起始地址，再把页的起始地址与最低12位相加，得到最终物理地址。

由于历史因素x86架构上除了有页管理外，还有段管理，所谓段管理其实比页管理更简单，内存被划分为若干个段，与页不同，段的大小不固定；逻辑地址加上这个段的首地址直接得到最终的物理地址。这个计算过程也是MMU完成的。实际上x86上一个逻辑地址，会被MMU先做段映射计算，再把结果作为页映射的输入，继续做页映射计算。段映射的结果通常称之为线性地址。而Linux默认划分了四个段，但是每个段的基地址都是0，所以经过段映射之后得到线性地址和逻辑地址相同，等同于越过了段管理，只使用了页管理。实际上只用页管理，也就够了，这都怪x86沉重的兼容包袱，不得不在代码上做一些特殊处理和额外操作。

<sup>7</sup> 具体如何划分，和CPU类型强相关。

<sup>8</sup> 高位在左。

---

## 1.7 O(1)

一个CPU同一时间只能执行一个任务，也就是进程。为了让系统中所有进程都可以获得

CPU执行，那么就需要调度。这个并不难理解，现实生活中例子很多。我们可以很容易想到——让每个进程执行一个固定的时间，时间到了就换别人，大家轮流坐庄，这个固定的时间就是时间片。如果这个时间片足够短，人们感知不到，每个进程跑一下就让给别人跑一下，最后看好像所有进程都在跑。这就是多任务分时操作系统。很简单的道理。

但有些任务长期运行，中断一小会儿，对它没有太大影响，比如编译源码这种批处理任务；有些任务只运行一小会儿，但要求立即响应，比如及键盘输入、鼠标移动这类交互式任务。如何在系统有比较高负载时，仍然可以快速响应用户的操作？这是进程调度的难点。这意味着内核要自动识别交互式进程并给予交互式进程更多的调度机会。总之时间片不能是固定值，进程不能平等对待。于是引入了优先级的概念。

Linux默认分为140个优先级，从0到139<sup>9</sup>。数值越低优先级越高。每一个进程都被赋予不同的优先级，而且根据进程行为改变优先级。最关键的就是识别出交互式进程，内核根据进程的休眠时间来推测。这种推测通常还比较准，交互式进程通常都在休眠等待用户输入，你APM再快，也快不过CPU吧。但具体的推测和计算公式却变得很玄乎，很难找到完美公式，这也是O(1)最被诟病的一点。

另外从运行队列中找到优先级最高的进程执行，这个看似简单，但却也是一个难点。遍历是不可取的，调度本身的开销要足够小。要快速的从运行队列中选择一个进程，这意味着必须采用巧妙的数据结构避免遍历和查找，让调度的开销和运行队列中的进程数量无关。这正是**传统**的著名的O(1)算法引以为傲的地方，也是其名字的由来。

O(1)为每一个优先级都分配了一个运行队列，而不是共享同一个。每次选择进程时，从最高优先级的队列上选择一个即可。从高到低依次遍历每一个优先级队列，取第一个非空的队列上的第一个进程即可。其最差时间取决于优先级队列的数量，而这个数量是一个常

---

<sup>9</sup> 并非这么简单，后面会对优先级有一个更详细的介绍。

---

数，和进程数无关。同时为了进一步加速确定第一个最高非空优先级队列，140个优先级队列放到一个数组中，数组的下标0~139恰好对应优先级的数值。0对应最高优先级。然后分配一个优先级位图。用140个bit代表140个优先级、140个数组下标，bit为0表示对应优先级队列空，为1表示非空。那么这个操作就转化为从140个bit中找到第一个非空的bit。这个操作会非常的快，并且有特殊的CPU指令支持。<sup>10</sup>

```
/**
 * __ffs - find first bit in word.
 * @word: The word to search
 *
 * Undefined if no bit exists, so code should check against 0 first.
 */
static __inline__ unsigned long __ffs(unsigned long word)
{
    __asm__ ("bsfq %1,%0\n\t: "=r" (word)
             : "rm" (word));
    return word;
}
```

这还不够，考虑这种情况——两个进程A和B，如果A的优先级高于B，那么A的时间片消耗完之后，A再次被放入所属的优先级队列，而由于A的优先级高于B，A再次被选择执行。

B永远无法得到执行！活活被饿死。这是不能接受的行为，即使平民也有吃饭的权利。于是

刚刚吃到CPU的人，要先等一等，暂时先不要进入队列，放到其他地方——这就是过期队列。

过期队列也有140个，按优先级存放，同样放在一个数组内，同样配置一个对应的位图。这样就有了两个优先级数组——活动数组和过期数组，分别存放活动队列和过期队列。时间片

消耗完的进程会被放入过期队列，等到所有活动队列中的所有进程都执行完毕，直接交换两个

优先级数组（指针），就瞬间完成了乾坤大挪移。十分巧妙简洁的方案。

---

<sup>10</sup> O(1)思想就是把对不确定数量的个体的遍历，变为对个体的有限种类的遍历，这一点可以被应用开发借鉴。同时遍历数组转化为遍历bit的技巧也可以参考。内核就是一部软件开发和设计宝典。

---

似乎是完美了，但还是有人不满意。交互式进程如果消耗完一个时间片，还需要执行，那么它需要等待活动队列中的所有进程都执行完一个时间片，才可以执行，这可能会造成比较明显的响应延迟。所以对于交互式进程，又为她开了一道后门——不放入过期队列。交互式进程消耗完一个时间片后，继续进入活动队列，优先享用CPU。但这潜藏着一种危险，万一这个原先被内核判断为交互式进程的进程改变了行为，变成了大量消耗CPU的进程，一直占着CPU不放，别人就没机会运行了。这并不比担心，因为消耗CPU时间多，意味着休眠少了，会受到惩罚，就不会被继续判断为受特殊照顾的交互式进程了。

以上对O(1)调度算法做了一个介绍，但注意到我前面对O(1)加了**传统**这一限定。这说明它是过去时了！Linux目前的调度算法叫做CFS。了解历史，了解一个事物的发展过程才能更深入的理解事物的当前状态。有了O(1)的豪华铺垫，对CFS的理解就自然和简单的多了。

## 1.8 CFS

CFS全称Completely **Fair** Scheduler。没错你注意到了**公平**！前面所说的O(1)重点就在于对特权的处理，在于如何体现**不公平**！正如其名，和原来的调度算法相比，CFS是一次设计理念的彻底革新，而这种革新出自同一个人之手，CFS和O(1)的作者都是Ingo Molnar。

CFS相比O(1)更加简洁。仅用几句话就可以描述它原理。没有优先级位图、没有优先级队列——在CFS中就没有优先级的概念！CFS使用一个红黑树保存运行队列，这是一个平衡的二叉树。每一个进程被分配一个**虚拟时钟（vruntime）**记录进程的运行时间。vruntime就是树的键值，树的最左节点就是运行时间最少的进程。当需要从运行队列挑选一个进程运行时，就直接取这个节点。这就意味着CFS的核心思想是**从运行队列中挑选占用CPU运行时间最少的进程执行**。而不是什么优先级最高的。——这就是CFS公平二字的

---

含义！

这种设计理念，天然的对交互式进程的处理，因为交互进程休眠多，占用CPU少，vruntime小，这意味着一旦它需要执行，进入运行队列，就会处于树的最左侧，会被优先选中投入运行。

而根据实际测试的结果来看，CFS要比传统O(1)推测进程行为、动态计算优先级、按优先级选择任务的方式，在桌面等交互式环境上性能表现得更好<sup>11</sup>。——看似傻瓜、懒惰的处理方式，其结果反而是更好的。也许任何复杂，总可以变得简单，只是我们还没想到或者根本就想去想甚至不愿意简单而已。

CFS是当前Linux标准内核采用的调度算法，所以我们继续看一下它的细节。

### 1.8.1 调度周期

在继续深入细节之前，必须交代一个重要的概念。优先级、时间片或vruntime等调度数据的计算和更新在时钟中断内执行，时钟中断以固定频率发生，这个频率就是我们看到的HZ。

```
# cat /proc/sys/kernel/HZ
250
```

250HZ，4ms一次中断，这就意味着调度检查4ms做一次。这就是调度周期——**schedule tick**。而时间片是进程被允诺的一次执行的时间，这个时间可能是多个调度周期，两个不是一个概念。

*HZ不仅影响调度，很多事情都是在时钟中断内做的，比如定时器。在时钟中断内检查定时器是否超时到期，这意味着250HZ时，定时器的精度是4ms，一个sleep 1us的调用，实际上要在4ms以上才能返回。如果我要支持更高精度的定时器就要增加HZ，但会给系统带*

---

<sup>11</sup> 现在Android上用的调度算法是BFS（Brain Fuck Scheduler）并非CFS，BFS交互性能更强、更简单。我认为BFS和CFS在核心理念上是差不多的，都是基于进程时间的调度。CFS是通用的调度算法，要照顾服务器等复杂的环境，引入了很多枝叶。而BFS专注于桌面环境，能省的都省了，精简高效。兼容性是性能杀手。

---

来

的更多的负担。增加到1000HZ，时钟精度提高1ms，还说的过去，要是需要提高到1us精度就要1000000HZ，这样的系统估计也没法干活儿了。这是一个两难的问题。

但这个两难的问题却被完美的解决了。硬件时钟能够以不固定的频率发生中断，或者说，硬件时钟能够在指定时间（理论上可以到ns级）后发生中断，这种工作模式被称为 *oneshot*。

至少在我分析的suse11（2.6.32）上，已经使用了oneshot工作模式<sup>12</sup>。/proc/sys/kernel/HZ消失

了。定时器可以精确到us级。你设置一个8us的定制器，那么8us后会收到一个中断。

没有了固定频率的滴答。那么schedule tick如何体现？很简单，直接设置一个4ms的定时器，触发后执行调度动作，然后再激活这个定时器。绝妙的方案。不过这主要是依赖强大的硬件特性。相关的代码定义在"kernel/time/tick-sched.c"内。

```
719 /**
720  * tick_setup_sched_timer - setup the tick emulation timer
721  */
722 void tick_setup_sched_timer(void)
723 {
724     struct tick_sched *ts = &__get_cpu_var(tick_cpu_sched);
725     ktime_t now = ktime_get();
726     u64 offset;
727
728     /*
729      * Emulate tick processing via per-CPU hrtimers:
730      */
731     hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
732     ts->sched_timer.function = tick_sched_timer;
733 }
734
735 "kernel/time/tick-sched.c" 819 lines --84%--
```

## 1.8.2 对休眠进程的和谐和奖励

如果一个休眠很久的进程醒来，其vruntime远小于树中的进程，如果它突然发飙，长时间运行，其他进程就没机会运行了。解决这个问题很简单，CFS维护了一个minruntime记录了树中最小的vruntime。休眠的进程再次进入树中，它的vruntime会被直接更新为

---

<sup>12</sup> 默认只要硬件支持，就会启用，主流的x86服务器都已经支持。



---

minruntime，这就是和谐。虽然你很好，常年让着大家，但和大家相差太多也不好。

同时为了照顾它的情绪，还会给它一点奖励，即在minruntime的基础上再减去一些。这

样它就是最左边了，但又不会和大家伙儿相差太远，不用干等着它连续跑半天追上来。

但是如果是某些进程挺坏的，一直运行，偶尔休眠一小点，故作谦让，立马又要运行，

如果按照上面对休眠进程的处理，对别人就会不公平。这种情况下，它之前长期运行累积的

vruntime，会比经minruntime计算来的值要大，那么直接保持它的vruntime不变就有效的避免

了这种情况了。

*相关处理的代码位置：*

```
726 static void
727 place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
728 {
729     u64 vruntime = cfs_rq->min_vruntime;
730
731     /*
732      * The 'current' period is already promised to the current tasks,
733      * however the extra weight of the new task will slow them down a
734      * little, place the new task so that it fits in the slot that
735      *
736      * "kernel/sched_fair.c" 2191 lines --31%--
```

### 1.8.3 优先级

CFS并不靠优先级来作为算法的基础，但它仍然有优先级的概念。依然是0~139个优先级。依然是值越低优先级越高。

100是分界线，低于100是实时优先级，高于或等于100是普通优先级。普通进程的默认优先级是120。除了优先级，还有nice值，nice值的范围是从-20~19对应优先级100~139，nice值越高说明你越好，懂得谦让，优先级越低。我们可以通过nice命令或系统调用调整进程的

---

优先级<sup>13</sup>。

优先级在CFS算法内如何体现？就体现在vruntime的这个v上。因为每个进程都有一个虚拟时钟，虽然硬件时钟是以固定的频率前进的，但虚拟时钟的频率可以改变，每一个进程都可以不同。这提供了很大的灵活性。让优先级高的进程的虚拟时钟走的慢一点，优先级低的虚拟时钟走的快一点，就自然体现出了优先级。

每一个优先级数值，被转化为一个权重值。

```
1386 static const int prio_to_weight[40] = {
1387 /* -20 */      88761,      71755,      56483,      46273,      36291,
1388 /* -15 */      29154,      23254,      18705,      14949,      11916,
1389 /* -10 */      9548,       7620,       6100,       4904,       3906,
1390 /*  -5 */      3121,       2501,       1991,       1586,       1277,
1391 /*   0 */      1024,        820,        655,        526,        423,
1392 /*   5 */       335,        272,        215,        172,        137,
1393 /*  10 */       110,         87,         70,         56,         45,
1394 /*  15 */        36,         29,         23,         18,         15,
"kernel/sched.c" 11396 lines --11%--
```

依靠硬件时钟以固定的频率更新进程的vruntime，但更新的值依据优先级计算的权重值

计算。优先级越高，vruntime走的越慢，越靠近树的左侧，越容易被调度到。

```
499 * Update the current task's runtime statistics. Skip current tasks that
500 * are not in our scheduling class.
501 */
502 static inline void
503 __update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
504              unsigned long delta_exec)
505 {
506     unsigned long delta_exec_weighted;
507
508     schedstat_set(curr->statistics.exec_max,
509                  max((u64)delta_exec, curr->statistics.exec_max));
510
511     curr->sum_exec_runtime += delta_exec;
512     schedstat_add(cfs_rq, exec_clock, delta_exec);
```

---

<sup>13</sup> 如果一个进程的nice值大于0，那么它所执行的用户态代码所占用的CPU就会被计入%ni字段。

---

```

513         delta_exec_weighted = calc_delta_fair(delta_exec, curr);
514
515         curr->vruntime += delta_exec_weighted;
516         update_min_vruntime(cfs_rq);
517     }
"kernel/sched_fair.c" 2191 lines --21%--

```

#### 1.8.4 时间片

CFS弱化了时间片的概念，没有固定的时间片。CFS树中的各个进程按照优先级对应的权重瓜分全部的CPU时间。具体做法如下：

CFS设置一个`ideal_runtime`，对应进程分到的时间片。

`ideal_runtime`动态计算，根据权重瓜分一个总的时间片。具体来说，由内核参数`sysctl_sched_latency`和`sysctl_sched_min_granularity`以及运行队列中的进程数量共同决定——进程数量比较少时，直接瓜分`sysctl_sched_latency`。如果超过一定数量（`sysctl_sched_latency`和`sysctl_sched_min_granularity`的比值）后，把`sysctl_sched_min_granularity`乘以进程数量，再按权重瓜分。

举例来说，默认情况下进程优先级相等，权重相同，如下80ms的总额被所有进程平分，  
如果是2个进程，那么每个进程的分到的时间片是40ms。  
而如果超过5个进程就会是16ms。

```

# sysctl -a |grep sched_la
kernel.sched_latency_ns = 80000000
# sysctl -a |grep gran
kernel.sched_min_granularity_ns = 16000000

```

另外`sysctl_sched_min_granularity`正如它的名字所指示的那样，还起到了一个确保进程最小执行时间的作用，如果进程一次执行的时间小于`sysctl_sched_min_granularity`，即使它的`vruntime`超过了树中最左侧的进程的`vruntime`一个时间片（`curr.vruntime - leftmost.vruntime > idealtime`），也不会切换。

```

881         if (delta_exec < sysctl_sched_min_granularity) //本次执行时间小于min_gran
882             return;
883

```

---

```
884         if (cfs_rq->nr_running > 1) {
885             struct sched_entity *se = __pick_next_entity(cfs_rq);
886             s64 delta = curr->vruntime - se->vruntime;
887
888             if (delta > ideal_runtime)
889                 resched_task(rq_of(cfs_rq)->curr);
890         }
891 }
892
"kernel/sched_fair.c" 2191 lines --39%--
```

调整sysctl\_sched\_min\_granularity和sysctl\_sched\_min\_granularity可以适应不同的场景，对

于桌面系统追求快速响应，就需要适当调低这两个值。对于服务器环境需要更好的处理性能，则需要适当提高这两个值。例如SUSE11的服务器版本默认值是80ms和16ms，而OpenSUSE 12的桌面版本的默认值是18ms和2.25ms。

### 1.8.5 组调度

---

组调度是这样一个特性，它可以让多进程成为一个组和其他组或组外进程平分CPU。如

果有两个优先级相同的进程A和B处于运行队列，大家权重相同，交替运行，各分得50%的CPU。如果又来了第三个进程C，那么就各分得33%的CPU。如果A和B放到一个组内，那么

A和B这一组整体再去和C平分CPU，最后C得50%，A得到25%，B得到25%。如果A和B的组

内又多了一个D，那么C仍然是50%，A、B、D平分剩下50%。无论组内多么繁忙，组内进程多么多，组外的进程不受影响！

这是一个很有用的特性，Mike Galbrai利用这个特性，提交了一个patch，仅用了200多行代码，让编译内核的同时还可以播放1080p的视频。桌面延迟下降几十倍。被linus称为killer feature。这个patch原理很简单，就是自动为每一个tty主进程创建一个组，而桌面（X window）单独使用一个tty。如果执行make -j 8开启的大量狂占CPU的进程，由于和桌面程序不属于同一个组中，不会冲击桌面程序的运行。

---

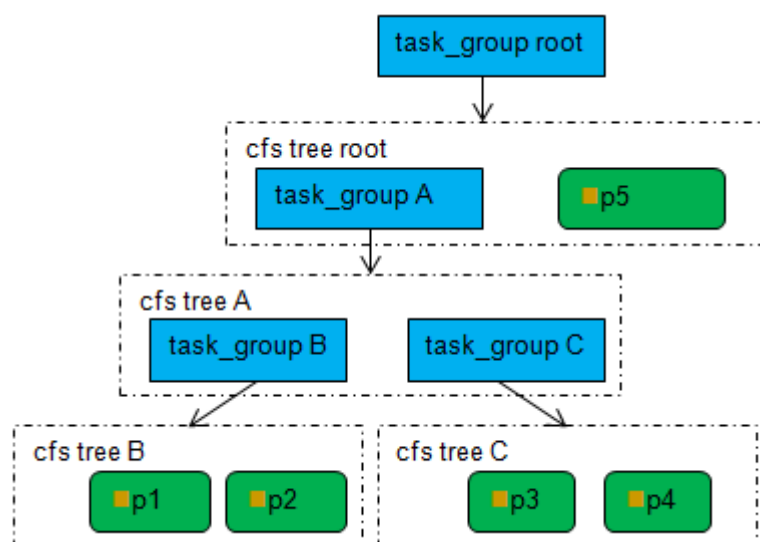
组调度作为cgroup<sup>14</sup>的子系统之一暴露给用户，你可以通过挂载cgroup文件系统很方便的划分组，做基于CPU占用率的分配和隔离。而不是像cpuset那样只能基于CPU个数。

在实现上，每一个组，即task\_group，也有自己的vruntime，一个task\_group的成员可以是进程也可以是其他task\_group。组成一个树状的结构，叶子就是进程。默认所有进程都属于根组。

组内任何一个成员的vruntime增加，都会增加该组的vruntime，组的vruntime通过和其他组或进程的vruntime比较进行调度选择。每一个task\_group都有自己的cfs树，树中的成员都按照前面所述的方式进行调度。

例如如下分组情况：

1. p1 ~ 5 5个进程处于RUNNING状态。
2. p5 和 task\_group A 位于cfs tree root中，属于组root。
3. B 和 C 位于cfs tree A 中，属于组A。
4. p1 和 p2 位于cfs tree B中，属于组B。
4. p3 和 p4 位于cfs tree C中，属于组C。



假设当前的vruntime都是0ms。当前即将运行的进程是p1，sysctl\_sched\_latency是80ms，

---

<sup>14</sup> Cgroup是Linux的一种资源控制技术，可以通过配置把系统的CPU、内存、IO、网络等资源分割给不同的进程，达到资源隔离保护的目的，可以理解成为一种超轻量级的虚拟化技术。

---

不考虑sysctl\_sched\_min\_granularity的影响，一个schedule tick是4ms：

在计算ideal\_runtime时，默认情况所有进程的优先级相同，权重相同。所以A和p5平分80ms，A得到40ms，B和C再平分这40ms得到20ms，以此类推，p1, p2, p3, p4分别得到10ms。

每一个tick，进程以及进程所属组，以及组的父组……一直推演到最顶部，它们的vruntime都会增加4ms<sup>15</sup>。

```
117 /* Walk up scheduling entities hierarchy */
118 #define for_each_sched_entity(se) \
119         for (; se; se = se->parent)
"kernel/sched_fair.c" 2191 lines --3%--
1997 /*
1998  * scheduler tick hitting a task of our scheduling class:
1999  */
2000 static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
2001 {
2002     struct cfs_rq *cfs_rq; //运行队列， cfs树
2003     struct sched_entity *se = &curr->se;
2004
2005     for_each_sched_entity(se) { //逐层遍历
2006         cfs_rq = cfs_rq_of(se);
2007         entity_tick(cfs_rq, se, queued); //每一层都进行cfs树的推进和调度处理
2008     }
2009 }
2010
"kernel/sched_fair.c" 2191 lines --90%--
```

这样3个tick之后，p1的vruntime变为12，同时B也变为12，A也变为12。在p1和p2所在的这一层的cfs tree B中，p2已经落后了p1 12ms，超过了p1分到的时间片。p2被选择运行。

又3个tick后，p2的vruntime变为12，B变为24，A变为24。同理cfs tree A中，C落后了B 24ms，超过了B分到的时间片，C被选择运行，而C是一个task\_group，选择它下属的cfs tree C中的p3<sup>16</sup>。

又3个tick后，p3变为12，C变为12，A变为36，其他不变，基于同样的规则，p3切换

---

<sup>15</sup> 由于中断可能被短暂的关闭而导致延迟，并非每一个tick都是准确的4ms。所以内核并非是直接增加4ms，

而是每次tick发生时，读取另外的硬件时钟源（例如tsc）的值，两次的差值，作为这次执行的实际时间。

<sup>16</sup> p3和p4相等，假设选择了p3。

为

p4。

又2个tick之后，p4变为8，C变为20，A变为44，其他不变，cfs tree root中的p5落后于A 44ms超过了A的时间片，p5被选中运行。

.....

所有的层次的树，都保持一个你追我赶的状态，维护一个平衡。最后宏观上看到的结果就是，p1 p2 p3 p4个占12~13%的CPU，而p5占50%的CPU。

未分组之前CPU的占用情况：

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
29212	root	20	0	4040	348	272	R	20	0.0	0:48.86	p3
29204	root	20	0	4040	348	272	R	20	0.0	1:08.04	p1
29208	root	20	0	4040	344	272	R	20	0.0	0:58.26	p2
29213	root	20	0	4040	344	272	R	20	0.0	0:42.93	p4
29214	root	20	0	4040	348	272	R	20	0.0	0:40.53	p5

按照如上方式分组之后的情况：

```
linux-en3s:/sys/fs/cgroup/cpu/A # echo 29204 > B/tasks
linux-en3s:/sys/fs/cgroup/cpu/A # echo 29208 > B/tasks
linux-en3s:/sys/fs/cgroup/cpu/A # echo 29212 > C/tasks
linux-en3s:/sys/fs/cgroup/cpu/A # echo 29213 > C/tasks
linux-en3s:/sys/fs/cgroup/cpu/A # cat tasks
linux-en3s:/sys/fs/cgroup/cpu/A # cat B/tasks
29204
29208
linux-en3s:/sys/fs/cgroup/cpu/A # cat C/tasks
29212
29213
linux-en3s:/sys/fs/cgroup/cpu/A # cat ../tasks |grep 29214
29214
linux-en3s:/sys/fs/cgroup/cpu/A #
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
29214	root	20	0	4040	348	272	R	50	0.0	2:37.04	p5
29204	root	20	0	4040	348	272	R	13	0.0	1:51.71	p1
29208	root	20	0	4040	344	272	R	13	0.0	1:41.92	p2
29212	root	20	0	4040	348	272	R	13	0.0	1:33.62	p3
29213	root	20	0	4040	344	272	R	12	0.0	1:28.07	p4

分组使得p5的CPU得到保证，只和组内成员竞争，无论其他组内的竞争多么激烈，只要p5所在组的情况不变，p5的得到的执行时间就不变。

### 1.8.6 多 CPU 的情况

我们前面有意的回避了多CPU的情况，无论O（1）还是CFS，每个CPU上都有一个运行

队列，调度的所有工作都是针对本CPU的，其他CPU上的进程和我不相干。对于CFS来

---

说，

每一个CPU都对应一个CFS树，独立进行**CFS树的推进**。

支持组调度以后，情况变得复杂了，一个组中的进程可能属于不同的CPU（我上面的例

子做了手脚，用taskset把5个进程都绑定到了一个CPU上）。实际上一个task\_group中会为每

一个CPU都建立一个运行队列，即cfs树；并且为每一个CPU建立一个调度实体

**sched\_entity**,

每一个实体都有一个**vruntime**。这就是影分身。每一个CPU上都有一个组的分身。

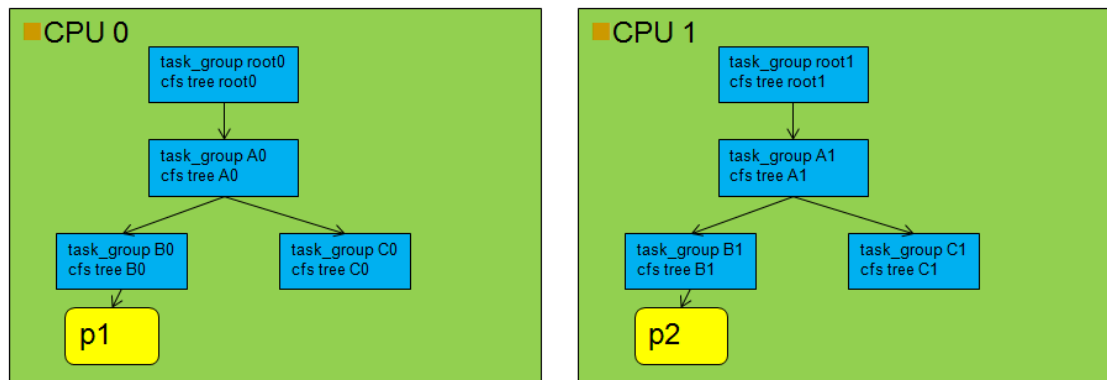
```
10108 int alloc_fair_sched_group(struct task_group *tg, struct task_group *parent)
10109 {
10110     .....          //省略
10124     for_each_possible_cpu(i) { //遍历所有cpu
10125         rq = cpu_rq(i);
10126
10127         cfs_rq = kzalloc_node(sizeof(struct cfs_rq), //分配一个cfs tree
10128                                GFP_KERNEL, cpu_to_node(i));
10129         if (!cfs_rq)
10130             goto err;
10131
10132         se = kzalloc_node(sizeof(struct sched_entity), //分配一个调度实体
10133                            GFP_KERNEL, cpu_to_node(i));
10134         if (!se)
10135             goto err;
10136
10137         init_tg_cfs_entry(tg, cfs_rq, se, i, 0, parent->se[i]);
10138     }
"kernel/sched.c" 11396 lines --88%--
```

继续前面的例子，如果p1~5 5个进程分属于不同的CPU那么大家分到的时间片都是80ms。

因为其所属cfs tree以及逐层往上的cfs tree中只有他们自己。

不要被代码展示的逻辑所迷惑，抛开组，以CPU的维度来理解，就很容易搞清楚了，实际上等价于这种情况：





这种情况下，p1和p2虽然都在task\_group B（地球上），但实际上是平行宇宙，p1和p2各自在各自的task\_group B，以及cfs tree B上，根本见不到面，没有任何竞争关系。在没其它进程的情况下，都得到100%的CPU。设计实现组调度这个特性的真是天才。

### 1.8.7 sched\_yield 的故事

sched\_yield是一个系统调用，调用进程主动放弃CPU，与sleep放弃的方式不同，sleep会让进程脱离运行队列，进入休眠状态。sched\_yield只是让进程让出CPU进入运行队列尾部

让别人执行。由于CFS没有什么尾部的概念，所以CFS下sched\_yield和传统的O(1)在行为上发生了很大的变化，具体参考[《SUSE11新调度算法CFS下的sched\\_yield问题》](#)。这个问题对理解CFS有很大的帮助。

## 1.9 schedule 函数

```
crash> bt 1784
PID: 1784   TASK: ffff880117b62180   CPU: 2   COMMAND: "gmain"
#0 [ffff8801138cd9c8] schedule at ffffffff8153c90a
#1 [ffff8801138cda50] schedule_hrtimeout_range_clock at ffffffff8153dd0d
#2 [ffff8801138cdae0] poll_schedule_timeout at ffffffff8115da54
#3 [ffff8801138cdb00] do_poll.isra.4 at ffffffff8115df92
#4 [ffff8801138cdb90] do_sys_poll at ffffffff8115ee76
#5 [ffff8801138cdf50] sys_poll at ffffffff8115efbc
#6 [ffff8801138cdf80] system_call_fastpath at ffffffff81546752
```

通过crash我们可以看到，每一个当前不占用CPU执行的进程的堆栈顶层函数一定是schedule。要完整理解Linux的进程调度机制，一定要理解这个函数的作用。

如前所述，每一个scheduler tick都会执行scheduler\_tick检查当前执行进程的时间片是否

---

超了，是否应该切换。但切换动作并不发生在scheduler\_tick函数内，而是通过schedule完成的。scheduler\_tick仅仅是做了一个标记。

```
2447 static inline void set_tsk_need_resched(struct task_struct *tsk)
2448 {
2449     set_tsk_thread_flag(tsk, TIF_NEED_RESCHED);
2450 }
"include/linux/sched.h" 2731 lines --88%--
```

这个标记在中断或系统调用返回之前检查，如果设置了就调用schedule执行切换。

```
902 retint_careful:
903     CFI_RESTORE_STATE
904     bt    $TIF_NEED_RESCHED,%edx //检查标记
905     jnc   retint_signal
906     TRACE_IRQS_ON
907     ENABLE_INTERRUPTS(CLBR_NONE)
908     pushq %rdi
909     CFI_ADJUST_CFA_OFFSET 8
910     call  schedule //执行切换
911     popq  %rdi
912     CFI_ADJUST_CFA_OFFSET -8
913     GET_THREAD_INFO(%rcx)
914     DISABLE_INTERRUPTS(CLBR_NONE)
915     TRACE_IRQS_OFF
916     jmp  retint_check
"arch/x86/kernel/entry_64.S" 1659 lines --52%--
```

考虑一个实际的例子，CPU当前运行着进程P，很happy，这时候发生了时钟中断，保存上下文进入内核态，执行中断处理函数，最终会执行scheduler\_tick，scheduler\_tick一看，丫的，时间到了，在P上画了一个叉叉。中断处理结束时，看到了这个叉叉，执行schedule函数而不是返回用户态继续执行应用代码。schedule完成进程切换。通过crash会看到下面所示的堆栈。

```
crash> bt 13631
PID: 13631 TASK: ffff880037922580 CPU: 1 COMMAND: "a.out"
#0 [ffff880021e39ef0] schedule at ffffffff8153c90a
#1 [ffff880021e39f78] retint_careful at ffffffff8153ee96
```

等到再次轮到P执行的时候，P从schedule返回继续执行中断处理尾部的其他工作，最终返回应用代码继续执行。

schedule通过调用调度算法注册的put\_prev\_task把当前占用CPU的prev进程放回运行队

列，在调用pick\_next\_task函数选择next进程。然后保存当前上下文，恢复next进程的上下文，执行next进程。因为next进程以前也是从schedule切走的<sup>17</sup>，它恢复的上下文就在schedule里面。所以一个进程如果不在CPU上跑，那么它一定停留在schedule里面，一个进程如果开始运行，那么一定是从schedule开始。

具体的上下文切换动作，由switch\_to宏实现，不同体系方法不同。如下是x86\_64的：

```

125 /* Save restore flags to clear handle leaking NT */
126 #define switch_to(prev, next, last) \
127     asm volatile(SAVE_CONTEXT \
128         "movq %%rsp,%P[threadrsp](%[prev])\n\t" /* save RSP */ \
129         "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */ \
130         "call __switch_to\n\t" \
131         ".globl thread_return\n" \
132         "thread_return:\n\t" \
133         "movq \"__percpu_arg([current_task])\",%%rsi\n\t" \
134         __switch_canary \
135         "movq %P[thread_info](%rsi),%r8\n\t" \
136         "movq %%rax,%rdi\n\t" \
137         "testl  %[_tif_fork],%P[ti_flags](%r8)\n\t" \
138         "jnz   ret_from_fork\n\t" \
139         RESTORE_CONTEXT \
140         : "=a" (last) \
141         __switch_canary_oparam \
142         : [next] "S" (next), [prev] "D" (prev), \
143         [threadrsp] "i" (offsetof(struct task_struct, thread.sp)), \
144         [ti_flags] "i" (offsetof(struct thread_info, flags)), \
145         [_tif_fork] "i" (_TIF_FORK), \
146         [thread_info] "i" (offsetof(struct task_struct, stack)), \
147         [current_task] "m" (per_cpu_var(current_task)) \
148         __switch_canary_iparam \
149         : "memory", "cc" __EXTRA_CLOBBER)
150 #endif
"arch/x86/include/asm/system.h" 463 lines --24%--

```

最关键的就两句，保存当前进程的栈地址恢复下一个进程的栈地址：

```

128     "movq %%rsp,%P[threadrsp](%[prev])\n\t" /* save RSP */ \
129     "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */

```

一旦这两句执行完毕，next的堆栈就恢复了，下面的代码就处于next的上下文中，然后再恢复其他寄存器就可以继续运行了。但唯独没有ip寄存器，ip寄存器并不存在保存恢复的问

<sup>17</sup> 新执行进程的情况，后面会有交代。

---

题，

代码是一条流下来的，只是外部环境（上下文）改变了，这就好比一个人吃烧饼，吃着吃着，

穿越了，还是继续吃烧饼，不过周边环境变了，原来吃完烧饼要去喝汤的，现在吃完烧饼可

能要去吃药了。

总之一句话，`schedule`内发生上下文的切换。而由于栈的切换，`schedule`返回之后执行的

函数就变了，宏观上看进程又继续从原来的函数位置继续运行了。

如果是一个进程主动调用了`schedule`函数是什么情况？这种情况就是进程主动放弃CPU休眠，`sleep poll select read write`等等阻塞进程的调用的内部实现最终都是调用`schedule`函数放弃CPU。比如本小节开头就是一个执行了`poll`系统调用的调用栈。

只要进程不在运行队列（不是R状态），那它就是主动调用了`schedule`函数。并且必然在等待某一个系统调用返回。

## 1.10 实时调度算法

CFS算法只处理优先级是100到139的普通进程。实时进程由单独的实时调度算法处理。

实时调度算法定义在`kernel/sched_rt.c`内。我们的进程可以通过调用`sched_setscheduler(2)`修改

调度策略，拥有实时优先级。例如CPU负载均衡算法中负责迁移进程到的`migration`内核线程就是一个实时<sup>18</sup>任务。其优先级是0，被`top`命令显示为RT。`top`命令显示的优先级范围是-100（RT）~39，对应内核的0~139。

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3	root	RT	0	0	0	0	S	0	0.0	0:00.01	migration/0

实时优先级的进程，不放在CFS的树中，有单独队列维护，总是优于普通进程被调度。

一旦被唤醒就会直接抢占当前运行的普通进程。采用简单的策略，比如FIFO——先来的先

---

<sup>18</sup> 只是相对普通任务而言，真正要做到实时，要靠专门的实时操作系统。

---

执行，没有时间片的概念，一旦投入运行除非它自己主动放弃CPU，无法被其他进程抢占。

如果我们的系统中有一些特殊进程需要保证优先执行和响应，可以设置实时调度模式。具体

参考man 2 sched\_setscheduler。

## 1.11 调度算法模块化

CFS调度算法的代码定义在kernel/sched\_fair.c内，通过代码的组织，我们也看到内核对调度算法模块化了，对调度算法做了抽象和封装，提供了统一接口，不同算法接口实现不同。

和IO调度算法<sup>19</sup>一样。封装调度算法的对象是sched\_class。

## 1.12 负载均衡

在多CPU的环境中，每个CPU上的负载可能会出现明显的不均衡，有的闲，有的忙，这就需要实现负载均衡。负载均衡程序（函数）定期<sup>20</sup>触发，检查各个CPU之间的负载情况，

根据需要决定是否在不同CPU之间迁移进程。

但是迁移是有成本的，除了负载均衡以及迁移本身的代码需要消耗CPU之外，迁移可能

会导致cache失效，造成性能下降。而不同的迁移距离成本也大大不同，比如同一个核的两个超线程之间迁移的成本肯定远小于两个核之间。两个核之间的迁移又小于两个物理CPU。

两个物理CPU又小于两个NUMA node<sup>21</sup>。这些负载均衡程序都需要考虑到。

Linux通过调度域（sched\_domain）来组织CPU。sched\_domain组成一个树状结构，类似一个公司的组织结构。最顶层是numa domain包括所有的CPU，往下是physical domain包

---

<sup>19</sup> 参见《Linux IO子系统知识总结》。

<sup>20</sup> 通过在scheduler\_tick内触发软中断SCHED\_SOFTIRQ实现。

<sup>21</sup> NUMA架构下，不同node上的CPU访问不同区域的内存有不同的速度。

---

括同一个numa node上的所有CPU，再往下是core domain，包括同一个物理CPU上的所有核，再往下是CPU domain，包括同一个核上的所有超线程。

负载均衡程序从末端的CPU domain开始依次保证本域层次内的均衡。每一个层次的域可以采取不同的策略。越往上迁移成本越高，迁移越少发生<sup>22</sup>。

大多数情况下，业务稳定运行，负载均衡程序工作良好，CPU负载是平稳的，并不需要我们操心。遇到特殊场景需要避免迁移，可以使用taskset(1)或sched\_setaffinity(2)把进程绑定到一个CPU上。

负载均衡的入口函数位置：

```
5110 static void run_rebalance_domains(struct softirq_action *h)
5111 {
5112     int this_cpu = smp_processor_id();
5113     struct rq *this_rq = cpu_rq(this_cpu);
5114     enum cpu_idle_type idle = this_rq->idle_at_tick ?
5115                                     CPU_IDLE : CPU_NOT_IDLE;
5116
5117     rebalance_domains(this_cpu, idle);
5118
5119 #ifdef CONFIG_NO_HZ
5120     /*
5121     "kernel/sched.c" 11396 lines --44%--
```

### 1.13 小结

到此调度就介绍了，花了很多篇幅，首先因为它是核心的核心，其次，调度不只是操作系统的概念，我们平常的应用开发过程中，很多情况下也会遇到涉及调度的业务处理流程，

有的甚至比进程调度都复杂些，Linux对调度的理解和实现、以及其中的一些设计思想、实现技巧都可以为我们所借鉴。

---

<sup>22</sup> 你们知道跨产品线调动远比跨项目组难的多得多。

---

## 2 进程状态

### 2.1 TASK\_RUNNING

表示进程正在CPU上执行或者处于运行队列中。

对应top看到的R状态。

### 2.2 TASK\_INTERRUPTIBLE

表示进程不在运行队列中，并且可以接收信号，可以被终止。sleep poll select等主动休眠系统调用就会进入这种状态。

对应top看到的S状态。

### 2.3 TASK\_UNINTERRUPTIBLE

表示进程不在运行队列中，不接收信号，不可以被终止（kill -9无效）。很多不希望被打断的等待过程会导致进程进入D状态，最典型就是IO。

对应top看到的D状态。

### 2.4 TASK\_STOPPED

表示进程收到了SIGSTOP等信号处于停止状态。直到再次收到对应的SIGCONT等信号才会恢复原来的状态。

对应top看到的T状态。

### 2.5 TASK\_TRACED

表示进程正在被调试跟踪。例如使用strace、gdb等跟踪调试工具时。

对应top看到的t状态。

### 2.6 EXIT\_ZOMBIE

表示进程已经结束，但父进程未调用wait系统调用获取结束状态。

对应top看到的Z状态。

关于僵尸进程的详细介绍可参考本人博文[《正确理解Linux下的（zombie）僵尸进程》](#)。

---

## 2.7 EXIT\_DEAD

EXIT\_ZOMBIE之后进入EXIT\_DEAD，表示进程已经彻底结束。这个状态是瞬间的，仅

用来避免并发wait导致的竞争问题，用户一般感知不到。

## 3 进程创建和退出

### 3.1 树

Linux的进程是一个单根(tree)的树状组织<sup>23</sup>。PID为1的进程在系统启动的时候，由内核负责创建起来。1号进程就是所有进程的祖先。

```
# pstree -A
init--+-acpid
      |-agetty
      |-atop
      |-cron
      |-dbus-daemon
      |-hald---hald-runner--+-hald-addon-acpi
      |                     |-hald-addon-cpuf
      |                     `--hald-addon-inpu
      |-irqbalance
      |-java---67*[{java}]
      |-klogd
      |-mcelog
      |-6*[mingetty]
      |-nscd---12*[{nscd}]
      |-rpcbind
      |-run.sh---java---298*[{java}]
      |-sshd---sftp-server
      |-sshd--+-sshd---bash---su---csh---top
      |       |-sshd---bash---su---csh
      |       `--sshd---bash---pstree
```

每一个进程必然有一个父进程。即使老祖宗1号进程也不是石头里面蹦出来，也有爹。

1

号进程的父进程是0号进程（ps命令可以看到），0号进程是一个特殊进程，实际上内核初

---

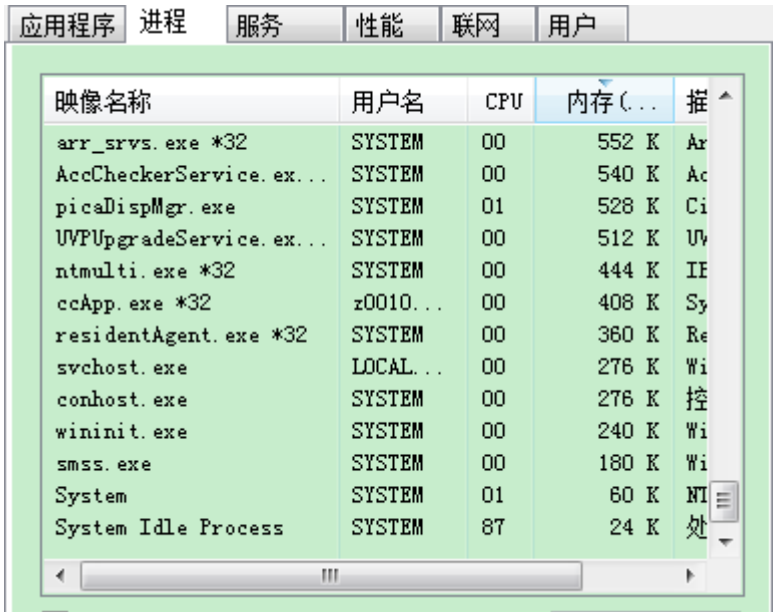
<sup>23</sup> 到处都是树。Linux师法自然。



始

化完毕之后它也一直存在，而且干的活儿非常重要——让CPU放松下来。CPU没活干的时候，就去执行这个进程，这个进程调用特殊的CPU指令，让CPU“休息休息”“降降温”。

Windows也有这样一个进程，作用是一样的。



映像名称	用户名	CPU	内存 (K)	操作
arr_srvs.exe *32	SYSTEM	00	552 K	Ar
AccCheckerService.exe...	SYSTEM	00	540 K	Ac
picaDispMgr.exe	SYSTEM	01	528 K	Ci
UVPUUpgradeService.exe...	SYSTEM	00	512 K	UV
ntmulti.exe *32	SYSTEM	00	444 K	IE
ccApp.exe *32	z0010...	00	408 K	Sy
residentAgent.exe *32	SYSTEM	00	360 K	Re
svchost.exe	LOCAL...	00	276 K	Wi
conhost.exe	SYSTEM	00	276 K	控
wininit.exe	SYSTEM	00	240 K	Wi
smss.exe	SYSTEM	00	180 K	Wi
System	SYSTEM	01	60 K	NT
System Idle Process	SYSTEM	87	24 K	处

我们有时候使用oprofile会看到有些比较空闲的环境，但内核态的函数占用比例很高，函数的名字都是包含“idle”字样。这就是0号进程干的**好事**了。

### 3.2 fork

一个进程是由它的父进程创建的，比如我们在命令行下执行a.out，那么这个进程就是被

shell程序比如bash创建的。bash首先读取我们的输入，然后调用相关系统调用创建这个子进

程。这个创建子进程的系统调用的名字不是create。而是fork——子进程不只是说仅仅把自己的ppid设置为父进程的pid，保持一个树状组织这么简单。fork出来的子进程完全复制了父进程的**代码和内存**！第一次接触的时候，总会觉得不可思议。想想下面的代码会打印多少个hello world？

```
int main()
{
    int i;
```

---

```
        for (i = 0; i < 10; i++)
        {
            fork();
            printf("hello world\n");
        }
    }
```

但是我要执行的是a.out程序，我a.out的代码如何运行？载入a.out的代码并执行通过另外一个系统调用execve完成。也就是说，Linux（Unix）把进程创建和程序加载分开了，先克隆一份父亲，后面再装入儿子的思想。

fork不接受任何参数，但却**返回两次**。这确实颠覆了我们朴素的理解。一个函数怎么可能

以返回两次？而且两次返回值不同？

```
# cat t.c
int main()
{
    int pid;
    if ((pid = fork()) == 0)
    {
        printf("child, my pid = %d\n", getpid());
    }
    else
    {
        printf("parent, my child pid = %d\n", pid);
    }
}

./a.out
parent, my child pid = 5116
child, my pid = 5116
```

一个函数并不能返回两次，除非它被执行了两次。如果同一个函数（一段代码）分别在

不同的进程内执行，那就可以分别返回不同的值。fork在父进程内返回新创建子进程的pid，

在子进程内返回0。

不过看来还是不好理解，那么不妨假设我们去按create这种直观的方式实现——在进程调用create系统调用后，我们为之分配一个新的pid，建立进程的地址空间然后**从磁盘上读取该程序的代码和数据**，并跳转到程序的入口函数执行它。这非常好理解。

---

如果不从磁盘上载入代码和数据，而是直接拷贝父进程的代码和数据以及父进程的运行状态，父进程从create返回，子进程也从create返回，但是返回值设置成不一样的。这就是fork了！

理解起来没有问题但做起来不一定容易。在代码实现上，这种机制的难点在于如何让新

创建进程也从fork返回，并且返回不同的值？复制简单，关键是复制之后，子进程的第一条

指令从哪里开始？显然不是用户态代码位置，这个位置一定是fork系统调用返回之前。

这个过程隐含在进程切换、和系统调用的处理机制内了。这个细节可以从以下相关内核代码找到。

```
460 ENTRY(system_call)
461     CFI_STARTPROC    simple
462     CFI_SIGNAL_FRAME
463     CFI_DEF_CFA      rsp,KERNEL_STACK_OFFSET
464     CFI_REGISTER     rip,rcx
465     /*CFI_REGISTER   rflags,r11*/
466     SWAPGS_UNSAFE_STACK
467     /*
468      * A hypervisor implementation might want to use a label
469      * after the swapgs, so that it can do the swapgs
```

"arch/x86/kernel/entry\_64.S" 1659 lines --28%--

```
1375 long do_fork(unsigned long clone_flags,
1376              unsigned long stack_start,
1377              struct pt_regs *regs,
1378              unsigned long stack_size,
1379              int __user *parent_tidptr,
1380              int __user *child_tidptr)
```

```
1381 {
```

"kernel/fork.c" 1767 lines --76%--

```
275 int copy_thread(unsigned long clone_flags, unsigned long sp,
276                 unsigned long unused,
277                 struct task_struct *p, struct pt_regs *regs)
278 {
279     int err;
```

"arch/x86/kernel/process\_64.c" 681 lines --40%--

```
125 /* Save restore flags to clear handle leaking NT */
```

---

```

126 #define switch_to(prev, next, last) \
127     asm volatile(SAVE_CONTEXT                                \
128         "movq %%rsp,%P[threadrsp](%[prev])\n\t" /* save RSP */ \
129         "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */ \
130         "call __switch_to\n\t"                               \
131         ".globl thread_return\n"                             \
132         "thread_return:\n\t"                                  \
133         "movq \"__percpu_arg([current_task]),%%rsi\n\t"      \
134         __switch_canary                                       \
135         "movq %P[thread_info](%%rsi),%%r8\n\t"               \
"arch/x86/include/asm/system.h" 463 lines --21%--

```

其关键过程如下：

我们用p代表父进程，c代表子进程。

1. p进程主动执行fork系统调用，切换至内核态，执行fork的实现代码。
2. p进程创建子进程的c，分配相关数据结构，比如最重要的task\_struct。
3. p进程把自己的地址空间拷贝给c。
4. p进程把自己的进入fork系统调用时的sp等寄存器，拷贝给c。这样c拥有了p刚进

### fork

时的上下文。

5. p进程为c设置一个标记（TIF\_FORK）。

```

296 set_tsk_thread_flag(p, TIF_FORK);
"arch/x86/kernel/process_64.c" 681 lines --38%--

```

6. p进程把c进程设置为就绪状态，放入CPU的运行队列。就从fork系统调用了。
7. 这时候，关键时刻到来了。我们前面介绍了系统调用返回时会检查调度标记（TIF\_NEED\_RESCHED），如果设置了就会执行schedule从运行队列中选择新的进程运行，假设p的时间片到了，那么就会选择，并且假设选择了c，schedule切到c的上下文以后，会判断前面设置的TIF\_FORK，接着会直接跳转到ret\_from\_fork，而不是从schedule返回，这是一个新的进程它没得返回。而ret\_from\_fork会进一步执行ret\_from\_sys\_call像父进程那样从系统调用返回。由于之前复制了父进程进入系统调用时的上下文，就好像子进程也执行了fork，但fork是一个空函数，直接返回了。

```

125 /* Save restore flags to clear handle leaking NT */
126 #define switch_to(prev, next, last) \
127     asm volatile(SAVE_CONTEXT                                \
128         "movq %%rsp,%P[threadrsp](%[prev])\n\t" /* save RSP */ \
129         "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */ \

```

---

```

130      "call __switch_to\n\t"
131      ".globl thread_return\n"
132      "thread_return:\n\t"
133      "movq \"__percpu_arg([current_task])\", %%rsi\n\t"
134      __switch_canary
135      "movq %P[thread_info](%%rsi), %%r8\n\t"
136      "movq %%rax, %%rdi\n\t"
137      "testl  %[_tif_fork], %P[ti_flags](%%r8)\n\t"
138      "jnz   ret_from_fork\n\t"
139      RESTORE_CONTEXT
140      : "=a" (last)
141      __switch_canary_oparam
142      : [next] "S" (next), [prev] "D" (prev),
143        [threadrsp] "i" (offsetof(struct task_struct, thread.sp)), \
144        [ti_flags] "i" (offsetof(struct thread_info, flags)), \
145        [_tif_fork] "i" (_TIF_FORK),
146        [thread_info] "i" (offsetof(struct task_struct, stack)), \
147        [current_task] "m" (per_cpu_var(current_task))
148      __switch_canary_iparam
149      : "memory", "cc" __EXTRA_CLOBBER)
150 #endif

```

### 3.3 COW

fork的时候，需要完全复制父进程的内存，但这种复制一开始是不需要的，通过地址映射机制，把子进程的地址空间也映射到相同的物理内存就可以了。到了后面，如果子进程或者父进程要修改内存，再分配新的物理内存进行复制并修改映射就可以了。这种技术叫做copy on write（COW）。

### 3.4 execve

进程可以在任何时候调用execve执行新程序，进程会被新程序完全替代。就好像它退出了一样。地址空间被重新分配。我们执行popen system等C库函数拉起外部程序，这些C库函数帮我们先做了fork，然后再在子进程内执行execve。

虽然原进程被清理替换，但打开的文件句柄会被execve保留。所以会出现程序通过

---

popen

调用外部程序或脚本，之后程序自己结束了，但程序建立的socket没有关闭，端口被还未停

止的外部程序或脚本占用的情况。

如果成功，execve并不会返回，因为原来的要返回的代码位置已经不存在了。在进入execve系统调用之前，发生用户态到内核态的切换，进程用户态的sp以及ip等会被保存。execve执行成功后，保存的sp和ip会被根据程序文件的内容重新初始化和设置。保存的ip会被设置为程序的入口地址。一旦从系统调用返回后，就自然执行了新程序的入口代码。

### 3.4.1 ELF

ELF是linux程序文件的格式，内核可以理解这种格式。相应的处理代码定义在fs/binfmt\_elf.c。

程序的代码、全局变量都在这个文件内体现。最关键的是入口地址。即execve返回之后的执行地址。这个地址在文件内保存。每个程序文件可能不一样。

```
# readelf -h a.out
ELF Header:
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
Class:                               ELF64
Data:                                 2's complement, little endian
Version:                             1 (current)
OS/ABI:                              UNIX - System V
ABI Version:                         0
Type:                                EXEC (Executable file)
Machine:                             Advanced Micro Devices X86-64
Version:                             0x1
Entry point address:                 0x400530
```

通过readelf命令我们可以读出来。可以看到这个地址对应的符号名是\_start。

```
# objdump -d a.out | grep -A 10 400530
0000000000400530 <_start>:
400530:    31 ed                xor    %ebp,%ebp
400532:    49 89 d1             mov    %rdx,%r9
400535:    5e                  pop    %rsi
400536:    48 89 e2             mov    %rsp,%rdx
400539:    48 83 e4 f0          and    $0xfffffffffff0,%rsp
40053d:    50                  push   %rax
```

---

```

40053e:    54                push    %rsp
40053f:  49 c7 c0 20 07 40 00  mov     $0x400720,%r8
400546:  48 c7 c1 90 06 40 00  mov     $0x400690,%rcx
40054d:  48 c7 c7 14 06 40 00  mov     $0x400614,%rdi
400554:  e8 9f ff ff       callq   4004f8 <__libc_start_main@plt>

```

这段代码是编译器帮我们生成的。\_\_libc\_start\_main会调用我们定义的main函数。

### 3.4.2 动态库的处理

但\_\_libc\_start\_main是一个动态符号，由动态库libc提供。

```

# ldd a.out
linux-vdso.so.1 (0x00007fff3cd14000)
libc.so.6 => /lib64/libc.so.6 (0x00007f3df2bd2000)
/lib64/ld-linux-x86-64.so.2 (0x00007f3df2f76000)
# readelf -s /lib64/libc.so.6 |grep __libc_start_main
5384: 000000000000212d0    439 FUNC      GLOBAL DEFAULT   12 __libc_start_main

```

我们知道在符号第一次被使用时，会解析并填充实际地址。这个解析和填充的动作是由ld-linux-x86-64.so.2这个库内的函数（\_dl\_runtime\_resolve）完成的。

不过ld-linux-x86-64.so.2本身也是一个动态库，\_dl\_runtime\_resolve本身也是动态符号，那么谁来解析并填充它的实际地址？答案是ld-linux-x86-64.so.2。但这时候它的代码还没执行呢？

实际上对于动态链接的程序，程序文件定义的Entry point会被忽略，内核会在程序文件内找到interpreter的定义，其值就是动态库加载器的路径/lib64/ld-linux-x86-64.so.2。

```

Elf file type is EXEC (Executable file)
Entry point 0x400530
There are 9 program headers, starting at offset 64

```

Program Headers:

Type	Offset	VirtAddr	PhysAddr
	FileSiz	MemSiz	Flags Align
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040
	0x00000000000001f8	0x00000000000001f8	R E 8
INTERP	0x0000000000000238	0x0000000000400238	0x0000000000400238
	0x000000000000001c	0x000000000000001c	R 1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000
	0x000000000000085c	0x000000000000085c	R E 200000
LOAD	0x0000000000000e18	0x0000000000600e18	0x0000000000600e18

---

```
0x0000000000000220 0x0000000000000230 RW 200000
DYNAMIC 0x0000000000000e40 0x000000000000600e40 0x000000000000600e40
```

内核会把ld-linux的Entry point设置为返回用户态的执行位置。也是\_start，这里会做一些

初始化动作，其中最重要的一个就是填充\_dl\_runtime\_resolve的地址。然后才会调用程序文件的\_start。

```
# readelf -a /lib64/ld-linux-x86-64.so.2 | grep Entry
Entry point address: 0x1640
# readelf -s /lib64/ld-linux-x86-64.so.2 | grep 1640
298: 00000000000001640 0 NOTYPE LOCAL DEFAULT 11 _start
```

### 3.5 exit

与创建使用fork系统调用对应的是，退出使用的是exit，exit由\_\_libc\_start\_main调用。

```
(gdb) disass 140737351973888
Dump of assembler code for function _dl_runtime_resolve:
0x00007ffff7defc00 <+0>: sub $0x38,%rsp
0x00007ffff7defc04 <+4>: mov %rax,(%rsp)
0x00007ffff7defc08 <+8>: mov %rcx,0x8(%rsp)
0x00007ffff7defc0d <+13>: mov %rdx,0x10(%rsp)
.....
0x00007ffff7a593b9 <+233>: callq *0x18(%rsp) //调用我们的main函数
0x00007ffff7a593bd <+237>: mov %eax,%edi
=> 0x00007ffff7a593bf <+239>: callq 0x7ffff7a6f6e0 <exit> //调用exit系统调用
0x00007ffff7a593c4 <+244>: xor %edx,%edx //这里不会执行到
0x00007ffff7a593c6 <+246>: jmpq 0x7ffff7a59309 <__libc_start_main+57>
```

exit就没有fork那么特殊了，就是一条流走下来，切入内核态，执行内核的进程退出代码。这部分代码定义在"kernel/exit.c"。

exit接收一个参数，就是退出码。这个退出码会保存在struct task\_struct里面。父进程通过wait系统调用可以获得这个退出码。我们在shell中执行一个程序，然后使用echo \$?获取

退出码实际上shell进程通过wait获取后保存到了内置变量\$?中。

exit会释放一切，释放地址空间，自然也解除了到物理内存的映射，内存可以被别人映射过去使用。关闭一切打开的文件（句柄）。最后会释放task\_struct对象，不过在这之前必须做一件事情，通知父进程自己的退出状态。通知通过给父进程发送SIGCHLD信号方式进行。而如果父进程调用wait，则会处理这个信号，并获得错误码。



---

发送信号之后，进程会进入EXIT\_ZOMBIE，直到父进程调用wait否则进程会一直保持EXIT\_ZOMBIE状态，即top看到的Z状态，即成为一个僵尸。

如果父进程使用sigaction(2)忽略了SIGCHLD信号，那么子进程退出时，将不会给它发送SIGCHLD信号，也就不会进入Z状态。

对于父进程提前退出的情况，子进程会成为孤儿进程，那么它的SIGCHLD发给谁？SIGCHLD发给1号进程init，在父进程退出时，其子进程会被过继给init，init成为新的父进程。所以也可以通过在main函数内做一次或多次fork，让父进程退出，在子进程内继续运行程序的方式，让自己的父进程变为1号，就可以避免成为僵尸进程了。

## 3.6 线程

线程的创建使用clone这个系统调用，这是pthread\_create的底层函数。

### NAME

clone, \_\_clone2 - create a child process

### SYNOPSIS

```
#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <sched.h>

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

与fork不同的地方在于，clone创建的“子进程”与父进程共享地址空间，并且其返回用  
户态后执行的入口函数是传入clone的第一个参数，一个函数地址。clone这个函数是C库封装  
的，内核对应的clone的实现sys\_clone也就是直接调用do\_fork，只是传递的参数不同。其实  
根本就没有fork这个系统调用了，C库封装的fork函数同样是执行clone这个系统调用。

```
532 asmlinkage long
533 sys_clone(unsigned long clone_flags, unsigned long newsp,
534           void __user *parent_tid, void __user *child_tid, struct pt_regs *regs)
535 {
536     if (!newsp)
537         newsp = regs->sp;
538     return do_fork(clone_flags, newsp, regs, 0, parent_tid, child_tid);
539 }
"arch/x86/kernel/process_64.c" 681 lines --73%--
```

---

但是注意到，进入内核之后，我们传给clone的线程入口函数已经不见了。那么怎么执行到我们的入口函数？与前面所述的fork过程一样，父进程和子进程都会从系统调用返回，即返

回到C库封装的clone函数内，clone函数可以根据系统调用返回值，判断出是子进程，然后执

行入口函数。内核本身并不需要知道线程的入口函数。本质还是fork，只不过fork出来的子进程和父进程内存完全共享，然后你自己在新进程内执行所谓的线程入口函数而已。

```
(gdb) bt
#0  work (arg=0x0) at test.c:8 //work是我们传递给pthread_create入口函数
#1  0x00002aaaaacd35f0 in start_thread () from /lib64/libpthread.so.0 // start_thread 是pthread传递给
clone的入口函数，这个函数调用work。
#2  0x00002aaaaafbd84d in clone () from /lib64/libc.so.6 //C库的clone调用pthread传入的start_thread
#3  0x0000000000000000 in ?? ()
(gdb) disass clone
Dump of assembler code for function clone:
.....
0x00002aaaaafbd84b <clone+107>: callq  *%rax    //调用start_thread
0x00002aaaaafbd84d <clone+109>: mov  %rax,%rdi //rax放的是start_thread的返回值，作为退出码
0x00002aaaaafbd850 <clone+112>: callq  0x2aaaaaf8c3e0 <_exit> //调用exit退出“进程”
```

线程的退出和进程的退出一样，都是执行exit，我们线程函数返回后，pthread的函数继续返回，然后到clone函数里面，它会执行exit。

但线程退出不会给父进程（主线程）发送SIGCHLD。不会进入Z状态。

## 4 进程间通讯（常用）

### 4.1 管道

pipe这个系统调用用于创建一个管道，pipe返回两个句柄，对应管道的两端，pipefd[0]是读端，pipefd[1]是写端。两个进程分别read和write两端，就可以完成数据传递。

#### SYNOPSIS

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

管道是单工的，即写端只能写，读端只能读，如果你想两个进程互相收发，只能创建两个管道。

pipe在一个进程内调用执行，然后把返回的句柄传递给另外一个进程，传递的方式是

---

fork，fork的时候子进程自然继承了父进程的句柄。

当我们在shell下执行ls | more这样的命令时，经过了以下过程，这个过程可以通过strace跟踪shell进程获取到。我们看一下关键过程：

```
1 13140 write(2, "ls | more", 12) = 12 //shell把输入的命令打印到终端
2 13140 write(2, "\n", 1) = 1
3 13140 pipe([3, 4]) = 0 //创建一个管道，对应3和4号句柄
5 13140 clone(child_stack=0 //fork 一个子进程
8 13140 clone( <unfinished ...> //再fork 一个子进程
16 20533 dup2(3, 0 <unfinished ...> //20533这个子进程把3号句柄复制到0号句柄上，实际上指向了标准输入，那么这个进程其实就是要执行more的。
20 20533 close(3 <unfinished ...> //关闭3号句柄
24 20532 dup2(4, 1) = 1 //20532这个子进程把4号句柄复制到1号句柄上，实际上指向了标准输出，那么这个进程其实就是要执行ls的
25 20532 close(4) = 0 //关闭4号句柄
26 20532 execve("/usr/bin/ls", ["ls"], [/* 69 vars */] <unfinished ...> //加载ls程序
27 20533 execve("/usr/bin/more", ["more"], [/* 69 vars */]) = 0 //加载more程序
```

此后ls程序执行起来之后，它并不知道1号句柄是终端还是管道，反正它的代码就是把内容写到1号句柄即标准输出上（printf），然后more同理就是读0号句柄，也就是读ls输出的内容。ls和more这两个程序内并没有处理管道的代码。

pipe的内核代码定义在，"fs/pipe.c"。

内部其实就一个缓冲区，使用一个文件封装了这个缓冲区，所以可以像读写文件那样操作

它。实际上存在一个名为pipefs的虚拟文件系统，不对外暴露，新建一个pipe就是在这个文件系统上新建一个文件。这是一贯玩法，实际上还存在sockfs，一个socket也会有一个文件与之对应，这个文件就放在sockfs这个虚拟文件系统上。我们可以通过proc查到一个pipe或socket对应的文件的inode号：

```
/proc/5062/fd:
total 0
dr-x----- 2 root root  0 Feb 19 14:59 ./
dr-xr-xr-x  8 root root  0 Feb 18 15:55 ../
```

---

```
lr-x----- 1 root root 64 Feb 19 14:59 3 -> pipe:[16761]
l-wx----- 1 root root 64 Feb 19 14:59 4 -> pipe:[16761]
lrwx----- 1 root root 64 Feb 19 14:59 5 -> socket:[16045]
```

这个缓冲区，一共16个页面，在x86上通常也就是64KB。缓冲区满了，write端继续write就会阻塞，缓冲区空了read端继续read就会阻塞。

## 4.2 信号

使用kill -l可以查看Linux支持的信号，信号用于进程间状态通知，传递简单信息。

```
# kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM     15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT    19) SIGSTOP    20) SIGTSTP
```

kill命令调用kill系统调用发送信号。

```
NAME
    kill - send signal to a process

SYNOPSIS
    #include <sys/types.h>
    #include <signal.h>
    int kill(pid_t pid, int sig);
```

kill的实现定义在"kernel/signal.c"。信号会被缓存到进程上的一个队列上，这个队列的长度通过ulimit控制。

```
# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 30513
```

kill在放入队列后，对进程做一个标记TIF\_SIGPENDING，然后返回。在进程中断或系统调用返回的时候会检查这个标记，并作信号处理。和调度切换一样的做法。

实际上信号的行为类似一个中断，可以理解为一种软中断。程序可能在任何位置被打断，所以信号处理函数内的操作要十分谨慎。

如果你用kill发送信号，它是作用于整个进程的，有线程都可能会处理这个信号，但只会处理一次，这个意思是，哪个线程先走到了TIF\_SIGPENDING的检查点，哪个就处理了这个信号。

---

如果想给指定的线程发送信号，就需要使用`tgkill`这个系统调用。它也是`pthread_kill`的底层函数。

### 4.3 共享内存

线程间内存是共享的，共享内存是一种使进程间可以共用一段物理内存的机制。

`shmget(2)`系统调用创建一段共享内存，返回一个id——`shmid`，`shmid`在系统内全局存在。进程把`shmid`传入`shmat(2)`系统调用，映射这一段共享内存到自己的地址空间内，就可以使用了。

一旦一段共享内存被创建，它就成为系统级的一个资源，即使所有使用它的进程退出了，它仍然存在，除非显示的执行`shmctl(2)`或`ipcrm(1)`删除它。这非常类似一个文件。实际上文件也是一种进程通讯方式。但和内核进程管理这部分没啥关系。

**Linux的共享内存，确实是使用文件封装了。这是Linux（Unix）的哲学。一切皆文件。**

所以我们通过`pmap`查看进程的地址空间，可以看到共享内存地址段，映射到一个名为"`SYSV`"开头的文件上。该文件位于一个`tmpfs`文件文件系统上。不对外部暴露。相关代码定义在"`ipc/shm.c`"。

`shmget`就是创建文件。`shmat`就是映射文件的到自己的地址空间，和`mmap`一个普通文件是一样的。

你可以使用`mount`命令挂载一个`tmpfs`文件系统到一个目录，进程访问这个目录内的文件，就是访问内存。如果多个进程同时`mmap`一个`tmpfs`文件系统上的文件到自己的地址空间，也就是起到共享内存作用了。`shmget`只不过把我们这种手工创建的方式封装起来了，并把底层的`tmpfs`屏蔽掉了而已。

正因为共享内存建立于`tmpfs`文件系统之上，所以共享内存会被统计到`free`的`cached`字段，因为`cached`字段统计的正是包含了文件映射的内存，即使这个文件不在磁盘上。参考本人博文《[说说free命令](#)》以及《[SUSE10和SUSE11可用内存统计方法](#)》。

### 4.4 锁

锁严格上讲并不是内核提供的进程间通讯方式。但内核却为锁做了一些工作。考虑锁

---

的一般实现：

```
while(condition ok) { sleep; }
```

一般condition就是探测一个变量。sleep导致进程放弃CPU，如果不sleep就是自旋锁。

用户态的自旋锁，并不需要和内核打交道。pthread提供了一个pthread\_spinlock\_t。

重点考虑休眠锁，sleep多长时间？太短了，浪费CPU。太长了，如果在休眠还未到期的时候，别人已经释放了锁，你却还在等待，则严重影响性能。

所以通常pthread\_mutex\_t等用户态的休眠锁都是利用内核的休眠和唤醒机制实现的。具体来说就是通过futex系统调用。全称是**Fast Userspace Locking system call**，如果你用strace看到一个进程调用futex非常频繁，那估计和锁有关。

#### NAME

futex - Fast Userspace Locking system call

#### SYNOPSIS

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int op, int val, const struct timespec *timeout,
          int *uaddr2, int val3);
```

futex的实现定义在"kernel/futex.c"

futex系统调用接收两个标记FUTEX\_WAIT和FUTEX\_WAKE，实际上对应两个内核函数futex\_wait和futex\_wake。

futex\_wait会把进程放入一个等待队列然后执行**schedule放弃CPU进入休眠状态**。

```
1702 static void futex_wait_queue_me(struct futex_hash_bucket *hb, struct futex_q *q,
1703                                 struct hrtimer_sleeper *timeout)
1704 {
1711     set_current_state(TASK_INTERRUPTIBLE); //设置成S状态
1712     queue_me(q, hb); //放到一个等待队列上，
1713     .....
1725     if (likely(!plist_node_empty(&q->list))) {
1726         /*
1727          * If the timer has already expired, current will already be
1728          * flagged for rescheduling. Only call schedule if there
1729          * is no timeout, or if it has yet to expire.
1730          */
1731         if (!timeout || timeout->task)
1732             schedule(); //放弃CPU
1733     }
1734     //从schedule函数返回，说明被唤醒了，修改为R状态继续运行。
1735     __set_current_state(TASK_RUNNING);
```

---

```
1735 }
1736
"kernel/futex.c" 2695 lines --63%--
```

futex\_wake会唤醒这个等待队列上的进程。所谓唤醒，就是把进程再放到CPU的运行队列上，这样如果被调度到，就会从schedule返回，继续执行。

拿不到锁的时候，会调用futex\_wait，进入休眠。释放锁的时候调用futex\_wake，等待锁的进程立即进入运行队列，没有sleep那样的延迟。

关于锁更多的探讨，可以参考本人博文《[多核环境下锁优化方法分析](#)》。

## 5 附录

### 5.1 Coredump

进程发生coredump是因为进程没办法继续运行了。有很多情况会导致这种结果。最典型的访问了非法地址，比如访问了空指针，0x0这个地址不在地址空间上，自然无法访问，你也无法运行了，如果继续运行，返回什么数据给你？所以内核会把进程core掉。实际上内核会给进程发送个信号，比如这种情况下会发送SIGSEGV信号，俗称段错误。收到这个信号的进程，会执行coredump。

不同的程序文件格式有不同的core dump方法，默认我们的程序都是ELF格式。

ELF的coredump代码定义在：

```
1886 static int elf_core_dump(long signr, struct pt_regs *regs, struct file *file, unsigned long limit)
1887 {
1888     int has_dumped = 0;
1889     mm_segment_t fs;
"fs/binfmt_elf.c" 2058 lines --89%--
```

所谓dump，就是把进程的内存全部保存下来，再加上CPU寄存器信息。位于内存中的栈自然被保存下来，这意味着根据core文件我们可以追溯代码的执行路径，找出core的代码位置和原因。

core文件包含了进程的全部内存，有时候会非常的大，比如使用了共享内存，但共享内存这部分通常对分析问题没多大作用。所以2.6.23以后，内核新增了一个控制开关，/proc/PID/coredump\_filter，通过它可以有选择的dump内存。具体可参考man 5 core。

只有core文件没法分析问题，因为core中虽然包含了代码段，但是它没有代码位置对应的符号信息，符号信息是保存在程序文件内的。所以使用gdb分析core的时候，同时需要

---

提供程序文件。

如果gcc编译的时候，增加了-g参数，会增加调试信息到程序文件内，分析core的时候，可以看到具体的源码行。这个非常方便，但会导致程序包比较大。Linux下有一个工具叫做strip，可以把程序文件中的调试信息甚至符号信息都抽离出来，保存在一个单独的文件内，通常称作debuginfo，不发布。

Linux的发行版，suse、redhat、ubuntu等等都是这么做的，它的所有rpm或deb包都是带-g编译的，然后使用strip抽离出一个单独的debuginfo包。如果某个系统程序挂了，找到对应的debuginfo就可以分析了。其实内核也这个样子。我们的应用程序可以参考这种做法。

## 6 参考资料

Linux Source Code 2.6.32

Understanding the Linux Kernel. 3rd. Edition

Linux Kernel Development. 3rd. Edition.