Linux进程、线程和调度(1)

讲解时间：9月13日、9月15日、9月19日、9月22日晚20点

宋宝华 <21cnbao@gmail.com>

报名直播或者录播：
http://edu.csdn.net/huiyiCourse/series_detail/60?utm_source=wx2

扫描二维码报名

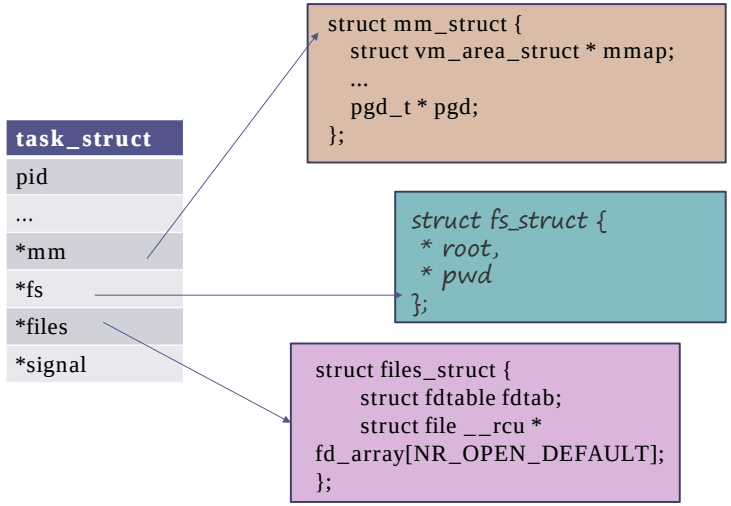
9.13第一次课大纲

* Linux进程生命周期(就绪、运行、睡眠、停止、僵死)
* 僵尸是个什么鬼？
* 停止状态与作业控制，cpulimit
* 内存泄漏的真实含义
* task_struct以及task_struct之间的关系
* 初见fork和僵尸

练习题

* fork的例子
* life-period例子，观察僵尸
* 用cpulimit控制CPU利用率

进程控制块PCB



pid

pid的数量是有限的

```
$ cat /proc/sys/kernel/pid_max
32768
```

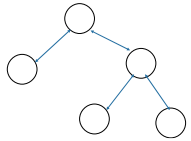
Fork炸弹

```
:(){ :|:& };:
```

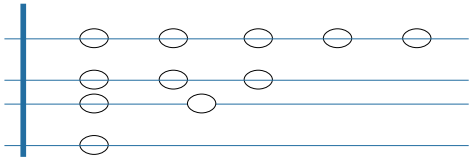
task_struct被管理

形成链表

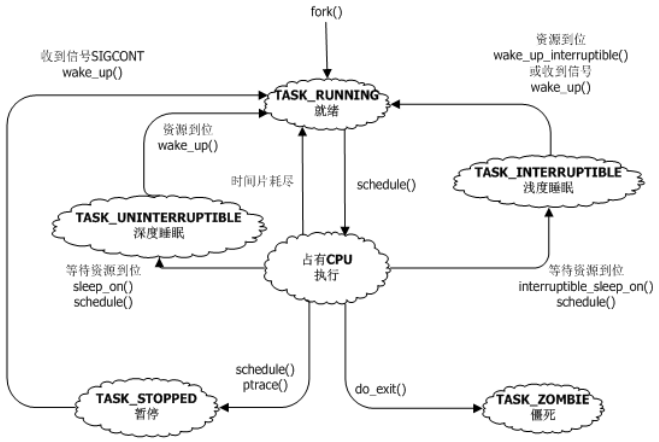
形成树



形成哈希：pid-> task_struct



进程生命周期



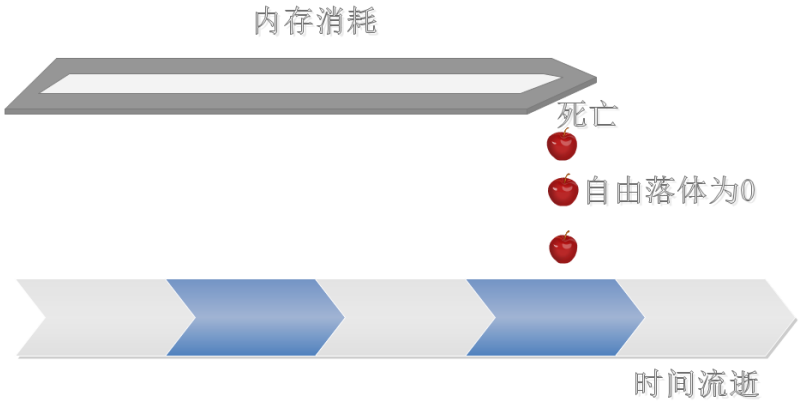
僵尸是什么

资源已经释放：无内存泄漏等
task_struct还在：父进程可以查到子进程死因

```
static int wait_task_zombie(struct wait_opts *wo, struct task_struct *p)
{
        int state, retval, status;
        pid_t pid = task_pid_vnr(p);
        uid_t uid = from_kuid_munged(current_user_ns(), task_uid(p));
        struct siginfo __user *infop;

        if (!likely(wo->wo_flags & WEXITED))
                return 0;

        if (unlikely(wo->wo_flags & WNOWAIT)) {
                int exit_code = p->exit_code;
                int why;

                get_task_struct(p);
                read_unlock(&tasklist_lock);
                sched_annotate_sleep();

                if ((exit_code & 0x7f) == 0) {
                        why = CLD_EXITED;
                        status = exit_code >> 8;
                } else {
                        why = (exit_code & 0x80) ? CLD_DUMPED : CLD_KILLED;
                        status = exit_code & 0x7f;
                }
```

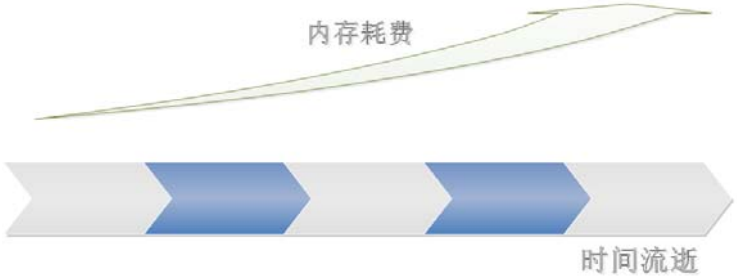
内存泄漏到底是什么?

不是：进程死了，内存没释放



内存泄漏到底是什么(cont.)?

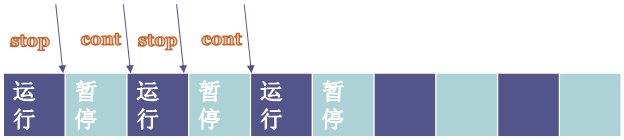
而是：进程活着，运行越久，耗费内存越多



作业控制

ctrl+ z, fg/bg
cpulimit

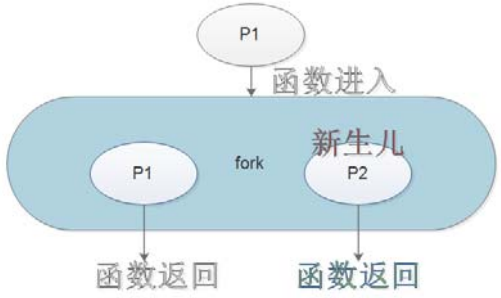
cpulimit -l 20 -p 10111
限制pid 为10111程序 的 cpu使用率不超过 10%



fork

打印几个hello?

```
1 main()
2 {
3         fork();
4         printf("hello\n");
5         fork();
6         printf("hello\n");
7         while(1);
8 }
```



fork(cont.)

怎么打印？

```
 6 int main(void)
 7 {
 8         pid_t pid,wait_pid;
 9         int status;
10
11         pid = fork();
12
13         if (pid==-1)    {
14                 perror("Cannot create new process");
15                 exit(1);
16         } else  if (pid==0) {
17                 printf("a\n");
18         } else {
19                 printf("b\n");
20         }
21
22         printf("c\n");
23         while(1);
24 }
```

子死父清场

```
pid = fork();

if (pid==-1)    {
        perror("Cannot create new process");
        exit(1);
} else  if (pid==0) {
        printf("child process id: %ld\n", (long) getpid());
        pause();
        _exit(0);
} else {
        wait_pid=waitpid(pid, &status, WUNTRACED | WCONTINUED);

        if (wait_pid == -1) {
                perror("cannot using waitpid function");
                exit(1);
        }

        if(WIFSIGNALED(status))
                printf("child process is killed by signal %d\n", WTERMSIG(status));
```

课程练习源码

https://github.com/21cnbao/process-courses

更早课程

- 《Linux总线、设备、驱动模型》录播：
http://edu.csdn.net/course/detail/5329

- 深入探究Linux的设备树
http://edu.csdn.net/course/detail/5627

Linux进程、线程和调度(2)

讲解时间：9月13日、9月15日、9月19日、9月22日晚20点

宋宝华 <21cnbao@gmail.com>

报名直播或者录播：

http://edu.csdn.net/huiviCourse/series_detail/60?utm_source=wx2

扫描二维码报名

麦当劳喜欢您来，喜欢您再来

扫描关注

Linuxer

9.15第二次课大纲

1.fork、vfork、clone
2.写时拷贝技术
3.Linux线程的实现本质
4.进程0和进程1
5.进程的睡眠和等待队列
6.孤儿进程的托孤，SUBREAPER

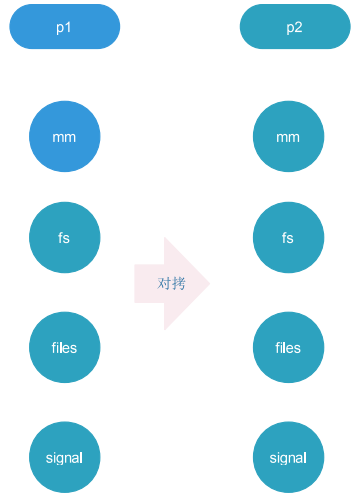
练习题

1.fork、vfork、Copy-on-Write例子
2.life-period例子，实验体会托孤
3.pthread_create例子，strace它
4.彻底看懂等待队列的案例

fork

- fork()
1. SIGCHLD

执行一个copy，但是任何修改都造成分裂，如：chroot, open, 写memory，mmap，sigaction....



Copy-on-write

最开始

virt1	phy1	原则上R+W

fork后

virt1	phy1	RD-ONLY
virt1	phy1	RD-ONLY

write后

virt1	phy1	原则上R+W

拷贝

virt1	phy2	原则上R+W

Mmu-less Linux

无copy-on-write,没有fork

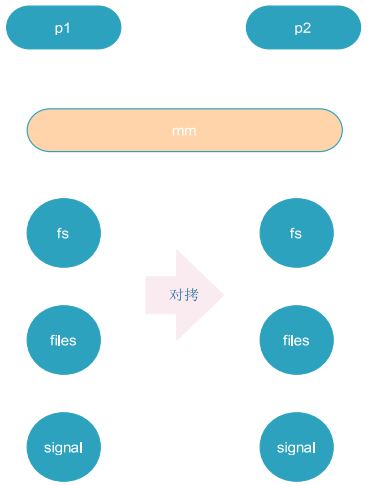
使用vfork:父进程阻塞直到子进程

1. exit
2. exec

vfork

- vfork()

1. CLONE_VM
2. CLONE_VFORK
3. SIGCHLD

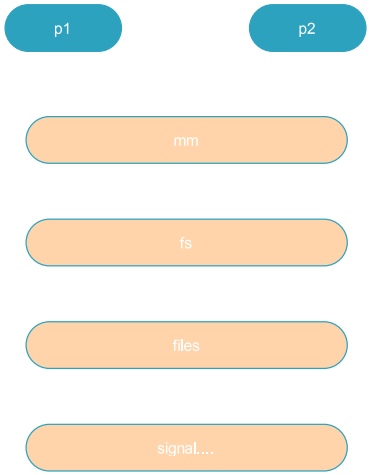


pthread_create-> clone

- clone()

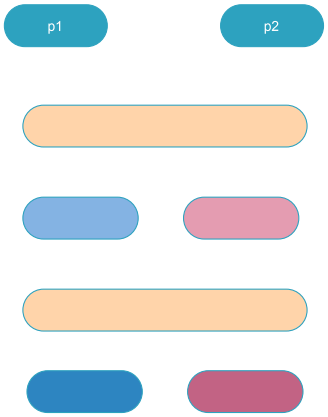
1. CLONE_VM
2. CLONE_FS
3. CLONE_FILES
4. CLONE_SIGHAND
5. CLONE_THREAD

共享资源，可调度



进程、线程与“人妖”

- clone

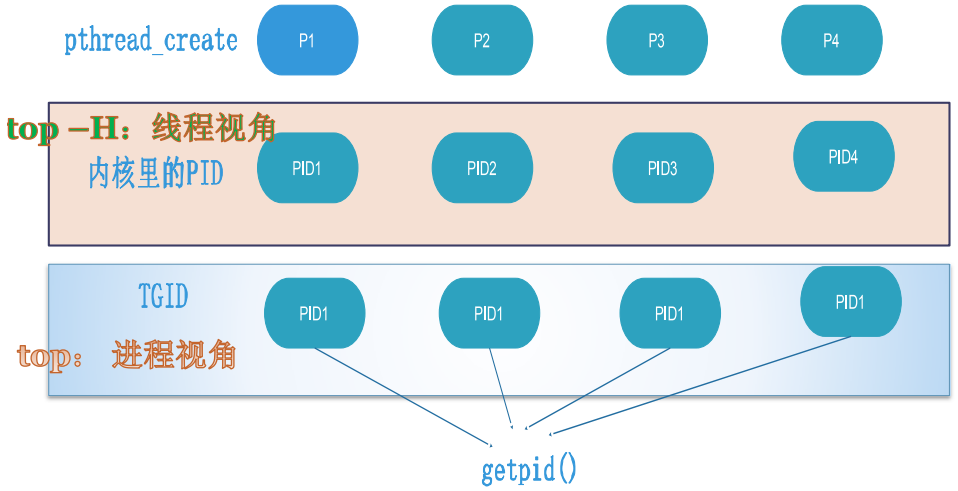


如果我们只clone一部分资源呢？

进程?
线程?
人妖?

妖有了仁慈的心,就不再是妖,是人妖

PID和TGID

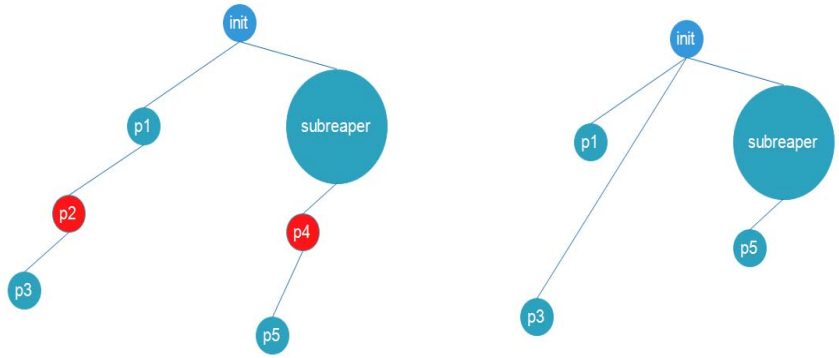


SUBREAPER与托孤

```
/* 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 是 Linux 3.4 加入的新特性。把它设置为非零值，当前进程就会变成 subreaper，会像 1 号进程那样收养孤儿进程了。

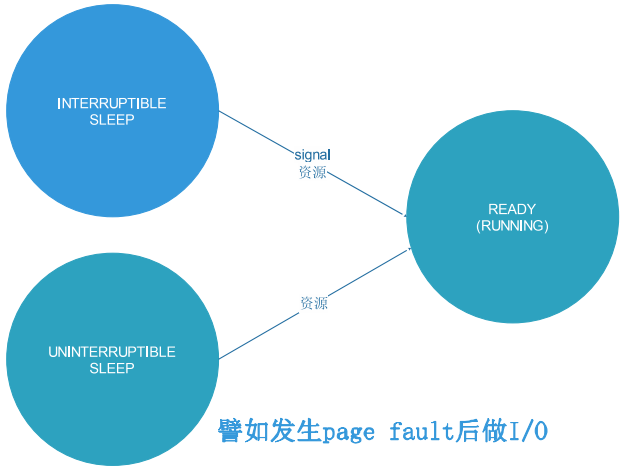
init vs. SUBREAPER



p2和p4死

睡眠

深度睡眠 vs. 浅度睡眠



譬如发生page fault后做I/O

wait queue

```
static ssize_t globalfifo_read(struct file *filp, char __user *buf,
                               size_t count, loff_t *ppos)
{
        int ret;
        struct globalfifo_dev *dev = container_of(filp->private_data,
                struct globalfifo_dev, miscdev);

        DECLARE_WAITQUEUE(wait, current);

        mutex_lock(&dev->mutex);
        add_wait_queue(&dev->r_wait, &wait);

        while (dev->current_len == 0) {
                if (filp->f_flags & O_NONBLOCK) {
                        ret = -EAGAIN;
                        goto out;
                }
                __set_current_state(TASK_INTERRUPTIBLE);
                mutex_unlock(&dev->mutex);

                schedule();
                if (signal_pending(current)) {
                        ret = -ERESTARTSYS;
                        goto out2;
                }

                mutex_lock(&dev->mutex);
        }

        if (count > dev->current_len)
                count = dev->current_len;

        if (copy_to_user(buf, dev->mem, count)) {
                ret = -EFAULT;
                goto out;
        } else {
                memcpy(dev->mem, dev->mem + count, dev->current_len - count);
                dev->current_len -= count;
                printk(KERN_INFO "read %d bytes(s),current_len:%d\n", count,
```

进程0和1

课程练习源码

https://github.com/21cnbao/process-courses

Linux进程、线程和调度(3)

讲解时间：9月13日、9月15日、9月19日、9月22日晚20点

宋宝华 <21cnbao@gmail.com>

扫描二维码报名

报名直播或者录播：
http://edu.csdn.net/huiyiCourse/series_detail/60?utm_source=wx2

麦当劳喜欢您来，喜欢您再来

扫描关注
Linuxer

9.19第三次课大纲

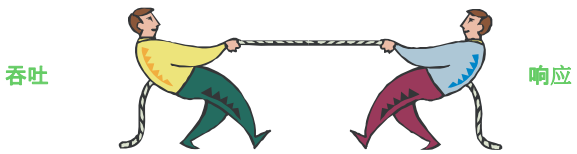
1. CPU/IO消耗型进程
2. 吞吐率 vs. 响应
3. SCHED_FIFO、SCHED_RR
4. SCHED_NORMAL和CFS
5. nice、renice
6. chrt

练习题

1. 运行2个高CPU利用率进程，调整他们的nice
2. 用chrt把一个死循环进程调整为SCHED_FIFO
3. 阅读ARM的big.LITTLE架构资料，并论述为什么ARM要这么做？

吞吐 vs. 响应

- 吞吐和响应之间的矛盾
- ✓ 响应：最小化某个任务的响应时间，哪怕牺牲其他的任务为代价
- ✓ 吞吐: 全局视野，整个系统的workload被最大化处理



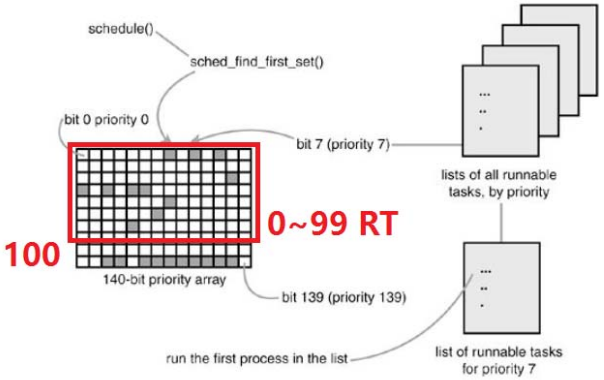
I/O消耗型vs. CPU消耗型

- IO bound: CPU利用率低，进程的运行效率主要受限于I/O速度；
- CPU bound: 多数时间花在CPU上面(做运算).

big.LITTLE

早期2.6:优先级数组和Bitmaps

- 0~139
- 某个优先级有TASK_RUNNING进程, 响应bit设置1。
- 调度第一个bitmap设置为1的进程



实时进程调度

- SCHED_FIFO：不同优先级按照优先级高的先跑到睡眠，优先级低的再跑；同等优先级先进先出。
- SCHED_RR：不同优先级按照优先级高的先跑到睡眠，优先级低的再跑；同等优先级轮转。

早期2.6:非实时进程的调度和动态优先级

- 在不同优先级轮转
- -20 ~ +19的nice值
- 根据睡眠情况，动态奖励和惩罚

惩罚!!

奖励

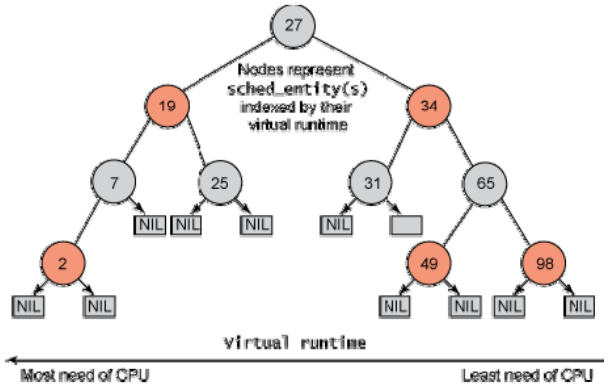
rt的门限

在period的时间里RT最多只能跑runtime的时间

/proc/sys/kernel/sched_rt_period_us
/proc/sys/kernel/sched_rt_runtime_us

CFS：完全公平调度

红黑树，左边节点小于右边节点的值
运行到目前为止vruntime最小的进程
同时考虑了CPU/IO和nice



CFS weight

vruntime += delta NICE_0_LOAD/ se.weight;*

```
static const int prio_to_weight[40] = {
 /* -20 */    88761,    71755,    56483,    46273,    36291,
 /* -15 */    29154,    23254,    18705,    14949,    11916,
 /* -10 */     9548,     7620,     6100,     4904,     3906,
 /*  -5 */     3121,     2501,     1991,     1586,     1277,
 /*   0 */     1024,      820,      655,      526,      423,
 /*   5 */      335,      272,      215,      172,      137,
 /*  10 */      110,       87,       70,       56,       45,
 /*  15 */       36,       29,       23,       18,       15,
};
```

调度相关的系统调用

System Call	Description
nice()	Sets a process's nice value
sched_setscheduler()	Sets a process's scheduling policy
sched_getscheduler()	Gets a process's scheduling policy
sched_setparam()	Sets a process's real-time priority
sched_getparam()	Gets a process's real-time priority
sched_get_priority_max()	Gets the maximum real-time priority
sched_get_priority_min()	Gets the minimum real-time priority
sched_rr_get_interval()	Gets a process's timeslice value
sched_setaffinity()	Sets a process's processor affinity
sched_getaffinity()	Gets a process's processor affinity
sched_yield()	Temporarily yields the processor

代码例子

设置SCHED_FIFO和RT优先级

```
struct sched_param the_priority;

the_priority.sched_priority  = 50;
pthread_setschedparam(pthread_self(), SCHED_FIFO,
&the_priority);
```

工具chrt和renice

设置SCHED_FIFO和50 RT优先级

```
# chrt -f -a -p 50 10576
```

设置nice

```
# renice -n -5 -g 9394
# nice -n 5 ./a.out
```

课程练习源码

https://github.com/21cnbao/process-courses

更早课程

- 《Linux总线、设备、驱动模型》录播：
http://edu.csdn.net/course/detail/5329

- 深入探究Linux的设备树
http://edu.csdn.net/course/detail/5627

Linux进程、线程和调度(4)

讲解时间：9月13日、9月15日、9月19日、9月22日晚20点

宋宝华 <21cnbao@gmail.com>

扫描二维码报名

报名看录播：
http://edu.csdn.net/course/detail/5995

麦当劳喜欢您来，喜欢您再来

扫描关注
Linuxer

9.22第四次课大纲

1. 多核下负载均衡
2. 中断负载均衡、RPS软中断负载均衡
3. cgroups和CPU资源分群分配
4. Android和Docker对cgroup的采用
5. Linux为什么不是硬实时的
6. preempt-rt对Linux实时性的改造

练习题

1. 用time命令跑1个含有2个死循环线程的进程
2. 用taskset调整多线程依附的CPU
3. 创建和分群CPU的cgroup，调整权重和quota
4. cyclictest

负载均衡

- RT 进程：N个优先级最高的RT分布到N个核
  - pull_rt_task()
  - push_rt_task()
- 普通进程
  - 周期性负载均衡
  - IDLE时负载均衡
  - fork和exec时负载均衡

CPU task affinity

- 设置affinity

```
int pthread_attr_setaffinity_np(pthread_attr_t *, size_t, const cpu_set_t *);
int pthread_attr_getaffinity_np(pthread_attr_t *, size_t, cpu_set_t *);
int sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask);
int sched_getaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask);
```

taskset

- taskset -a -p 01 19999
- taskset -a -p 02 19999
- taskset -a -p 03 19999

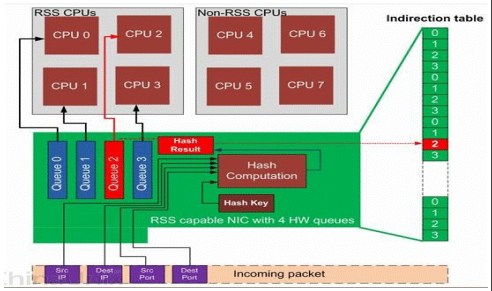
IRQ affinity

- 分配IRQ到某个CPU

```
[root@boss ~]# echo 01 > /proc/irq/145/smp_affinity
[root@boss ~]# cat /proc/irq/145/smp_affinity
      00000001
```

- mq ethernet

```
/proc/irq/74/smp_affinity 000001
/proc/irq/75/smp_affinity 000002
/proc/irq/76/smp_affinity 000004
/proc/irq/77/smp_affinity 000008
…
```



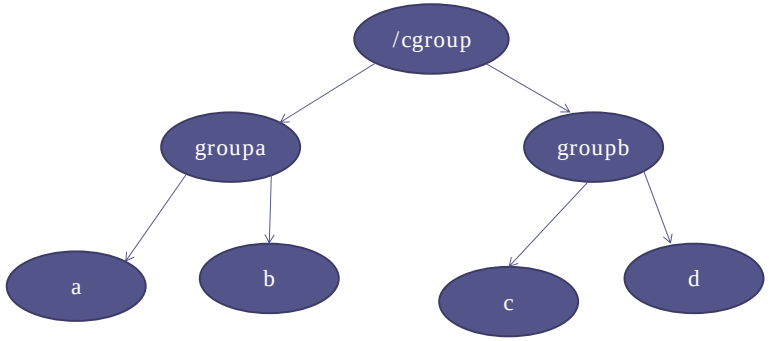
多核间的softIRQ scaling

- RPS 将包处理负载均衡到多个CPU

```
#例如
[root@machine1 ~]# echo fffe > /sys/class/net/eth1/queues/rx-0/rps_cpus
fffe

#观察
[root@machine1 ~]# watch -d "cat /proc/softirqs | grep NET_RX"
```

cgroup

- 定义不同cgroup CPU分享的share
- 定义某个cgroup在某个周期里面最多跑多久



Android和cgroup

- apps, bg_non_interactive

Shares:

apps: cpu.shares = 1024

bg_non_interactive: cpu.shares = 52

Quota:

apps:

cpu.rt_period_us: 1000000 cpu.rt_runtime_us: 800000

bg_non_interactive:

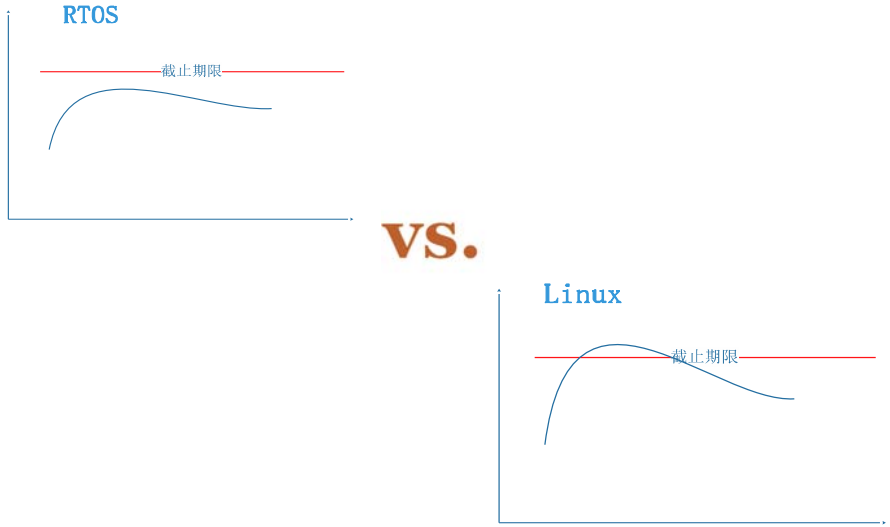
cpu.rt_period_us: 1000000 cpu.rt_runtime_us: 700000

Docker和cgroup

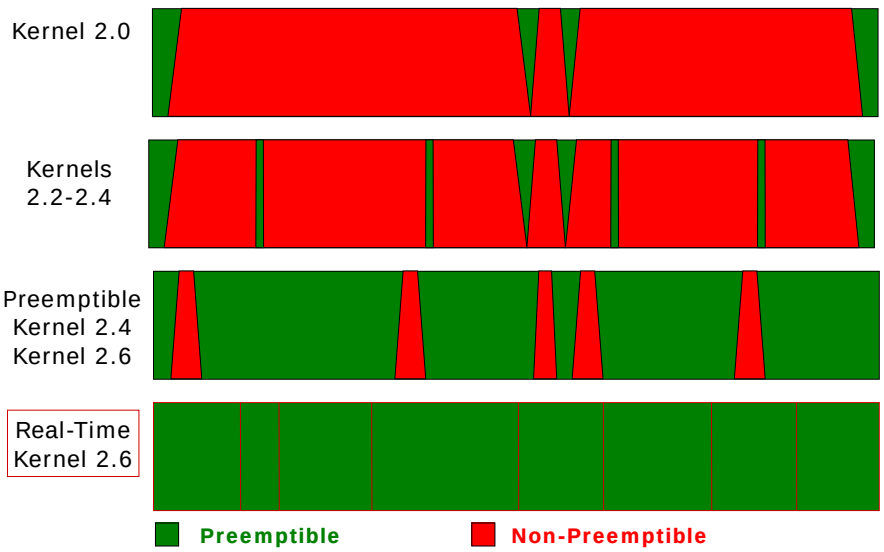
- Docker使用cgroup调配容器的CPU资源

```
$ docker run --cpu-quota 25000 --cpu-period 10000 --cpu-shares 30
linuxep/lepv0.1

baohua@ubuntu:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
3f39ca25d14d

baohua@ubuntu:/sys/fs/cgroup/cpu/docker$ cd 3f39c...
baohua@ubuntu:/sys/fs/cgroup/cpu/docker/3f39c...$ ls
cgroup.clone_children  cgroup.procs  cpuacct.stat  cpuacct.usage
cpuacct.usage_percpu  cpu.cfs_period_us  cpu.cfs_quota_us  cpu.shares  cpu.stat
notify_on_release  tasks
baohua@ubuntu:/sys/fs/cgroup/cpu/docker/3f39c...$ cat cpu.cfs_quota_us
25000
baohua@ubuntu:/sys/fs/cgroup/cpu/docker/3f39c...$ cat cpu.cfs_period_us
10000
baohua@ubuntu:/sys/fs/cgroup/cpu/docker/3f39c...$ cat cpu.shares
30
```

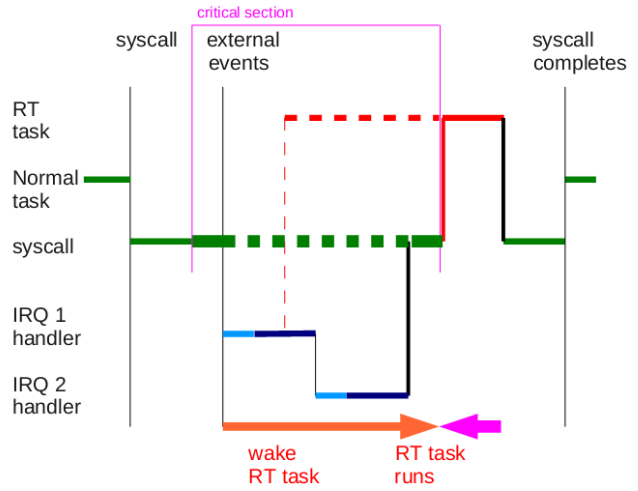
Hard realtime – 可预期性



Kernel越发支持抢占

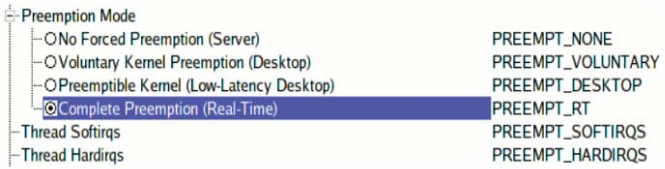


Linux为什么不硬实时



PREEMPT_RT补丁

- spinlock迁移为可调度的mutex,同时报了raw_spinlock_t
- 实现优先级继承协议
- 中断线程化
- 软中断线程化



课程练习源码

https://github.com/21cnbao/process-courses