目录

[目录 1](#_Toc47715928)

[第一部分 进程的内容 3](#_Toc47715929)

[1.什么是进程？ 3](#_Toc47715930)

[2 linux进程的组织方式 5](#_Toc47715931)

[3.进程的生命周期 5](#_Toc47715932)

[3.1.什么是僵尸进程： 6](#_Toc47715933)

[3.2 stop状态： 7](#_Toc47715934)

[3.3 睡眠 7](#_Toc47715935)

[3.4调度： 8](#_Toc47715936)

[4.fork 8](#_Toc47715937)

[5.什么是内存泄漏？ 8](#_Toc47715938)

[第二部分 进程和线程的本质 10](#_Toc47715939)

[1.进程拥有资源 10](#_Toc47715940)

[2.写时拷贝技术： 11](#_Toc47715941)

[3. vfork的 mm指针直接指向父进程mm； 12](#_Toc47715942)

[4.线程 13](#_Toc47715943)

[5.人妖 14](#_Toc47715944)

[6.PID 14](#_Toc47715945)

[7.进程死亡： 15](#_Toc47715946)

[8.睡眠 15](#_Toc47715947)

[9. 0进程 16](#_Toc47715948)

[第三部分 进程调度 18](#_Toc47715949)

[1. 吞吐率和响应 18](#_Toc47715950)

[2. Linux任务类型 19](#_Toc47715951)

[3任务调度 20](#_Toc47715952)

[第四部分 负载均衡 23](#_Toc47715953)

[1.线程的负载均衡 23](#_Toc47715954)

[2.中断负载均衡 24](#_Toc47715955)

[RPS补丁 26](#_Toc47715956)

[3.linux不是硬实时系统： 28](#_Toc47715957)

[第5部分 Cgroup： 33](#_Toc47715958)

[(1)按群组分配资源 33](#_Toc47715959)

[(2)设置群的quota配额: 35](#_Toc47715960)

[1.sched\_autogroup: 35](#_Toc47715961)

[2 dock和cgroup 36](#_Toc47715962)

[3 systemd和cgroup 36](#_Toc47715963)

[4 cpuset和cgroup 37](#_Toc47715964)

[第六部分 深度睡眠 38](#_Toc47715965)

[1.深度睡眠特点及存在原因 38](#_Toc47715966)

[2.深度睡眠对load average的影响 39](#_Toc47715967)

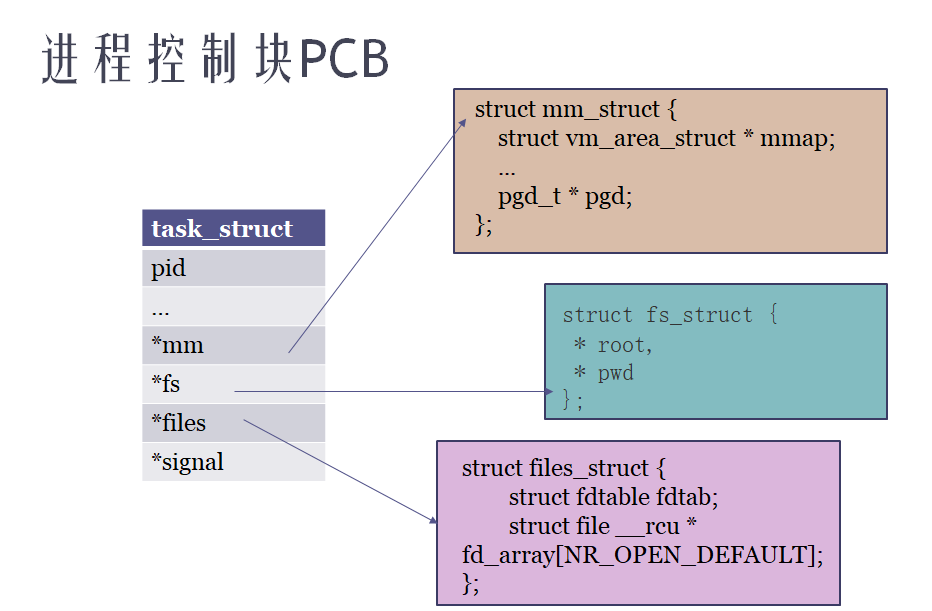
[3.TASK\_KILLABLE 40](#_Toc47715968)

# 第一部分 进程的内容

## 1.什么是进程？

进程：资源的封装单位；

linux用一个PCB来描述进程，即task\_struct， 其包含mm,fs,files,signal…



1. root目录，是一个进程概念，不是系统概念；

apropos chroot

man chroot 2

如下图，将分区/dev/sda5挂载到/mnt/a，调用chroot，改变root目录，当前进程下的文件b.txt即位于当前进程的根目录。



(2)fd也是进程级概念；

(base) leon@leon-Laptop:/proc/29171$ ls fd -l

总用量 0

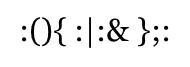
lrwx------ 1 leon leon 64 5月 16 10:26 0 -> /dev/pts/19

lrwx------ 1 leon leon 64 5月 16 10:26 1 -> /dev/pts/19

lrwx------ 1 leon leon 64 5月 16 10:26 2 -> /dev/pts/19

(3)pid，系统全局概念

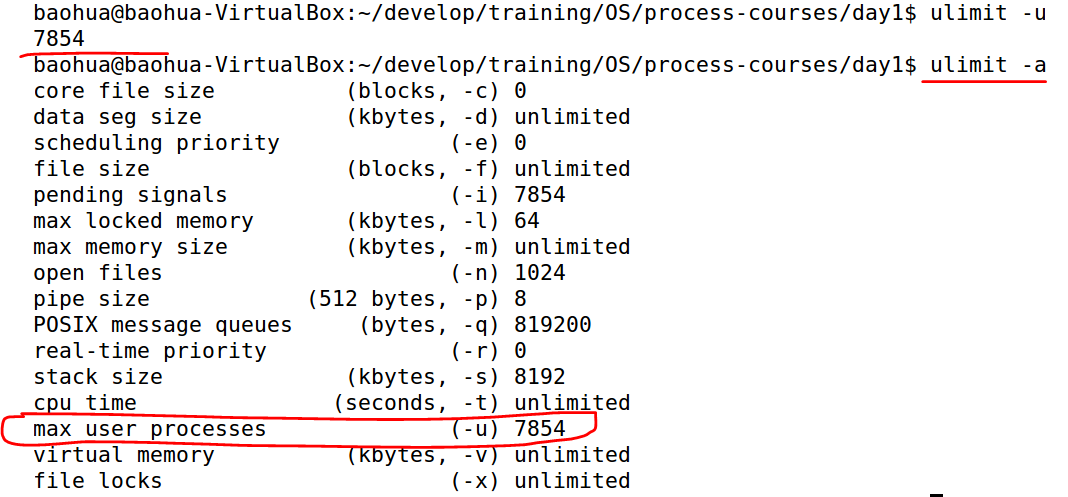
Linux总的PID是有限的，用完PID



每个用户的PID也是有限的

ulimit -u 最大进程数

ulimit –a

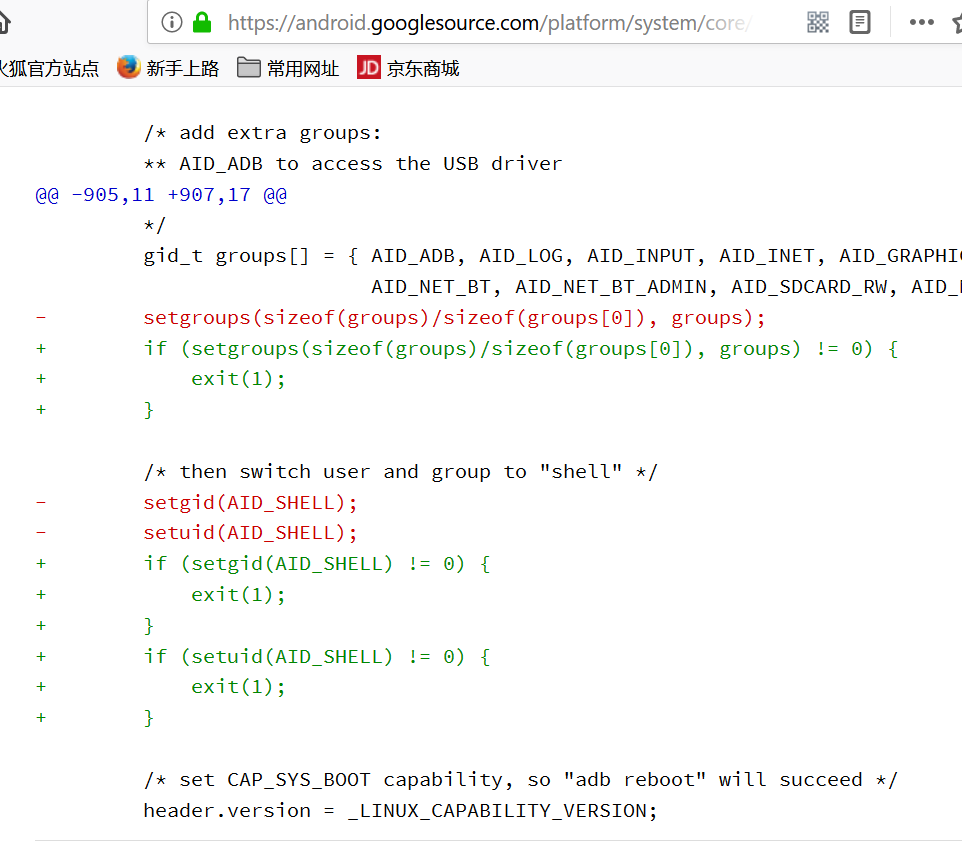


(base) leon@leon-Laptop:/proc/29171$ cat /proc/sys/kernel/pid\_max

32768

案例：android2.2漏洞

Android提权漏洞分析——rageagainstthecage: <https://blog.csdn.net/feglass/article/details/46403501>



Setuid(shell):rootshell用户 PID用完时，降权失败，依然具有root权限

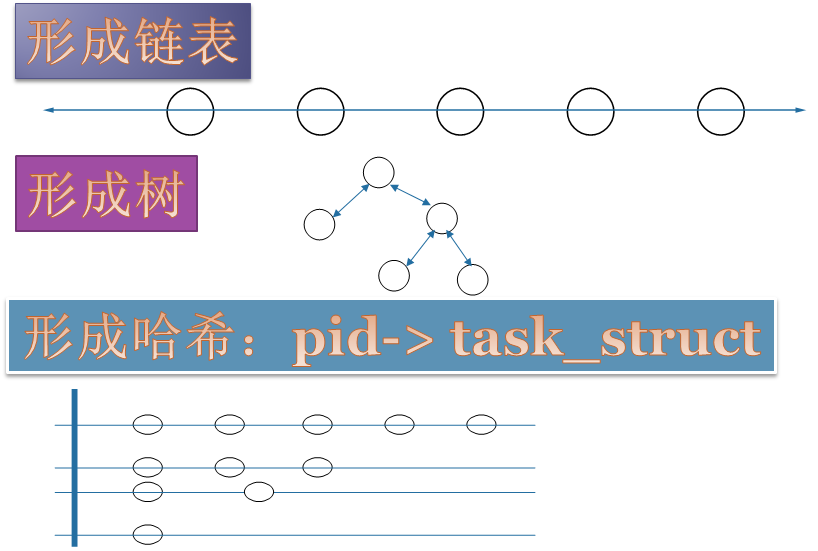
解决办法，检查返回值

软件工程符合墨菲定律，解决办法，代码写出闭环。

## 2 linux进程的组织方式

linux里的多个进程，其实就是管理多个task\_struct，那他们是怎么组织联系的呢？

组织task\_struct的数据结构：



a.链表，遍历进程

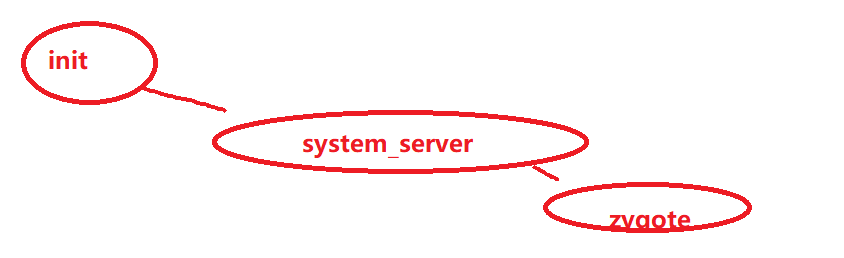
b.树：方便查找父子相关进程

c.哈希表：用于快速查找

用三种数据结构来管理task\_struct，以空间换时间。

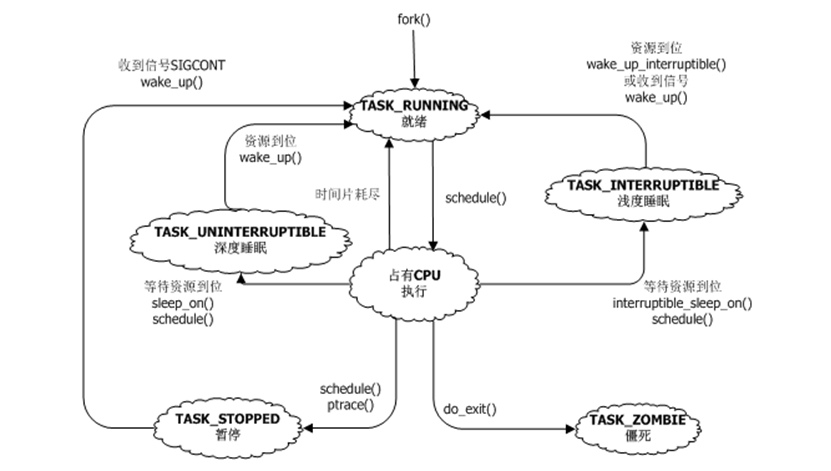
父进程监控子进程，linux总是白发人送黑发人。父进程通过wait，读取task\_struct的退出码，得知进程死亡原因。并且清理子进程尸体。

Android/或者服务器，都会用由父进程监控子进程状态，适时重启等；



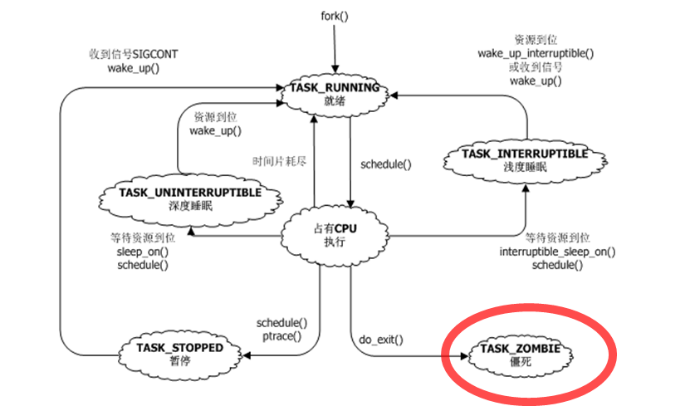
## 3.进程的生命周期

有六个状态：就绪，运行，睡眠(深度/浅度睡眠)，暂停，僵尸

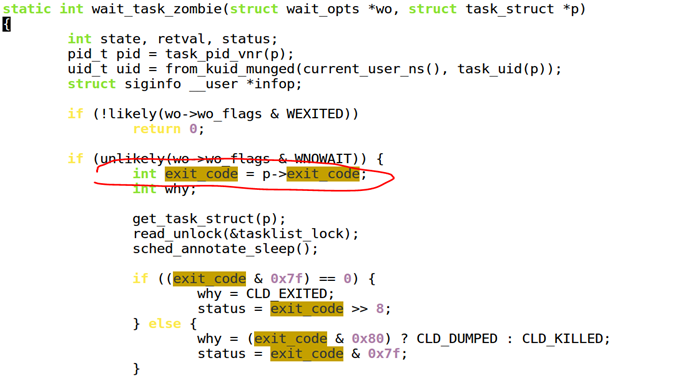


### 3.1.什么是僵尸进程：

子进程死掉，父进程还没清理尸体，没火化。



子进程死亡后，首先变成僵尸， mm,fs等所有资源已经释放，只剩task\_struct躯壳还没被父进程清理,父进程通过wait\_pid获得，wait结束，僵尸所有资源(task\_struct)被释放。



父进程用waitpid()查看task\_struct退出码，检测子进程退出状态；

Waitpid()调用完成，则子进程彻底消失。

僵尸进程状态：子进程死亡，还没被父进程清理，资源已经被释放，只剩下task\_struct。

清理办法：kill父进程。

僵尸进程被杀死的假象：

当一个进程里有多个子线程，主线程退出其他线程仍然运行；

top是以进程视角看线程，所以造成僵尸进程亦然可以被杀的假象；

### **3.2 stop状态**：

其他进程控制其停止

Ctrl+z：让进程暂停；发信号19

Fg:进程在前台继续跑；

Bg:让进程在后台继续跑

Fg/bg实际上是发continue信号18

用于作业控制。

Kill –l

cpulimit -l 20 -p 3637

限制CPU占用率为20，实际用ctrl+z fg/bg实现的。

cpulimit是暴力的方法，更好的用CGROUP

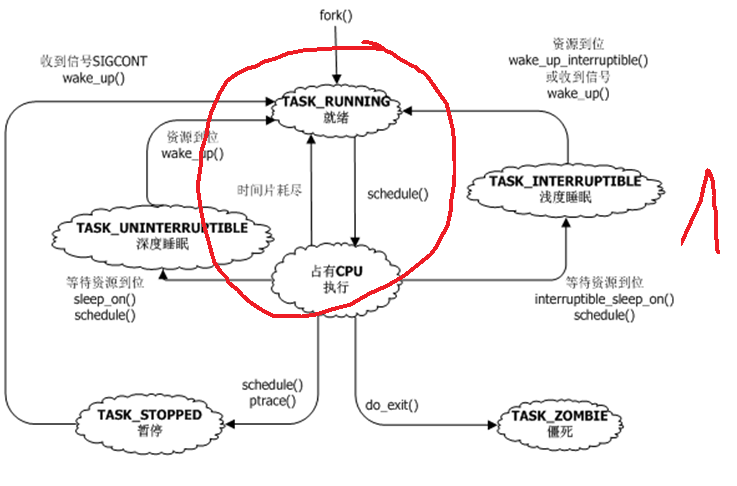
### 3.3 睡眠

进程本身主动睡眠，等待资源，深度睡眠/浅度睡眠。

### 3.4调度：

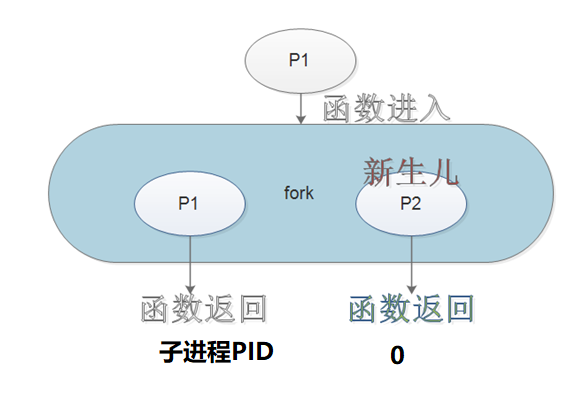
只管在就绪态和运行态进程的切换，一个运行态，多个就绪态。

调度进程只等CPU，其他资源全部就绪。Linux就绪和占用都是用task\_running标识符



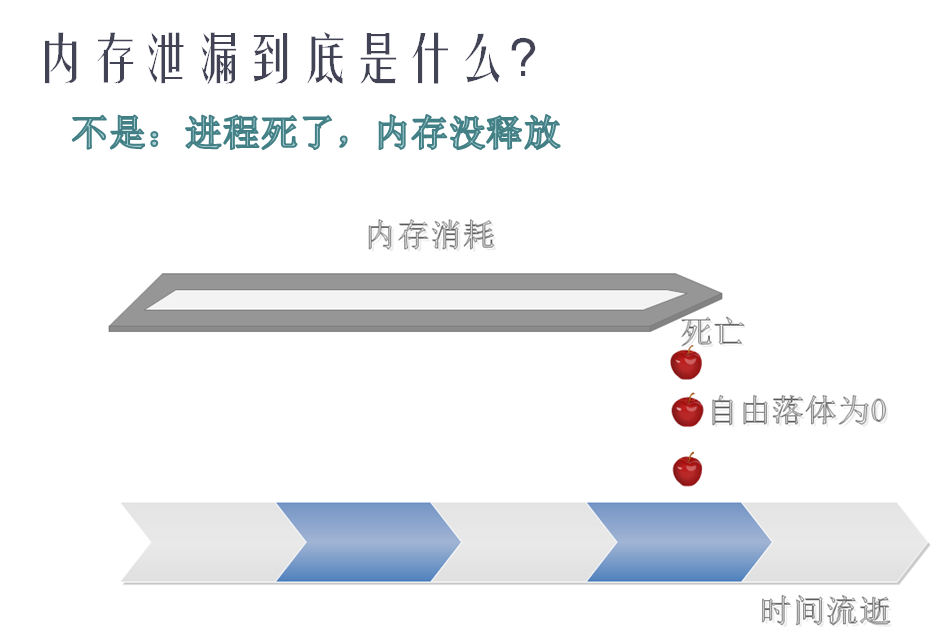
## 4.fork

创建进程，子进程拷贝父进程的task\_struct资源。



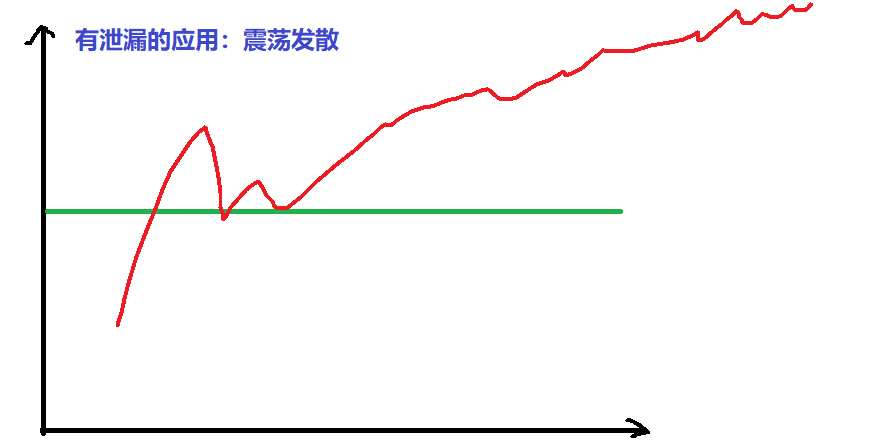
## 5.什么是内存泄漏？

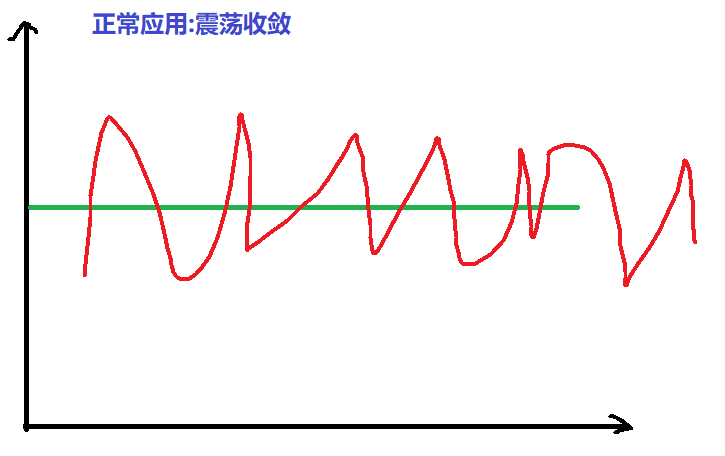
进程没死，运行越久，耗费内存越多，叫内存泄漏（程序死亡时，所占内存会全部释放）；



判断内存泄漏方法：连续多点采样(6，7，8，9每小时采样，统计剩余内存是否收敛)，正常的程序，内存震荡收敛。随时间增加，内存消耗不断增多，且不收敛，则一定是内存泄漏；

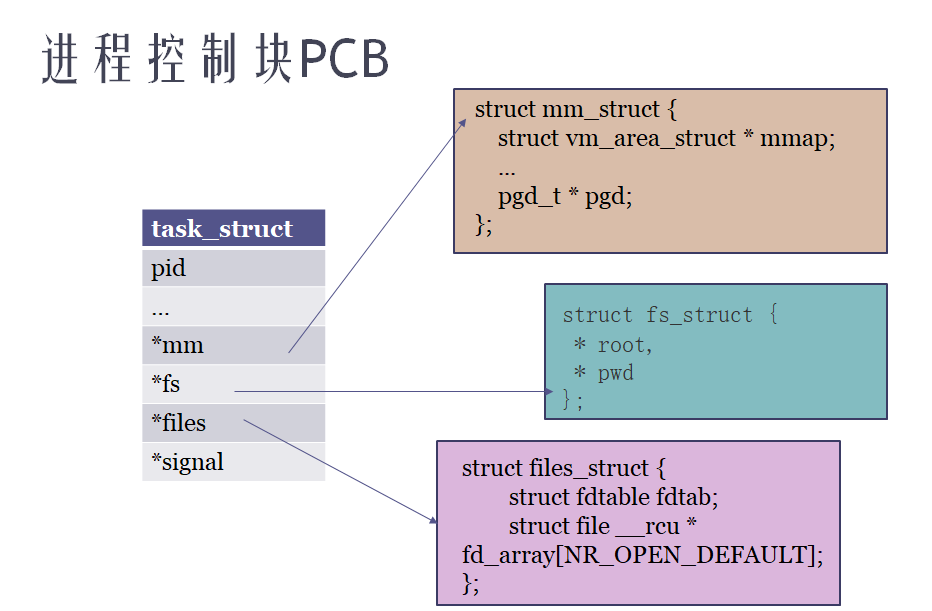
发散



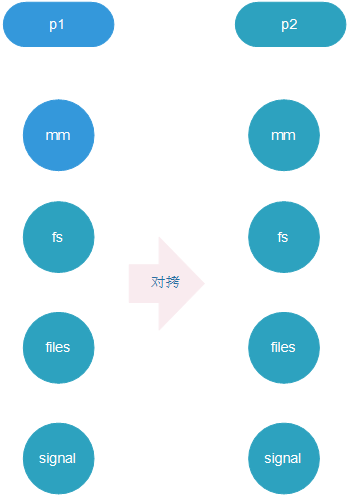


# 第二部分 进程和线程的本质

1.进程拥有资源mm,fs,files,signal…



Fork创建一个新进程，也需要创建task\_struct所有资源；实际上创建一个新进程之初，子进程完全拷贝父进程资源，如下图示：



比如fs结构体：



子进程会拷贝一份fs\_struct，

\*p2\_fs = \*p1\_fs;

pwd路径和root路径与父进程相同，子进程修改当前路径，就会修改其p2\_fs->pwd值；父进程修改当前路径，修改p1\_fs->pwd;

其他资源大体与fs类似，最复杂的是mm拷贝，需借助MMU来完成拷贝；即写时拷贝技术：

## 2.写时拷贝技术：

#include <sched.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

int data = 10;

int child\_process()

{

printf("Child process %d, data %d\n",getpid(),data);

data = 20;

printf("Child process %d, data %d\n",getpid(),data);

\_exit(0);

}

int main(int argc, char\* argv[])

{

int pid;

pid = fork();

if(pid==0) {

child\_process();

}

else{

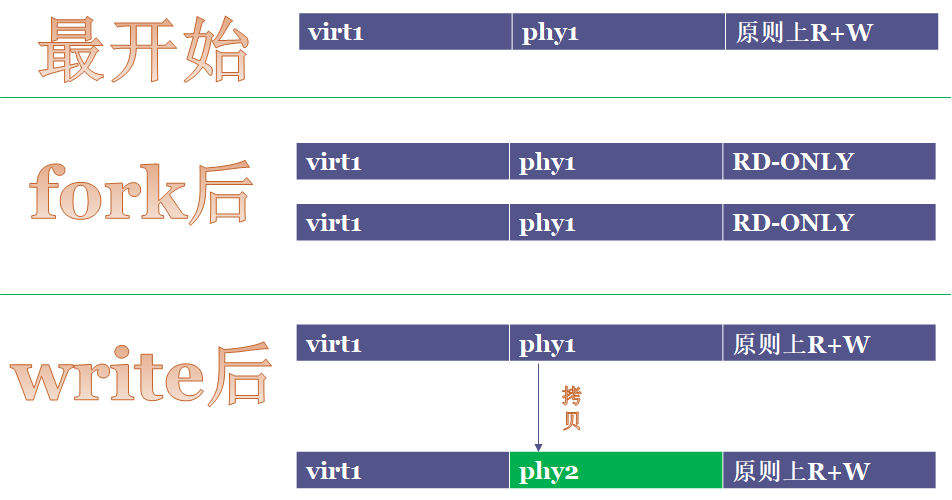
sleep(1);

printf("Parent process %d, data %d\n",getpid(), data);

exit(0);

}

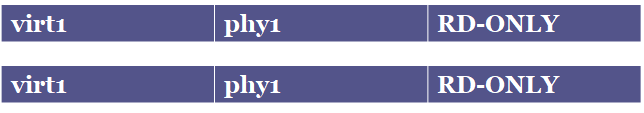
}



**第一阶段**：只有一个进程P1，数据段可读可写：



**第二阶段**，调用fork之后创建子进程P2，P2完全拷贝一份P1的mm\_struct，其指针指向相同地址，即P1/P2虚拟地址，物理地址完全相同，但该内存的页表地址变为只读；



**第三阶段**：当P2改写data时，子进程改写只读内存，会引起内存缺页中断，在ISR中申请一片新内存，通常是4K，把P1进程的data拷贝到这4K新内存。再修改页表，改变虚实地址转换关系，使物理地址指向新申请的4K，这样子进程P2就得到新的4K内存，并修改权限为可读写，然后从中断返回到P2进程写data才会成功。整个过程虚拟地址不变，对应用程序员来说，感觉不到地址变化。

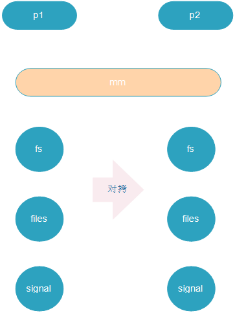
谁先写，谁申请新物理内存；

Data=20；这句代码经过了赋值无写权限，引起缺页中断，申请内存，修改页表，拷贝数据…回到data=20再次赋值，所以整个执行时间会很长。

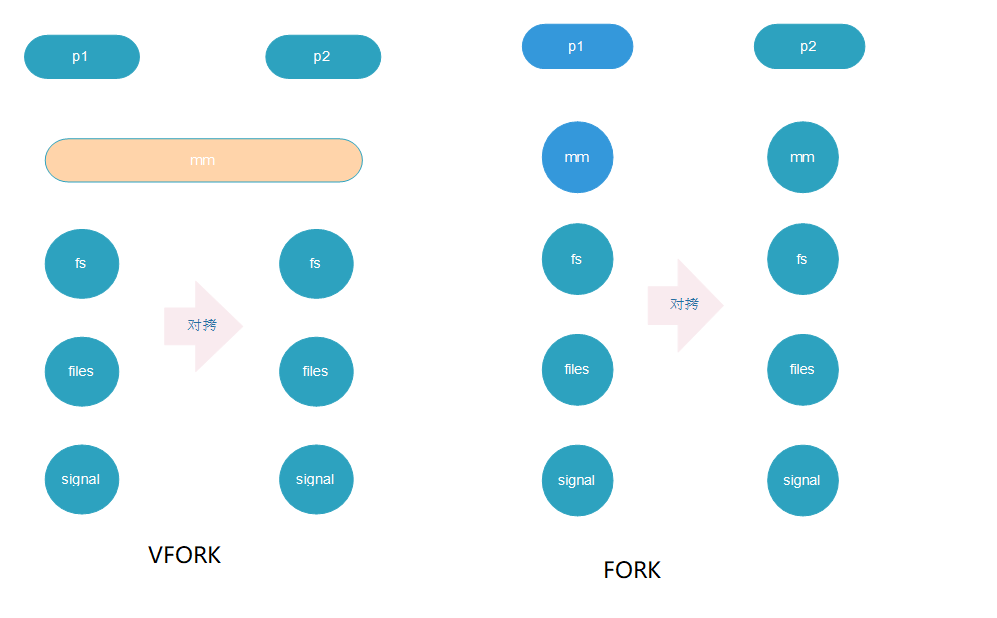
这就是linux中的写时拷贝技术(copy on write), 谁先写谁申请新内存，没有优先顺序；

cow依赖硬件MMU实现，没有MMU的系统就没法实现cow，也就不支持fork函数,只有vfork;

## 3. vfork的 mm指针直接指向父进程mm；



除了mm共享，其他资源全都拷贝一份，而fork是所有资源都对拷一份，对比如下图



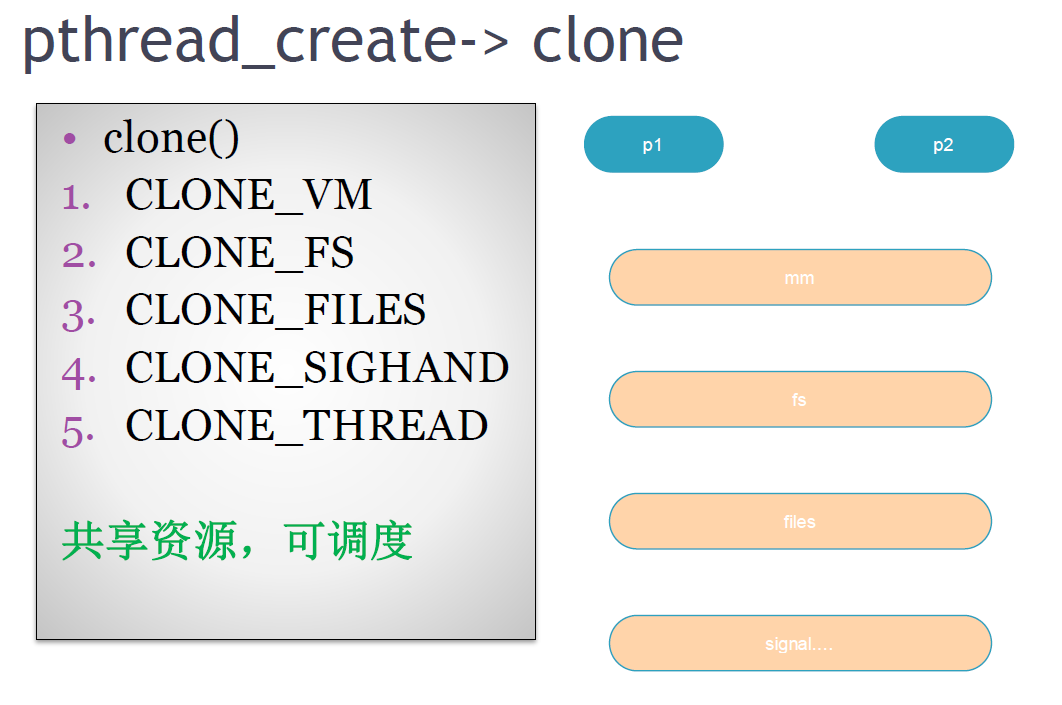
不同点：vfork会阻塞：

vfork后，父进程会阻塞，直到子进程调用exit()或exec，否则父进程一直阻塞不执行；

上面代码改用vfork，打印输出10，20，20

## 4.线程

clone函数创建一个新进程，不执行任何拷贝，所有资源都等同vfork中的mm共享，task\_struct里只有指针指向父进程task\_struct；



也就是子进程与父进程完全共享资源，但是又可以被独立调度，实际上这就是linux中的线程本质；

pthread\_create()函数就是调用clone()函数(带有clone\_flags)创建新task\_struct，其内部mm,fs等指针全都指向父进程task\_struct；

Linux中创建进程(fork，vfork)和线程(pthread\_create)，在内核都是调用do\_fork()-->clone()，参数clone\_flags标记表明哪些资源是需要克隆的，创建线程时，所有资源都克隆；

从调度的角度理解线程，从资源角度来理解进程，内核里只要是task\_struct，就可以被调度；linux中的线程又叫轻量级进程lwp;

ret = pthread\_create(&tid1, NULL, thread\_fun, NULL);

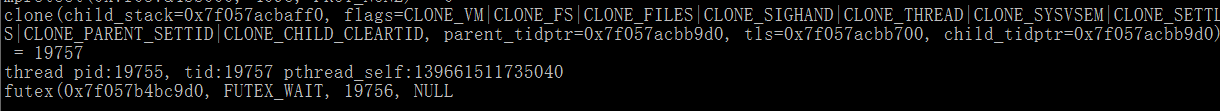
if (ret == -1) {

perror("cannot create new thread");

return -1;

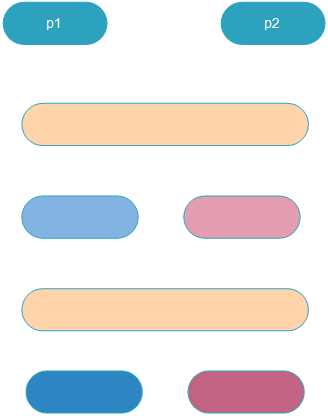
}

strace ./a.out



## 5.人妖

如上述，资源全部共享是线程，不共享是进程；那假如修改clone函数中的clone\_flags，使共享其中部分资源，如下图示：



这时候创建的既不是进程也不是线程，妖有了仁慈的心,就不再是妖,是人妖；

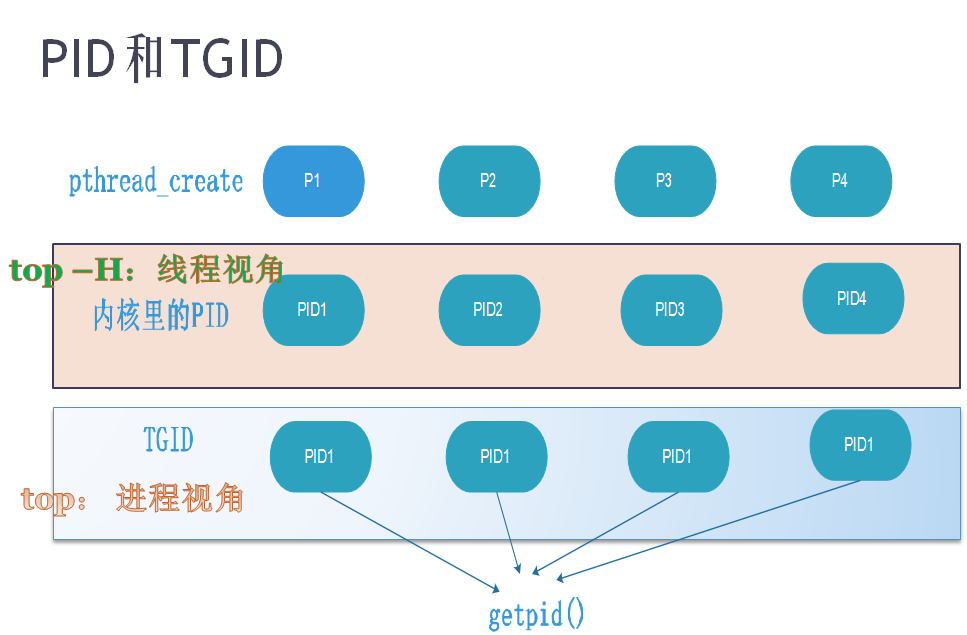
Linux是可以调用clone创建人妖的，不过没实际必要～

## 6.PID

Linux 的每个线程都会创建task\_struct，会有个独立的PID；

POSIX标准规定，在多线程中调用getpid()应该获得相同的PID；

为兼容POSIX标准，linux增加了一层TGID, 调用getpid()实际上是去TGID层获取PID，TGID中PID均相同，保留了线程在内核中不同的PID，如下图所示：



top命令看到的是进程TGID，所有线程相同；

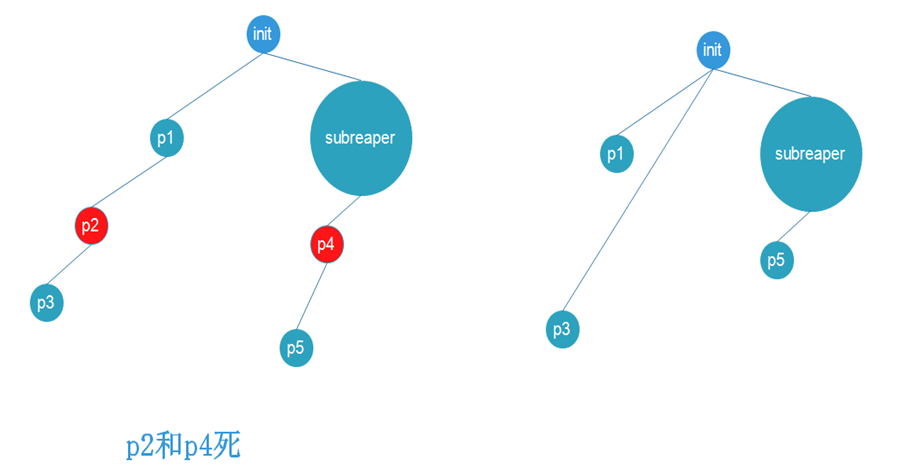
top –H命令是从线程视角，此时的PID是task\_struct中实际的PID;

## 7.进程死亡：

7.1子进程先死亡，父进程去清理，所谓白发人送黑发人，不清理则变成僵尸进程；

7.2 若父进程先死，子进程变成孤儿，一般托付给init，新版linux3.4引入subreaper，可以托付给中间进程subreaper。

父进程先死亡后，子进程沿tree向上找最近的subreaper挂靠，找不到subreaper，就挂在init。



/\* Become reaper of our children \*/

if (prctl(PR\_SET\_CHILD\_SUBREAPER, 1) < 0) {

log\_warning("Failed to make us a subreaper: %m");

if (errno == EINVAL)

log\_info("Perhaps the kernel version is too old (< 3.4?)");

}

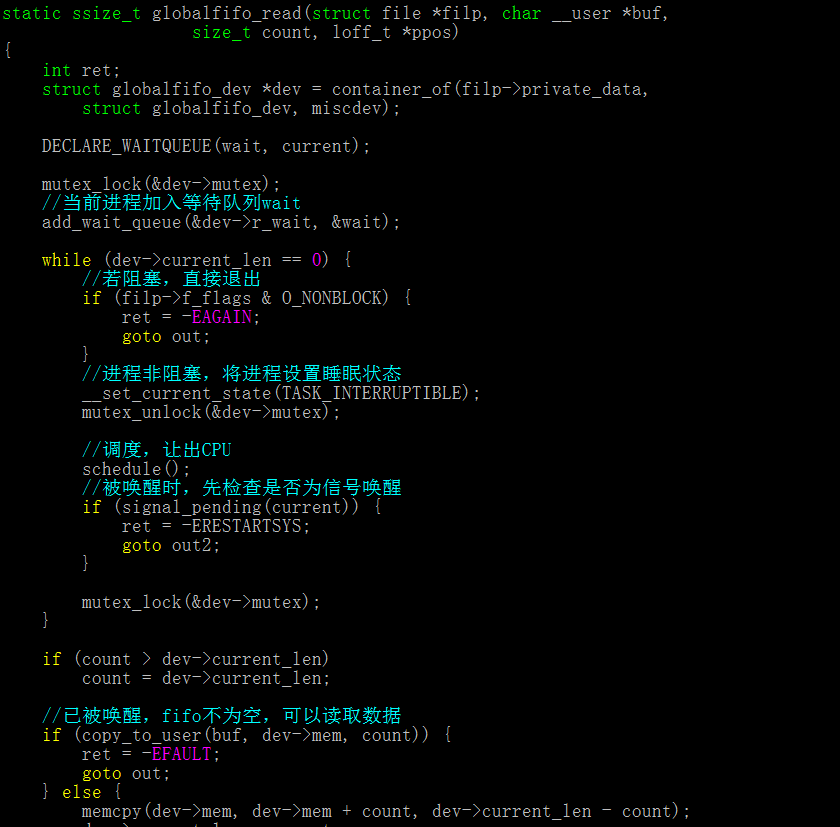
PR\_SET\_CHILD\_SUBREAPER设置为非零值，当前进程就会变成subreaper，会像1号进程那样收养孤儿进程；

## 8.睡眠

当进程需要等待硬件I/O资源的时候，可以设置为睡眠状态，一般驱动做成浅度睡眠，硬盘等资源会置入深度睡眠(不会被信号唤醒)；

睡眠是把task\_struct挂在wait queue上，比如多个进程都在等待串口，当串口可用时，唤醒等待队列上所有进程；

以下为《linux设备驱动开发详解》中案例注释

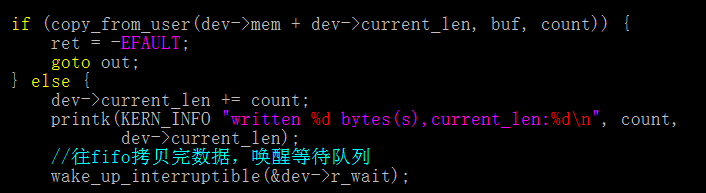


注：上图有个错误，while循环中，应该为

“若非阻塞，直接退出”；

“进程阻塞，将进程设置睡眠状态”;

当读取fifo为空即dev->current\_len==0时，将进程加入等待队列睡眠，schedule()让出CPU, fifo中写入数据时将等待队列唤醒，此函数中schedule()处继续执行；唤醒动作在write函数中执行；



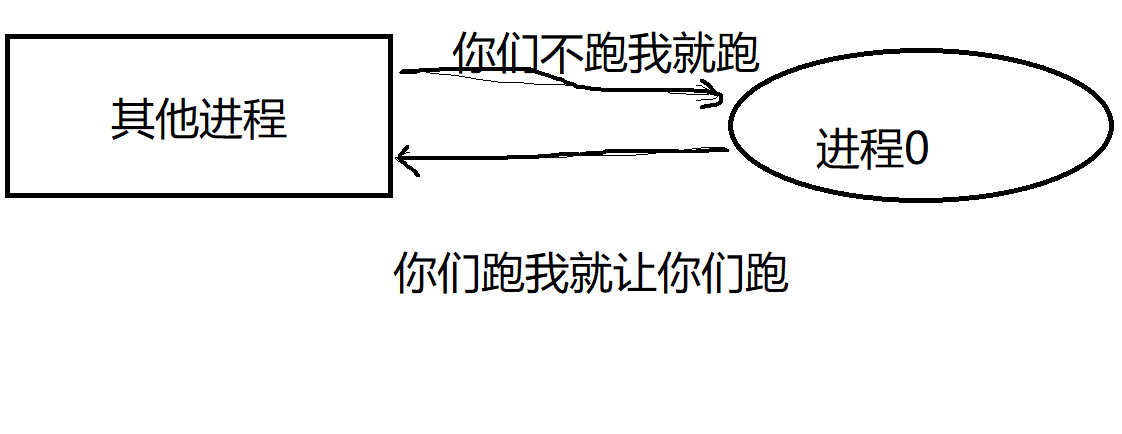
唤醒后检查唤醒原因，若为IO唤醒，正常读取数据；若为信号唤醒，直接退出；

## 9. 0进程

0进程是唯一没有通过fork() 创建的进程，是系统中所有其它用户进程的祖先进程，其创建1号进程(init进程后)，退化为idle进程，也叫swapper进程；

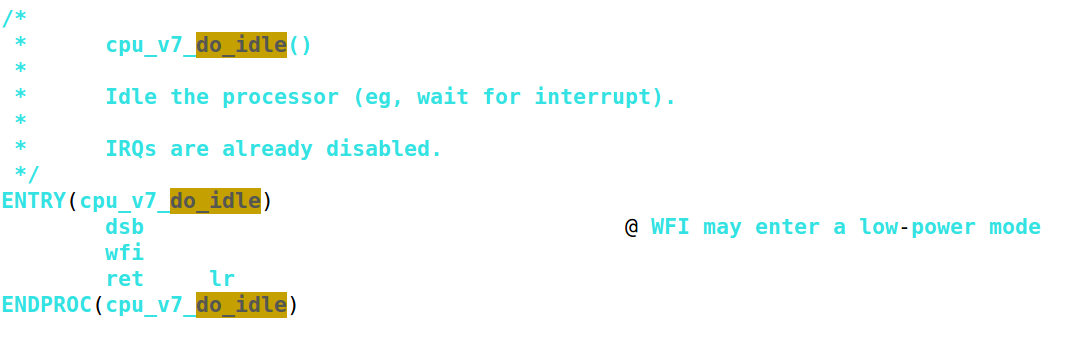
top命令中的id时间即为idle进程运行时间；

idle进程：优先级是最低的，当系统中没有任何进程运行时，即执行idle进程，idle将CPU置入低功耗模式，有任何其他进程被唤醒，idle即让出CPU；



idle进程的设计，实际上是将“跑”与“不跑”的问题，统一为“跑”的问题。极巧妙的简化了系统设计，降低进程之间的耦合度。(将检查系统是否空闲，设置CPU低功耗模式的功能放在idle实现，其他进程都不用关心CPU工作模式)

ARM版本实现如下：



wfi ==> wait for interrupt

对于用户空间来说，进程的鼻祖是init进程，所有用户空间的进程都由init进程创建和派生，只有init进程才会设置SIGNAL\_UNKILLABLE标志位。

如果init进程或者容器init进程要使用CLONE\_PARENT创建兄弟进程，那么该进程无法由init回收，父进程idle进程也无能为力，因此会变成僵尸进程(zombie)。

# 第三部分 进程调度

## 1. 吞吐率和响应

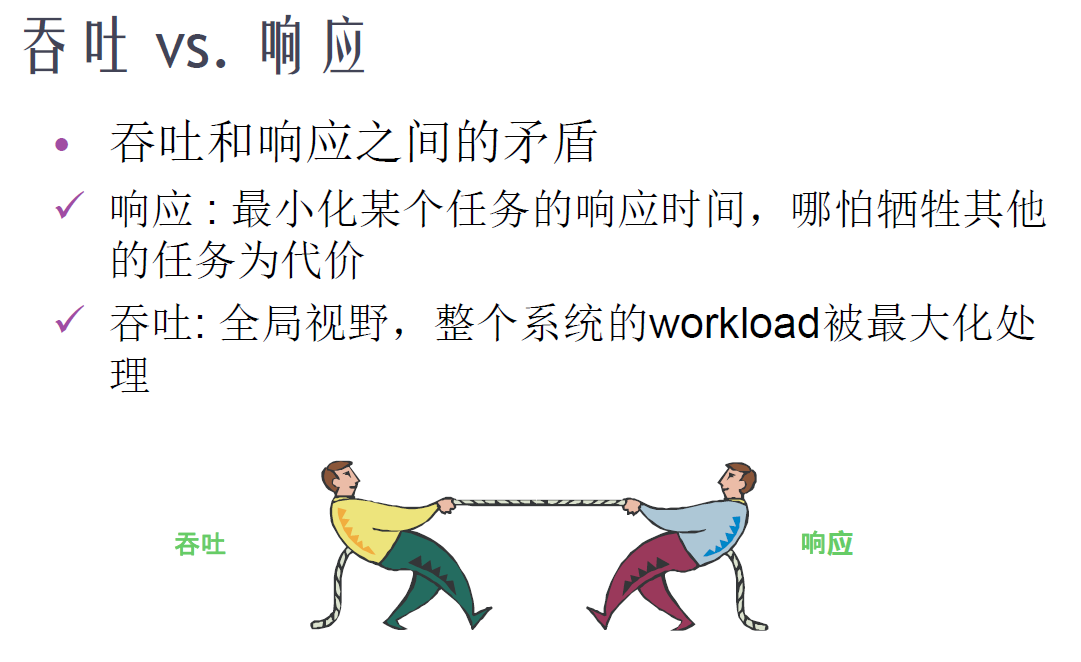
**吞吐**：单位时间内做的有用功；

**响应**：低延迟。

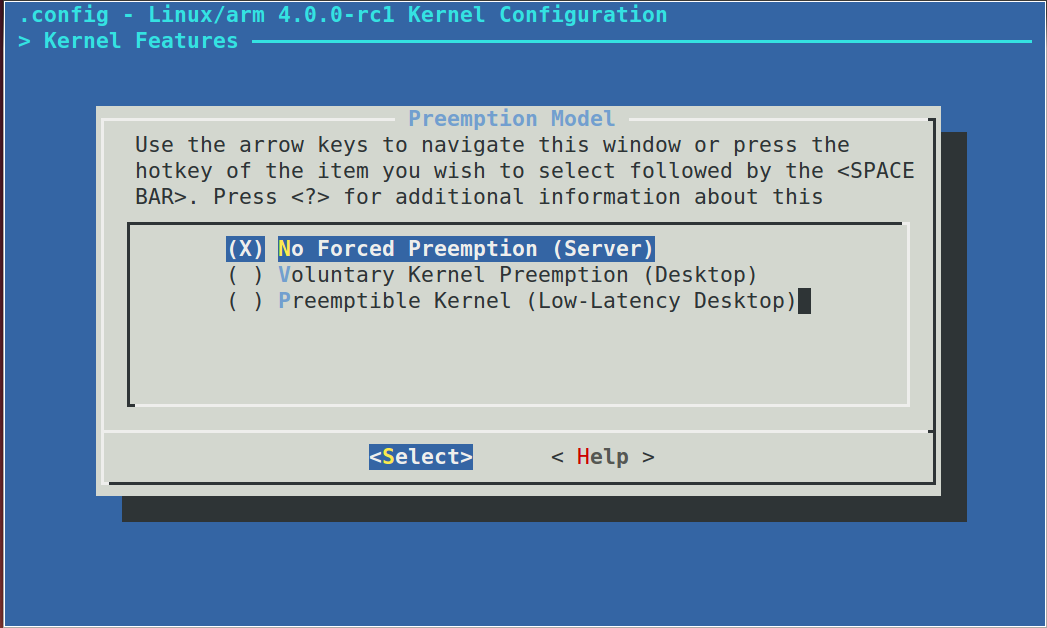
吞吐追求的整个系统CPU做有用功，响应追求的是某个特定任务的延迟低；

1GHZ的CPU切换线程保存恢复现场约几个微妙级别，看似消耗不了太多时间，但是由于系统的局部性原理，会保存当前线程数据的缓存，切换线程会打乱局部性引起cache miss，而CPU访问cache速度远大于内存访问，这样综合看来上下文切换花销还是很大的。无用功占用较多CPU；

所以追求吞吐量和低延迟，这两个目标是矛盾的



编译内核选项有如下

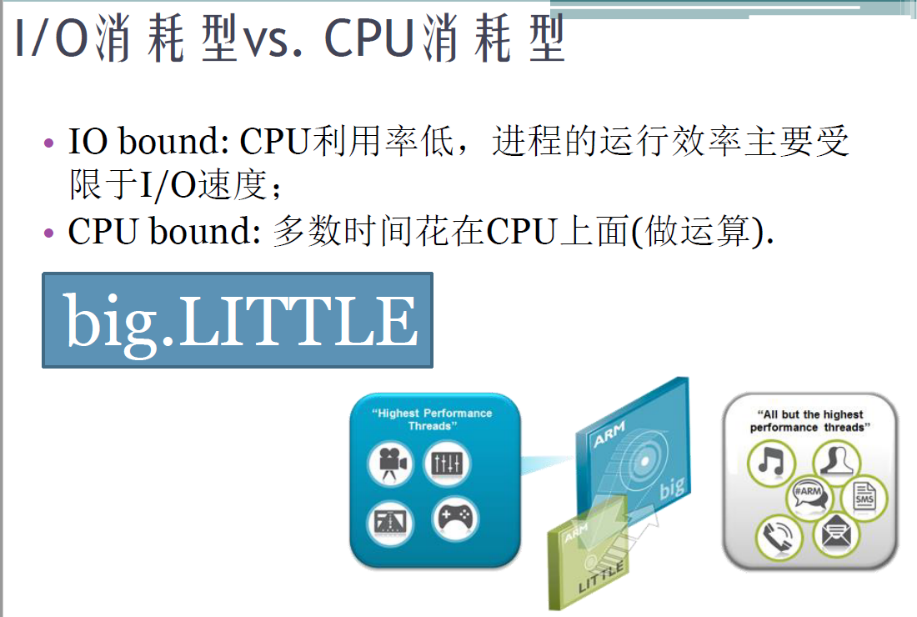


服务器版追求吞吐量，配置为不抢占；桌面版或手机更追求响应，配置为低延迟；

调度器一般讲的是最后一种，低延迟抢占；

## 2. Linux任务类型

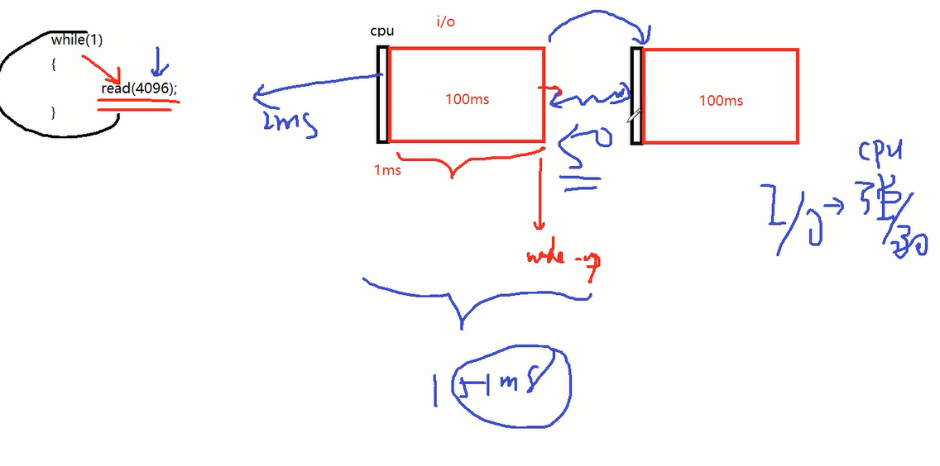
问题2: 一个典型的系统内任务分两种：CPU消耗型和IO消耗型



**CPU型的任务**，通常要求高性能，但是不追求低延迟，优先级较低；

**IO型的任务**，通常优先级更高，**追求低延迟**，对CPU性能不敏感；

如下图，一个读IO循环过程，占用CPU时间极短（1ms），IO占用100ms ，若CPU性能降一倍，执行CPU占用2ms对整体时间影响不大(总时间102ms)；但是如果CPU不能及时响应，一个读写周期响应延迟100ms，那整体时间变为约200ms，整体性能降低一半；



所以，IO型任务只关注响应速度，对CPU性能不敏感；

基于此原理ARM公司设计了big.LITTLE架构，比如在手机上有8个核，设计为四大核，四小核；大小核指令集完全兼容，调度器调配IO型的任务跑在小核上(对CPU弱不敏感)，CPU型任务放在大核上跑。这样用4+1(四小核等效)核的功耗实现了8核的性能；

调度器要在吞吐和延迟之间找到某个均衡；

## 3任务调度

1调度原理：

涉及两个概念，**策略**和**优先级**；

内核所有进程优先级为0~139之间；

0~99：采用**RT策略**，常用的有SCHED\_FIFO, SCHED\_RR;

SCHED\_FIFO, 属于霸占型的，高优先级执行完，才会执行低优先级；

SCHED\_RR, 不同优先级与CHED\_FIFO相同，属于霸占型，同等优先级轮转；

比如有4个进程：P1\_FIFO\_3, P2\_CHED\_RR\_2, P3\_CHED\_RR\_2, P4\_FIFO\_4

执行过程，P2/P3轮转执行完之后，执行P1, 最后执行P4。

100~139：普通线程，采用非RT策略，对应nice(-20, 19).

这里所有优先级都是可以抢占的，但nice 值越低，分配CPU资源越多。

2.所有调度，SCHER\_FIFO/SCHER\_RR或设置nice都是针对task\_struct；

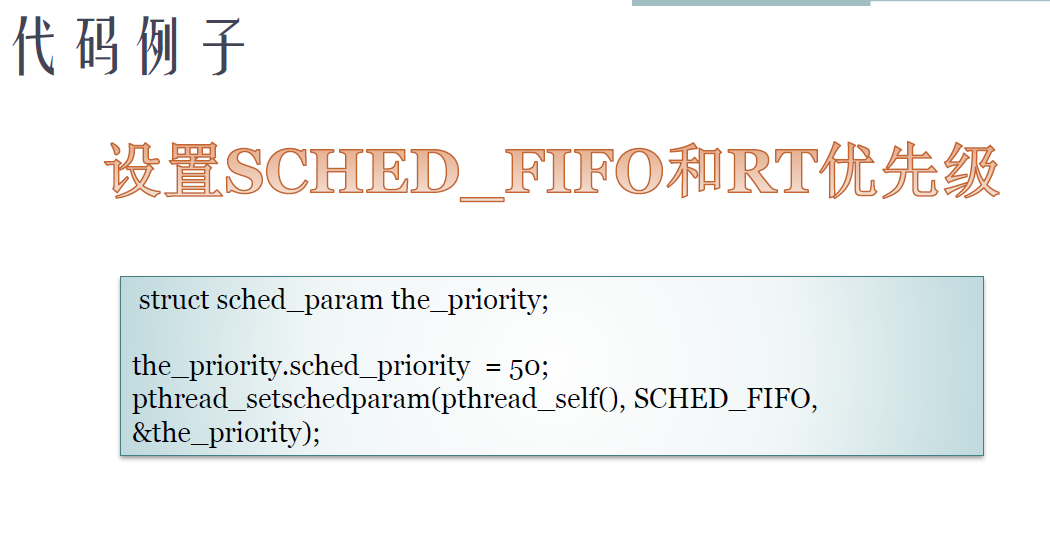
3.将进程设置为FIFO策略:

修改进程25020的策略为FIFO, 优先级50

sudo chrt –f –a –p 50 25020

这样进程是按FIFO策略调度，占有CPU最高为80%(普通进程可以接近100%)，CPU占有率降低(但不可抢占)，此时IO延迟反而变大，鼠标操作变慢；

设置FIFO线程api



内核里优先级=99-50=49

内核态数字越低，优先级越高；用户态相反；

4.每个task\_struct的nice都可以单独设置(所有普通线程)；

Nice值设置都是指普通线程，RT策略不支持nice；

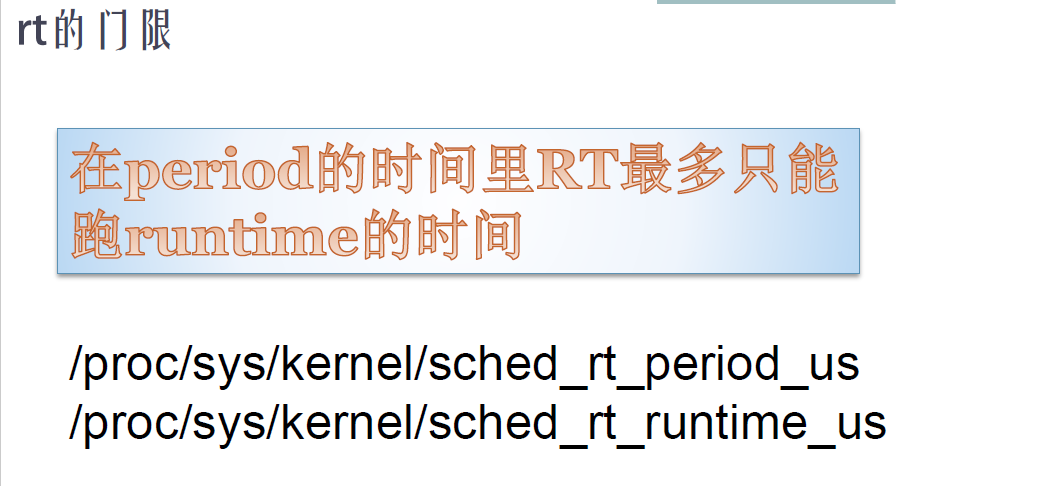


nice()默认是0

调度，在不同nice值进程间轮转，

2.6早期内核对进程采取奖励和惩罚算法，越睡眠越奖励，越消耗CPU，越惩罚，动态调整nice值，实现极其复杂，后来升级两个补丁；

**补丁1：RT熔断机制，设置rt门限值**



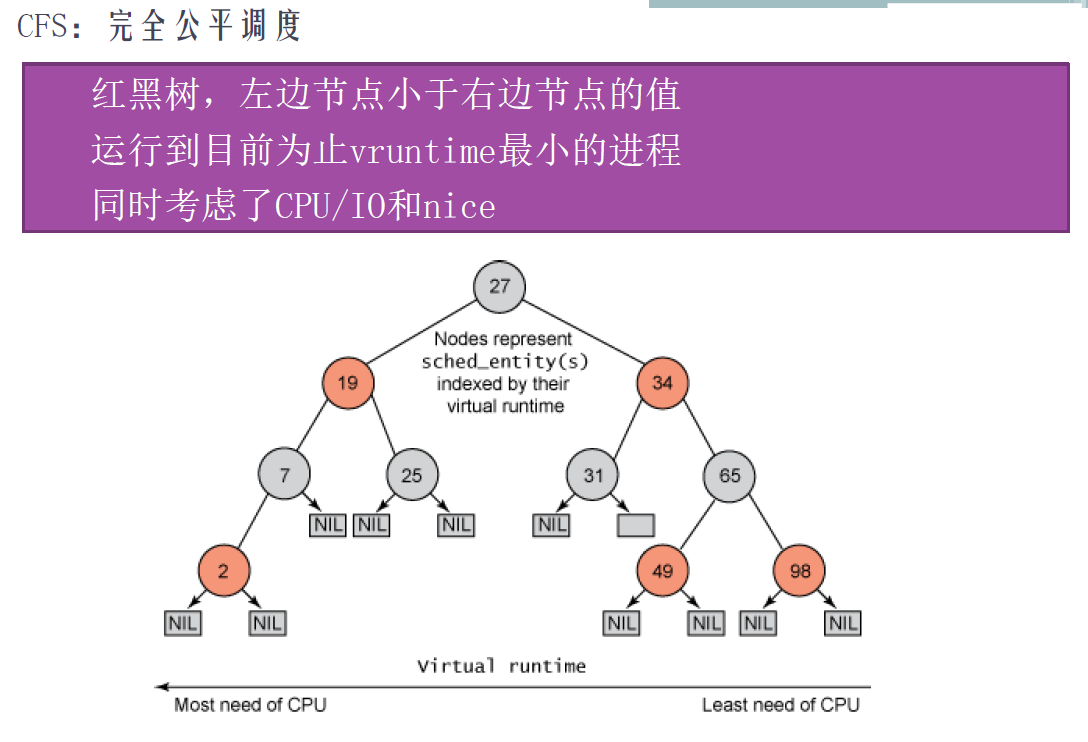
默认runtime 0.95s，period 1s,1s内RT最多跑了0.95s自动熔断；

$cd /proc/sys/kernel

sudo sh -c 'echo 800000 > sched\_rt\_runtime\_us'

设置RR/FIFO策略熔断最高位800ms，RR/FIFO策略进程最多占用CPU 80%。(默认是95%)

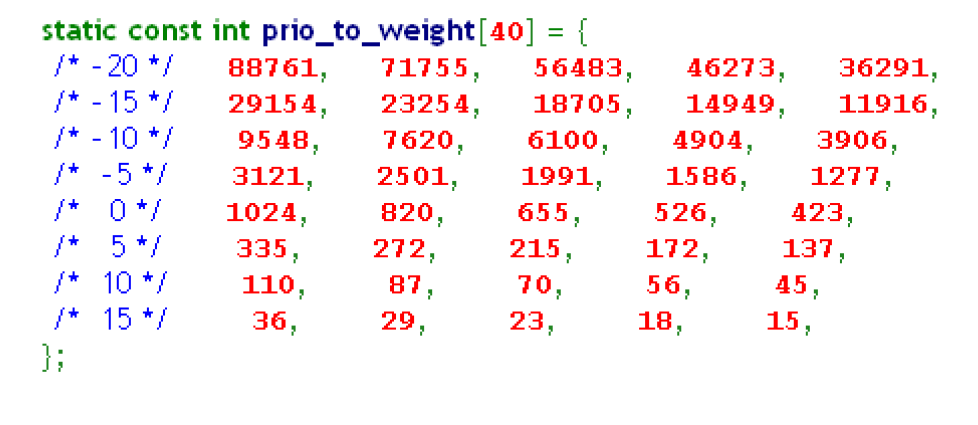
**补丁2：普通进程调度算法CFS（complete fare schedule）**



虚拟运行时间，vtime = ptime \* 1024/weight

ptime：物理时间

weight:权重参数



nice=0，虚拟时间等于物理时间；

nice值越小，对应weight分母越大，vtime增长越慢，实际对应ptime占用越多；

vtime机制很好的平衡了I/O型，CPU型任务；

I/O型喜欢睡眠，ptime比较小，所以vtime自然较小，会偏向于挂在树左边；

同理，优先级越高的CPU型，其weight值越大，vtime也会越小，亦偏向挂载树左边；

即CFS用很简单的方式实现了历史上复杂的睡眠补偿，消耗惩罚，动态调整等功能；

修改进程的nice值：

sudo renice -n -5 -g 24856 //24856进程的所有线程nice都设置为-5

**综上，linux调度算法过程：**

1.首先执行SCHECH\_RR/SCHECH\_FIFO进程，待他们执行到睡眠或者熔断，CPU切换到普通线程；

2.普通线程按CFS算法调度，在普通线程间轮转；

线程调度优先与线程是否在内核态无关，只由优先级和策略决定。用户态内核态只涉及权限问题；

# 第四部分 负载均衡

## 1.线程的负载均衡

对task\_struct做负载均衡；

分布式系统中，linux的每个核都自动以劳动为乐，(共产主义社会)。

**分别对RT任务和普通线程做负载均衡**：

RT任务：将n个优先级最高的线程自动分配到n个核；

pull\_rt\_task()

push\_rt\_task()

普通任务：

周期性负载均衡，在时钟tick会检查哪个核空闲，优先使空闲核工作(从负载重的核pull任务，或push任务给空闲核，每个CPU以劳动为乐)；

idle时负载均衡；某个核进入idle状态，会主动pull任务执行；

fork和exec时负载均衡；创建的新进程，会放到最闲的核去跑；

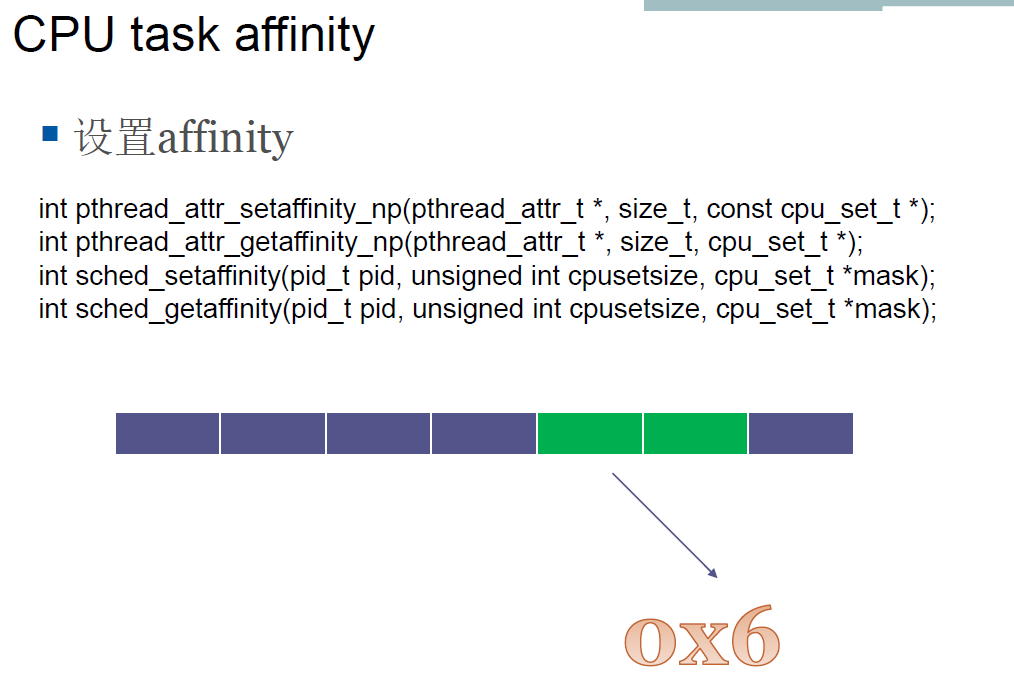
**软亲和性（affinity）：**Linux 内核进程调度器天生就具有被称为软 CPU 亲和性（affinity）的特性，这意味着进程通常不会在处理器之间频繁迁移。这种状态正是我们希望的，因为进程迁移的频率小就意味着产生的负载小。

**硬亲和性（affinity）：**就是利用linux内核提供给用户的API，强行将进程或者线程绑定到某一个指定的cpu核运行。

需要绑定CPU的原因：

1.提高cache命中率；

2.在一些非对称CPU，比如NUMA中，防止一个CPU去另外一个插槽的内存读写数据；

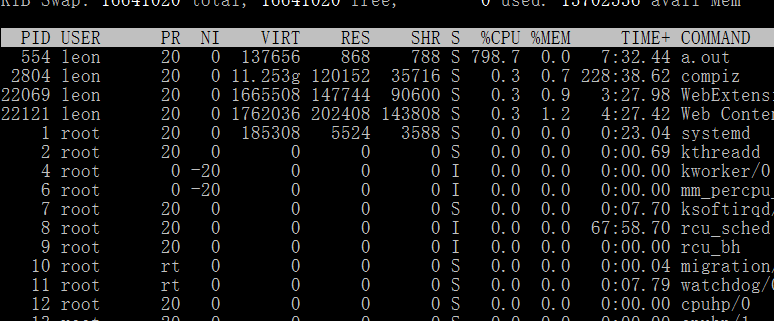


也可以用taskset工具设置：-a 所有线程

pidof ./a.out

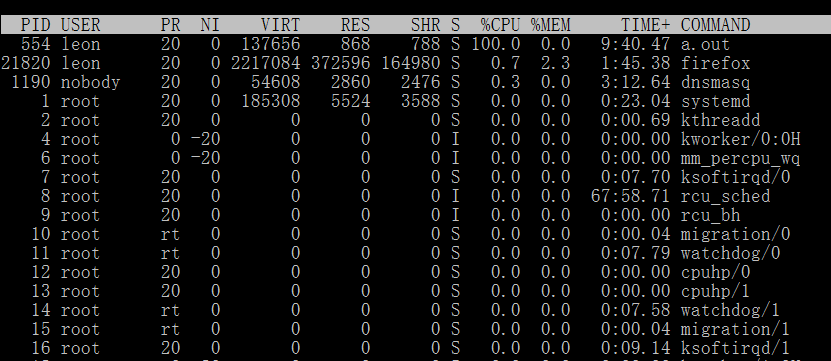


进程a.out占用CPU800%(八核)



设置只在2号核跑

taskset -a -p 02 554



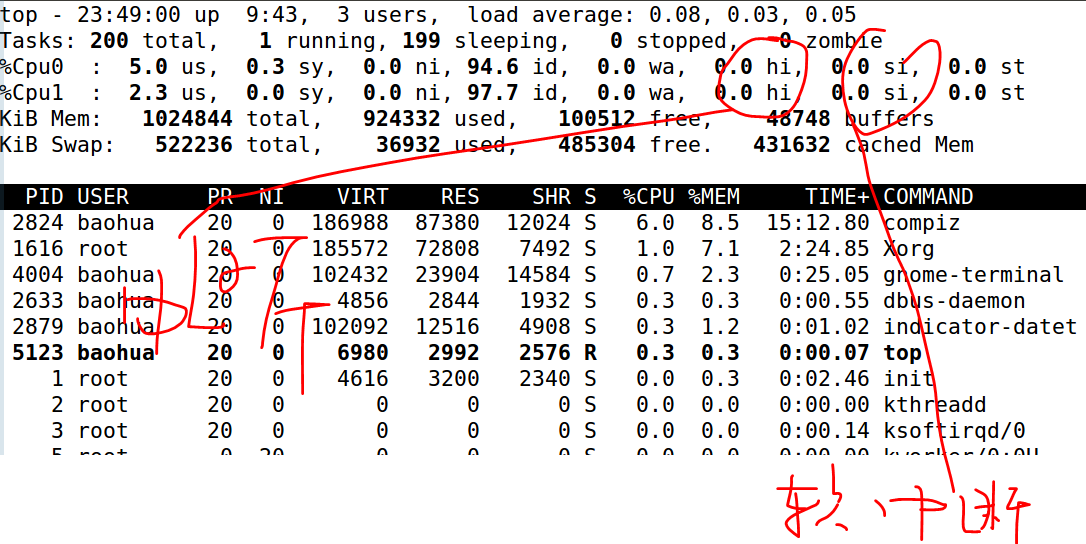
CPU占有率降至100%，所有线程只在2号核上跑；

## 2.中断负载均衡

负载除了进程还有中断，执行完中断，才会去执行task\_struct任务；

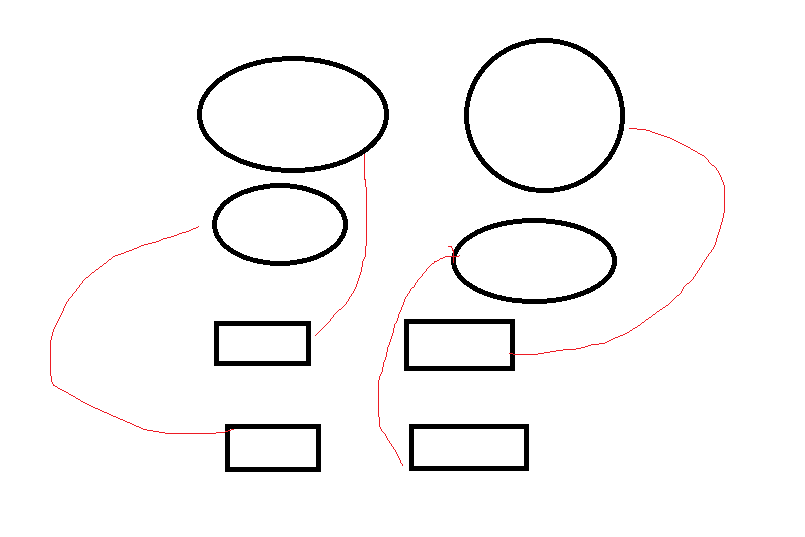
硬件中断一般比较短(linux2.6.34后内核不支持中断嵌套)，比如网卡收包，在硬中断接收完包，会调用软中断(处理tcp/ip包，软中断可以嵌套)，软中断结束后，会调用vtime最小的线程；

top命令的hi表示硬中断时间(isr，屏蔽中断)，si表示软中断，

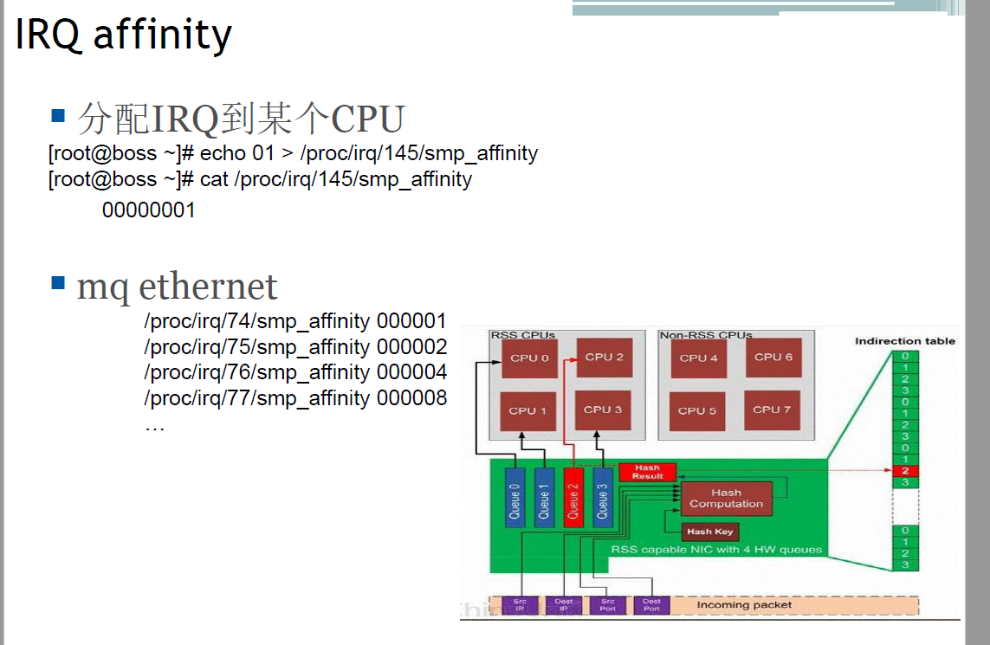


当网络流量大的时候，CPU花在硬中断和软中断时间比较多，这时候需要做中断的负载均衡；

现在新网卡一般有多个队列，假如在一个4核的系统，网卡有4个队列，每个队列可以单独产生中断，硬件支持负载均衡；那么可以将一个队列绑定到一个核，这样，所有CPU都会参与网卡发送包服务；



设定方法：将每个中断号，分别设置affinity，绑定到指定CPU

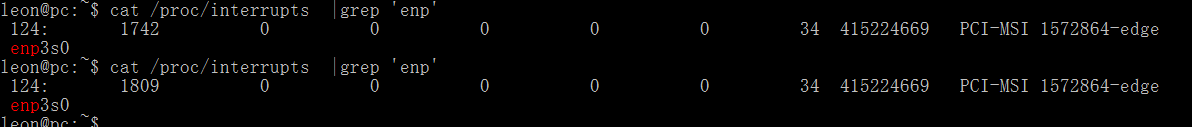


cat /proc/interrupts |grep 'enp'

124: 0 0 0 0 0 0 0 34 415207449 PCI-MSI 1572864-edge enp3s0



sudo sh -c "echo 3 > /proc/irq/124/smp\_affinity"



有的网卡只有一个队列：单个核抛出的软中断只能在这个核跑，那么一个队列抛出的软中断(tcp/ip层处理)只能在这一个核执行，其他核会空闲；

测试：

客户端：$ echo "不要回答！不要回答！不要回答！" | nc 10.10.100.16 1235

服务端：$ nc -l -p 1235

### RPS补丁

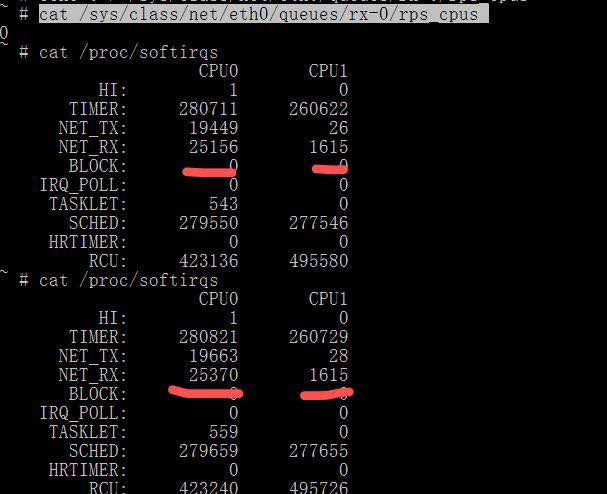
Google推出了rps补丁，可以把TCP/IP协议栈的处理，均衡到多个核上去，这个技术叫RPS；

比如我手头板子是单个网卡队列收包，默认是单核上执行

#cat /sys/class/net/eth0/queues/rx-0/rps\_cpus

0

收包只在CPU0上执行



nc -l -p 1235

监听1235端口收到数据时，只有CPU0继续增加；

**多核间的softIRQ scaling**

设置rps在CPU0/1上均衡

echo 3 > /sys/class/net/eth0/queues/rx-0/rps\_cpus

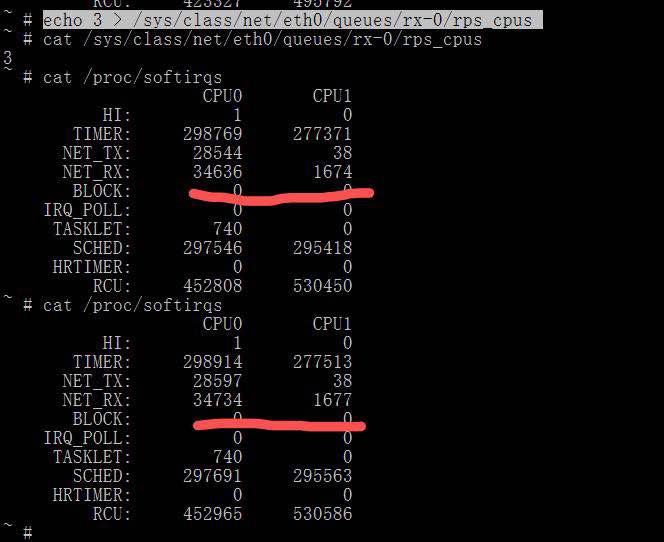
watch –d “cat /proc/softirqs | grep NET\_RX”

/proc/interrupt

/proc/softirq

分别查看硬中断和软中断次数

top +1显示多核的负载



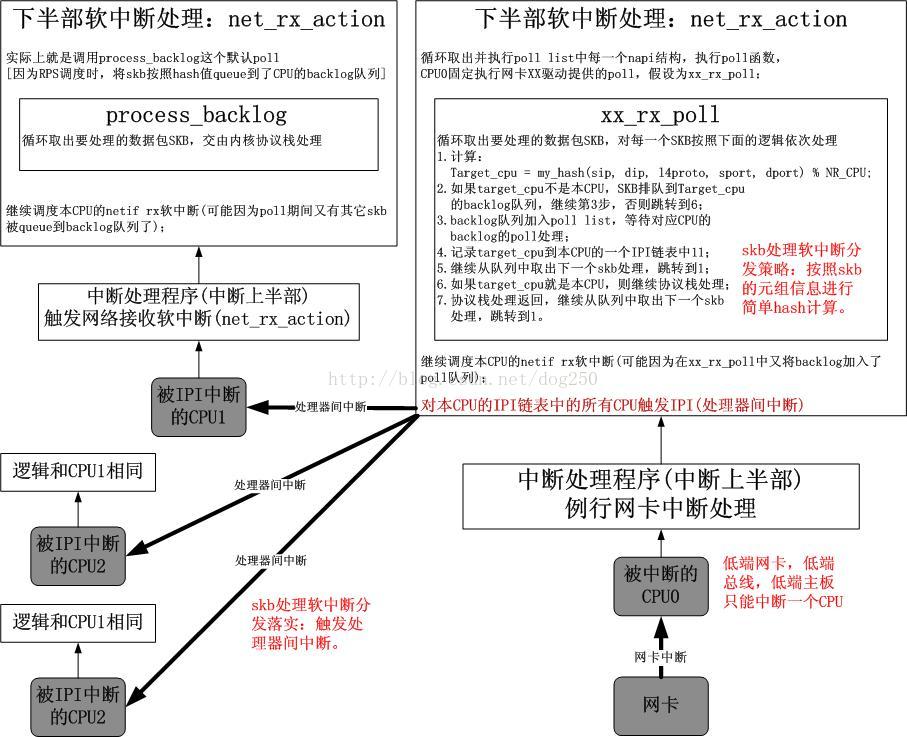
由上可以看到CPU0/1都开始处理网络接收包

**经实验得出结论**：

1. 硬中断绑定的CPU核，一定会响应softirqs，不管对应rps\_cpus位设置与否；

2.当rps\_cpus位设置时，对应CPU核会响应softirqs；

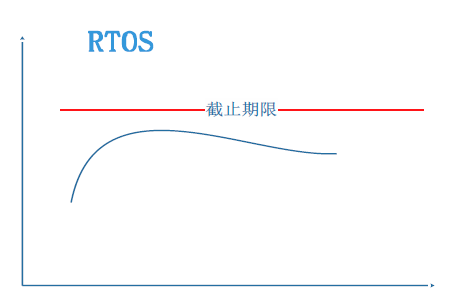
RPS原理：由单一CPU核响应硬件中断，将大量网络接收包，通过多核间中断方式分发给其他空闲核，如下网络图清晰说明情况

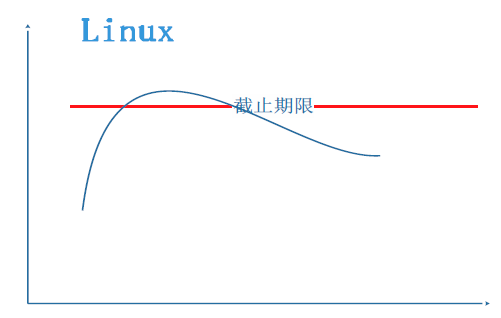


上图引用自<https://blog.csdn.net/dog250/article/details/46705657>

## 3.linux不是硬实时系统：

硬实时：满足可预期，非一定越快越好，强调的是截止期限的可预期性；

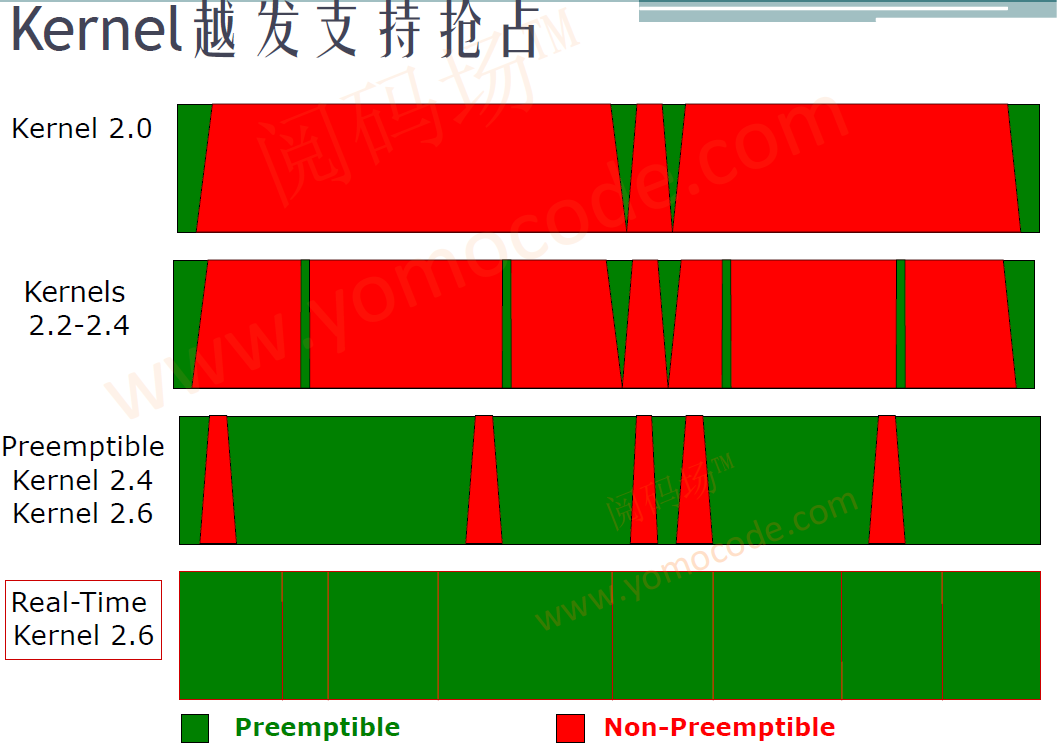




Linux的设计决定了她不是硬实时系统，有些情况(比如spin\_lock)不可抢占，其不可预期，是软实时系统；

用什么系统由场景决定，并非硬实时一定优于软实时。

Linux的抢占算法演变，越新的内核支持抢占越多；

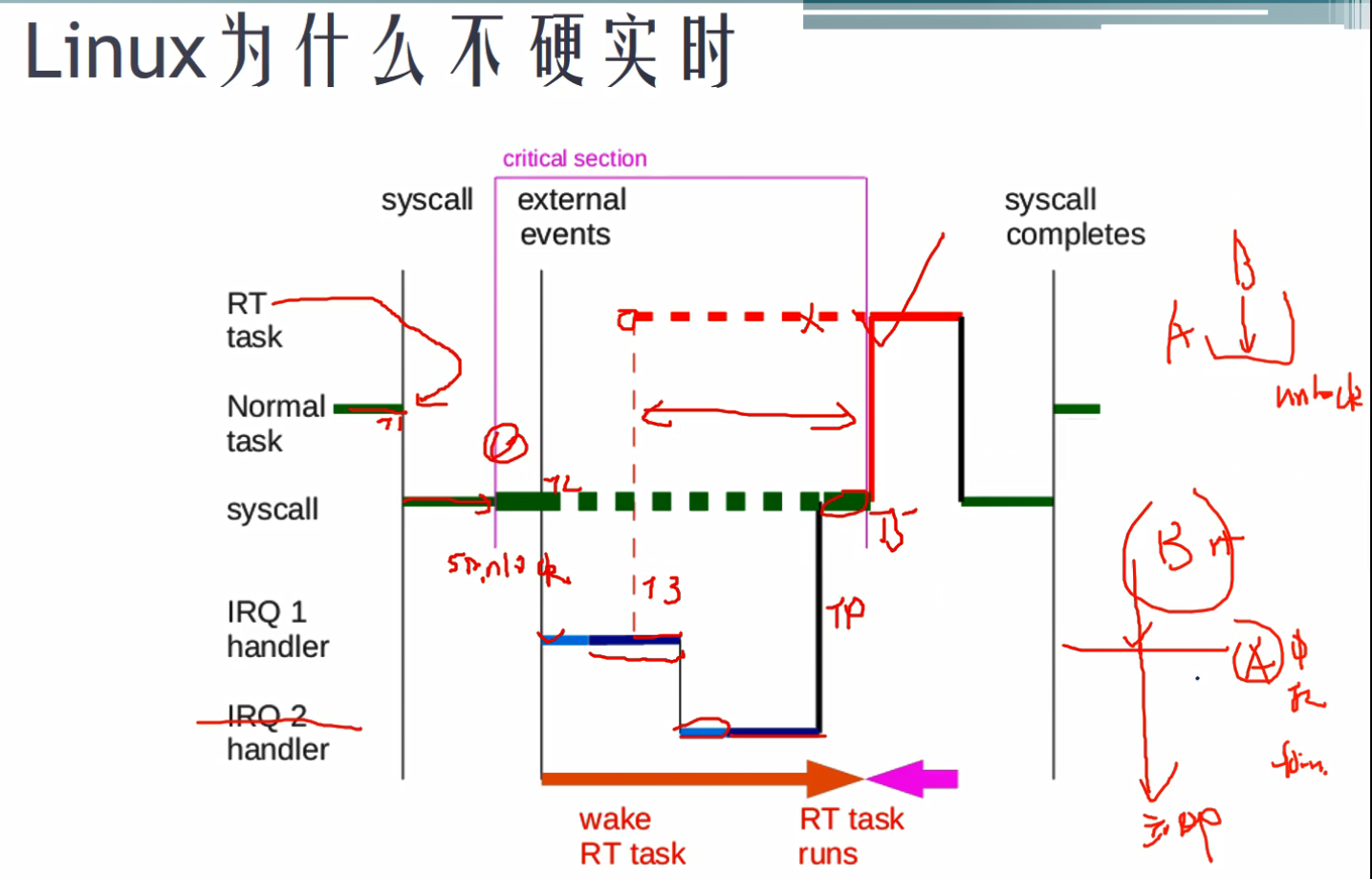


Linux的CPU消耗主要分以下几种：



内核态的spin\_lock，是自旋锁，用户态不是；

1,2,3都不可抢占，所以linux不是一个硬实时系统，**硬实时系统要求任何时刻都可以抢占**。



T3时刻唤醒一个更高优先级RT线程, RT线程只能到T5时刻(硬中断，软中断都执行完，并且释放spin\_lock之后)才能抢占，等待时间是未知的。

要解决linux硬实时性，需要打rt补丁，

补丁不在mainline中，维护网站;

<https://wiki.linuxfoundation.org/realtime/start>

<https://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git/>

rt补丁

将所有中断线程化，

去掉软中断，

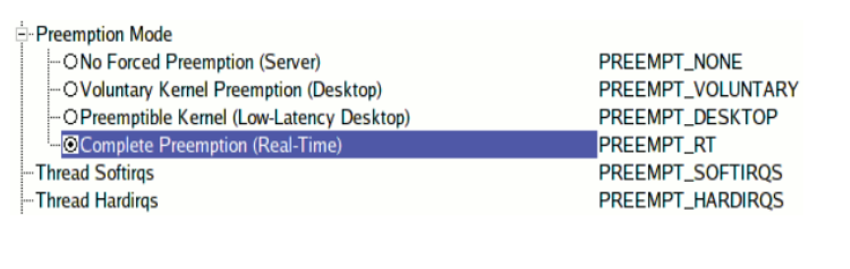
spin\_lock改成mutex实现，即系统中只有上图第4类进程，任何线程都可以被抢占；

使延迟变为可预期，即成为硬实时系统。

不是每一个内核版本都有相应rt补丁。

rt可以将linux延迟做到100us量级(1G CPU)，同时吞吐会下降。

打上补丁后如下图



Server:不抢占

Desktop: kernel不抢占

Low-Latency Destktop：手机用，kernel也抢占

Real-Time：完全抢占；

调度器实时后，linux还不一定实时，有无数内存坑，比如内存管理部分，ROW，写内存时发现内存尚未分配，此时无法保持实时性。

其他方法：

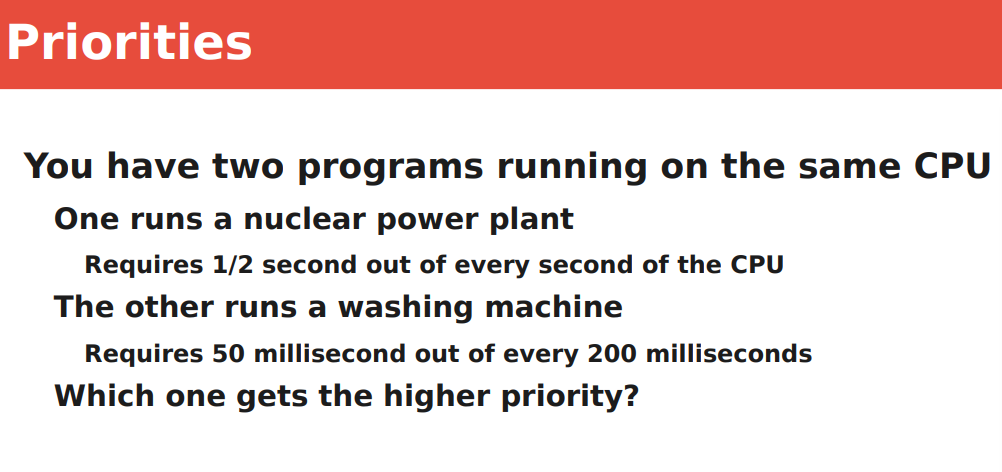
同时运行两个系统，实时任务放在RTOS跑，非实时任务放在Linux跑。

比如单反，一般用两个系统，传统拍照用实时系统，涉及网络相关任务放在linux，利用linux强大的网络协议栈。

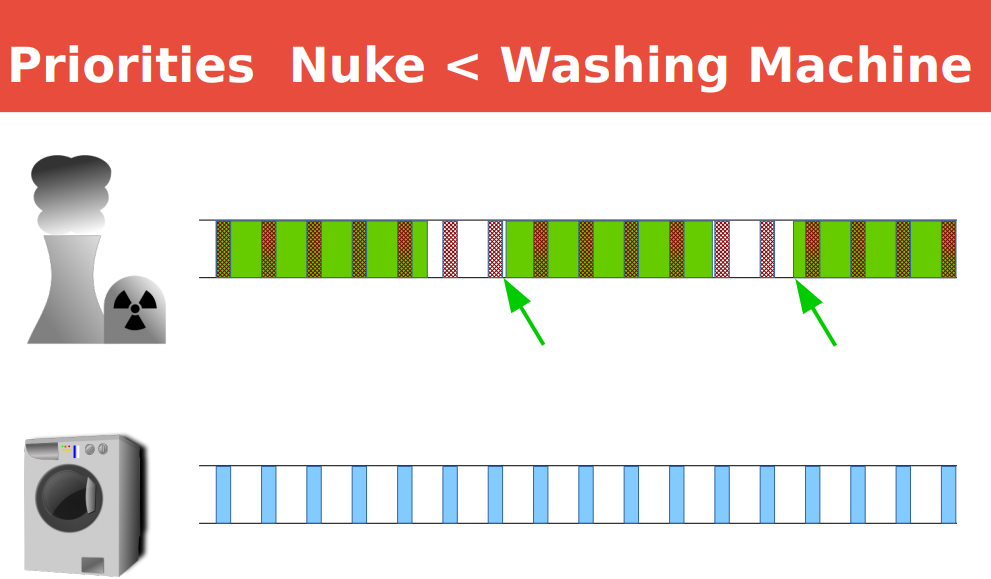
4.Deadline调度算法

<https://elinux.org/images/f/fe/Using_SCHED_DEADLINE.pdf>

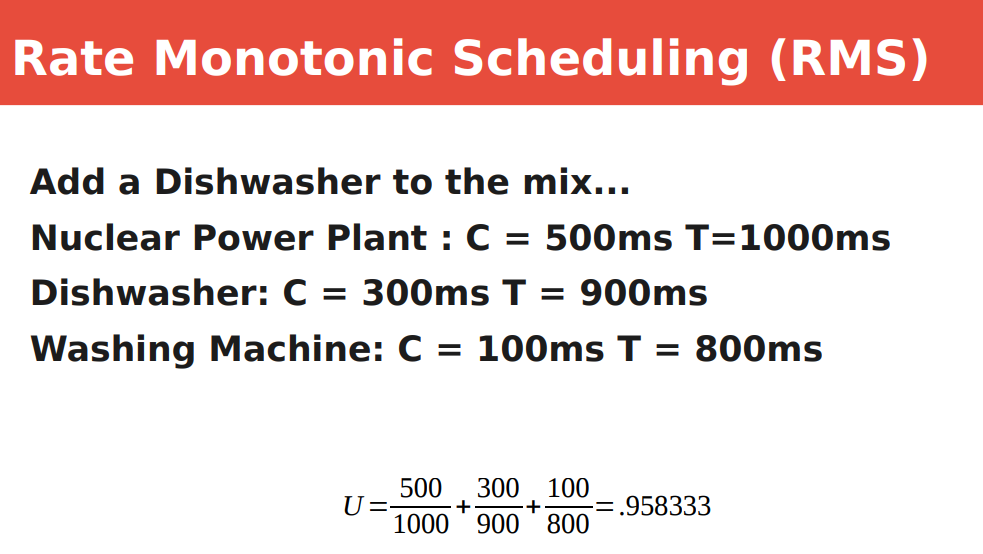
如下场景需求，系统有两个任务，核电站任务要求运行1/2s, 洗衣机要求运行20ms/200ms;



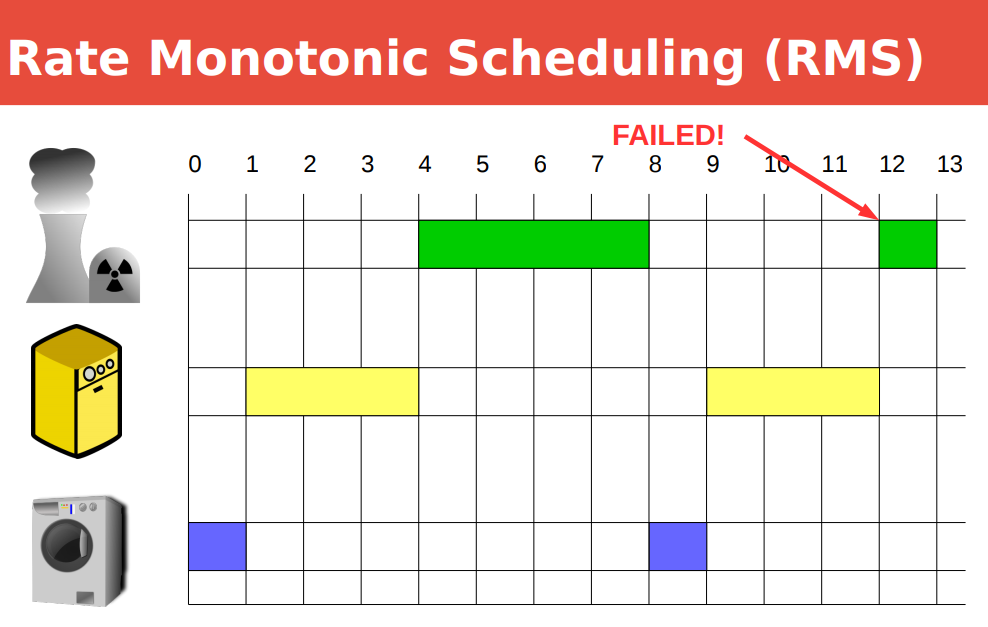
用传统RT或CFS调度算法，必须设置洗衣机优先级更高才能满足需求。(核电站优先级更高将独占CPU超过0.5s, 下一个周期无法满足CPU需求)



但当系统有多个任务，比如核电站要求800ms周期内必须跑500ms，洗碗机要求300/900ms, 洗衣机要求100/800ms：



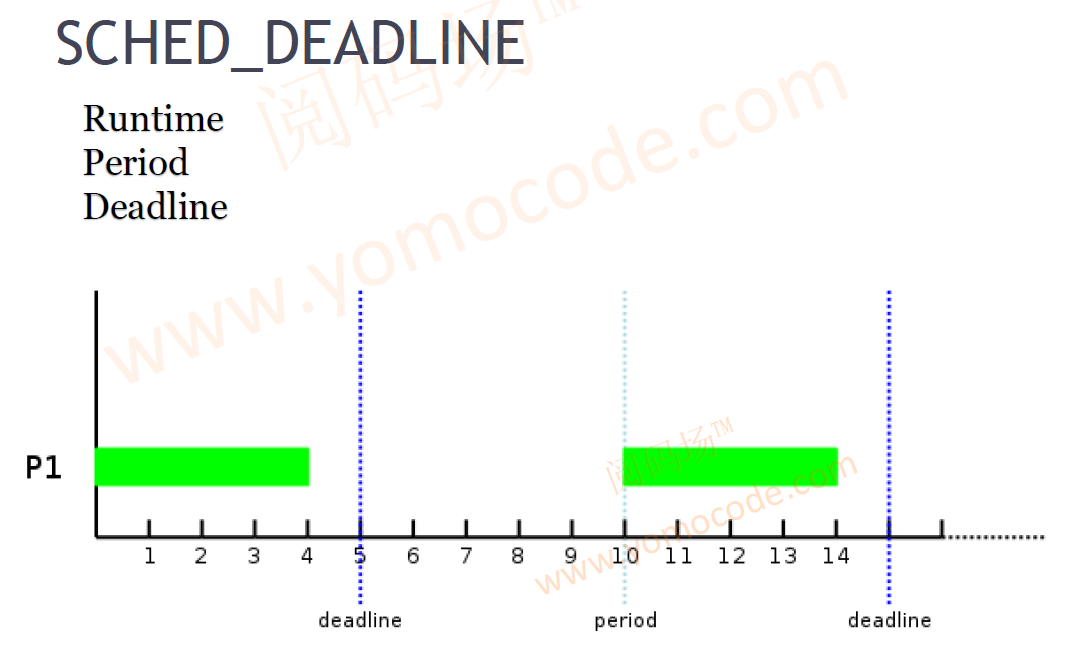
总的CPU消耗是小于100%的。但是按照传统RR算法，无法满足调度需求；



Linux提供了一个Rate Monotonic Scheduling：

**Deadline调度算法思想**：当产生一个调度点的时候，DL调度器总是选择其Deadline距离当前时间点最近的那个任务并调度它执行，谁更紧急，谁先跑；

Deadline需要设置三个参数，周期Period，Deadline, Runtime



deadline调度器虽然可以保证每个RT任务在deadline之前完成，但是并不能保证每一个任务的最小响应时间。

内核实例，sched-deadline.c

设置允许普通用户修改app调度策略

sudo setcap 'cap\_sys\_nice=eip' ./a.out

pidof ./a.out

sudo chrt -d -T 1000000 -P 3000000 -D 2000000 0 pid

sudo chrt -p pid

$ sudo chrt -p 6276

chrt: unknown scheduling policy

pid 6276's current scheduling policy: pid 6276's current scheduling priority: 0

chrt 版本太低，未支持显示deadline参数；

Deadline在top显示rt进程，

练习

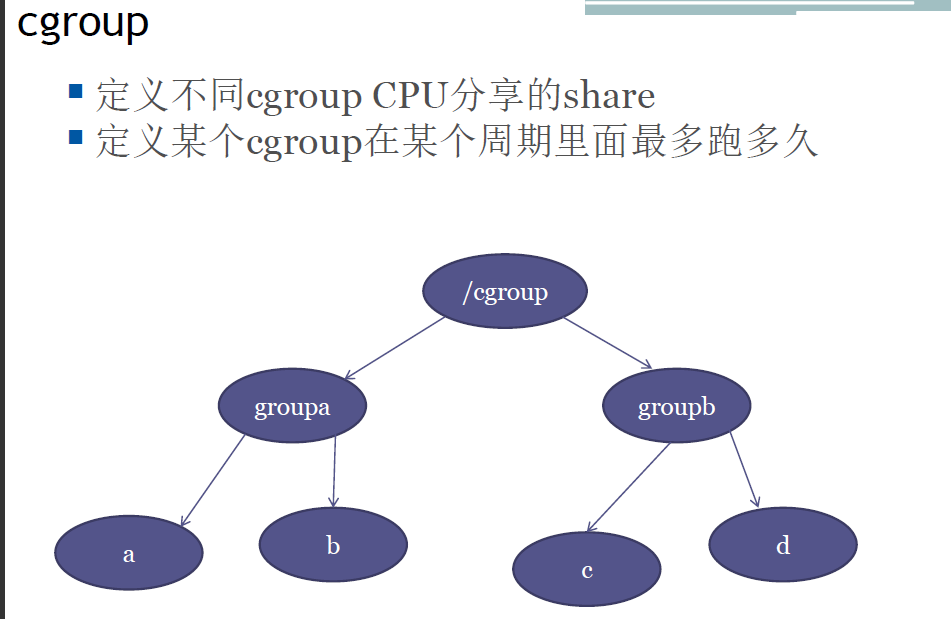
cyclictest工具

用time命令跑1个含有2个死循环线程的进程

用taskset调整多线程依附的CPU

# 第5部分 Cgroup：

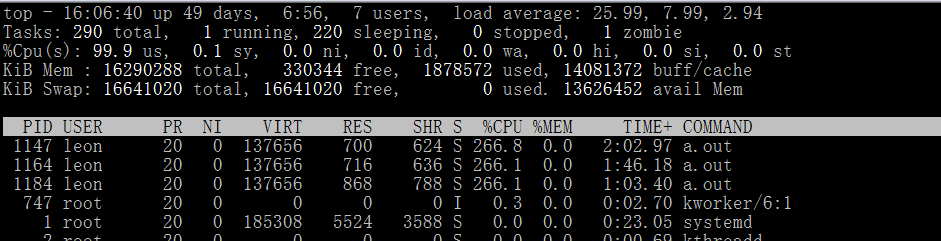
按群分组，分配资源，二级分配；



## (1)按群组分配资源

将进程分组，多个组按权重分配CPU配额，单个组内任务再按进程调度算法分配

案例：启动三个a.out，一共耗CPU800%(八核)



创建两个CGROUP：A和B

cd /sys/fs/cgroup/cpu

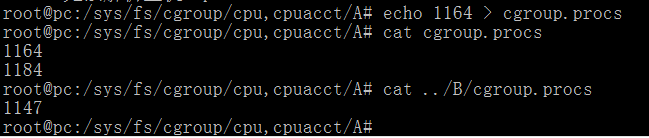
sudo mkdir A

sudo mkdir B

关注A/B中两个成员，

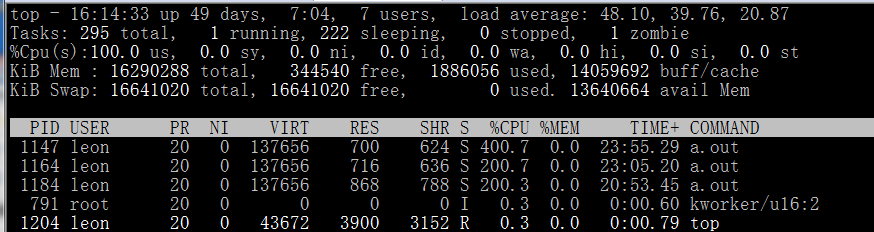
cgroup.procs：本群组包含的进程

cpu.shares：本组权重值，与其他组竞争的权重值；



将进程1164/1184加入组A,1147加入组B

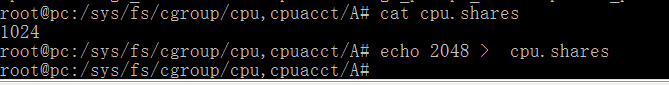
查看CPU占用情况：



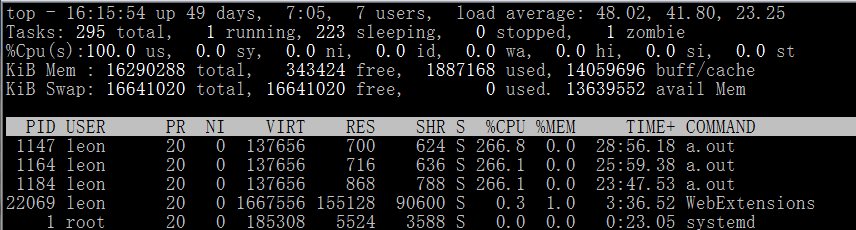
即组A两进程之和与组B相当，因为默认权重相等；

修改组A权重值为2048

echo 2048 > cpu.shares

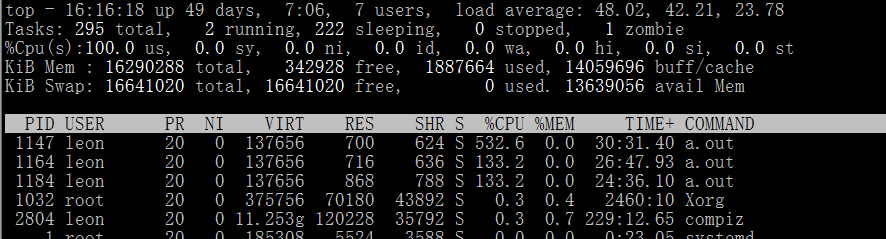


三进程CPU占有率变为均等



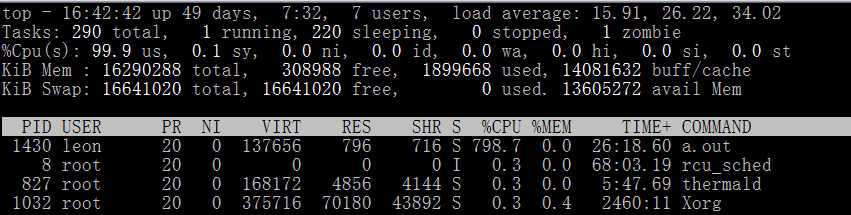
修改组A权重值为512

echo 512 > cpu.shares



上图看到组A的进程1164，1184CPU占用率加起来大概是组B一半

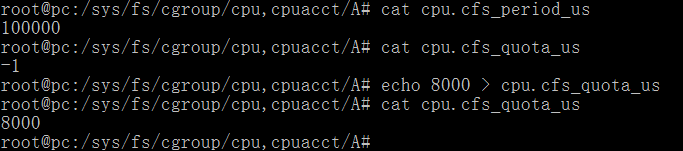
这里cgroup.procs里的进程号是动态的，进程死亡，这里进程号就会消失；



## (2)设置群的quota配额:

cpu.cfs\_perid\_us: 默认100ms；

cpu.cfs\_quota\_us： 设置该群组最大可占用CPU比例；



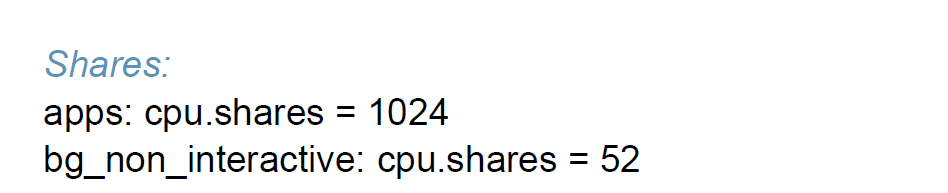
群A的CPU占用最大8000/100000; cfs\_quota\_us可以超过100%, 最多不超过CPU个数；

类似网络带宽，按付费给予配额。

还可以进一步分配每个CPU的具体任务，参考cgroup文档。

cgroup应用案例1：

Android新版本中，将任务分为前端进程组，后端进程组，分配不同权重



这样前端进程会分配更多CPU，后端进程(比如后台运行支付宝和微信等)组占用CPU较少，合理的提高了CPU利用率，提高交互型，用户体验更好。

cgroup应用案例2：

在云服务中，用Cgroup限制CPU配额，根据支付费用分配CPU资源，即使有多余CPU资源存在，付费用户得到的CPU资源也不会比原本更多。

## 1.sched\_autogroup:

So I think this is firmly one of those “real improvement” patches. Good job. By Linus

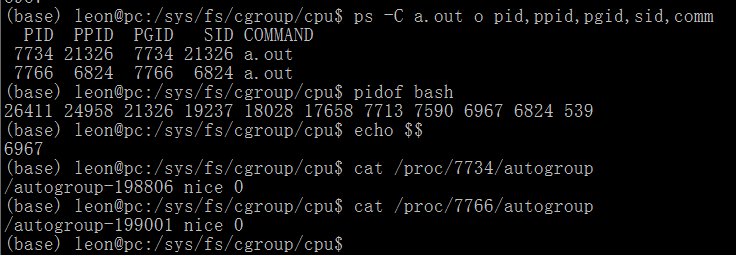
每一个seesion，自动创建一个group, 也可以关闭

leon@pc:/proc/sys/kernel$ cat sched\_autogroup\_enabled

1

群默认权重是一样的，所有每个新seesion的CPU占有率都相同，也可以设置nice;

每个seesion的所有进程都在一个group；



## 2 dock和cgroup

启动每个docker是一个group，可以设置其CPU占用quota

docker run --cpu-quota 25000 --cpu-period 10000 --cpu-shares 30

linuxep/lepv0.1

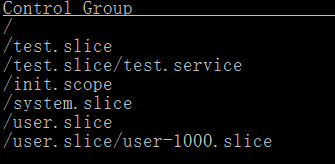
## 3 systemd和cgroup

systemd是应用层用来管理服务群组的层级关系；跟cgroup非常类似，

它是一个动态过程，只有设置资源限制，才会跟cgroup线程一一对应起来；

systemd cg 相关命令

systemd-cgls



systemd-cgtop

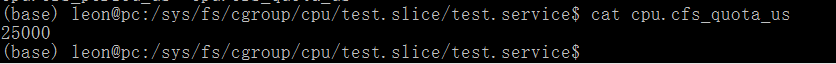
systemd-run

systemctl

set-property 限制cgroup的资源

sudo systemd-run -p --unit=test --slice=test ./a.out 4

sudo systemd-run -p CPUQuota=25% --unit=test --slice=test ./a.out 4

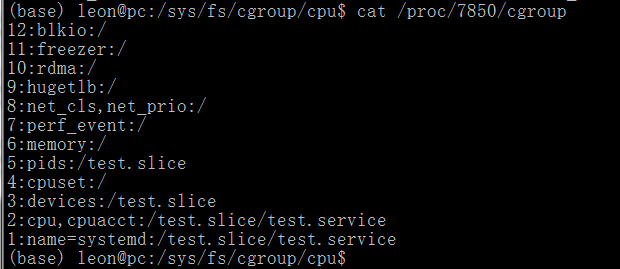


─test.slice

│ └─test.service

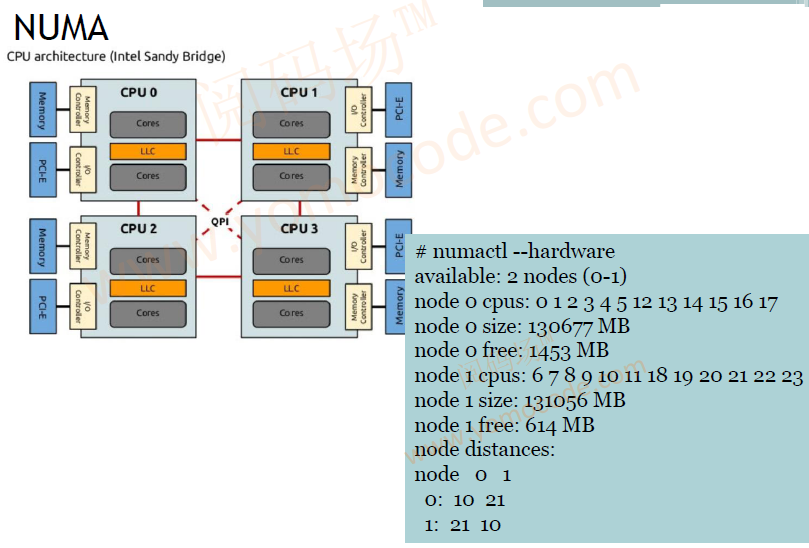
│ └─7850 /home/leon/xxx/./a.out 4

cgroup的cpu目录会生成对应test配置文件



## 4 cpuset和cgroup

将CPU添加到cgroup群组，将进程添加入该群组，就限制在该群组的CPU运行，特别适用NUMA的场景



可以设置cpuset.cpus

cpuset.mems

在:/sys/fs/cgroup/cpuset创建组A，然后往A添加进程，设置A目录下的cpuset.cpus。

leon@:echo 0-1 > cpuset.cpus

该组的进程就只限制在指定CPU上运行；

同样可以设置内存限制；

# 第六部分 深度睡眠

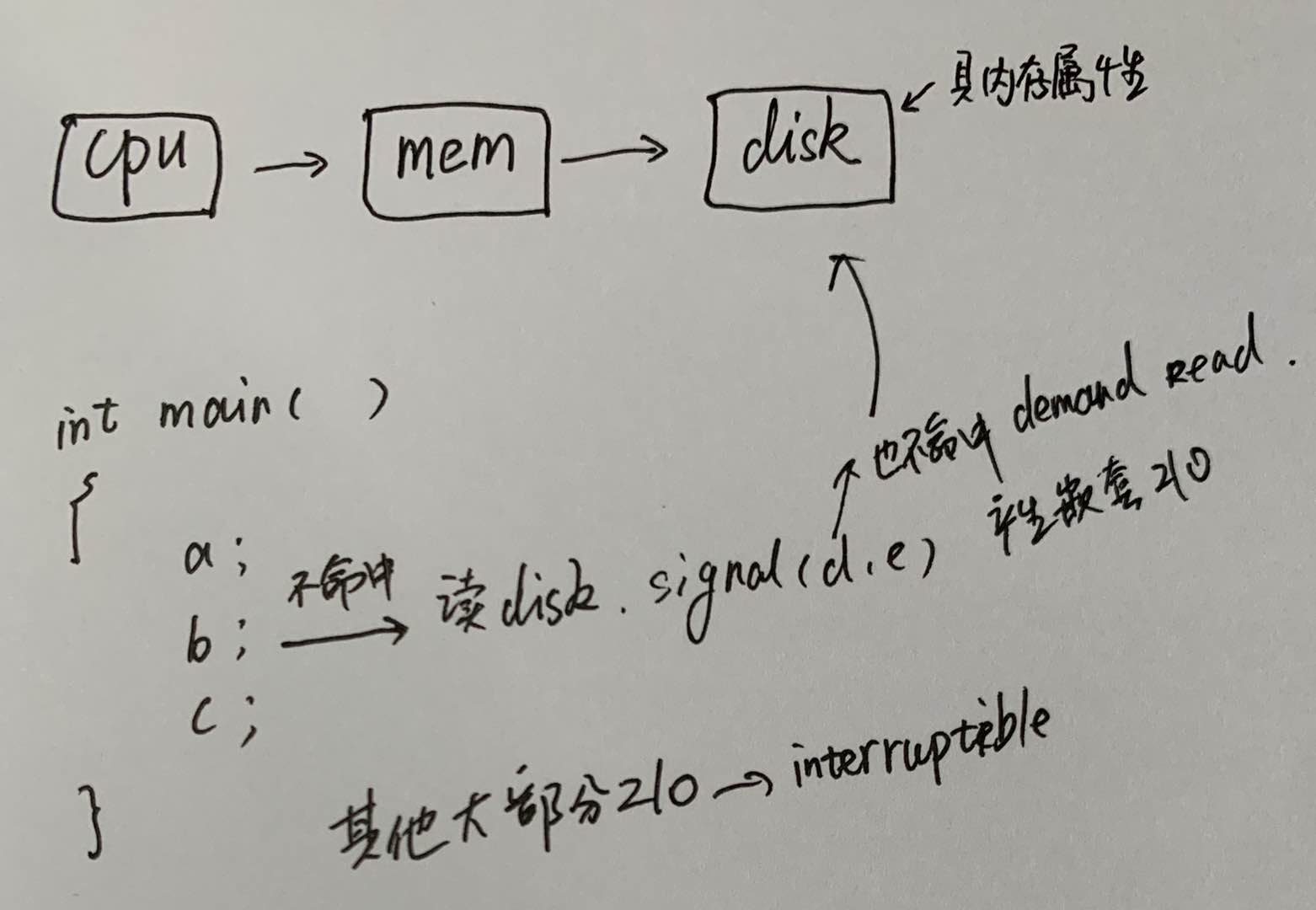
## 1.深度睡眠特点及存在原因

**深度睡眠TASK\_UNINTERRUPTIBLE**：不可被信号唤醒；

**浅度睡眠TASK\_INTERRUPTIBLE**：唤醒方式，等到需要的资源，响应信号；

深度睡眠场景：

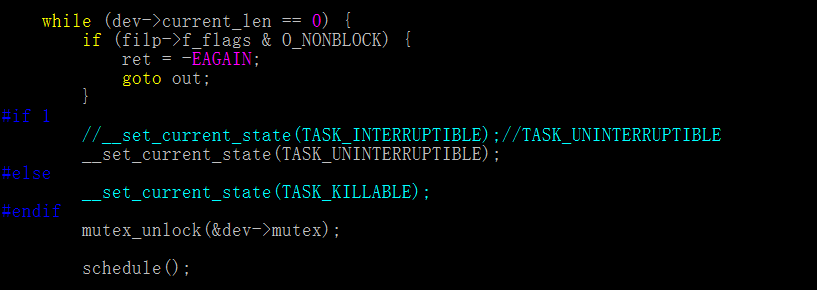
有些场景是不能响应信号的，比如读磁盘过程是不能打断的，NFS也是；



执行程序过程中，可能需要从磁盘读入可执行代码，假如在读磁盘过程中，又有代码需要从磁盘读取，就会造成嵌套睡眠。逻辑做的太复杂，所以读磁盘过程不允许打断，即只等待IO资源可用，不响应任何信号；

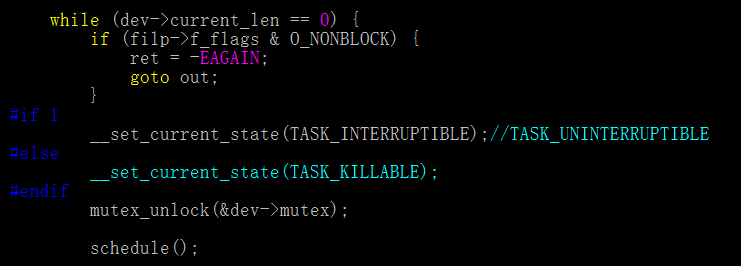
应用程序无法屏蔽也无法重载SIGKILL信号，深度睡眠可以不响应SIGKILL kill-9信号；

**深度睡眠案例**



此时该进程读IO过程，是无法用kill -9杀死的；

一般绝大部分场景，都设置为浅睡眠



ps aux: D+

深度睡眠进程显示D+状态

## 2.深度睡眠对load average的影响

top load average:

cat /dev/global\_fifo

Kill -2 pid

Kill -9 pid

无法杀死，load average变大，CPU依然空闲；

多加一个进程

cat /dev/global\_fifo，

**load average继续变大，CPU依然空闲；**

load average：包括CPU消耗和IO的总预期，此时虽然不消耗CPU，但是等待IO消耗了时间，load average显示的是随时间变化的平均负载预期；

D状态会增加load average；

对于同一个程序，不同执行环境：

执行性能=APP(消耗CPU)+DISK(I/O消耗)；

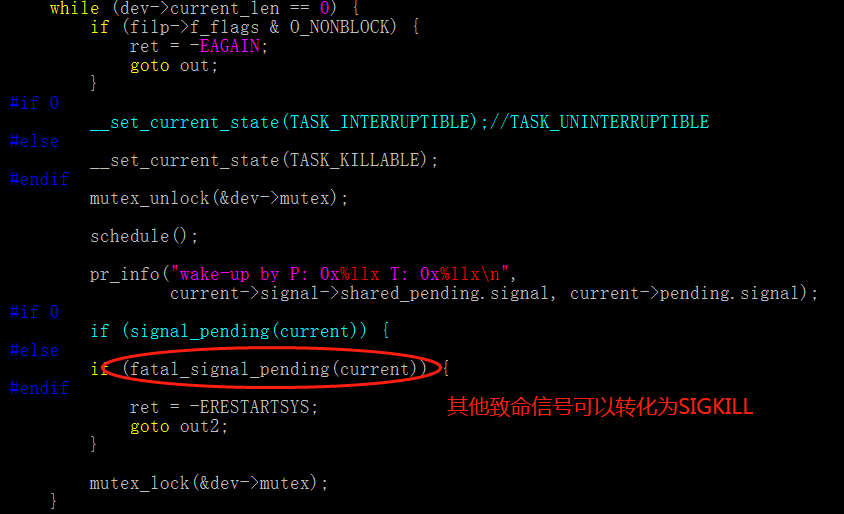
## 3.TASK\_KILLABLE

**只响应致命信号 D状态；**

不跳转APP响应信号，不会产生递归睡眠；

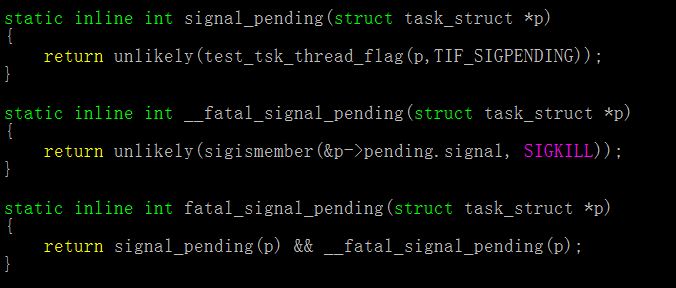
\_\_set\_current\_state(TASK\_KILLABLE);

可以响应信号9

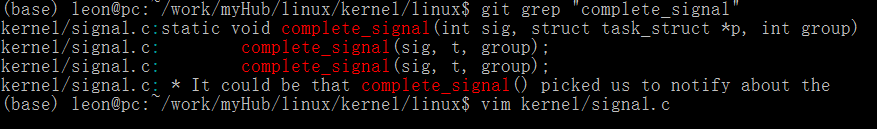


kill -2 pid ? //发现被杀死，什么原因呢

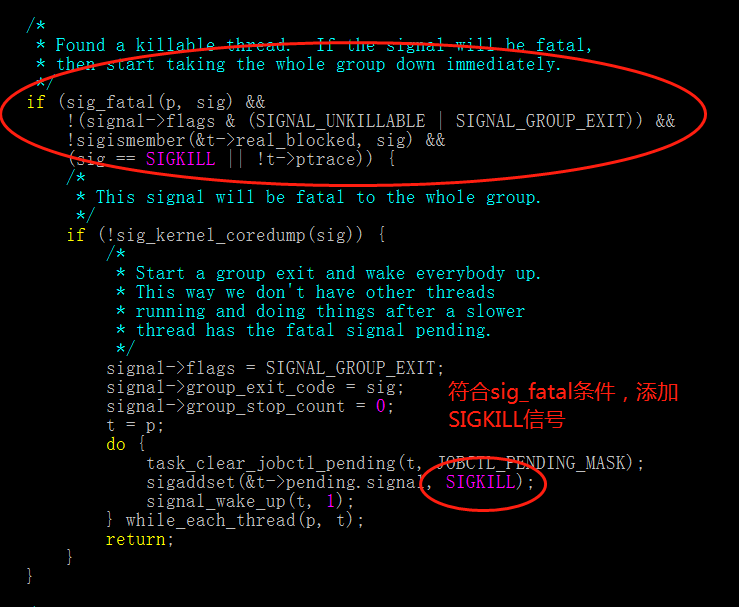
dmesg



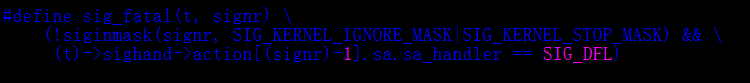
git grep "complete\_signal"



**满足sig\_fatal**，判断独占signal\_pending，符合条件，内核添加一个SIGKILL信号，导致-2变成-9



条件：



#define sig\_fatal(t, signr) \

(!siginmask(signr, SIG\_KERNEL\_IGNORE\_MASK|SIG\_KERNEL\_STOP\_MASK) && \

(t)->sighand->action[(signr)-1].sa.sa\_handler == SIG\_DFL)

满足信号：不忽略，也不是stop，行为缺省。(缺省信号定义基本就是死，Core/Term等)

那么添加SIGKILL信号；

与fatal\_signal()必须9信号，不一样；

Ps:如果应用层捕获了2信号，那么就不满足sig\_fatal，kill - 2杀不死了；

**综上，**满足sig\_fatal，kill -2可以杀TASK\_KILLABLE进程；