奔跑吧LINUX内核* 入门篇

第八章进程管理

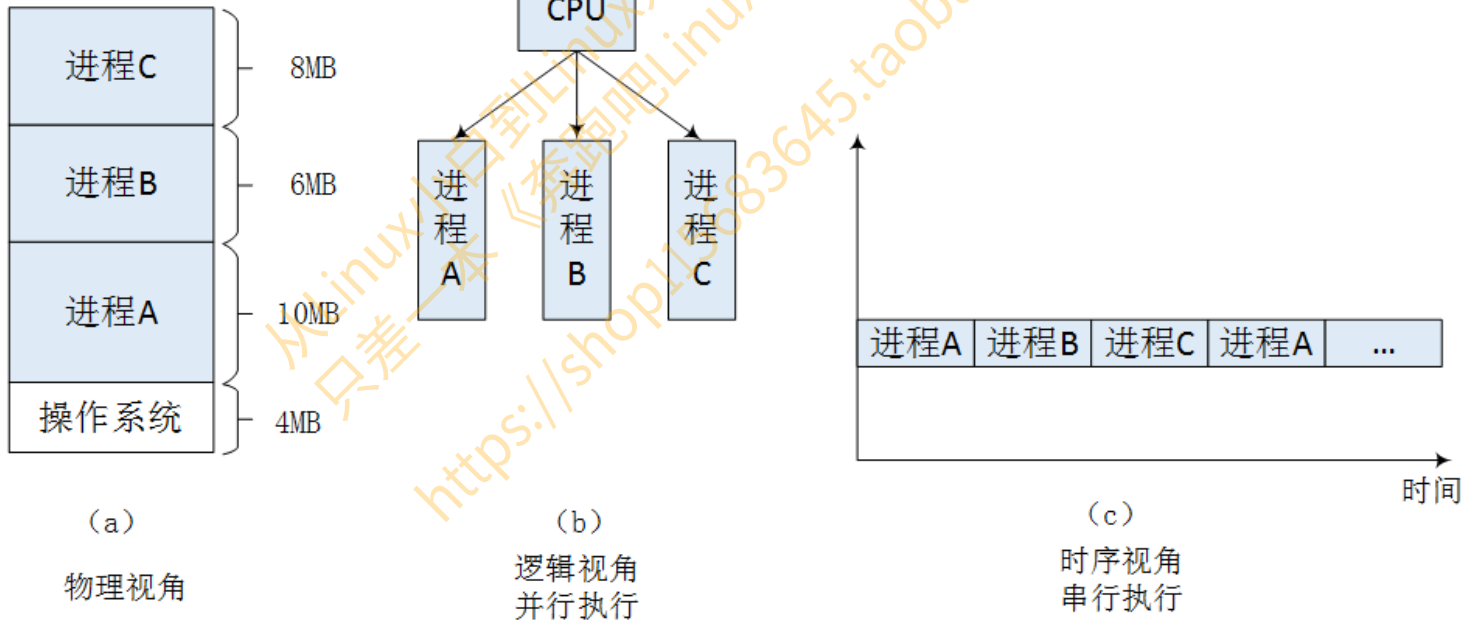
笨叔叔

目 录

进程简介

为什么要有进程

- 上个世纪50年代，工作（job）-> 进程（process）
- 进程 = 程序 + 执行



进程和程序总结

- 程序通常指的是完成特定任务的一系列指令集合或者指的是一个可执行文件
- 进程是一个有生命的个体，它不仅仅包含代码段数据段等信息，还有很多运行时需要的资源
- 进程是一段执行中的程序
- 进程是操作系统分配内存、CPU时间片等资源的基本单位。
- 进程是用来实现多进程并发执行的一个实体，实现对CPU的虚拟化

进程控制块PCB

- 对进程所必须拥有的资源做一个抽象，这个抽闲描述称为进程控制块PCB
- Linux 0.11内核的PCB

```c
struct task_struct {
/* these are hardcoded - don't touch */
	long state;	/* -1 unrunnable, 0 runnable, >0 stopped */
	long counter;
	long priority;
	long signal;
	struct sigaction sigaction[32];
	long blocked;	/* bitmap of masked signals */
/* various fields */
	int exit_code;
	unsigned long start_code,end_code,end_data,brk,start_stack;
	long pid,father,pgrp,session,leader;
	unsigned short uid,euid,suid;
	unsigned short gid,egid,sgid;
	long alarm;
	long utime,stime,cutime,cstime,start_time;
	unsigned short used_math;
/* file system info */
	int tty;		/* -1 if no tty, so it must be signed */
	unsigned short umask;
	struct m_inode * pwd;
	struct m_inode * root;
	struct m_inode * executable;
	unsigned long close_on_exec;
	struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
	struct desc_struct ldt[3];
/* tss for this task */
	struct tss_struct tss;
};
```

Linux 4.0内核

- Linux 4.0内核的task_struct数据结构，多达上百个成员

```
1278 struct task_struct {
1279         volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
1280         void *stack;
1281         atomic_t usage;
1282         unsigned int flags;     /* per process flags, defined below */
1283         unsigned int ptrace;
1284
1285 #ifdef CONFIG_SMP
1286         struct llist_node wake_entry;
1287         int on_cpu;
1288         struct task_struct *last_wakee;
1289         unsigned long wakee_flips;
1290         unsigned long wakee_flip_decay_ts;
1291
1292         int wake_cpu;
1293 #endif
1294         int on_rq;
1295
1296         int prio, static_prio, normal_prio;
1297         unsigned int rt_priority;
1298         const struct sched_class *sched_class;
1299         struct sched_entity se;
1300         struct sched_rt_entity rt;
1301 #ifdef CONFIG_CGROUP_SCHED
1302         struct task_group *sched_task_group;
1303 #endif
1304         struct sched_dl_entity dl;
1305
1306 #ifdef CONFIG_PREEMPT_NOTIFIERS
1307         /* list of struct preempt_notifier: */
1308         struct hlist_head preempt_notifiers;
```

进程属性相关信息

- state成员：用来记录进程的状态，
- pid成员：这个是进程唯一的进程标识符（Process Identifier
- flag成员：用来描述进程属性的一些标志位
- exit_code和exit_signal成员：用来存放进程退出值和终止信号，这样父进程可以知道子进程的退出原因。
- pdeath_signal成员：父进程消亡时发出的信号。
- comm成员：存放可执行程序的名称。
- real_cred和cred成员：用来存放进程的一些认证信息

调度相关的信息

- prio成员：保存着进程的动态优先级，是调度类考虑的优先级。
- static_prio成员：静态优先级。内核不存储nice值，取而代之的是static_prio。
- normal_prio成员：基于static_prio和调度策略计算出来的优先级。
- rt_priority成员：实时进程的优先级。
- sched_class成员：调度类。
- se成员：普通进程调度实体。
- rt成员：实时进程调度实体。
- dl成员：deadline进程调度实体。
- policy成员：进程的类型，比如普通进程还是实时进程。
- cpus_allowed成员：进程可以在哪几个CPU上运行。

进程之间的关系

- real_parent成员：指向当前进程的父进程的task_struct数据结构。
- children成员：指向当前进程的子进程的链表。
- sibling成员：指向当前进程的兄弟进程的链表。
- group_leader成员：进程组的组长。

内存管理和文件管理相关信息

- mm成员：指向进程所管理的内存的一个总的抽象的数据结构mm_struct。
- fs成员：保存一个指向文件系统信息的指针。
- files成员：保存一个指向进程的文件描述符表的指针。

进程控制块总结

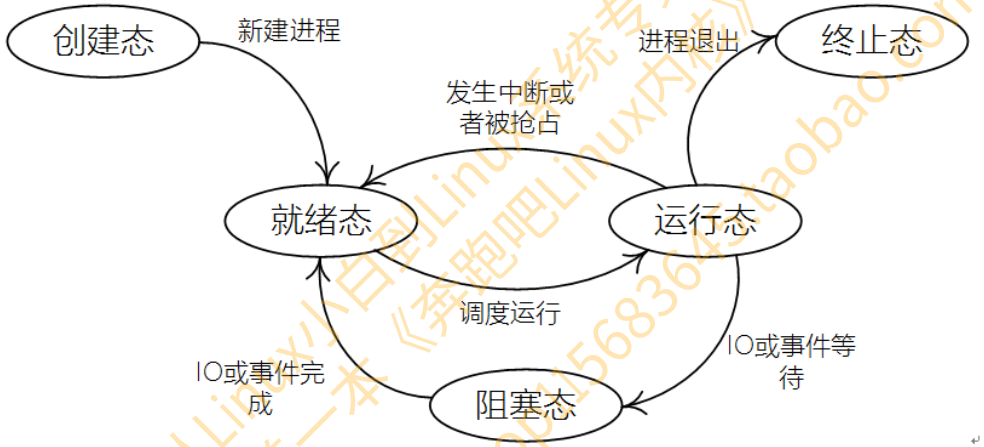
- 进程的运行状态
- 程序计数器
- CPU寄存器，保存上下文
- CPU调度信息
- 内存管理信息
- 统计信息
- 文件相关信息

进程生命周期

- 进程间经常需要相关沟通和交流，比如文本进程需要等待键盘的输入

- 进程状态包括：
  - ✓ 创建态：创建了新进程。
  - ✓ 就绪态：进程获得了可以运作的所有资源和准备条件。
  - ✓ 执行态：进程正在CPU中执行。
  - ✓ 阻塞态：进程因为等待某项资源而被暂时踢出了CPU。
  - ✓ 终止态：进程消亡。

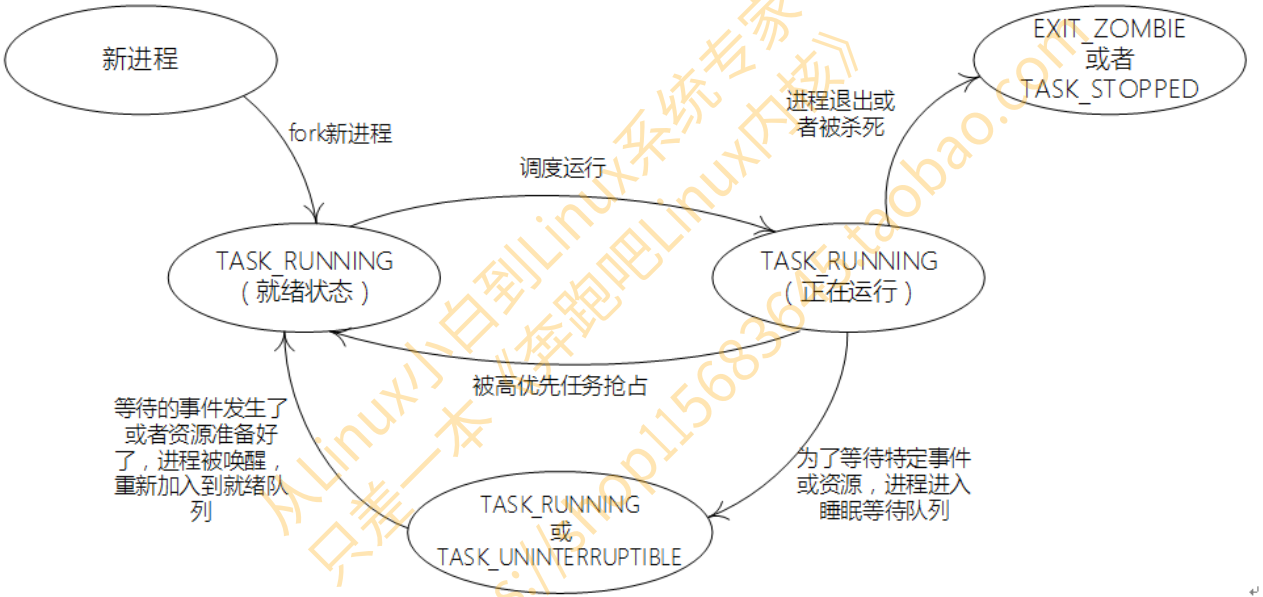
经典操作系统里的进程状态图



Linux内核里的进程状态

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
#define __TASK_TRACED           8
/* in tsk->exit_state */
#define EXIT_DEAD               16
#define EXIT_ZOMBIE             32
```

Linux内核的进程状态转换图



进程状态的设置

- 进程状态设置

  p->state = TASK_RUNNING;

- 内核提供的API：

```
#define set_task_state(tsk, state_value)        \
    set_mb((tsk)->state, (state_value))

#define set_current_state(state_value)          \
    set_mb(current->state, (state_value))
```

进程标识符PID

- 进程标识符PID （Process Identifier）：用来识别进程的唯一号码
- PID的类型是一个int类型，所以默认最大值是32768

- 为了循环使用PID编号，内核使用bitmap机制来管理当前已经分配的PID编号和空闲的PID编号

- getpid()系统调用返回当前进程的tgid值而不是线程的pid值
- 系统调用gettid会返回线程的PID

进程的家族关系

- 父进程，子进程，兄弟进程，祖孙进程

成员	描述
real_parent	指向创建了该进程A的进程描述符，如果进程A的父进程不存在了，则指向进程1（init进程）的进程描述符。
parent	指向进程的当前父进程，通常和real_parent一致。
children	所有的子进程都链成一个链表，这是链表头
sibling	所有兄弟进程都链接成一个链表，链表头在父进程的sibling成员中。

第一个进程

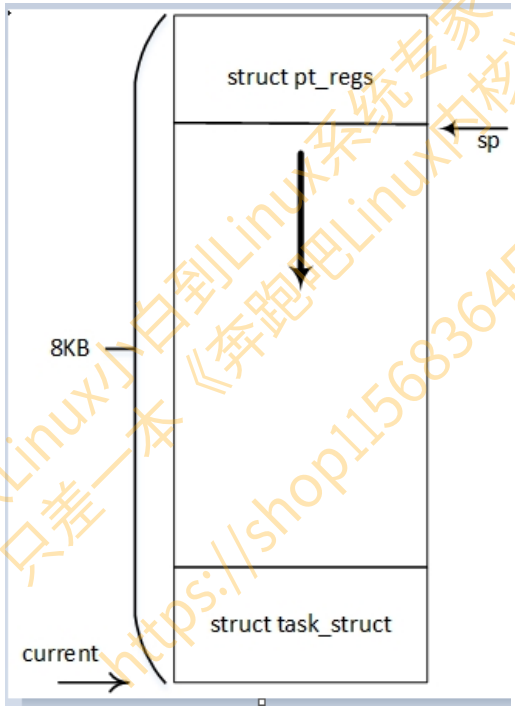
➢ 系统第一个进程是怎么初始化的?

```
/*
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x1fffff (=2MB)
 */
#define INIT_TASK(tsk)  \
{                                                                    \
        .state          = 0,                                         \
        .stack          = &init_thread_info,                         \
        .usage          = ATOMIC_INIT(2),                            \
        .flags          = PF_KTHREAD,                                \
        .prio           = MAX_PRIO-20,                               \
        .static_prio    = MAX_PRIO-20,                               \
        .normal_prio    = MAX_PRIO-20,                               \
        .policy         = SCHED_NORMAL,                              \
        .cpus_allowed   = CPU_MASK_ALL,                              \
        .nr_cpus_allowed= NR_CPUS,                                   \
        .mm             = NULL,                                      \
        .active_mm      = &init_mm,                                  \
        .restart_block = {                                           \
                .fn = do_no_restart_syscall,                         \
        },                                                           \
        .se             = {                                          \
                .group_node     = LIST_HEAD_INIT(tsk.se.group_node), \
        },                                                           \
        .rt             = {                                          \
                .run_list       = LIST_HEAD_INIT(tsk.rt.run_list),   \
                .time_slice     = RR_TIMESLICE,                      \
        },                                                           \
        .tasks          = LIST_HEAD_INIT(tsk.tasks),                 \
        INIT_PUSHABLE_TASKS(tsk)                                     \
```

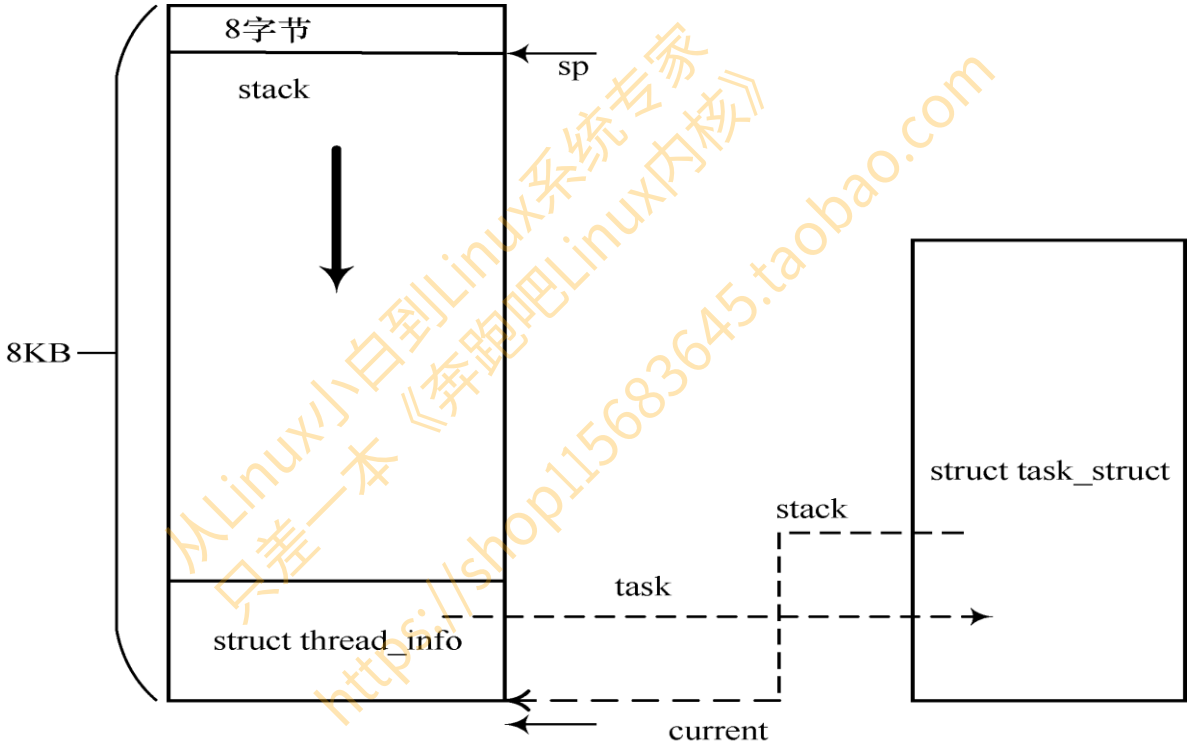
include/linux/init_task.h

current

- 如何获取当前进程的task_struct数据结构?



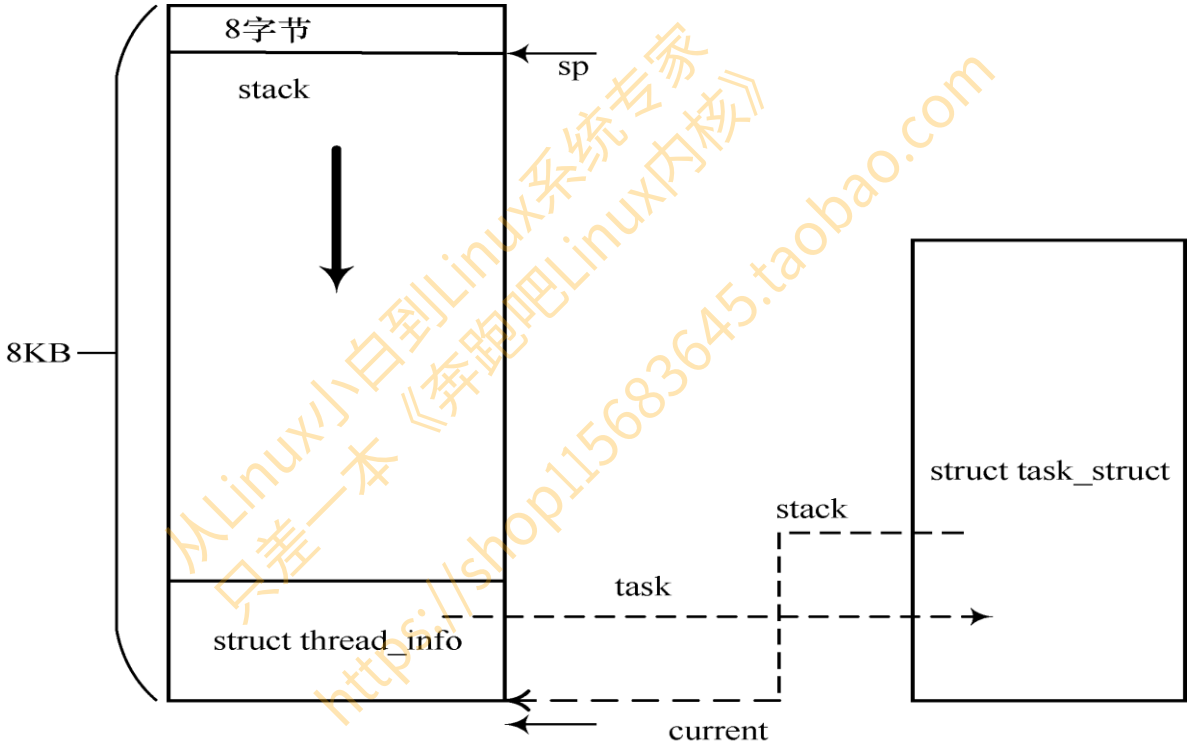
Linux内核里内核栈和task_struct 之间的关系



task_struct 和task_thread_info

```
47 /*
48  * low level task data that entry.S needs immediate access to.
49  * __switch_to() assumes cpu_context follows immediately after cpu_domain.
50  */
51 struct thread_info {
52         unsigned long           flags;          /* low level flags */
53         int                     preempt_count;  /* 0 => preemptable, <0 => bug */
54         mm_segment_t            addr_limit;     /* address limit */
55         struct task_struct      *task;          /* main task structure */
56         struct exec_domain      *exec_domain;   /* execution domain */
57         __u32                   cpu;            /* cpu */
58         __u32                   cpu_domain;     /* cpu domain */
59         struct cpu_context_save cpu_context;    /* cpu context */
60         __u32                   syscall;        /* syscall number */
61         __u8                    used_cp[16];    /* thread used copro */
62         unsigned long           tp_value[2];    /* TLS registers */
63 #ifdef CONFIG_CRUNCH
64         struct crunch_state     crunchstate;
65 #endif
66         union fp_state          fpstate __attribute__((aligned(8)));
67         union vfp_state         vfpstate;
68 #ifdef CONFIG_ARM_THUMBEE
69         unsigned long           thumbee_state;  /* ThumbEE Handler Base register */
70 #endif
71 };
```

Linux内核里核栈和task_struct 之间的关系



小结

- 了解什么是进程，为啥需要有进程
- 进程和程序的区别
- 进程控制块都包含哪些内容
- Linux 0.11内核和Linux 4.0内核里控制块里都包含哪些主要的成员？
- Linux内核里进程的生命周期是怎么样的？
- 进程的PID
- 进程的家族关系
- 系统第一个进程是怎么初始化的、
- 如何去获取当前进程的task-struct数据结构？

进程的创建和终止

进程是怎么诞生的?

- 我们写一个test程序，在shell里运行，那会不会创建一个新进程?
- Linux 0.11内核里的fork实现。

```
207 .align 2
208 sys_fork:
209         call find_empty_process
210         testl %eax,%eax
211         js 1f
212         push %gs
213         pushl %esi
214         pushl %edi
215         pushl %ebp
216         pushl %eax
217         call copy_process
218         addl $20,%esp
219 1:      ret
220
```

Linux 0.11里的copy_process()函数

```c
/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in it's entirety.
 */
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
		long ebx,long ecx,long edx,
		long fs,long es,long ds,
		long eip,long cs,long eflags,long esp,long ss)
{
	struct task_struct *p;
	int i;
	struct file *f;

	p = (struct task_struct *) get_free_page();
	if (!p)
		return -EAGAIN;
	task[nr] = p;
	*p = *current;	/* NOTE! this doesn't copy the supervisor stack */
	p->state = TASK_UNINTERRUPTIBLE;
	p->pid = last_pid;
	p->father = current->pid;
	p->counter = p->priority;
	p->signal = 0;
	p->alarm = 0;
	p->leader = 0;		/* process leadership doesn't inherit */
	p->utime = p->stime = 0;
	p->cutime = p->cstime = 0;
	p->start_time = jiffies;
	p->tss.back_link = 0;
	p->tss.esp0 = PAGE_SIZE + (long) p;
	p->tss.ss0 = 0x10;
	p->tss.eip = eip;
	p->tss.eflags = eflags;
```

Linux 0.11内核fork进程实现要点

- 每个进程需要有一个内核栈。不管是4KB还是8KB。这个内核栈需要承载两样东西，一个是task-struct数据结构本身，另外一个内核栈
- 继承父进程的task-struct数据结构，然后进行微调
- 设置进程的栈。
- 拷贝父进程的进程地址空间给子进程

Linux 内核的创建进程的API

```
fork实现：
    do_fork(SIGCHLD, 0, 0, NULL, NULL);

vfork实现：
    do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0, 0, NULL, NULL);

clone实现：
    do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr);

内核线程：
    do_fork(flags|CLONE_VM|CLONE_UNTRACED, (unsigned long)fn, (unsigned
long)arg, NULL, NULL);
```

Fork函数

- 子进程会从父进程那里继承了整个进程地址空间，包括进程上下文、进程堆栈、内存信息、打开的文件描述符、进程优先级、根目录、资源限制、控制终端等。

```
FORK(2)                                  Linux Programmer's Manual                                  FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a  new  process by duplicating the calling process.  The new process is referred to as the child process.  The
       calling process is referred to as the parent process.

       The child process and the parent process run in separate memory spaces.  At the time of fork() both memory spaces have the same
       content.   Memory  writes,  file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect
       the other.
```

- 返回值：
  - ✓ 父进程会返回子进程的PID
  - ✓ 子进程会返回0

vfork函数

- vfork：父进程会一直阻塞，直到子进程调用exit()或者exec()为止。

```
#include <sys/types.h>
#include <unistd.h>

pid_t vfork(void);
```

- vfork()函数通过系统调用进入到Linux内核，然后通过do_fork()函数来实现。

```
SYSCALL_DEFINE0(vfork)
{
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
                0, NULL, NULL);
}
```

clone函数

- clone()函数用来创建用户线程

```
/* glibc库的封装*/
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...);

/* 原始的系统调用*/
long clone(unsigned long flags, void *child_stack,
           void *ptid, void *ctid,
           struct pt_regs *regs);
```

- clone()函数通过系统调用进入到Linux内核，然后通过do_fork()函数来实现。

```
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
        int __user *, parent_tidptr,
        int __user *, child_tidptr,
        int, tls_val)
{
    return do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr);
}
```

内核线程创建

- 内核线程创建API：

```
kthread_create(threadfn, data, namefmt, arg...)
kthread_run(threadfn, data, namefmt, ...)
```

do_fork函数

- do_fork函数原型：

```
[kernel/fork.c]

long do_fork(unsigned long clone_flags, unsigned long stack_start, unsigned
long stack_size, int __user *parent_tidptr, int __user *child_tidptr)
```

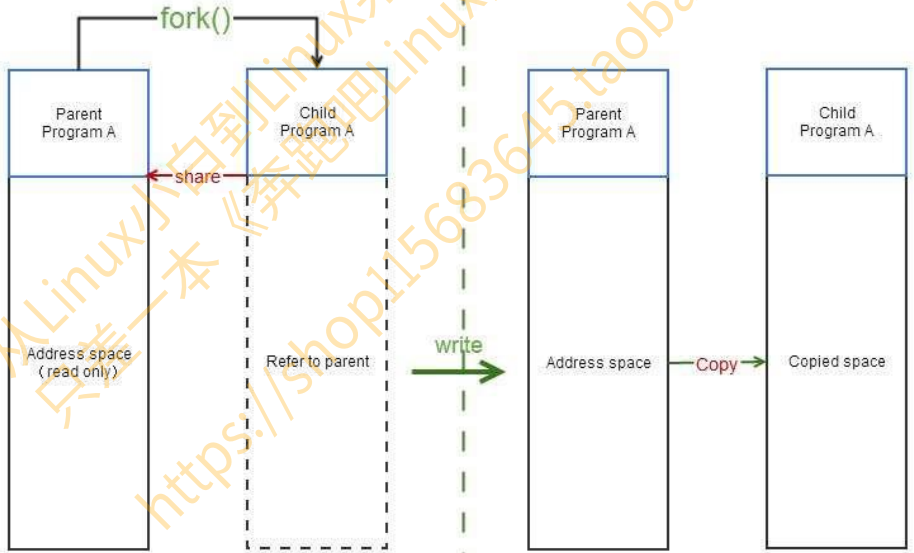
- do_fork()函数有5个参数。
  - ✓ clone_flags：创建进程的标志位集合。
  - ✓ stack_start：用户态栈的起始地址。
  - ✓ stack_size：用户态栈的大小，通常设置为0。
  - ✓ parent_tidptr和child_tidptr：指向用户空间中地址的两个指针，分别指向父子进程的PID。

常见clone标志位

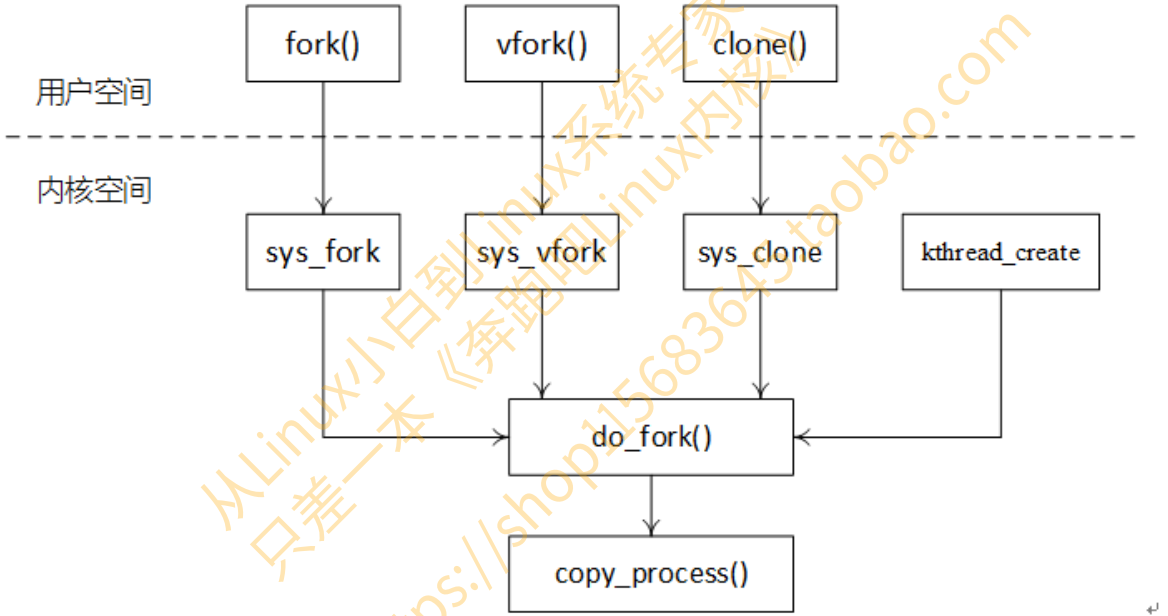
参数标志	含义
CLONE_VM	父子进程共享进程地址空间
CLONE_FS	父子进程共享文件系统信息
CLONE_FILES	父子进程共享打开的文件
CLONE_SIGHAND	父子进程共享信号处理函数以及被阻断的信号
CLONE_PTRACE	父进程被跟踪（ptrace），子进程也会被跟踪。
CLONE_VFORK	在创建子进程时启用Linux内核的完成机制（completion）。wait_for_completion()会使父进程进入睡眠等待，直到子进程调用execve()或exit()释放虚拟内存资源。
CLONE_PARENT	指定子进程和父进程拥有同一个父进程
CLONE_THREAD	父子进程在同一个线程组里
CLONE_NEWNS	为子进程创建新的命名空间
CLONE_SYSVSEM	父子进程共享System V等语义
CLONE_SETTLS	为子进程创建新的TLS (thread local storage)
CLONE_PARENT_SETTID	设置父进程的TID

写时复制技术

- 写时复制技术（copy on write），简称CoW
- 写时复制技术就是父进程在创建子进程的时候，不需要拷贝进程地址空间的内容给子进程，只需要拷贝父进程的进程地址空间的页表给子进程即可，这样父子进程就共享了相同的进程地址空间。



fork, vfork, clone和do_fork之间的关系



进程的终止

- 进程主动终止主要有如下两个途径：
  - ✓ 从main()函数返回，链接程序会自动添加对exit()系统调用。
  - ✓ 主动调用exit()系统调用。

- 进程被动终止主要有如下三个途径：
  - ✓ 进程收到一个自己不能处理的信号。
  - ✓ 进程在内核态执行的时候产生了一个异常。
  - ✓ 进程收到SIGKILL等终止信号。

- exit()系统调用把退出码转换成内核要求的格式并且调用do_exit()函数来处理

```
SYSCALL_DEFINE1(exit, int, error_code)
{
    do_exit((error_code&0xff)<<8);
}
```

Linux内核中线程的实现

- 进程是资源管理的最小单位，线程是程序执行的最小单位
- Linux线程的实现：在内核里和普通进程是一样的，都是task_struct数据结构来描述。称为轻量级进程。
- Pthread库

0号进程和1号进程

- 进程0是指的的Linux内核初始化阶段从无到有创建的一个内核线程
  - ✓ 进程0
  - ✓ idle进程
  - ✓ swapper进程

```
[init/init_task.c]
struct task_struct init_task = INIT_TASK(init_task);
[include/linux/init_task.h]
#define INIT_TASK(tsk)    \
{                                   \
    .state          = 0,                    \
    .signal    = &init_signals,                 \
    ...
}
```

- Linux内核初始化函数start_kernel()在初始化完内核所需要的所有数据结构之后会创建另外一个内核线程，这个内核线程就是进程1或者叫做init进程

僵尸进程和托孤进程

- 当一个进程通过exit()系统调用已经终止之后，进程处于僵尸状态（ZOMBIE）。
- 当父进程通过调用wait()系统调用来获取已终结的子进程的信息之后，内核才会去释放子进程的task_struct数据结构。

```
asmlinkage long sys_wait4(pid_t pid, int __user *stat_addr,
                          int options, struct rusage __user *ru);
asmlinkage long sys_waitid(int which, pid_t pid,
                           struct siginfo __user *infop,
                           int options, struct rusage __user *ru);
asmlinkage long sys_waitpid(pid_t pid, int __user *stat_addr, int options);
```

- 啥是托孤进程?
- 　如果父进程先于子进程消亡，那么子进程就变成孤儿进程

僵尸进程和托孤进程

- 进程睡眠 – wait_event() 把当前进程加入到等待队列wq中
- 等待队列头（wait_queue_head_t）

进程的唤醒

- wake_up()函数：唤醒等待队列中所有的进程

- 面试题目：wakeup函数唤醒的是一个进程还是这个等待队列中的所有进程?

进程调度

进程属性之CPU消耗型和IO消耗型

- CPU消耗型：CPU-Bound。大部分时间都用在执行代码上，一直占用CPU
- IO消耗型：IO-Bound。大部分时间用来提交IO请求或者等待IO请求，很少占用CPU。
- 通常IO消耗型的进程优先级要高于CPU消耗型。

task_struct中的关于优先级的几个成员

- static_prio是静态优先级，在进程启动时分配
- normal_prio是基于static_prio和调度策略计算出来的优先级，在创建进程时会继承父进程的normal_prio
- prio保存着进程的动态优先级，是调度类考虑的优先级，有些情况下需要暂时提高进程优先级，例如实时互斥量等。rt_priority是实时进程的优先级。

```
#define MAX_USER_RT_PRIO        100
#define MAX_RT_PRIO             MAX_USER_RT_PRIO

#define MAX_PRIO                (MAX_RT_PRIO + NICE_WIDTH)
#define DEFAULT_PRIO            (MAX_RT_PRIO + NICE_WIDTH / 2)

/*
 * Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice)      ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio)      ((prio) - DEFAULT_PRIO)

```

include/linux/sched/prio.h

Linux 0.11内核调度算法

```
104 void schedule(void)
105 {
106         int i,next,c;
107         struct task_struct ** p;
108
109 /* this is the scheduler proper: */
110
111         while (1) {
112                 c = -1;
113                 next = 0;
114                 i = NR_TASKS;
115                 p = &task[NR_TASKS];
116                 while (--i) {
117                         if (!*--p)
118                                 continue;
119                         if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
120                                 c = (*p)->counter, next = i;
121                 }
122                 if (c) break;
123                 for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
124                         if (*p)
125                                 (*p)->counter = ((*p)->counter >> 1) +
126                                         (*p)->priority;
127         }
128         switch_to(next);
129 }
130
```

不到20行的调度算法，经典

Linux 0.11内核的时间片

```
175 .align 2
176 timer_interrupt:
177         push %ds                # save ds,es and put kernel data space
178         push %es                # into them. %fs is used by _system_call
179         push %fs
180         pushl %edx              # we save %eax,%ecx,%edx as gcc doesn't
181         pushl %ecx              # save those across function calls. %ebx
182         pushl %ebx              # is saved as we use that in ret_sys_call
183         pushl %eax
184         movl $0x10,%eax
185         mov %ax,%ds
186         mov %ax,%es
187         movl $0x17,%eax
188         mov %ax,%fs
189         incl jiffies
190         movb $0x20,%al          # EOI to interrupt controller #1
191         outb %al,$0x20
192         movl CS(%esp),%eax
193         andl $3,%eax            # %eax is CPL (0 or 3, 0=supervisor)
194         pushl %eax
195         call do_timer           # 'do_timer(long CPL)' does everything from
196         addl $4,%esp            # task switching to accounting ...
197         jmp ret_from_sys_call
198
```

时钟中断来了

```
305 void do_timer(long cpl)
306 {
307         if ((--current->counter)>0) return;
308         current->counter=0;
309         if (!cpl) return;
310         schedule();
311 }
312
```

递减当前进程的时间片，当前进程时间片用完之后，会发生调度

时间片

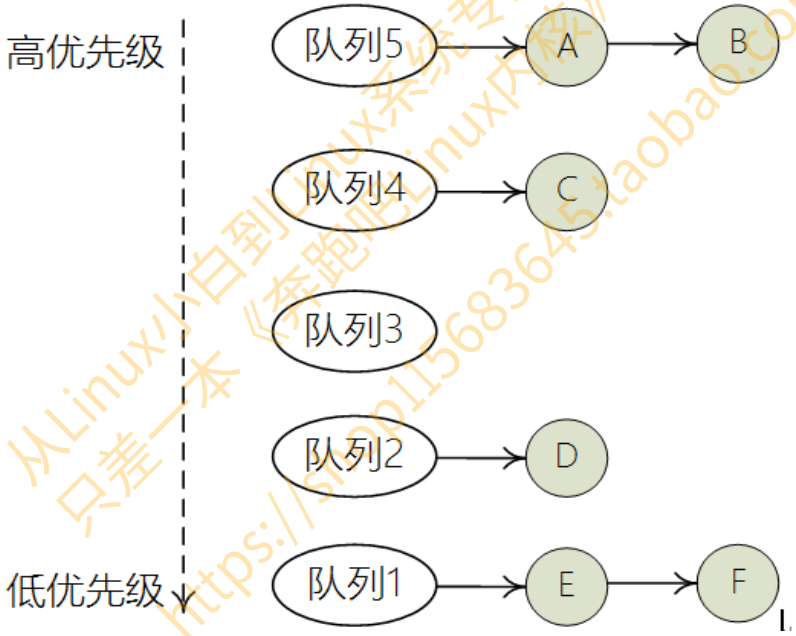
- 时间片 time slice -表示进程在被抢占和调度之前所能持续运行的时间片
- Linux 0.11内核里的 task_struct->counter就是一个时间片
- 时间片随着时钟中断的到来会 逐步递减。
- 当时间片用完了，应该选择下一个进程运行
- 当系统所有的进程的时间片都用完了，那该怎么办?

O(n)调度器的缺点

- Linux 0.11 ~ Linux 2.4实现的调度器算法是O(n)
- O(n)：选择下一个进程时间复杂度为O(n)
- 可扩展性差

经典多级调度算法Multi-level Feedback Queue

- 1962年Corbato提出多级反馈队列算法。（Multi-level feedback Queue）

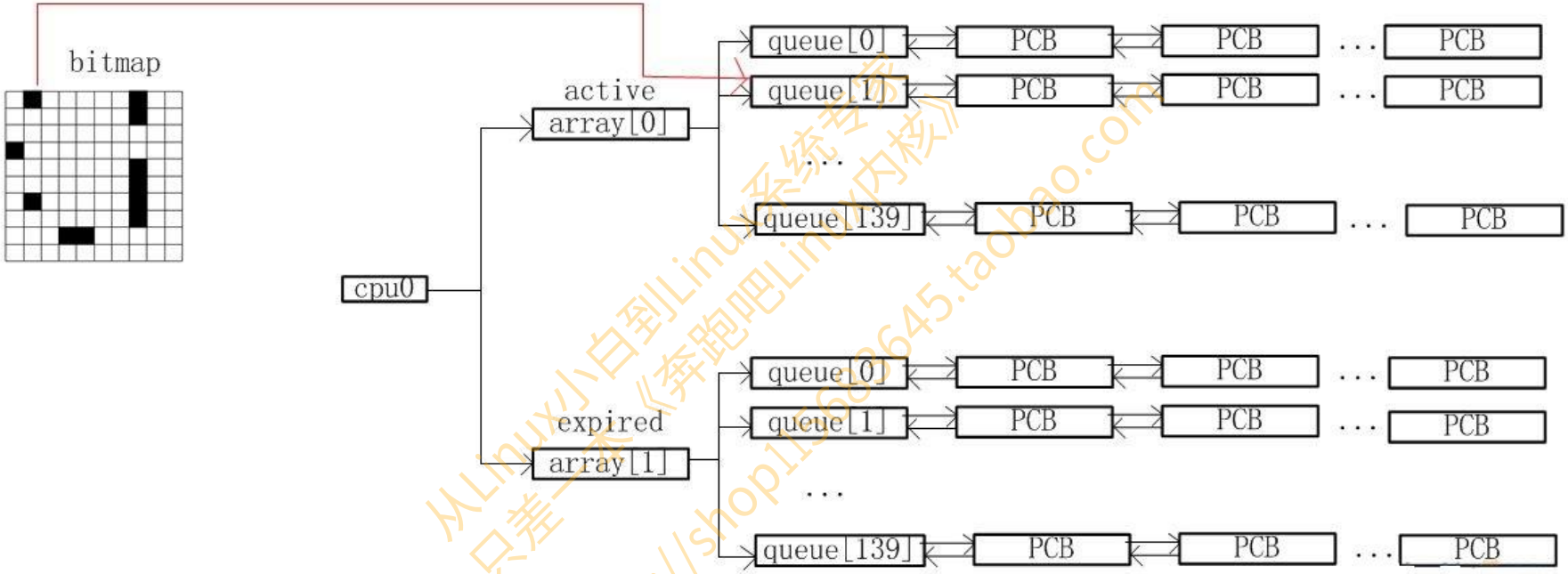


多级反馈队列算法的精髓

- 规则1：如果进程A的优先级大于进程B的优先级，那么调度器选择进程A
- 规则2：如果进程A和进程B优先级一样，那么它们同属一个队列里，使用轮转调度算法来选择
- 规整3：当一个新进程进入调度器时，把它放入到最高优先级的队列里
- 规整4a：当一个进程吃满了时间片，说明这是一个CPU消耗型的进程，那么需要把优先级降一级，从高优先级队列中迁移到低一级的队列里。
- 规整4b：当一个进程在时间片还没有结束之前放弃CPU，那说明是一个IO消耗型的进程，那么优先级保持不变，维持原来的高优先级。

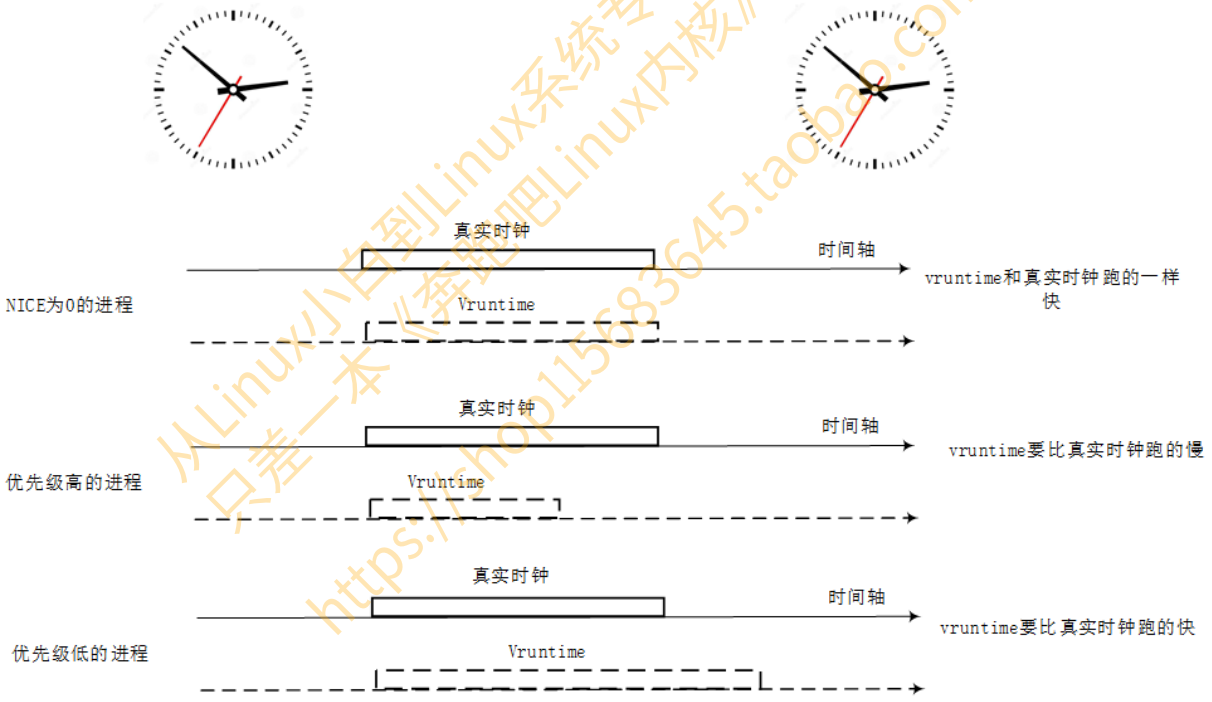
Linux 2.6内核的O(1)调度算法

- 红帽公司Ingo Molnar设计的O(1)调度算法，该调度算法的核心思想是基于多级反馈队列算法。
- 每个CPU有一个就绪队列
- 就绪队列有多个链表组成，每个优先级一个链表



CFS调度器

- 虚拟时钟vruntime：每个进程的虚拟时间是实际运行时间相对NICE值为0的权重的比例值



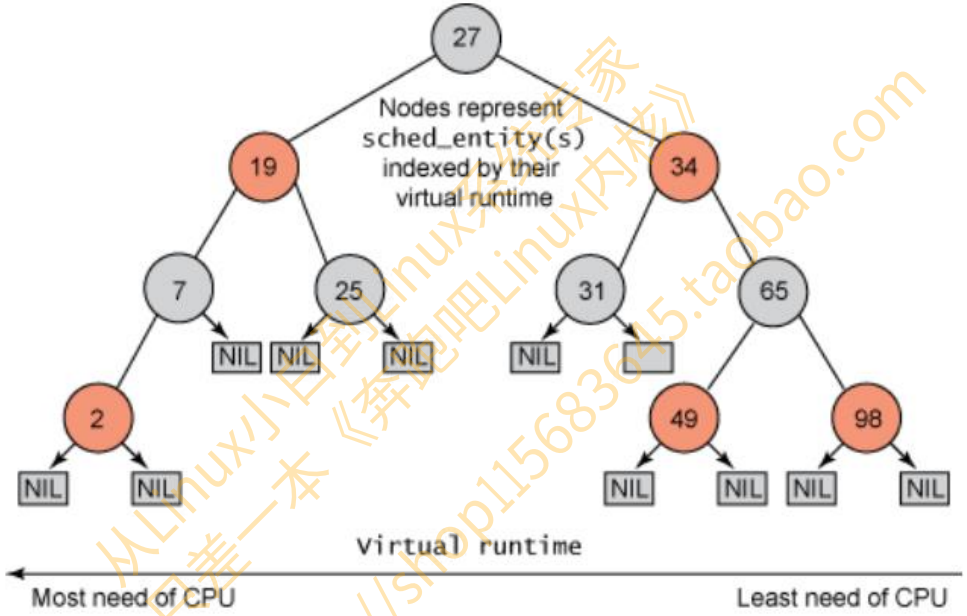
CFS调度器

- vruntime计算公式：

$$vruntime = \frac{delta\_exec * nice\_0\_weight}{weight}$$

- vruntime表示进程虚拟的运行时间
- delta_exec表示实际运行时间
- nice_0_weight表示nice为0的权重值
- weight表示该进程的权重值

CFS调度器：选择下一个进程



CFS调度器总选择红黑树最左边的叶子节点作为下一个运行的进程

CFS调度器：tick时钟中断

- 更新红黑树的信息和vruntime
- 判断当前进程是否需要被调度 need_reschedule
- CFS抛弃了传统的时间片概念

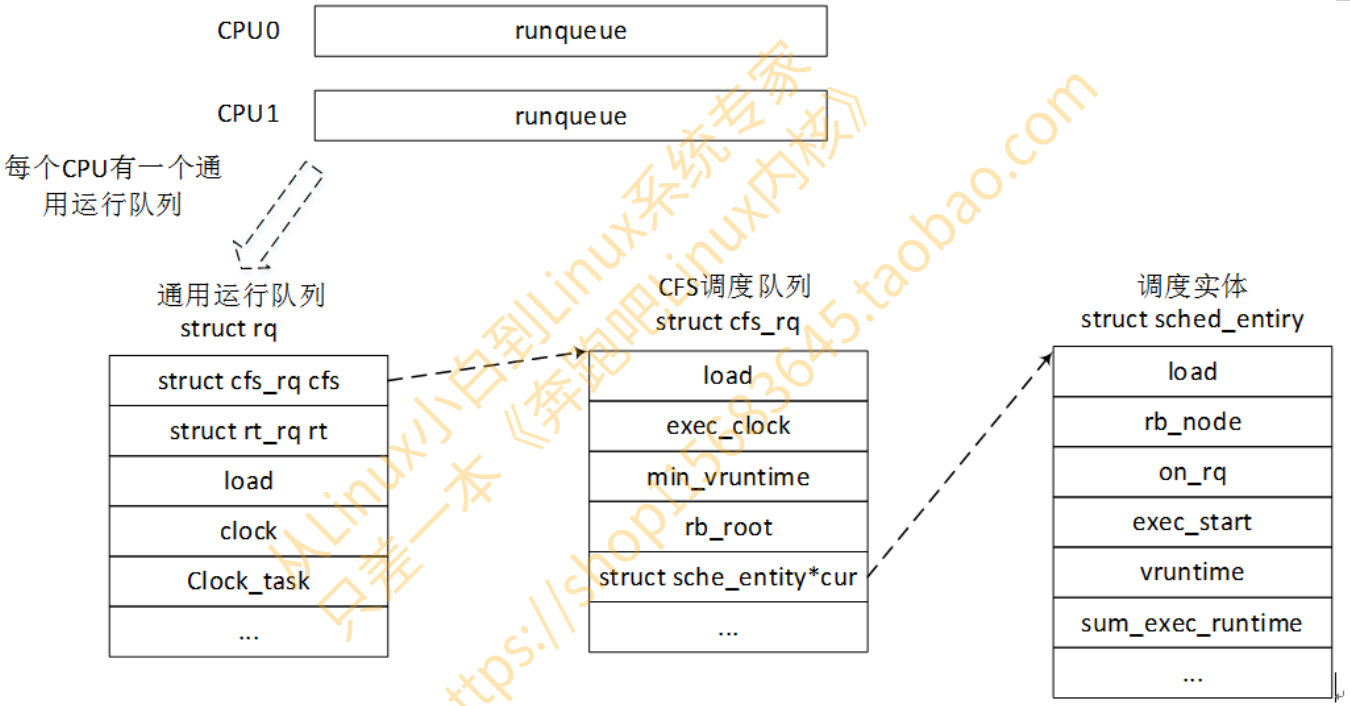
调度器类

- Linux内核实现多套的调度算法，用调度类来实现
  - ✓ SCHED_FAIR：普通进程，使用CFS调度算法
  - ✓ SCHED_RT：实时进程
  - ✓ SCHED_ DEADLINE：deadline进程
  - ✓ SCHED_IDLE：idle进程

- sched_getscheduler()—用户空间程序系统调用API设置和获取内核调度器的调度策略和参数。

```
33 /*
34  * Scheduling policies
35  */
36 #define SCHED_NORMAL            0
37 #define SCHED_FIFO              1
38 #define SCHED_RR                2
39 #define SCHED_BATCH             3
40 /* SCHED_ISO: reserved but not implemented yet */
41 #define SCHED_IDLE              5
42 #define SCHED_DEADLINE          6
```

调度器主要数据结构总结



进程调度时机点

- 调度时机：
  - ✓ （1）阻塞操作：互斥量（mutex）、信号量（semaphore）、等待队列（waitqueue）等。
  - ✓ （2）在中断返回前和系统调用返回用户空间时，去检查TIF_NEED_RESCHED标志位以判断是否需要调度。
  - ✓ （3）将要被唤醒的进程（Wakeups）

- 唤醒进程什么时候被调度？
  - ✓ 内核支持抢占
  - ✓ 内核不支持抢占

进程切换switch_to()

- 调度器的职责是选择下一个进程来运行，而进程切换就是负责具体落实该进程的执行
- 切换的本质：
  - ✓ 保存上一个进程的上下文
  - ✓ 装载下一个进程的上下文到CPU

```
33 struct cpu_context_save {
34         __u32   r4;
35         __u32   r5;
36         __u32   r6;
37         __u32   r7;
38         __u32   r8;
39         __u32   r9;
40         __u32   sl;
41         __u32   fp;
42         __u32   sp;
43         __u32   pc;
44         __u32   extra[2];
45 };
```

保存进程切换上下文的数据结构
arch/arm/include/asm/thread_info.h

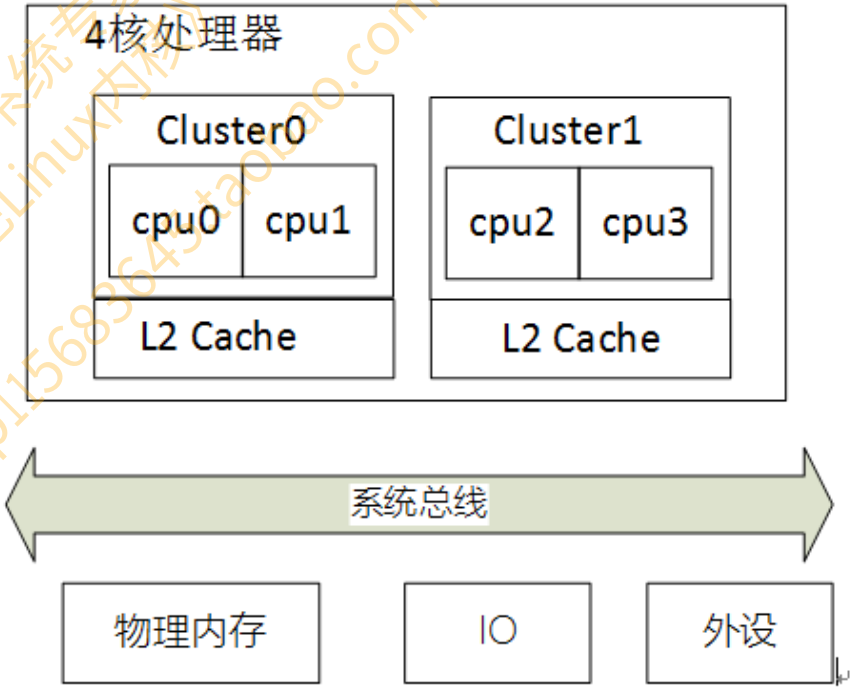
```
.globl cpu_switch_to
cpu_switch_to:
        stmia r0!, {r4 - sl, fp, sp, lr}  @store regs on stack
        ldmia r1, {r4 - sl, fp, sp, pc}   @load all regs saved previously
```

进程切换核心代码

多核调度

SMP处理器

- SMP（Symmetrical Multi-Processing）：对称多处理器技术
  - ✓ 相同的CPU核心
  - ✓ 相同的运行频率
  - ✓ 访问同一条总线
  - ✓ 访问相关的内存和外设



调度域和调度组

- CPU物理属性
  - ✓ 超线程
  - ✓ 多核
  - ✓ 处理器

- struct sched_domain_topology_level来描述CPU的层次关系
- struct sched_domain数据结构来表示调度域
- 使用struct sched_group数据结构来表示调度组

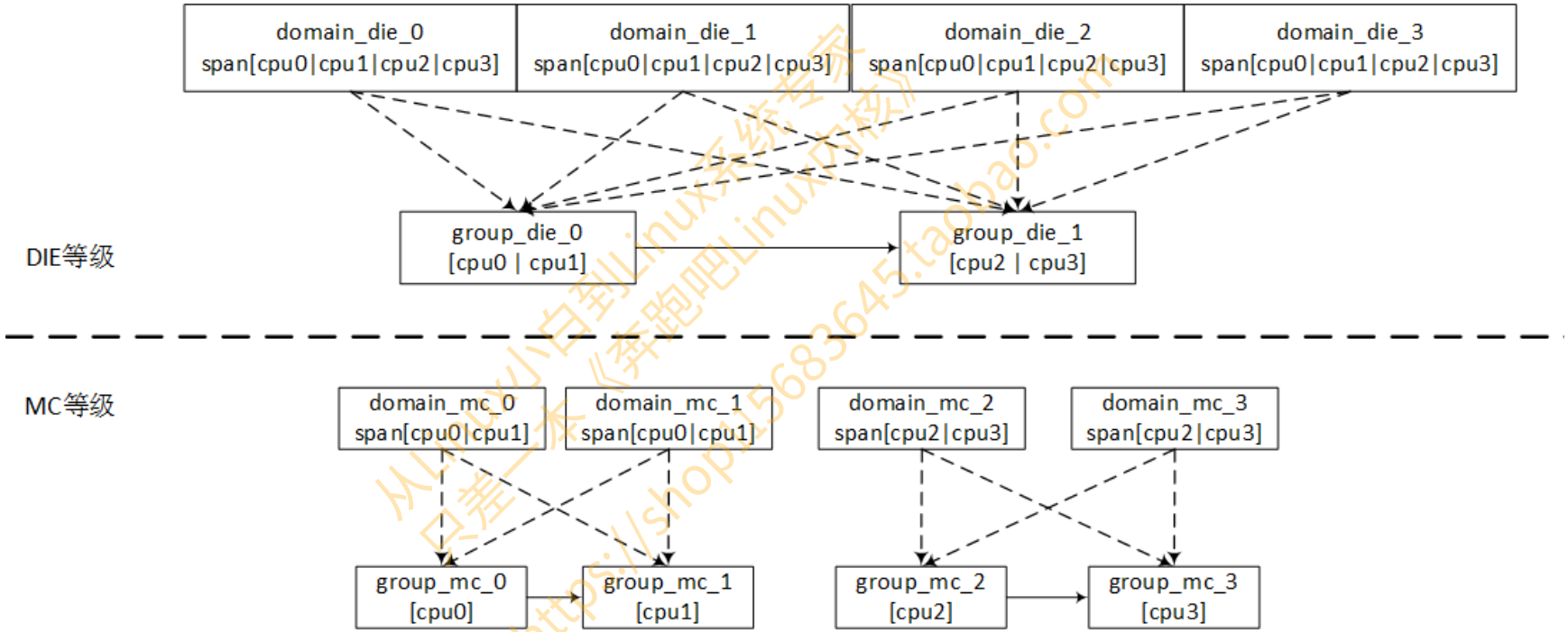
CPU 分类	Linux 内核分类	说明
超线程（SMT, Simultaneous MultiThreading）	CONFIG_SCHED_SMT	一个物理核心可以有两个执行线程，被称为超线程技术。超线程使用相同CPU资源且共享L1 cache，迁移进程不会影响Cache利用率
多核（MC）	CONFIG_SCHED_MC	每个物理核心独享L1 cache，多个物理核心可以组成一个cluster，cluster里的CPU共享L2 cache
处理器（SoC）	内核称为DIE	SoC级别

Linux内核里默认的调度层次关系

```
6272 /*
6273  * Topology list, bottom-up.
6274  */
6275 static struct sched_domain_topology_level default_topology[] = {
6276 #ifdef CONFIG_SCHED_SMT
6277         { cpu_smt_mask, cpu_smt_flags, SD_INIT_NAME(SMT) },
6278 #endif
6279 #ifdef CONFIG_SCHED_MC
6280         { cpu_coregroup_mask, cpu_core_flags, SD_INIT_NAME(MC) },
6281 #endif
6282         { cpu_cpu_mask, SD_INIT_NAME(DIE) },
6283         { NULL, },
6284 };
6285
```

kernel/sched/core.c

调度域和调度组之间的关系图



负载计算

- 在SMP负载均衡算法里拿什么去判断一个CPU是否忙或者闲呢?
- PELT（Pre-entity Load Tracking）：不仅考虑权重，而且跟踪每个调度实体的负载情况

PETL算法

- 统计多个实际的PI周期，并使用一个衰减系数来计算过去的PI周期对负载的贡献。
- 计算公式：

$$L = L0 + L1 * y + L2 * y^2 + L3 * y^3 + \cdots + L32 * y^{32} + \cdots$$

- L0~Ln：表示第n个周期的负载
- y是一个预先选定好的衰减系数

一个调度实体负载计算公式

- 最终计算公式：

$$load\_avg\_contrib = \frac{runnable\_avg\_sum * weight}{runnable\_avg\_period}$$

  - ✓ load_avg_contrib：表示进程最终的负载贡献值
  - ✓ weight：调度实体的权重值
  - ✓ runnable_avg_sum：调度实体的可运行状态下的总衰减累加时间
  - ✓ runnable_avg_period：调度实体在调度器中的总衰减累加时间

- 一个CPU上负载 = 所有进程上的load_avg_contrib的总和

SMP负载均衡算法

- SMP负载均衡机制从注册软中断开始，每次系统处理调度tick时会检查当前是否需要处理SMP负载均衡。

- rebalance_domains()函数是负载均衡的核心入口

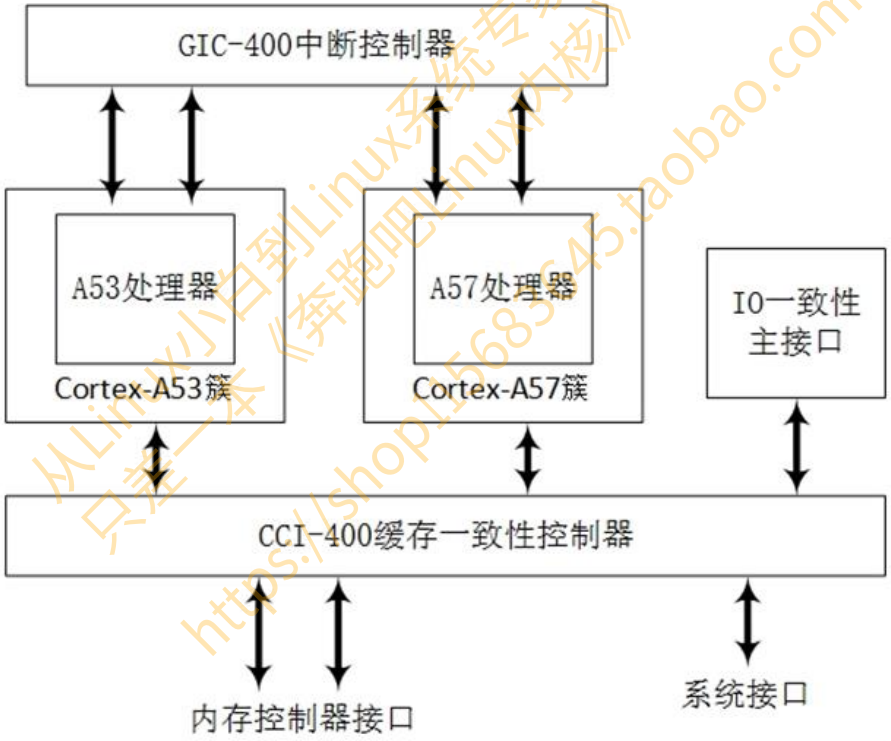
SMP负载均衡算法

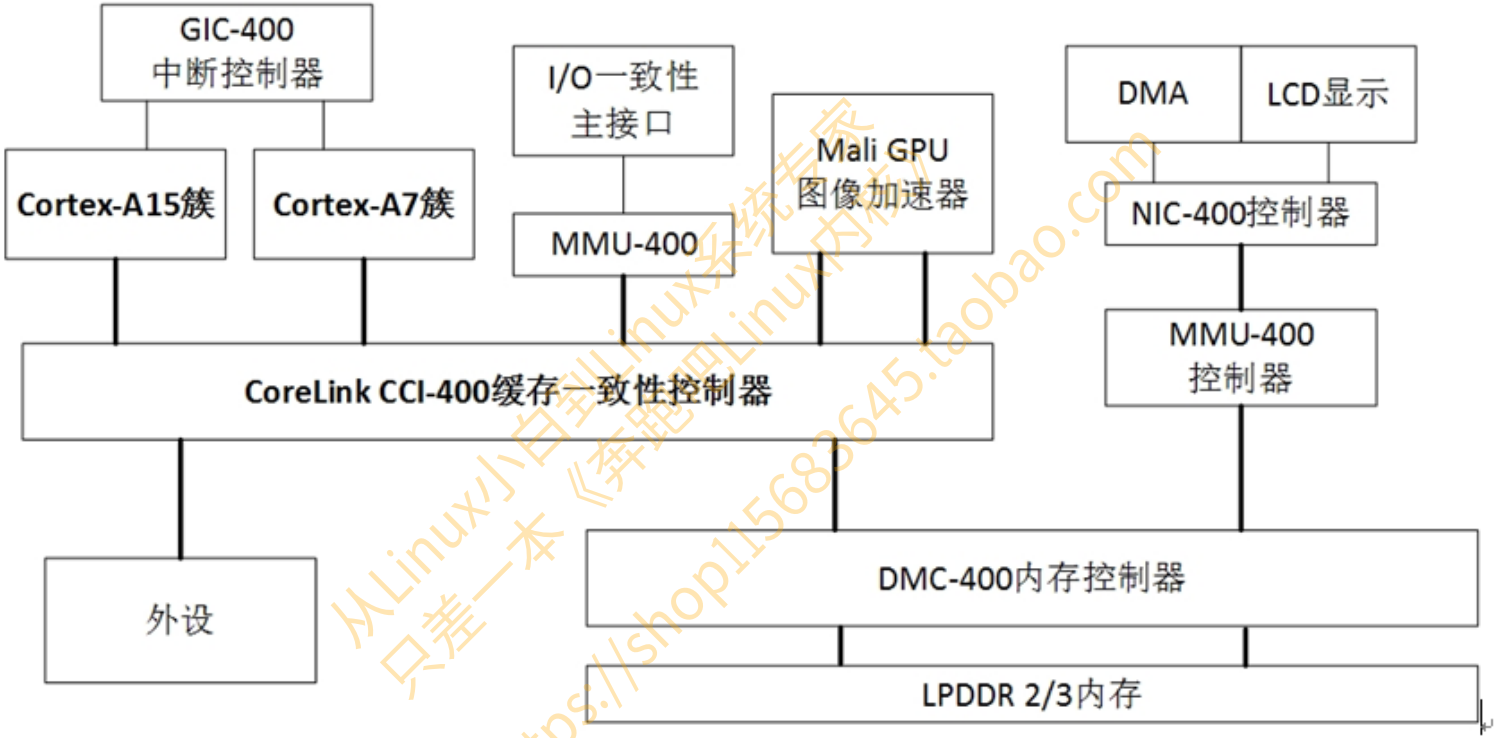
- **load_balance()函数主要流程总结如下：**
  - ✓ 负载均衡以当前CPU开始，由下至上地遍历调度域，从最底层的调度域开始做负载均衡。
  - ✓ 允许做负载均衡的首要条件是当前CPU是该调度域中第一个CPU，或者当前CPU是idle CPU。
  - ✓ 在调度域中查找最繁忙的调度组，更新调度域和调度组的相关信息，最后计算出该调度域的不均衡负载值（imbalance）。
  - ✓ 在最繁忙的调度组中找出最繁忙的CPU，然后把繁忙CPU中的进程迁移到当前CPU上，迁移的负载量为不均衡负载值。

ARM大小核调度

大小核

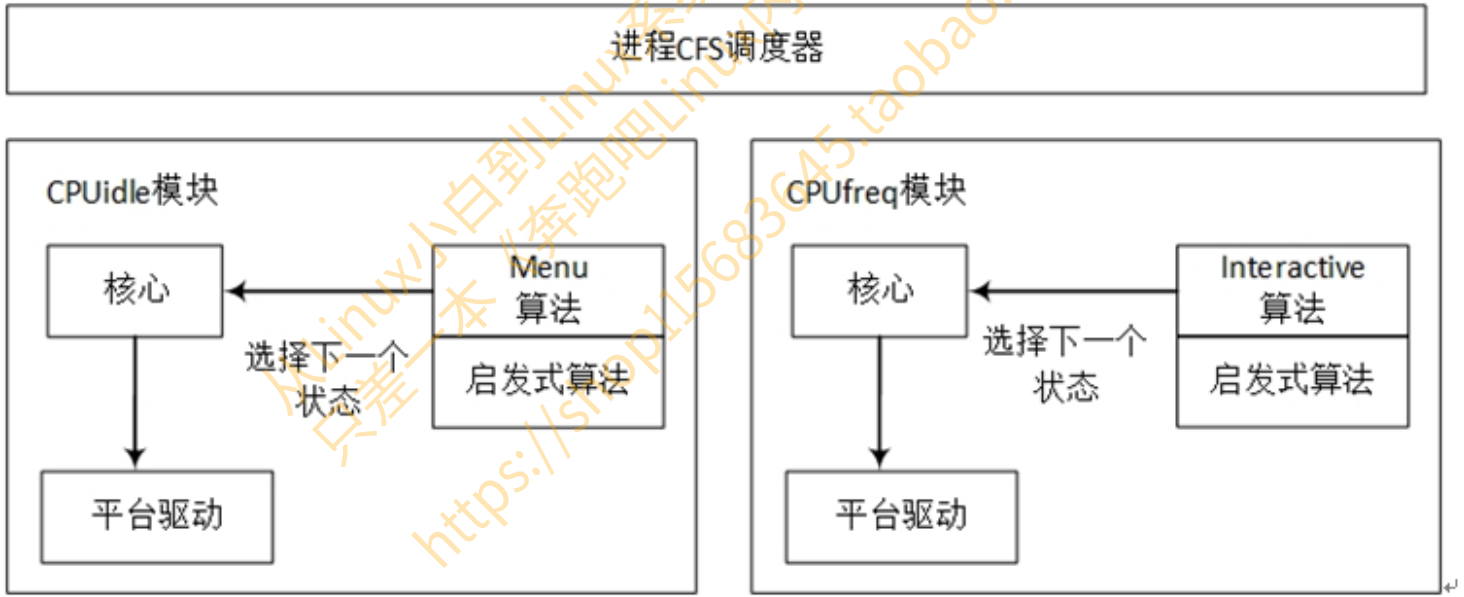
- Cortext-A15之后，ARM推出大小核这个概念



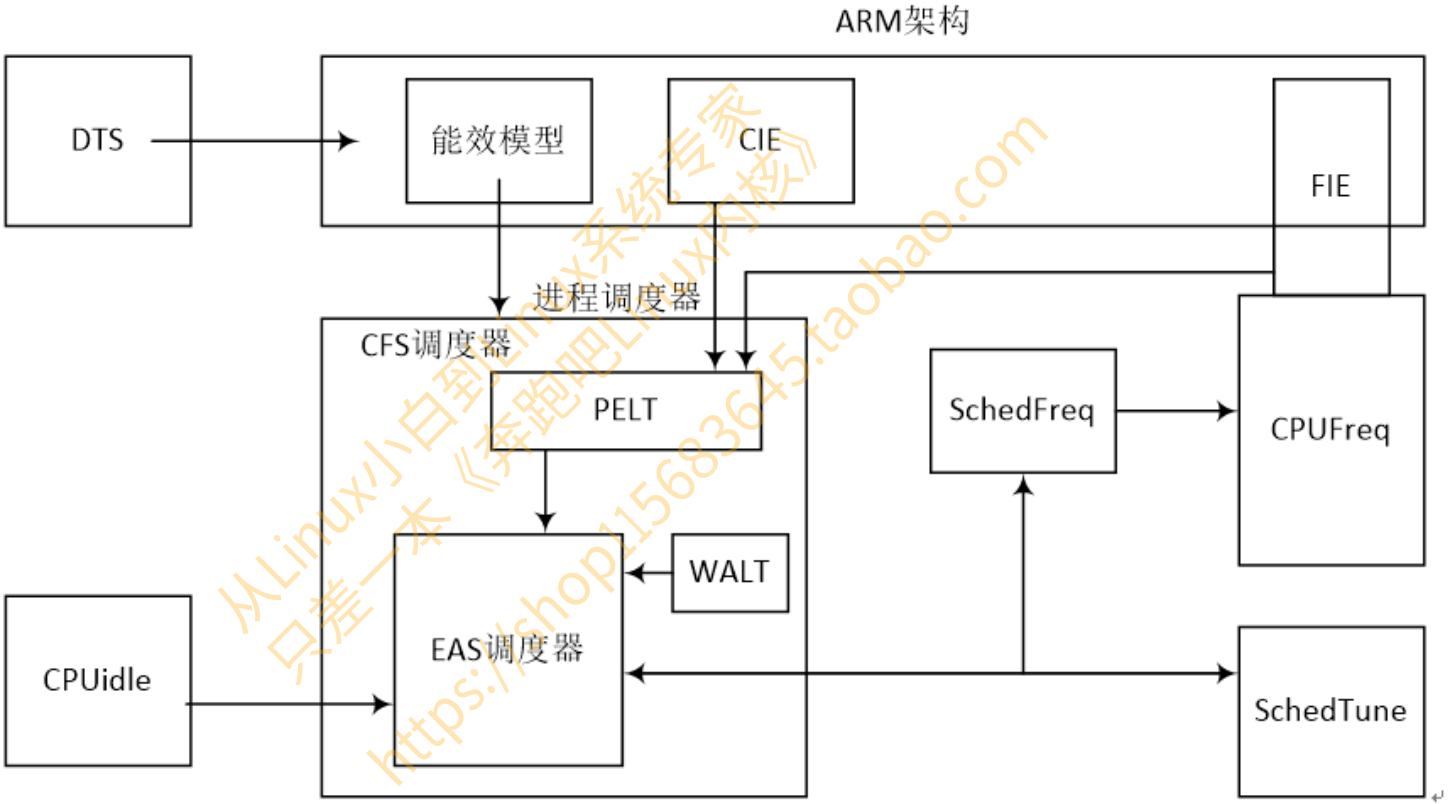


EAS绿色节能调度器

- HMP调度器，最早由ARM开发，后来被EAS替代
- 为啥要有EAS调度器？Energy Aware Scheduling



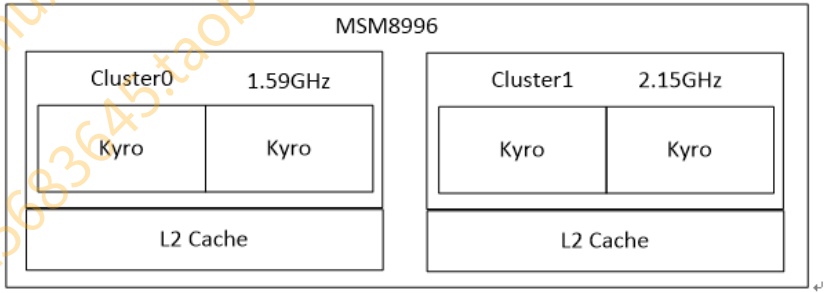
EAS调度器架构



能效模型

- 考虑CPU的计算能力（capacity）和功耗（energy）两方面的因素
- SoC的能效模型 定义在dts文件里，由SoC厂商提供

```
energy-costs {
    CPU_COST_0: core-cost0 {
        busy-cost-data = <
            149 90
            188 111
            225 133
            257 160
            281 182
            315 210
            368 251
            406 306
            428 332
            469 379
            502 438
            538 494
            581 550
            611 613
            648 670
            684 752
            729 848
            763 925
        >;
        idle-cost-data = <
            2 2 0
        >;
    };
```



高通820处理器

arch/arm/boot/dts/qcom/msm8996

CPU计算能力公式

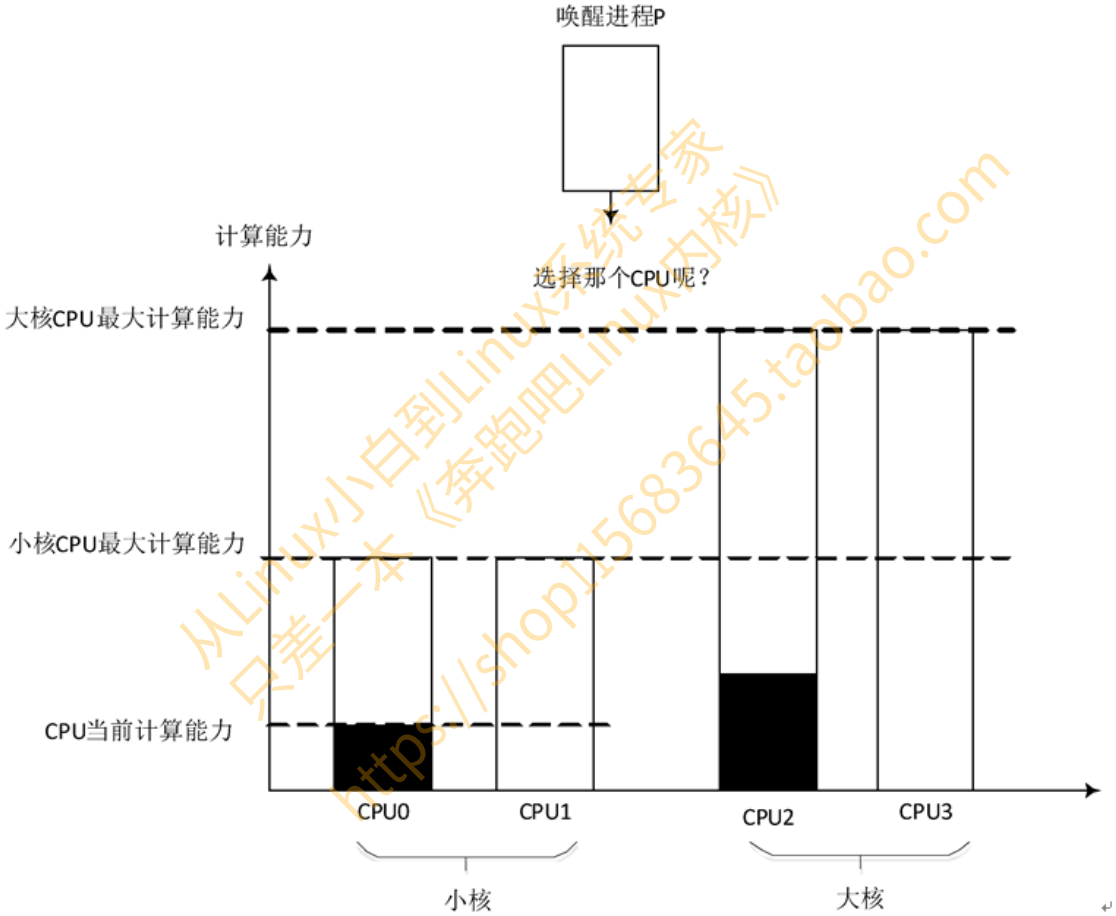
- FIE（Frequency Invairent Engine）
- 计算CPU负载时考虑CPU频率的变化

- CIE（CPU Invariant Engine）
- 考虑不同CPU架构的计算能力对负载的影响

- CPU能力计算公式

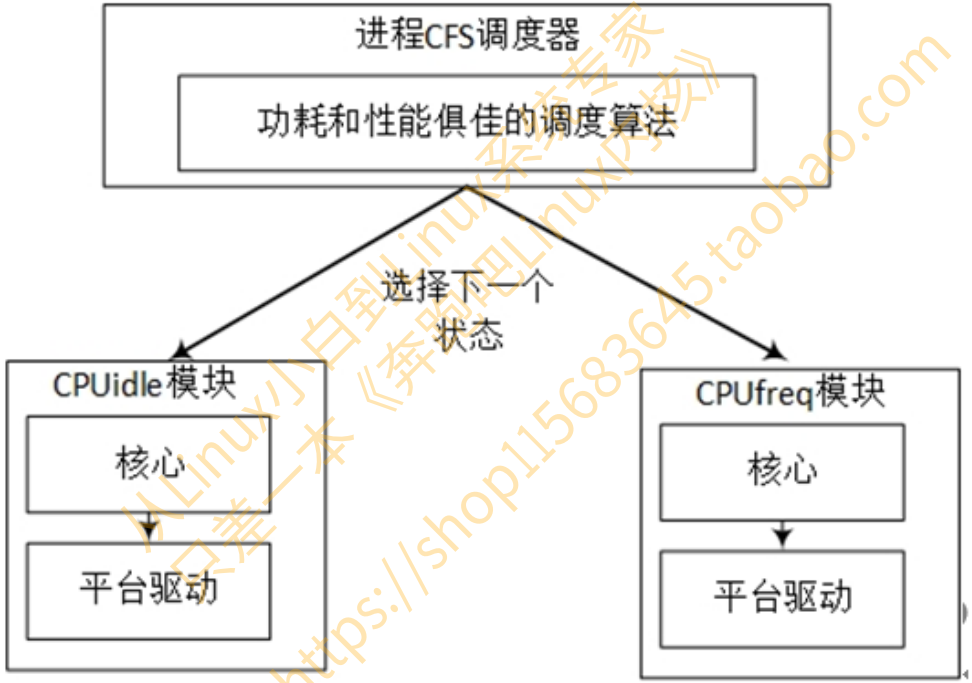
$$cpu\_capacity\_org = cpu\_scale * \frac{cpu\_max\_freq}{system\_max\_freq}$$

EAS如何实现大小核调度?



EAS对CPU调频的改进

- EAS把将调度器和CPUfreq整合到一起



EAS大小核调度器总结

- 量化的计算能力
- 量化的能效模型
- 考虑CPU频率
- 考虑CPU核心不同
- 复用Linux中的CPU调度域拓扑关系图
- 整合CPUfreq模块

实验

实验1：fork和clone

- 实验目的
  - ✓ 了解和熟悉Linux中fork系统调用和clone系统调用的用法。
- 实验步骤
  - ✓ 1）使用fork()函数创建一个子进程，然后在父进程和子进程中分别使用printf语句来判断谁是父进程和子进程。
  - ✓ 2）使用clone()函数创建一个子进程。如果父进程和子进程共同访问一个全局变量，结果会如何？如果父进程比子进程先消亡，结果会如何？
  - ✓ 3）请思考，如下代码中会打印几个“_”？

实验2：内核线程

- 实验目的
  - ✓ 了解和熟悉Linux内核中是如何创建内核线程的。
- 实验步骤
  - ✓ 1）写一个内核模块，创建一组内核线程，每个CPU一个内核线程。
  - ✓ 2）在每个内核线程中，打印当前CPU的状态，比如ARM通用寄存器的值。
  - ✓ 3）在每个内核线程中，打印当前进程的优先级等信息。

实验3：后台守护进程

- 实验目的
  - ✓ 通过本实验了解和熟悉Linux是如何创建和使用后台守护进程的。
- 实验步骤
  - ✓ 1）写一个用户程序，创建一个守护进程。
  - ✓ 2）该守护进程每隔5秒去查看当前内核的日志中是否有oops错误。

实验4：进程权限

- 实验目的
  - ✓ 了解和熟悉Linux是如何进行进程的权限管理的。
- 实验步骤
  - ✓ 1）写一个用户程序，限制该程序的一些资源，比如进程的最大虚拟内存空间等。

实验5：设置优先级

- 实验目的
  - ✓ 了解和熟悉Linux中getpriority()和setpriority()系统调用的用法。
- 实验步骤
  - ✓ 1）写一个用户进程，使用setpriority()来修改进程的优先级，然后使用getpriority()函数来验证。
  - ✓ 2）可以通过一个for循环来依次修改进程的优先级（-20~19）。

实验6： per-cpu变量

- 实验目的
  - ✓ 学会Linux内核中per-cpu变量的用法。
- 实验步骤
  - ✓ 1）写一个简单的内核模块，创建一个per-cpu变量，并且初始化该per-cpu变量，修改per-cpu变量的值，然后输出这些值。

BACKUP

第1季旗舰篇课程目录

课程名称	时长
序言一： Linux内核学习方法论	0:09:13
序言二： 学习前准备	
序言2.1 Linux发行版和开发板的选择	0:13:56
序言2.2 搭建Qemu+gdb单步调试内核	0:13:51
序言2.3 搭建Eclipse图形化调试内核	0:10:59
实战运维1：查看系统内存信息的工具（一）	0:20:19
实战运维2：查看系统内存信息的工具（二）	0:16:32
实战运维3：读懂内核log中的内存管理信息	0:25:35
实战运维4：读懂 proc meminfo	0:27:59
实战运维5：Linux运维能力进阶线路图	0:09:40
实战运维6：Linux内存管理参数调优(一)	0:19:46
实战运维7：Linux内存管理参数调优(二)	0:31:20
实战运维8：Linux内存管理参数调优(三)	0:22:58
运维高级如何单步调试RHEL— CENTOS7的内核一	0:15:45
运维高级如何单步调试RHEL— CENTOS7的内核二	0:41:28
vim:打造比source insight更强更好用的IDE(一)	0:24:58
vim:打造比source insight更强更好用的IDE(二)	0:20:28
vim:打造比source insight更强更好用的IDE(三)	0:23:25
实战git项目和社区patch管理	
2.0 Linux内存管理背景知识介绍	
奔跑2.0.0 内存管理硬件知识	0:15:25
奔跑2.0.1 内存管理总览一	0:23:27
奔跑2.0.2 内存管理总览二	0:07:35
奔跑2.0.3 内存管理常用术语	0:09:49
奔跑2.0.4 内存管理究竟管些什么东西	0:28:02
奔跑2.0.5 内存管理代码框架导读	0:38:09
2.1 Linux内存初始化	
奔跑2.1.0 DDR简介	0:06:47
奔跑2.1.1 物理内存三大数据结构	0:19:39
奔跑2.1.2 物理内存初始化	0:11:13
奔跑2.1 内存初始化之代码导读一	0:43:54
奔跑2.1 内存初始化之代码导读二	0:23:31

奔跑2.1 代码导读C语言部分（二）	0:21:28
2.2 页表的映射过程	
奔跑2.2.0 ARM32页表的映射	0:08:54
奔跑2.2.1 ARM64页表的映射	0:10:58
奔跑2.2.2 页表映射例子分析	0:11:59
奔跑2.2.3 ARM32页表映射那些奇葩的事	0:09:42
2.3 内存布局图	
奔跑2.3.1 内存布局一	0:10:35
奔跑2.3.2 内存布局二	0:13:30
2.4 分配物理页面	
奔跑2.4.1 伙伴系统原理	0:10:10
奔跑2.4.2 Linux内核中的伙伴系统和碎片化	0:11:14
奔跑2.4.3 Linux的页面分配器	0:21:37
2.5 slab分配器	
奔跑2.5.1 slab原理和核心数据结构	0:18:36
奔跑2.5.2 Linux内核中slab机制的实现	0:16:56
2.6 vmalloc分配	
奔跑2.6 vmalloc分配	0:15:48
2.7 VMA操作	
奔跑2.7 VMA操作	0:16:42
2.8 malloc分配器	
奔跑2.8.1 malloc的三个迷惑	0:17:41
奔跑2.8.2 内存管理的三个重要的函数	0:17:38
2.9 mmap分析	
奔跑2.9 mmap分析	0:23:14
2.10 缺页中断处理	
奔跑2.10.1 缺页中断一	0:31:07
奔跑2.10.2 缺页中断二	0:16:58
2.11 page数据结构	
奔跑2.11 page数据结构	0:29:41
2.12 反向映射机制	
奔跑2.12.1 反向映射机制的背景介绍	0:19:01
奔跑2.12.2 RMAP四部曲	0:07:31
奔跑2.12.3 手撕Linux2.6.11上的反向映射机制	0:07:35
奔跑2.12.4 手撕Linux4.x上的反向映射机制	0:10:08
2.13 回收页面	
奔跑2.13 页面回收一	0:16:07
奔跑2.13 页面回收二	0:11:41

奔跑2.11 page数据结构	0:29:41
2.12 反向映射机制	
奔跑2.12.1 反向映射机制的背景介绍	0:19:01
奔跑2.12.2 RMAP四部曲	0:07:31
奔跑2.12.3 手撕Linux2.6.11上的反向映射机制	0:07:35
奔跑2.12.4 手撕Linux4.x上的反向映射机制	0:10:08
2.13 回收页面	
奔跑2.13 页面回收一	0:16:07
奔跑2.13 页面回收二	0:11:41
2.14 匿名页面的生命周期	0:26:16
2.15 页面迁移	0:19:07
2.16 内存规整	0:24:03
2.17 KSM	0:28:17
2.20 Meltdown漏洞分析	
奔跑2.20.1 Meltdown背景知识	0:10:13
奔跑2.20.2 CPU体系结构之指令执行	0:11:25
奔跑2.20.3 CPU体系结构之乱序执行	0:11:03
奔跑2.20.4 CPU体系结构之异常处理	0:03:48
奔跑2.20.5 CPU体系结构之cache	0:10:56
奔跑2.20.6 进程地址空间和页表及TLB	0:17:39
奔跑2.20.7 Meltdown漏洞分析	0:06:04
奔跑2.20.8 Meltdown漏洞分析之x86篇	0:12:07
奔跑2.20.9 ARM64上的KPTI解决方案	0:25:39
代码导读	
奔跑2.1 内存初始化之代码导读一	0:43:54
奔跑2.1 内存初始化之代码导读二	0:23:31
奔跑2.1 代码导读C语言部分（一）	0:27:34
奔跑2.1 代码导读C语言部分（二）	0:21:28
代码导读3页表映射	1:12:40
代码导读4分配物理页面	0:55:57
git入门和实战	
git入门与实战：节目总览	0:08:48
git入门与实战1：建立本地的git仓库	0:30:53
git入门与实战2：快速入门	0:12:45
git入门与实战3：分支管理	0:24:27
git入门与实战4：冲突解决	0:20:20
git入门和实战5：提交更改	0:12:15
git入门和实战6：远程版本库	0:13:26
git入门和实战7：内核开发和实战	0:15:52
git入门和实战8：实战rebase到最新Linux内核代码	0:18:07
git入门和实战9：给内核发补丁	0:13:57

第2季旗舰篇课程目录	
课程名称	时长
进程管理	
进程管理1基本概念	0:52:16
进程管理2进程创建	0:53:24
进程管理3进程调度	0:54:51
进程管理4多核调度	0:49:38
中断管理	
中断管理1基本概念	1:04:27
中断管理2中断处理part1	0:46:28
中断管理2中断处理part2	0:10:19
中断管理3下半部机制	0:55:57
中断管理4面试题目	1:13:57
锁机制	
锁机制入门1基本概念	0:56:16
锁机制入门2-Linux常用的锁	0:54:01

实战死机专题课程目录	
课程名称	时长
上集x86_64	
实战死机专题（上集）part1-kdump+crash介绍	0:30:09
实战死机专题（上集）part2-crash命令详解	0:28:15
实战死机专题（上集）part3-实战lab1	0:12:38
实战死机专题（上集）part4-实战lab2	0:11:03
实战死机专题（上集）part4-实战lab3	0:06:48
实战死机专题（上集）part4-实战lab4	0:15:28
实战死机专题（上集）part4-实战lab5	0:12:21
实战死机专题（上集）part4-实战lab6	0:24:07
实战死机专题（上集）part4-实战lab7	0:59:34
下集arm64	
实战死机专题(下集)part1	0:13:19
实战死机专题(下集)part2	0:20:47
实战死机专题(下集)part3	0:11:22
实战死机专题(下集)part4	0:33:01

全程约5小时高清，140多页ppt，8大实验，基于x86_64的**Centos 7.6**和**arm64**，提供全套实验素材和环境。全面介绍kdump+crash在死机黑屏方面的实战应用，全部案例源自线上云服务器和嵌入式产品开发实际案例！

微店 奔跑吧Linux内核

扫码识别

微店二维码

淘宝店二维码

《奔跑吧Linux内核 * 入门篇》相关的免费视频，或者更多更精彩更in的内容，请关注奔跑吧Linux社区微信公众号