华 清 远 见 ®

FARSIGHT

嵌 入 式 培 训 专 家

学习LINUX内核

www.embedu.org

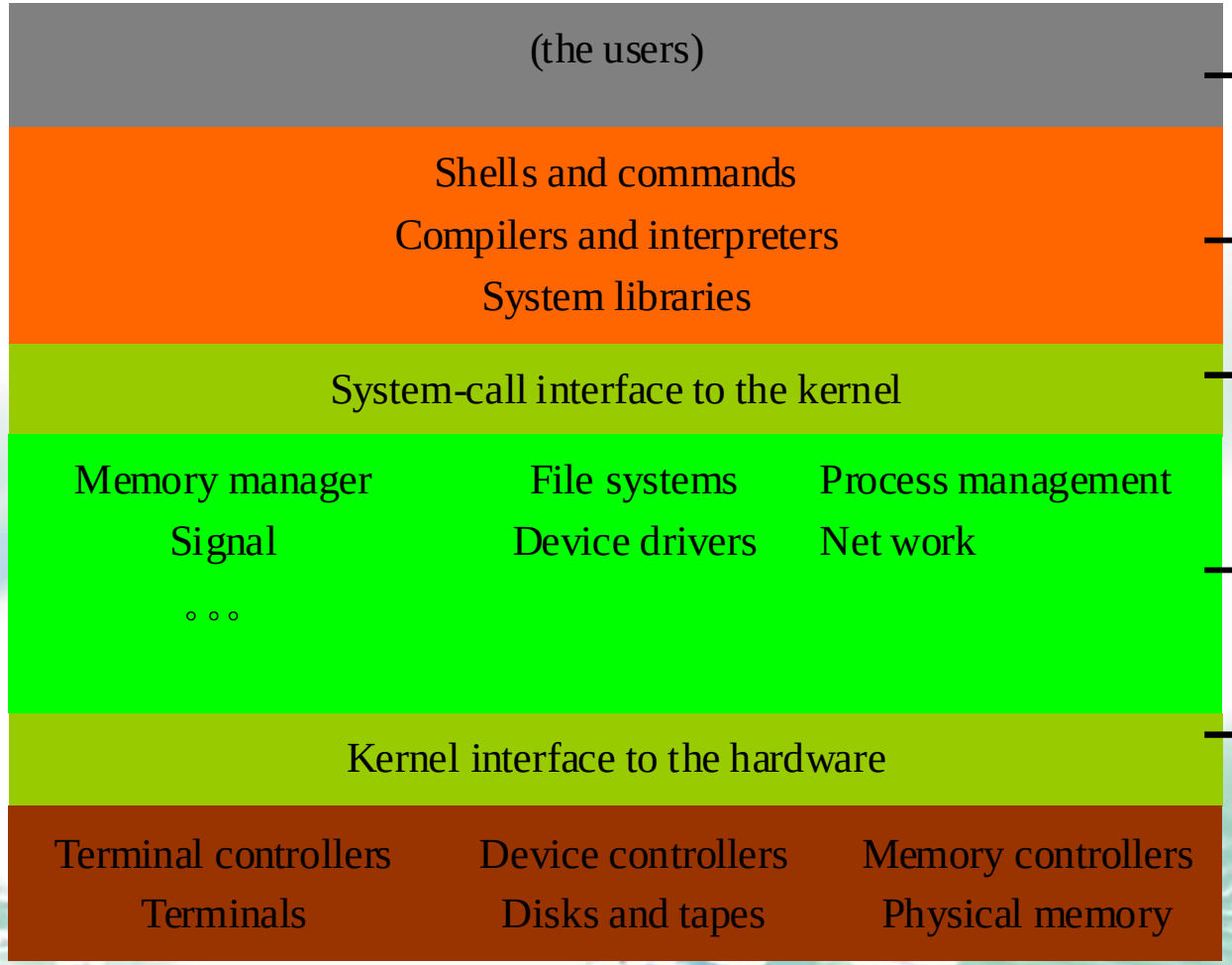
C-SKY

Linux 简介

- Linux是免费的、源代码开放的、符合POSIX标准规范的操作系统 。
  版本历史：
  - 1991年，诞生
  - …
  - 2001年，Linux2.4版内核发布
  - 2003年，Linux2.6版内核发布

- Linux特性
  - 抢占式多任务处理
  - PMMU -- 页式内存管理
  - VFS – 虚拟文件系统
  - 网络功能（如，支持TCP/IP ）
  - 动态加载模块
  - 支持SMP
  - 支持绝大多数的32位和64位CPU 等

一个典型的Linux操作系统的结构



用户应用程序

Shell，lib

System call

Kernel implementation

对硬件资源的管理

最简单也是最复杂的操作

Linux 基本概念

- 系统调用
- 内存管理
- 进程管理
- 虚拟文件系统（VFS）
- 信号机制
- 内核初始化过程

Ø 提纲

- 用户态和内核态
- 系统调用意义
- 系统调用方法

用户态和内核态

（CPU：ckcore）	内核态	用户态
标志	PSR最高位1	PSR最高位0
运行指令	无限制	特权指令不可执行
地址空间(MMU)	0~4G可访问	0~2G可访问

- Ckcore的特权指令有：MFCR、MTCR、PSRSET、PSRCLR、RFI、RTE、STOP、WAIT、DOZE
- 这里所说的地址空间是虚拟地址而不是物理地址

- 区分用户态和内核态目的在于安全考虑：
  - 禁止用户程序和底层硬件直接打交道

    (最简单的例子，如果用户程序往硬件控制寄存器写入不恰当的值，可能导致硬件无法正常工作)
  - 禁止用户程序访问任意的物理内存

    (否则可能会破坏其他程序的正常执行，如果对内核所在的地址空间写入数据的话，会导致系统崩溃)

用户程序如何同设备打交道？

例如，用户需通过网卡发送数据

- 硬件被linux 内核隔离，只能通过内核实现。
- 不可能直接调用操作系统的函数：不可行，也不安全。

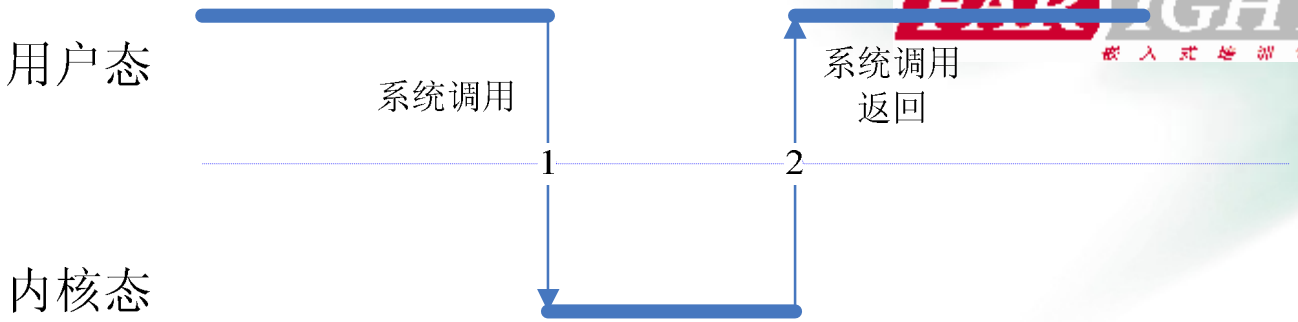
Ø Linux提供的解决方法：系统调用

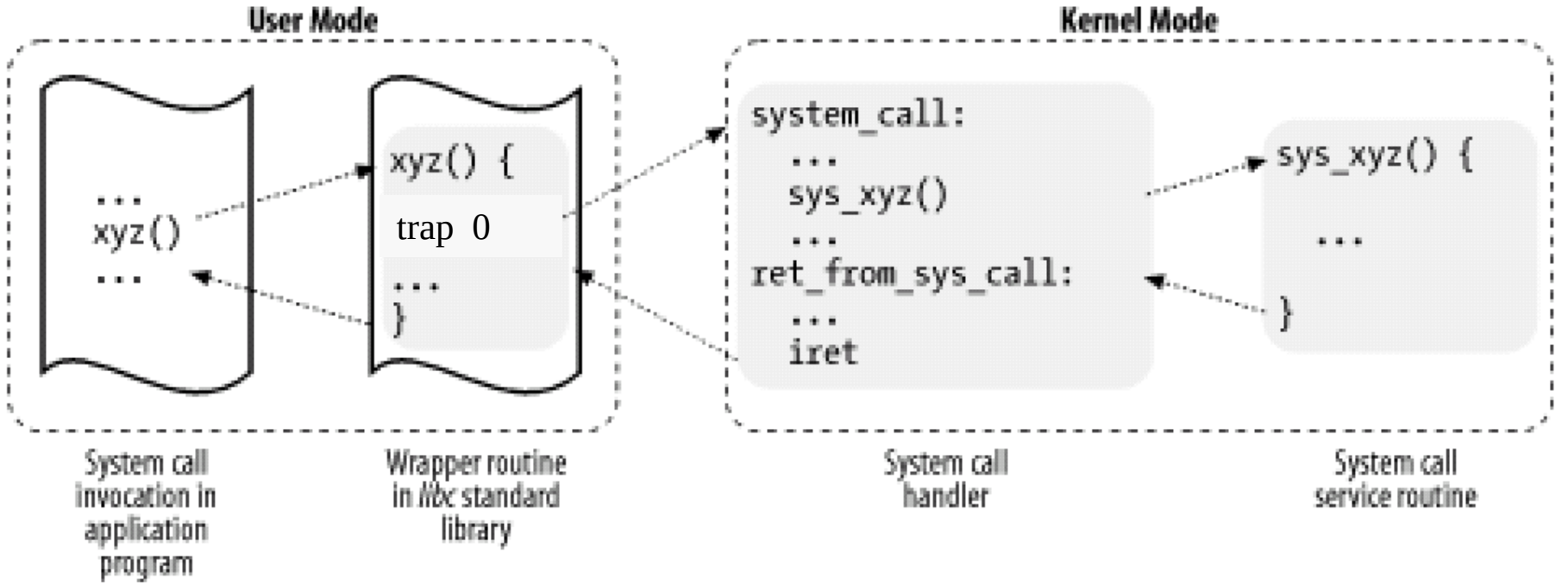
系统调用的意义

- 操作系统为用户态进程与硬件设备进行交互提供了一组接口——系统调用
  - 把用户从底层的硬件编程中解放出来
  - 极大的提高了系统的安全性
  - 使用户程序具有可移植性
- 基于ckcore的Linux kernel使用“trap 0”指令进行系统调用

- 系统调用过程：
  - 执行陷阱异常指令trap 0 à
  - 进入异常后，处理器PSR最高位被硬件置1，实现 普通用户 到 特权用户 的转变 à
  - 根据系统调用号（r1传入），调用相应函数，满足用户需求à
  - 系统调用返回，重新回到用户态，用户获得资源。
- API和系统调用完全不同：
  - API只是一个函数定义
  - 系统调用通过“软中断”向内核发出一个明确的请求

系统调用图解


用户态
内核态
系统调用
系统调用
返回
1
2


User Mode
Kernel Mode
...
xyz()
...
xyz() {
trap 0
...
}
system_call:
...
sys_xyz()
...
ret_from_sys_call:
...
iret
sys_xyz() {
...
}
System call invocation in application program
Wrapper routine in libc standard library
System call handler
System call service routine

- Linux 2.6 提供了300多个系统调用，用户可以通过这些系统调用，及它们的组合实现对设备的操作。
- 通常，应用程序开发并不直接和系统调用打交道，而是用C库提供的一层包装函数。

如，malloc() **à** sbrk() **à** sys_brk **à** 内核函数

sys_brk 是45号系统调用,C库中它的系统调用方式可能是:

```
…
movi   r1, 45
trap    0
…
```

Linux 基本概念

- 系统调用
- 内存管理
- 进程管理
- 文件系统
- 信号机制
- 内核初始化过程

Ø 提纲

- 虚拟内存
- 虚拟内存到物理内存映射方法
- 物理内存和虚拟空间的管理
- 页面异常处理
- 页面交换策略
- slab分配器
- ioremap

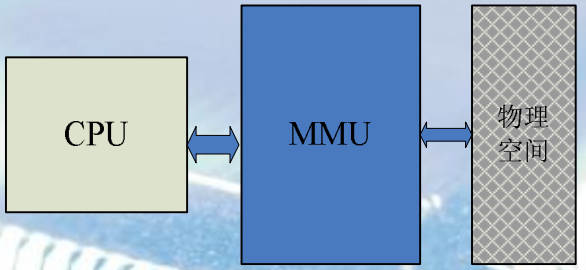
虚拟内存

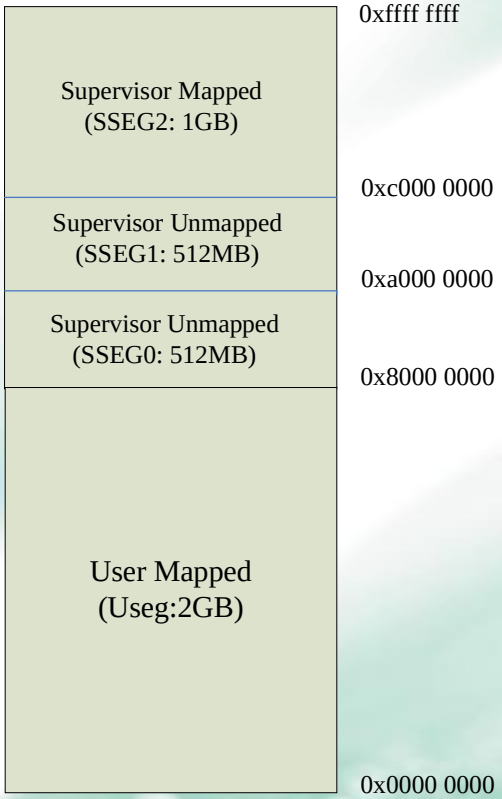
- 物理内存有限，是一种稀缺资源
- 32位系统中，每个进程独立的占有4G虚拟空间。
- 虚拟内存优势：
  - Ø用户程序开发方便
  - Ø保护内核不受恶意或者无意的破坏
  - Ø隔离各个用户进程

Ckcore的MMU虚拟地址空间

- 在保护模式下，即MMU开启时：

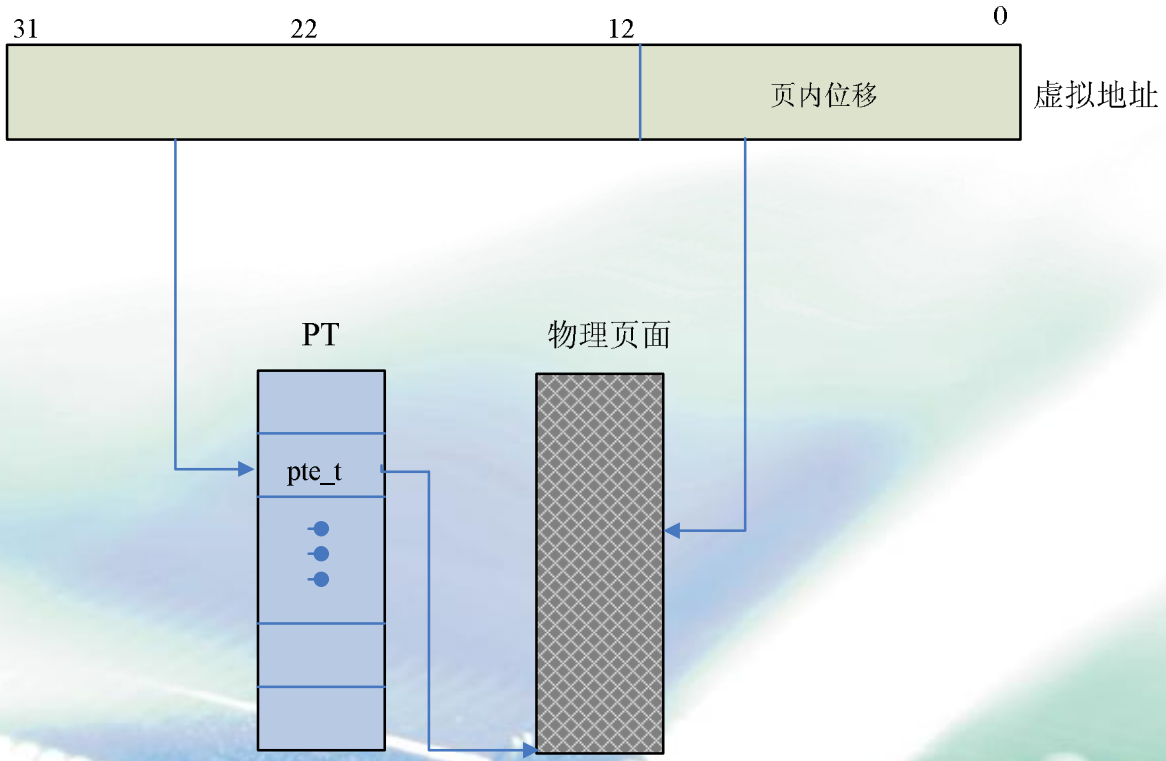
USEG：用户程序可访问；需建立映射
SSEG0：只有内核可访问；直接映射到0~512M PA；可cache
SSEG1：只有内核可访问；也映射到0~512M PA；不可cache
SSEG2：只有内核可访问；需建立映射





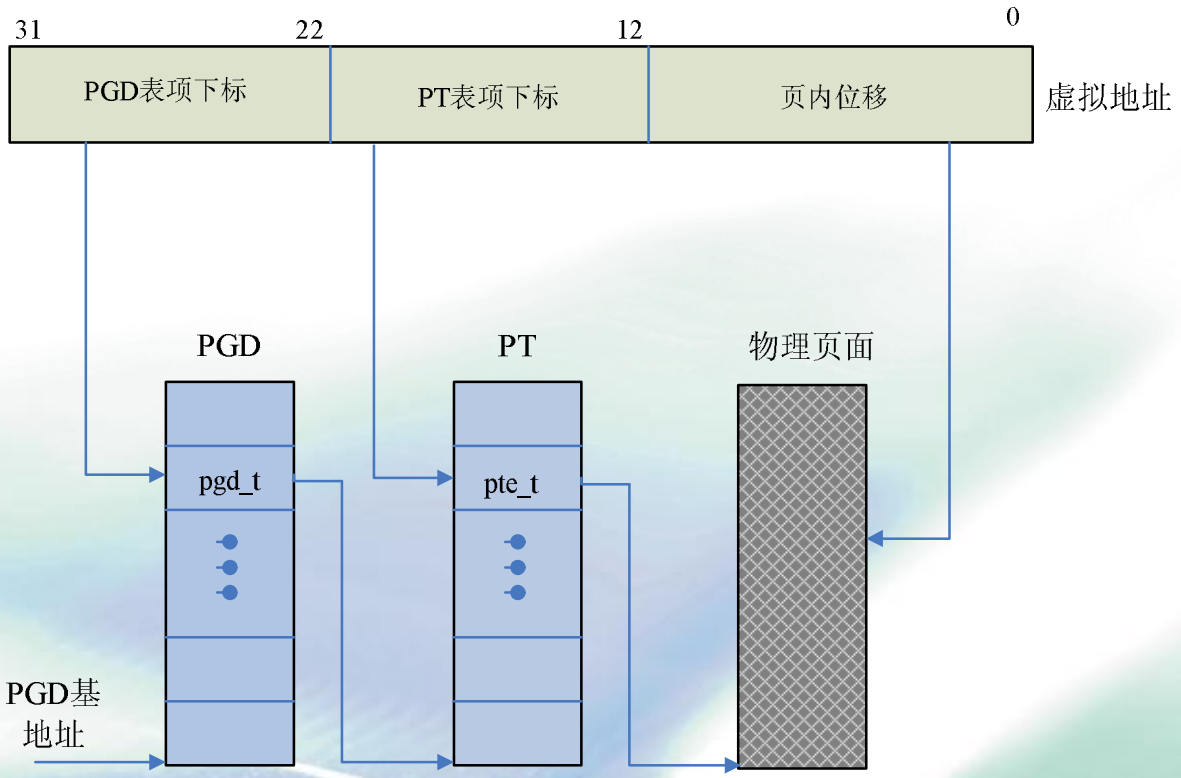
在ckcore Linux中，2G以上是内核空间，2G以下是用户空间。

虚拟地址到物理地址转换



问题：内核为每个进程管理一个PT，占据内存 4MB。

虚拟地址到物理地址转换：多级映射



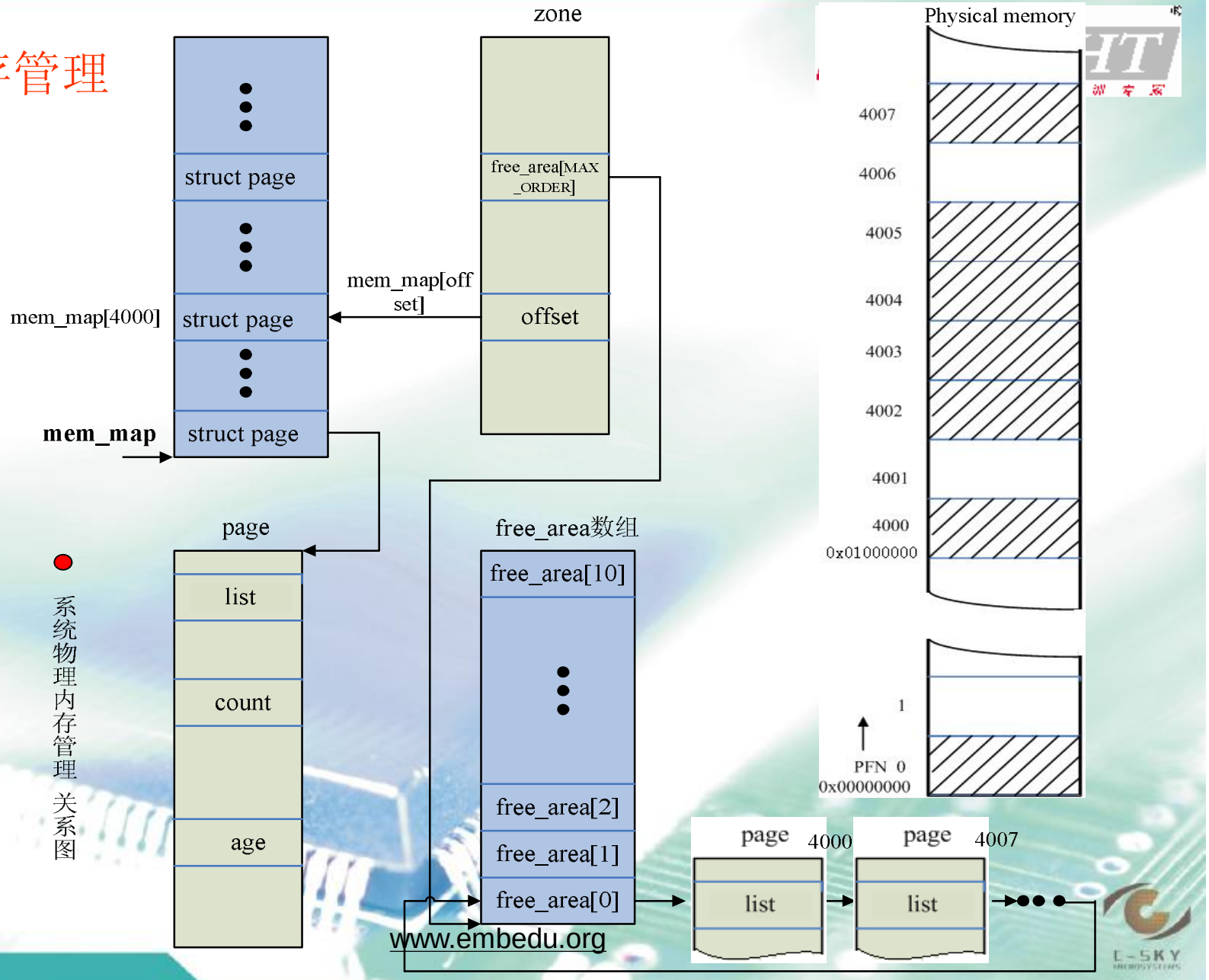
内核为每个进程管理一个PGD表， PT则需要时再分配。占内存 4kB+4KB*N

- 物理内存获取过程：

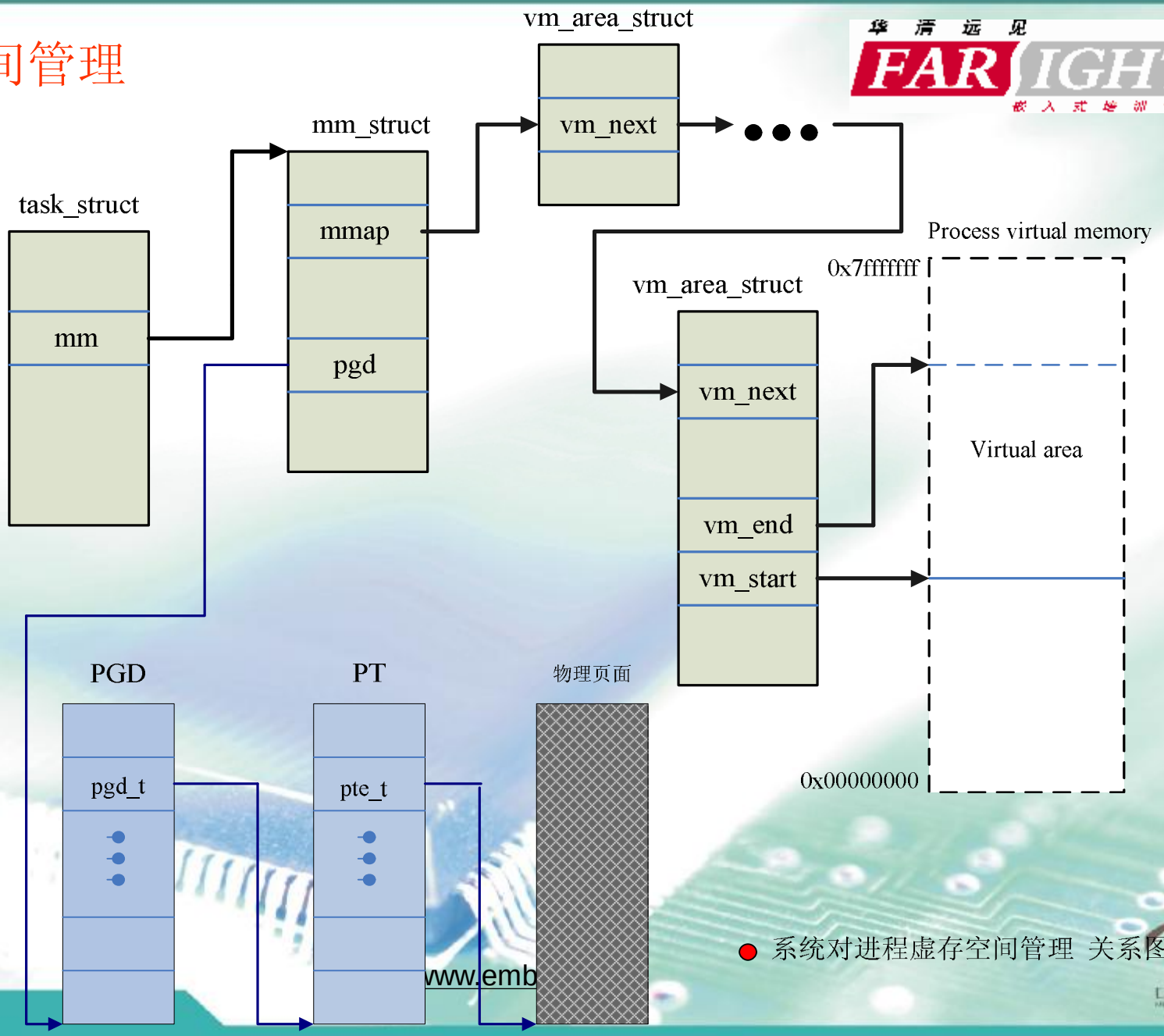
  用户程序请求物理内存 à 内核分配物理页面 à 内核填写对应页表项 à 用户程序获得物理内存

- 从这个过程看出，内存管理的核心内容是：物理页面分配 和 所有进程页表的维护，即物理内存管理 和 虚拟空间管理。

物理内存管理


zone
struct page
free_area[MAX_ORDER]
mem_map[offset]
mem_map[4000]
struct page
offset
mem_map
struct page
page
free_area数组
list
free_area[10]
count
age
free_area[2]
free_area[1]
free_area[0]
系统物理内存管理关系图
Physical memory
4007
4006
4005
4004
4003
4002
4001
4000
0x01000000
1
PFN 0
0x00000000
page 4000
page 4007
list
list
www.embedu.org

虚拟空间管理



系统对进程虚存空间管理 关系图

缺页异常处理

- 物理页面并不预先分配，而是用到时发现缺少再分配。
- 典型的页面分配过程：

  用户程序访问虚拟空间 **à**

  未建立映射，产生缺页异常 **à**

  异常处理函数分配物理页面，并填好页表 **à**

  异常返回，用户程序再次访问该虚拟空间

Linux页面交换策略

- 把暂时不用的页面存放到磁盘上，为其它急用的进程腾出空间，到需要时再从磁盘上读出来

- 以时间换空间，因此：
  - 有实时性要求的系统不宜使用
  - 嵌入式只有Flash闪存的系统不适合用

slab分配器

- 由于操作系统在运行中会不断产生、使用、释放大量重复的对象， 所以对这样的重复对象的生成进行改进可以大大提高效率

- Slab将缓存分为两种：

  1、专用高速缓存：用来存放内核使用的数据结构，例如：mm，skb，vm等等

  2、普通高速缓存：指存放一般的数据，比如内核为指针分配一段内存

ioremap() :外部设备存储空间的地址映射

- 通常的内存页面的管理，都是先分配虚存空间，再为此区间分配物理内存页面，并建立映射。
- ioremap()则相反：要先有一个物理存储区间，如外设卡上的存储器出现在总线上的地址。再“反向”从该总线地址出发找到一片虚存区间，并建立映射。
- 因为只有内核才能对外部设备进行操作，所以相应的虚存区间是内核空间（2GB以上）。

Linux 基本概念

- 系统调用
- 内存管理
- 进程管理
- 虚拟文件系统（VFS）
- 信号机制
- 内核初始化过程

Ø 提纲

- 进程描述符
- 进程的状态
- 进程调度
- 进程的切换
- 创建和撤销进程
- 0号线程和1号进程

简单说进程的概念

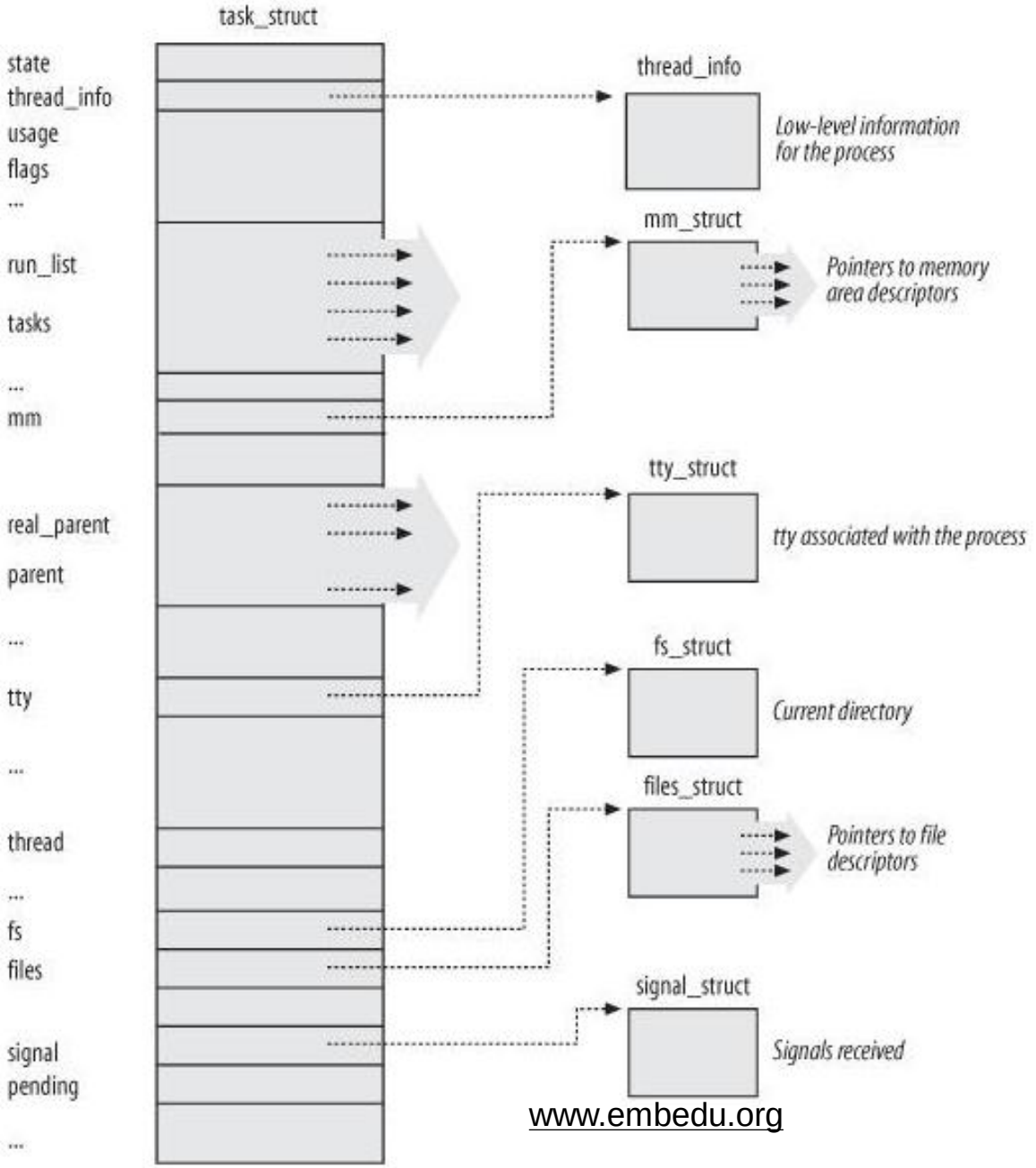
- 进程是执行程序的一个实例
- 进程和程序的区别
  - 几个进程可以并发的执行一个程序
  - 一个进程可以顺序的执行几个程序

标识一个进程

- 使用进程描述符（task_struct）
  - 进程和进程描述符之间有非常严格的一一对应关系
- 使用PID (Process ID，PID)
  - 每个进程有唯一的PID号，存放在进程描述符的pid域中

进程描述符

- 进程描述符提供了内核所需了解的进程信息
  - include/linux/sched.h
    struct task_struct
  - 数据结构很庞大
    - 基本信息
    - 管理信息
    - 控制信息



Linux2.6进程的状态

```
00173: #define TASK_RUNNING         0
00174: #define TASK_INTERRUPTIBLE   1
00175: #define TASK_UNINTERRUPTIBLE   2
00176: #define __TASK_STOPPED       4
00177: #define __TASK_TRACED        8
00178: /* in tsk->exit_state */
00179: #define EXIT_ZOMBIE        16
00180: #define EXIT_DEAD          32
00181: /* in tsk->state again */
00182: #define TASK_DEAD          64
00183: #define TASK_WAKEKILL         128
```

TASK_RUNNING：可运行。

TASK_INTERRUPTIBLE与**TASK_UNINTERRUPTIBLE**：睡眠等待。

interrupt指软中断，即信号。前者可被信号唤醒，后者不能。

TASK_STOPPED：暂停状态。主要用于调试。

TASK_TRACED：进程被监控。

TASK_DEAD：死亡状态。进程退出（调用do_exit()）。

TASK_WAKEKILL：在接收到致命信号时唤醒进程。

Linux2.6最新进程状态特性：

1. TASK_DEAD状态区分两个不同的exit_state：

EXIT_ZOMBIE：进程已终止，它正等待其父进程收集关于它的一些统计信息。

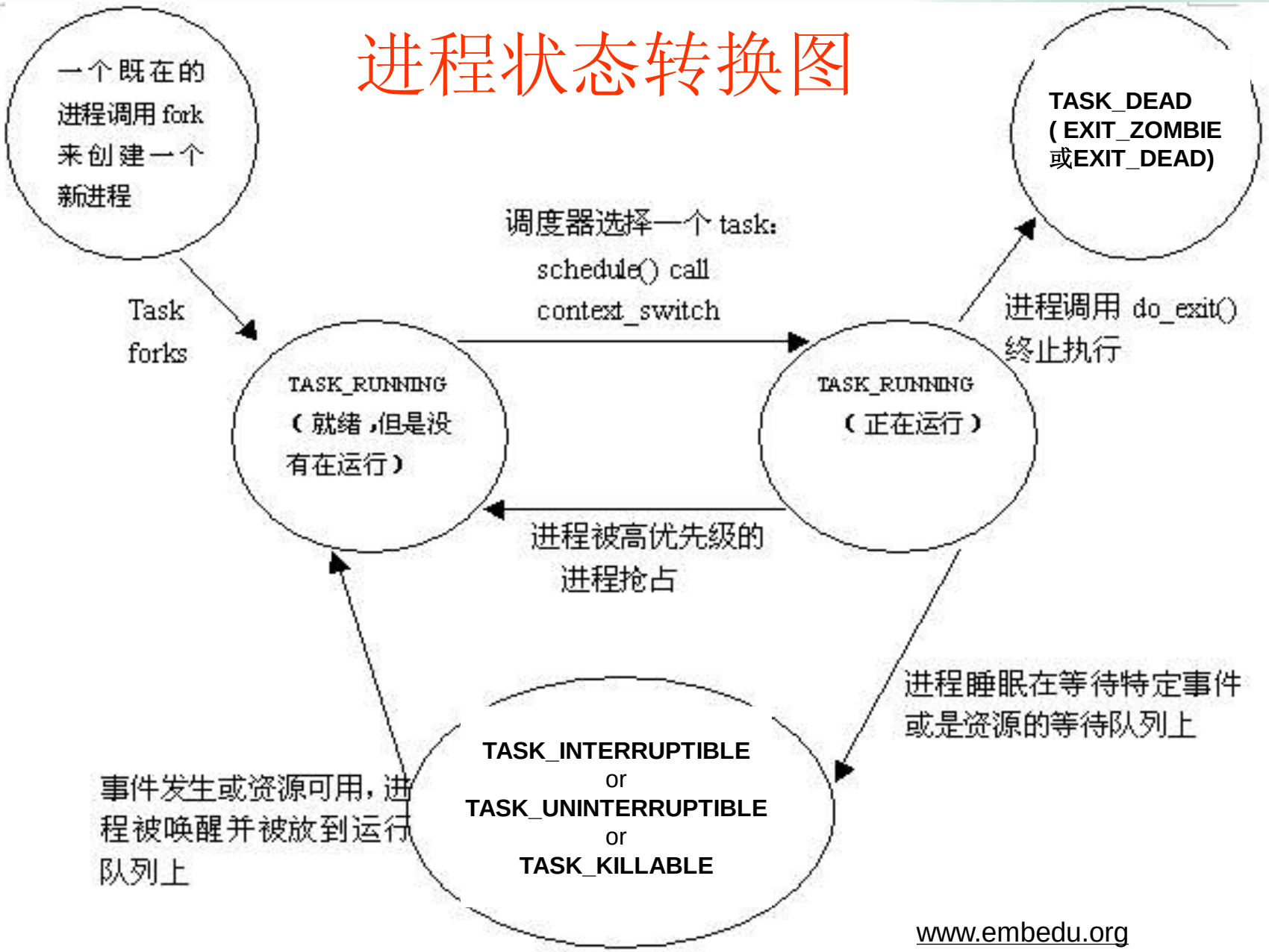
EXIT_DEAD：最终状态。其父进程已经通过 wait4() 或 waitpid() 调用收集所有统计信息，为了防止其他执行线程对同一进程也执行wait()类系统调用。

2. 添加TASK_WAKEKILL 用于在接收到致命信号时唤醒进程：

```
#define TASK_KILLABLE   (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED    (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED     (TASK_WAKEKILL | __TASK_TRACED)
```

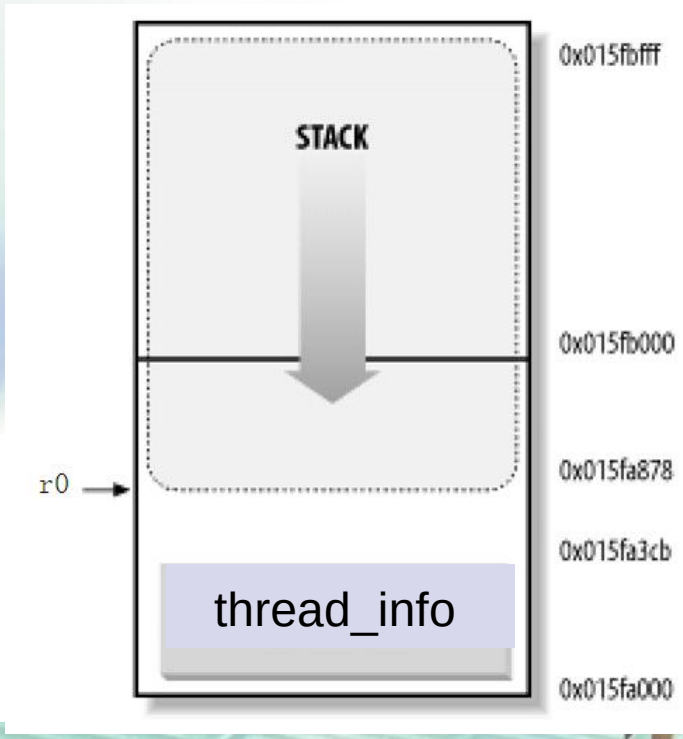
TASK_KILLABLE可以替代TASK_UNINTERRUPTIBLE在某些地方的应用。

进程状态转换图


一个既在的进程调用 fork 来创建一个新进程
Task forks
TASK_RUNNING
（就绪，但是没有在运行）
调度器选择一个 task:
schedule() call
context_switch
TASK_RUNNING
（正在运行）
TASK_DEAD
(EXIT_ZOMBIE
或EXIT_DEAD)
进程调用 do_exit()
终止执行
进程被高优先级的进程抢占
进程睡眠在等待特定事件或是资源的等待队列上
TASK_INTERRUPTIBLE
or
TASK_UNINTERRUPTIBLE
or
TASK_KILLABLE
事件发生或资源可用，进程被唤醒并被放到运行队列上

进程的内核堆栈

- Linux为每个进程分配一个8KB大小的内存区域，用于存放该进程两个不同的数据结构：
  - thread_info
  - 进程的内核堆栈



- 进程处于内核态时使用不同于用户态堆栈
- 内核控制路径所用的堆栈很少，因此对栈和描述符来说，8KB足够了

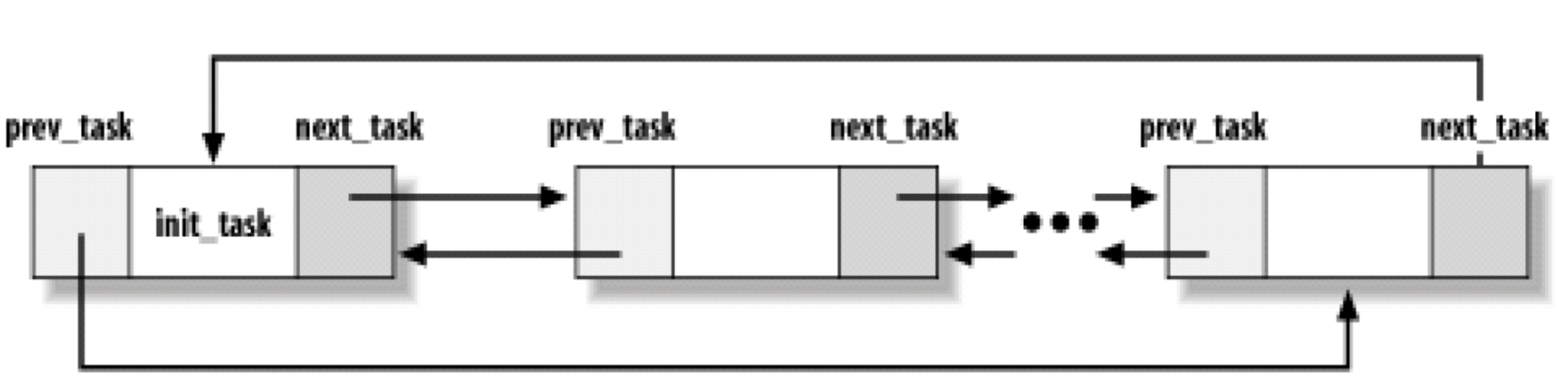
再看进程的概念

- 进程：有独立的内核堆栈（如8k），一个task_struct，专用的用户空间，并利用这些资源运行一个或者多个程序。
- 如果没有用户空间，则是 内核线程(thread);
  如果共享用户空间，则称为 用户线程。
- 进程和线程是进程调度的基本单位。

进程的管理

内核维护着若干进程链表，用于进程的管理和调度。

- 所有进程链表



进程可运行链表

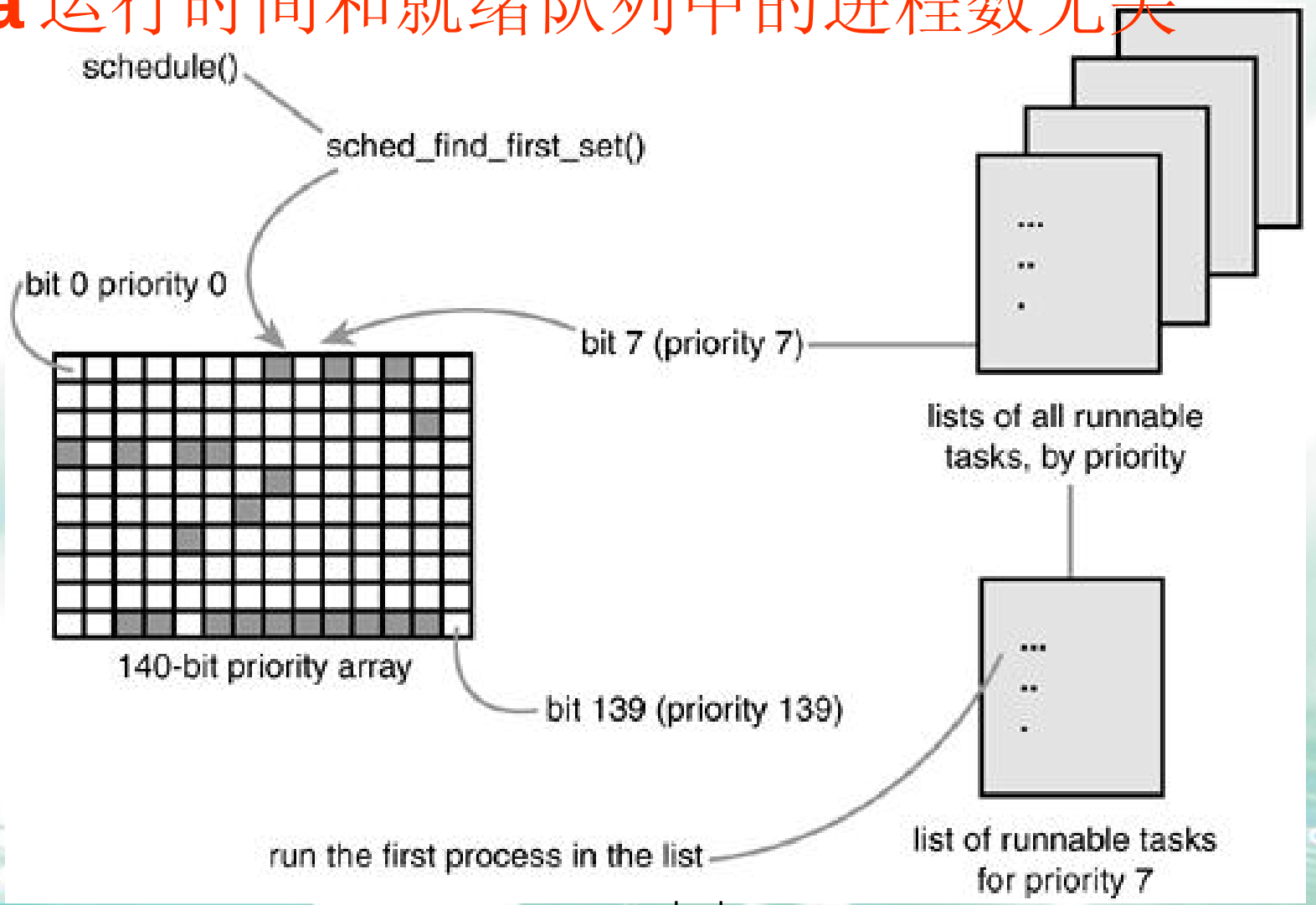
Ø内核把所有处于TASK_RUNNING状态的进程组织成一个可运行双向循环队列。

Ø调度函数通过扫描整个可运行队列，取得最值得执行的进程投入执行。避免扫描所有进程，提高调度效率。

问题：可运行进程（n）越多，每次调度的开销越大。这是2.4的O(n)调度算法的缺点。

Linux 2.6 新引入O(1)调度算法

à运行时间和就绪队列中的进程数无关



调度时机

A. 用户进程自愿放弃CPU，如执行sleep()系统调用；

B. 系统调用中，需要等待时，直接调用schedule()进行调度；

C. 系统调用、中断或异常处理完成后，返回到用户空间前，若当前进程的描述符中的need_resched = 1，则发生调度；

内核是否为preemptible：视C中的中断处理完成，但返回内核空间时，是否仍有调度时机。

进程切换(process switching)

- 内核挂起正在CPU上执行的进程，并恢复以前挂起的某个进程的执行，叫做进程切换，任务切换，上下文切换。

进程上下文

- 包含了进程执行需要的所有信息
  - 用户地址空间
    包括程序代码，数据，用户堆栈等
  - 控制信息
    进程描述符，内核堆栈等
  - 硬件上下文

硬件上下文

- 进程恢复执行前必须装入寄存器的一组数据，包括：
  - 通用寄存器
  - 系统寄存器

- 所有的进程共享CPU的寄存器。因此，内核在恢复一个进程执行之前，必须确保每个寄存器装入了挂起进程时的值。这样才能正确的恢复一个进程的执行。

- 在linux中，一个进程的上下文主要保存在thread_info和thread_struct, 以及内核态堆栈中。

- thread_info

```
struct thread_info {
    struct task_struct    *task;        /* main task structure */
    unsigned long         flags;
    struct exec_domain    *exec_domain;  /* execution domain */
    int                   preempt_count; /* 0 => preemptable, <0 => BUG */
    mm_segment_t          addr_limit;

    struct restart_block  restart_block;
    struct pt_regs        *regs;
};
```

thread_struct

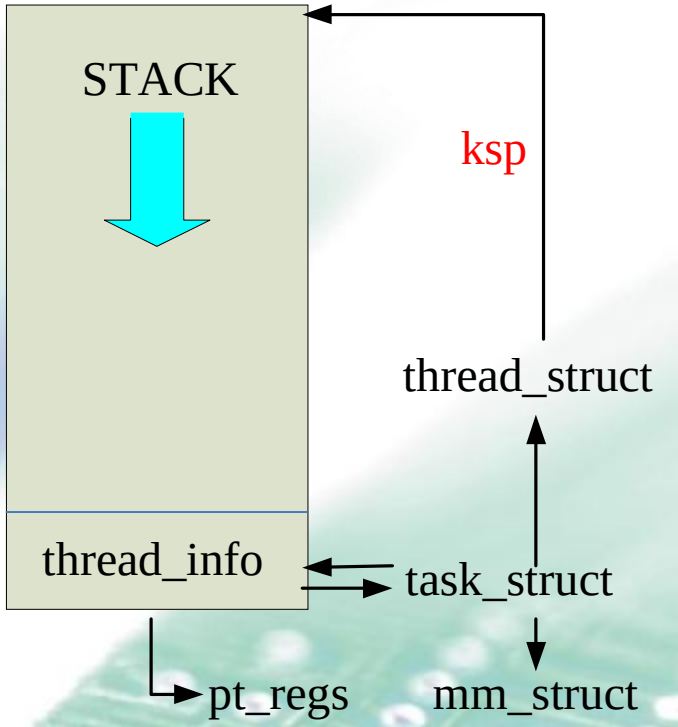
```
struct thread_struct {
    unsigned long  ksp;        /* kernel stack pointer */
    unsigned long  usp;        /* user stack pointer */
    unsigned long  sr;        /* saved status register */
    unsigned long  crp[2];     /* cpu root pointer */
    unsigned long  esp0;       /* points to SR of stack frame */

    /* Other stuff associated with the thread. */
    unsigned long address;      /* Last user fault */
    unsigned long baduaddr;
    unsigned long error_code;
    unsigned long trap_no;
};
```

pt_regs

```
struct pt_regs {
        unsigned long   pc;
        long            r1;
        long            syscallr2;
        unsigned long   sr;
        long            r2;
        long            r3;
        long            r4;
        long            r5;
        long            r6;
        long            r7;
        long            r8;
        long            r9;
        long            r10;
        long            r11;
        long            r12;
        long            r13;
        long            r14;
        long            r15;
};
```

- 进程切换的关键：内核堆栈的切换
  thread_info和内核态堆栈紧密联系在一起，切换内核态堆栈就意味着切换当前thread_info。

- 进程执行需要的所有信息：
  - 用户地址空间
    包括程序代码，数据，用户堆栈
  - 控制信息
    进程描述符，内核堆栈等
  - 硬件上下文



current task

- 用current宏获取当前task_struct：

```
static inline struct task_struct *get_current(void)
{
    return current_thread_info()->task;
}

#define current (get_current())
```

```
static inline struct thread_info *current_thread_info(void)
{
    unsigned long sp;

    __asm__ __volatile__(
                        "mov %0, r0\n\t"
                        :"=r"(sp));

    return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));
}
```

进程的创建

- Linux提供了几个系统调用来创建：
  - fork，vfork和clone系统调用创建新进程
  - 在内核中，它们都是调用do_fork实现的

- fork：父进程的所有数据结构复制一份给子进程

  clone：有选择的复制，其它通过指针共享

  vfork：只复制task_struct和内核堆栈，所以子进程只是线程（没独立的用户空间）。

```
asmlinkage int ckcore_fork(struct pt_regs *regs)
{
    return do_fork(SIGCHLD, rdusp(), regs, 0,NULL,NULL);
}

asmlinkage int ckcore_vfork(struct pt_regs *regs)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, rdusp(), regs, 0,NULL,NULL)
;
}

asmlinkage int ckcore_clone(struct pt_regs *regs)
{
    unsigned long clone_flags;
    unsigned long newsp;
    int __user *parent_tidptr, *child_tidptr;

    /* syscall2 puts clone_flags in r2 and usp in r3 */
    clone_flags = regs->r2;
    newsp = regs->r3;
    parent_tidptr = (int __user *)regs->r4;
    child_tidptr = (int __user *)regs->r5;
    if (!newsp)
        newsp = rdusp();
    return do_fork(clone_flags, newsp, regs, 0, parent_tidptr, child_tidptr);
}
```

execve系统调用

- 通常，子进程从fork返回后会调用exec来开始执行新的程序，如：

```
If (result = fork() == 0){
      /* 子进程代码 */
    …
    if (execve(“new_program”,…)<0)
                perror(“execve failed”);
                exit(1);
}else if (result<0){
    perror(“fork failed”)
}
/* result==子进程的pid，父进程将会从这里继续执行*/
…
```

进程的撤销

- 进程终止的一般方式是exit()系统调用。
  - 这个系统调用可能由编程者明确的在代码中插入
  - 另外，在控制流到达主过程[C中的main()函数]的最后一条语句时，执行exit()系统调用
- 内核可以强迫进程终止
  - 当进程接收到一个不能处理或忽视的信号时
  - 当在内核态产生一个不可恢复的CPU异常而内核此时正代表该进程在运行

线程0

- 在系统初始化阶段由start_kernel()函数从无到有手工创建的一个内核线程
  - 包括：进程描述符、内核态堆栈、mm、fs、files、signals等
  - 线程0最后的初始化工作是创建init内核线程，此后运行cpu_idle，线程0成为idle线程。

```
static void rest_init(void)
{
        kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
        unlock_kernel();
        current->need_resched = 1;
        cpu_idle();
}
```

进程1

- 当调度程序选择到init线程时，init线程开始执行init()函数。
- init()函数最后调用execve()系统调用装入可执行程序init。从此，init内核线程变成一个普通的进程。
- init进程从不终止，因为它创建和监控操作系统外层的所有进程的活动

Linux 基本概念

- 系统调用
- 内存管理
- 进程管理
- 虚拟文件系统（VFS）
- 信号机制
- 内核初始化过程

Ø 提纲

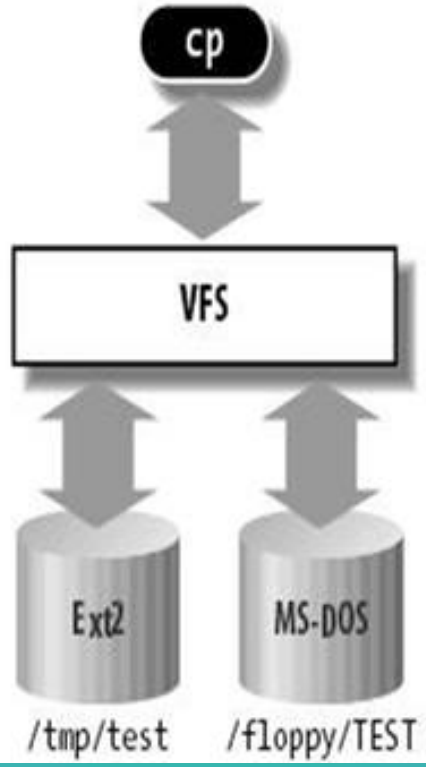
- VFS基本概念
- 主要对象类型
- 存储介质管理
- 操作具体磁盘
- 根文件系统的挂载
- VFS主要系统调用

虚拟文件系统 Virtual Filesystem Switch

- VFS是一个软件层，是用户应用程序与具体文件系统实现之间的抽象层：
  - 对用户界面：一组标准的、抽象的文件操作，以系统调用提供，如read()、write()、open()等。
  - 对具体文件系统界面：主体是file_operations结构，全是函数指针，提供函数跳转表。

VFS在一个简单文件复制操作中的作用

- 假设用户输入shell命令：$cp /floppy/TEST /tmp/test
  - 在cp命令中，它通过VFS提供的系统调用接口进行文件操作。




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

VFS支持的文件系统类型

- VFS支持三种主要类型的文件系统：
  - 基于磁盘的文件系统：它们管理在本地磁盘分区中可用的存储空间。
    - 包括：ext2、ext3、ReiserFS、 MINIX、 MS-DOS、 NTFS 、VFAT 、 DVD文件系统、 JFS 等等。
  - 网络文件系统：用于访问属于其他网络计算机的文件系统所包含的文件
    - NFS、Coda、AFS、SMB、NCP
  - 特殊文件系统
    - 不同于上述两大类，不管理具体的磁盘空间
    - 如/proc

- VFS有下列主要对象类型：
  - 超级块对象（superblock ）

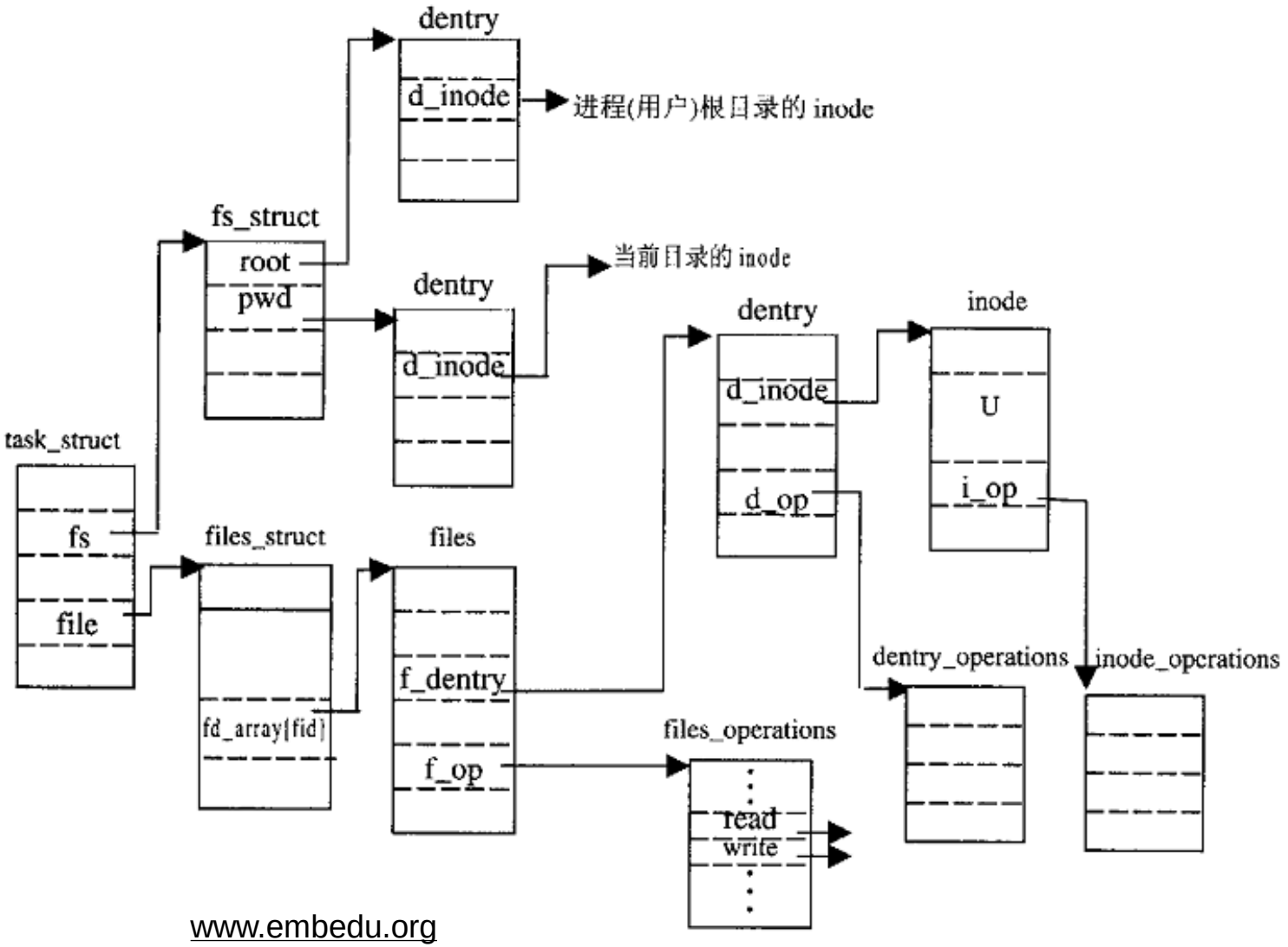
    存放文件系统相关信息：例如文件系统控制块
  - 索引节点对象（inode ）

    存放具体文件的一般信息：文件控制块/inode
  - 目录项对象（dentry ）

    存放目录项与文件的链接信息
  - 文件对象（file ）

    存放已打开的文件和进程之间交互的信息

对象关系图


dentry
d_inode
进程(用户)根目录的 inode
fs_struct
root
pwd
dentry
d_inode
当前目录的 inode
dentry
d_inode
d_op
inode
U
i_op
task_struct
fs
file
files_struct
fd_array[fid]
files
f_dentry
f_op
files_operations
read
write
dentry_operations
inode_operations

存储介质对文件的管理

以ext2为例，涉及的对象有：

- 引导块
- 磁盘超级块

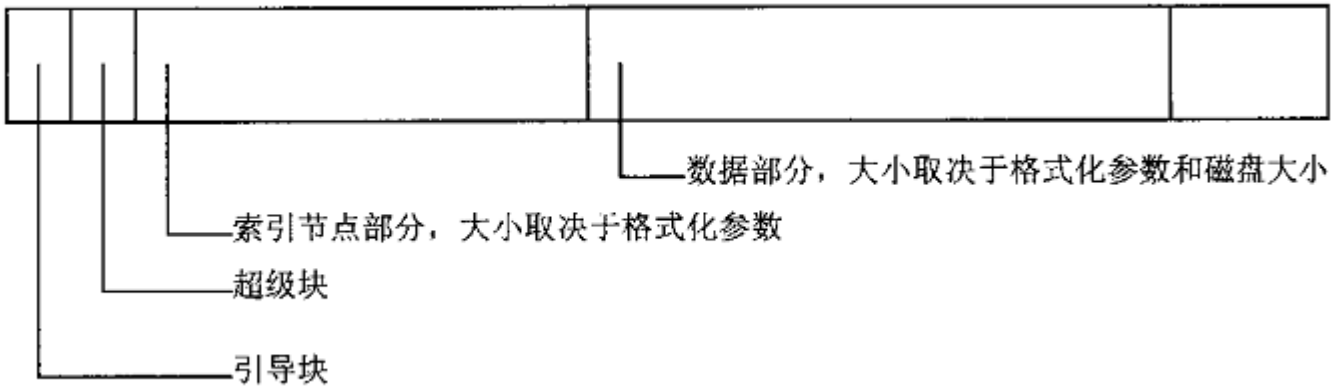
  在设备上的逻辑位置是固定的

- 文件的组织和管理信息

  即索引节点ext2_inode

- 文件中的数据

  包括目录的内容，即目录项ext2_dir_entry_2



- 各种结构在内存和磁盘中的对应关系

  super_block 对应 磁盘超级块

  inode 对应 索引节点ext2_inode

  dentry 对应 目录项ext2_dir_entry_2

- 文件系统的安装：从一个存储设备上读入超级块，在内存中建立起一个super_block结构。再将此设备上的根目录与文件系统中的空白目录挂上钩。

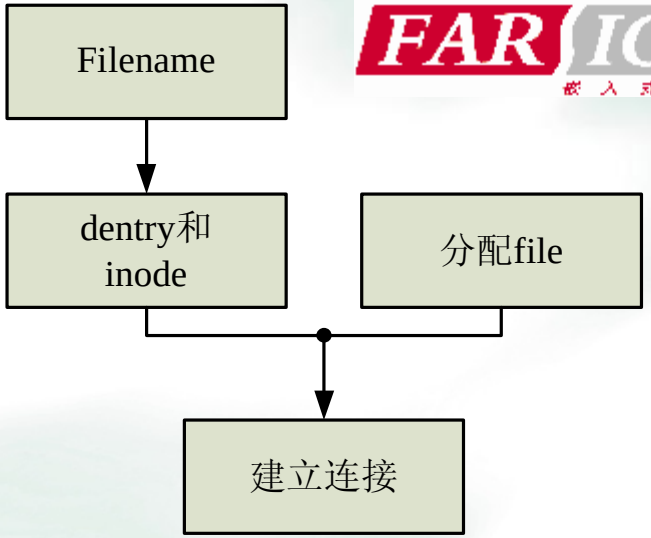
- 每个文件都有目录项和索引节点在磁盘上，只有在需要时才在内存中建立起相应的dentry和inode结构。

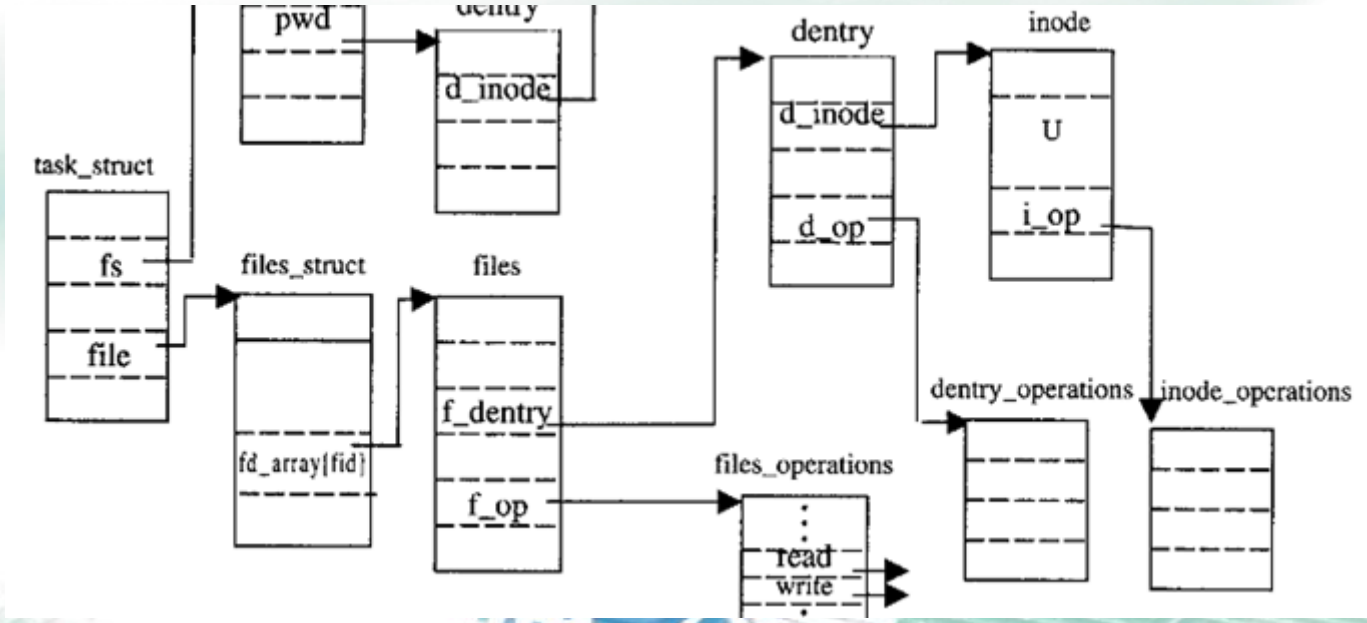
从路径名到目标节点

根据给定的文件路径名在内存中找到或建立代表着目标文件或目录的dentry结构和inode结构。

文件的打开

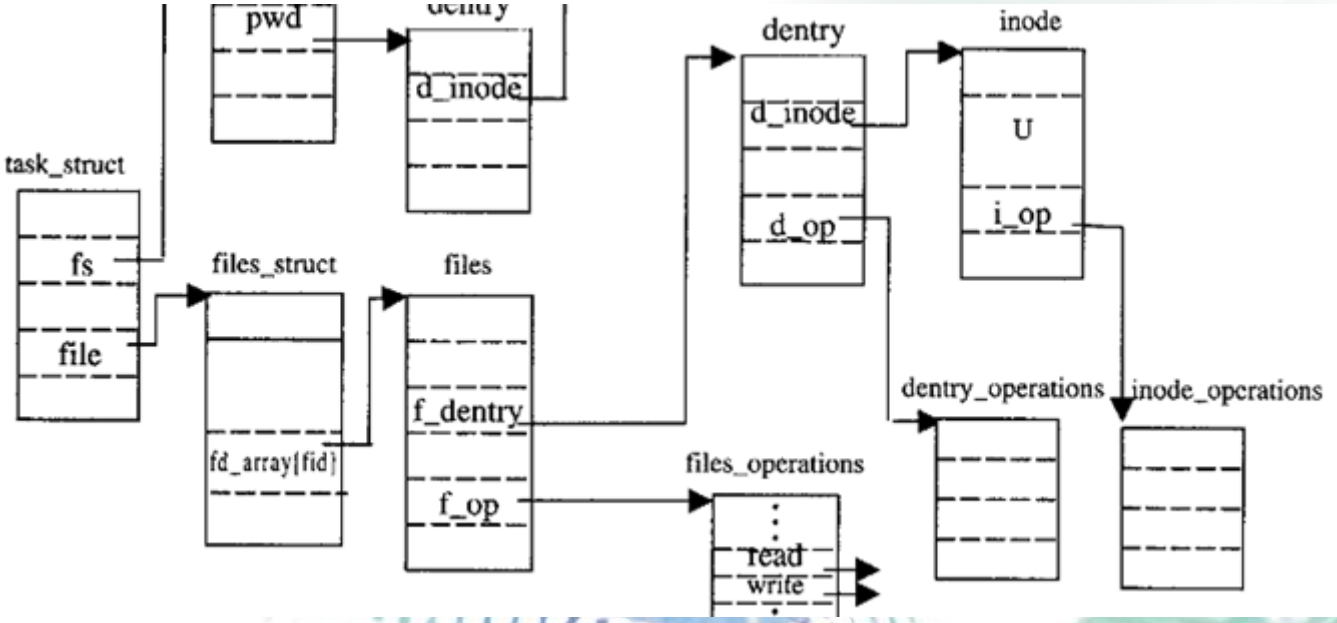
在进程与文件之间建立起连接。

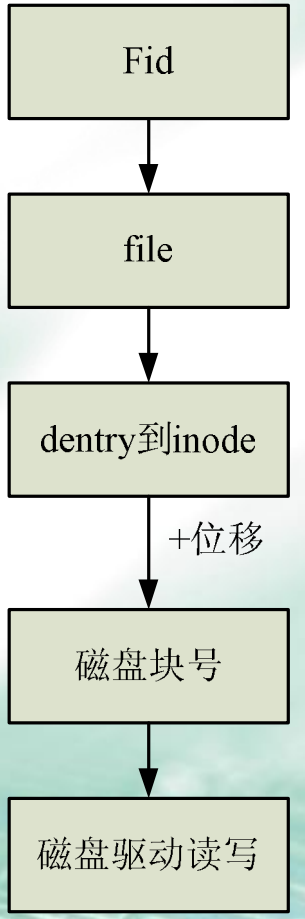




文件的读和写

打开文件后，才能对文件进行读/写。





- 挂载根文件系统
  - 第一阶段：安装一个特殊的文件系统，仅提供作为初始安装点的空目录：init_mount_tree
  - 第二阶段：在该目录上挂载根文件系统

- 挂载一个文件系统
  - 系统调用sys_mount
- 卸载一个文件系统
  - 系统调用sys_umount

如，mount -t ext2 /dev/fd0 /mnt

VFS所处理的系统调用

- mount、umount：挂载/卸载文件系统
- sysfs：获取文件系统信息
- statfs、fstatfs、ustat：获取文件系统统计信息
- chroot：更改根目录
- chdir、fchdir、getcwd：操纵当前工作目录
- mkdir、rmdir：创建/删除目录
- getdents、readdir、link、unlink、rename：对目录项进行操作
- readlink、symlink：对符号链接进行操作
- chown、fchown、lchown：更改文件所有者
- chmod、fchmod、utime：更改文件属性
- open、close、create…

Linux 基本概念

- 系统调用
- 内存管理
- 进程管理
- 虚拟文件系统（VFS）
- 信号机制
- 内核初始化过程

Ø 提纲

- 信号概念
- 信号的安装
- 信号的发送
- 信号的响应
- 用户信号处理函数运行机制

信号

- 信号是一种软中断机制，用于进程间通信。

	中断	信号
说法	处理器收到中断请求	进程收到信号
中断源	其它处理器、外设、处理器本身	其它进程、内核、本进程
同异步	异步，指令结束后检测	异步，系统空间返回用户空间前夕检测
屏蔽	可屏蔽	可屏蔽

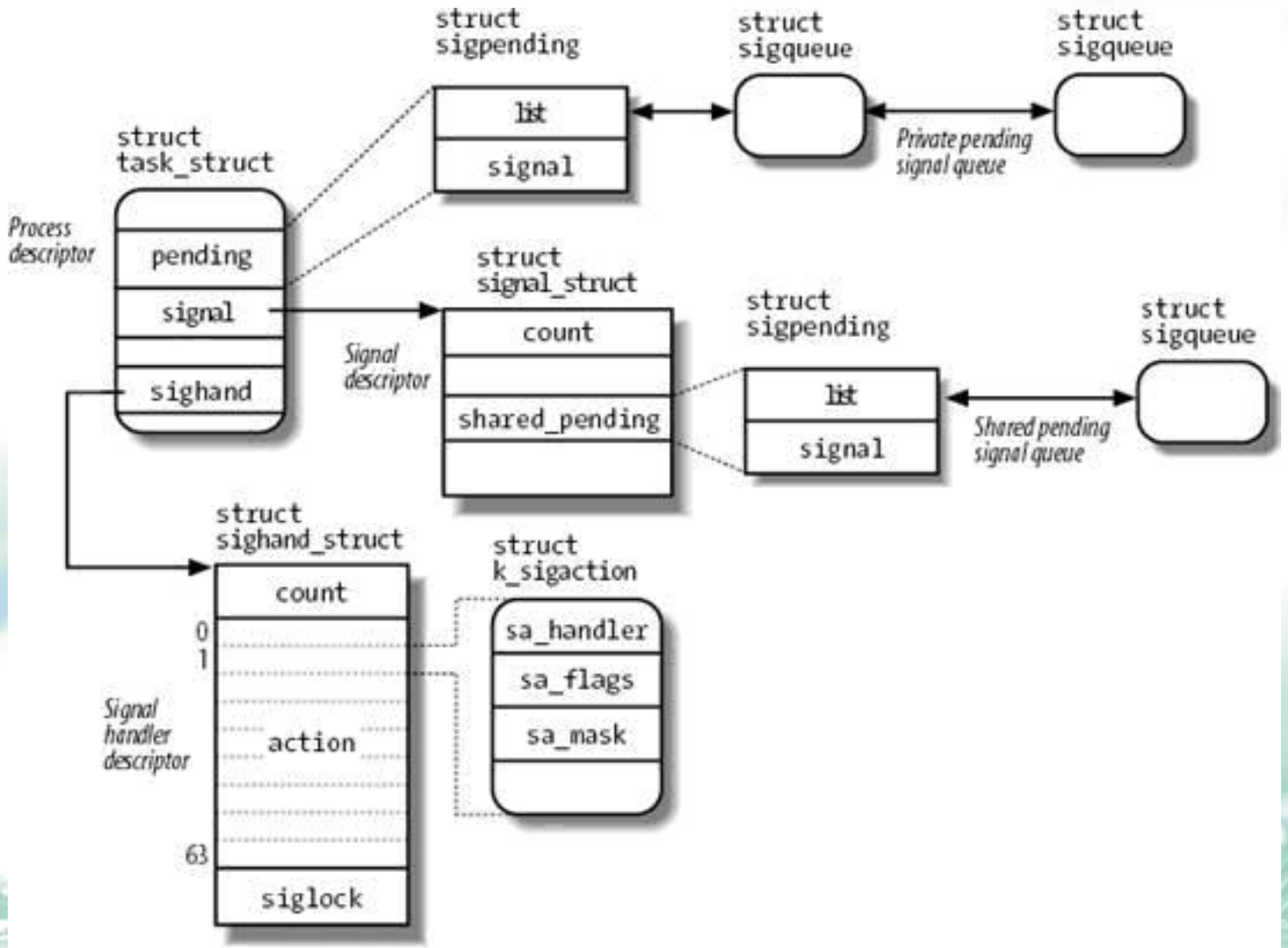
- 通常使用一个数字来标识一个信号。
- 信号可以被发送到一个进程或一组进程。
- 早期Unix系统只定义了32种信号，现在已经扩充到了64种，但是同时兼容原来的32种。
- 使用信号的两个主要目的是：
  - 通知进程已经发生了特定的事件。
  - 强迫进程执行相应的信号处理程序（系统默认或者用户自己定义）。

信号的安装

- 两个目的：
  - 进程将要处理哪个信号
  - 收到该信号时，进程将执行何种操作
- 对于未安装的信号，进程将执行信号的默认动作。
- 用户可用下面两个函数安装信号：
  - sys_signal(signum, handler)
  - sys_sigaction(signum, act, oact)
- 信号安装的具体体现：将handler或act赋值给 current->sighand->action[sig-1]

```
struct sighand_struct {
    atomic_t        count;
    struct k_sigaction  action[_NSIG];
    spinlock_t      siglock;
    wait_queue_head_t   signalfd_wqh;
};
```

进程信号相关结构关系：


struct sigpending
struct sigqueue
struct sigqueue
list
signal
Private pending signal queue
struct task_struct
Process descriptor
pending
signal
sighand
struct signal_struct
count
shared_pending
Signal descriptor
struct sigpending
struct sigqueue
list
signal
Shared pending signal queue
struct sighand_struct
count
0
1
action
63
siglock
Signal handler descriptor
struct k_sigaction
sa_handler
sa_flags
sa_mask

信号的发送

- 信号可以发送给指定的一个或者多个进程。
- task_struct中有个pending域：

```
struct sigpending pending;
struct sigpending{
    struct list_head list;
    sigset_t signal;
};
```

```
struct sigqueue {
    struct list_head list;
    int flags;
    siginfo_t info;
    struct user_struct *user;
};
```

其中，signal是标识未处理信号集的位图， list是sigqueue队列。

- 信号发送具体体现：在指定进程task_struct的未处理信号集中添加该信号。
- 可靠信号和不可靠信号：是否支持排队

信号发送函数

- int kill(pid_t pid,int signo)

  将信号发送给一个或一组进程。

  对于pid的值

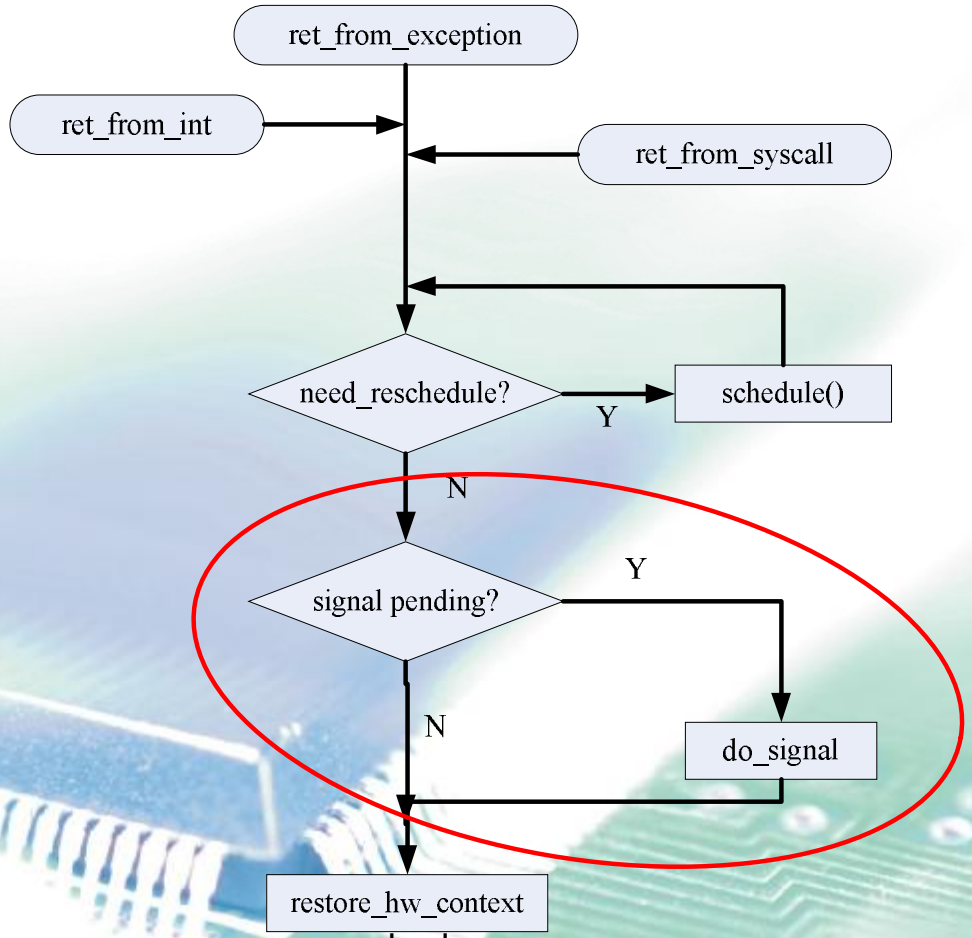
  1. ＞0，发送信号给指定的进程;
  2. ＝0，把信号发送给同组的所有进程;
  3. ＝-1，把信号发送给除0号、1号以及current之外的所有进程;
  4. ＜-1，把信号发送给指定的进程组中的所有的进程。

- int sigqueue(pid_t pid, int sig, const union sigval val)

  只能向一个进程发送信号，支持发送信号带附加信息。

信号的响应

- 响应时机：


ret_from_exception
ret_from_int
ret_from_syscall
need_reschedule?
Y
schedule()
N
signal pending?
Y
N
do_signal
restore_hw_context

响应方式：

1. 显式的忽略信号
2. 执行系统默认的缺省操作，可以是：
   - Terminate：进程被杀死
   - Dump：进程被杀死，且如果可能，创建包含进程上下文的可用于调试的core文件
   - Ignore：简单的忽略信号
   - Stop：进程被停止，状态置为TASK_STOPPED
   - Continue：如果进程被挂起，则状态置为TASK_RUNNING。否则忽略该信号
3. 捕获信号
   - 为了执行用户希望的对某个事件的处理，可以由用户指定某个信号的处理函数。

do_signal()

一位一位的检查当前被挂起的非阻塞信号，调用sighand中action数组对应的处理方法:

- 如果是SIG_IGN（忽略信号）,忽略信号，返回;
- 如果是SIG_DFL（缺省操作）,找到对应的缺省处理方式;
- 如果信号有用户注册的处理程序，就调用handle_signal()强迫执行该处理程序。

handle_signal()

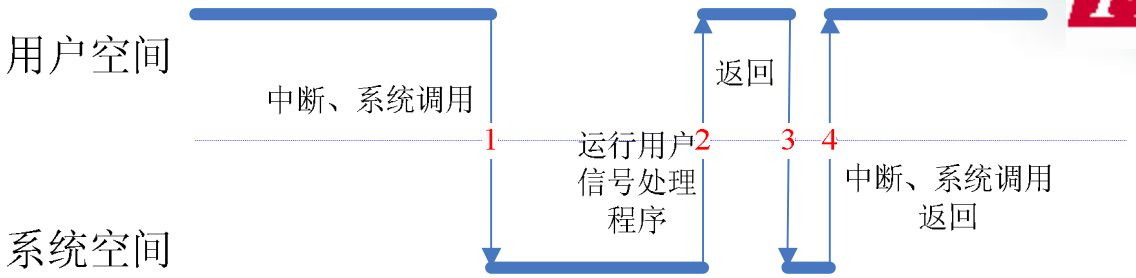
handle_signal运行在内核态，而信号处理程序在用户态的代码段中，运行在用户态。

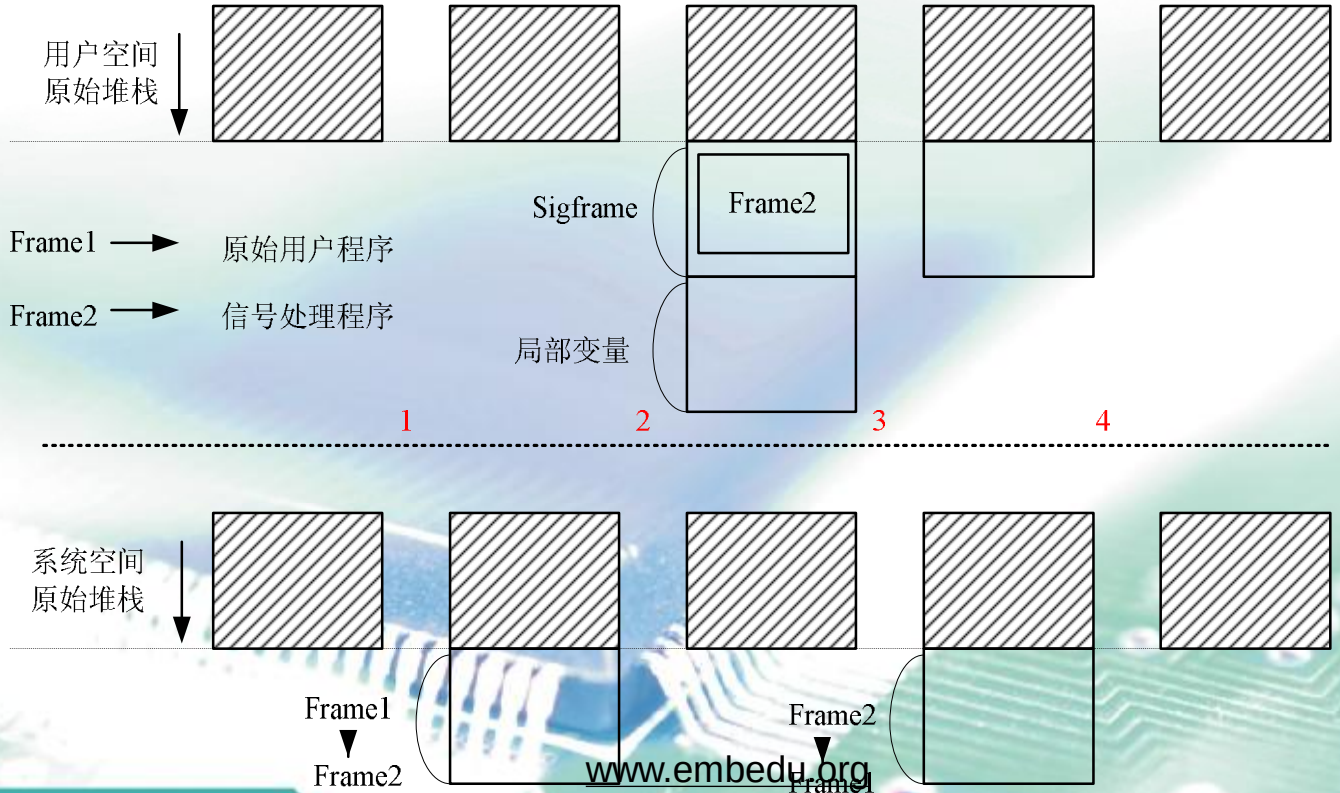
问题：

1. 必须返回用户态执行信号处理程序；
2. 必须按照原来进入内核的方式返回用户态；
3. 一旦返回用户态，内核堆栈就被清空，如何保存内核堆栈的内容

Linux采用的解决办法：

- 把保存在内核态堆栈中的上下文拷贝到当前进程的用户态堆栈中；
- 建立好信号处理程序所需的堆栈环境；
- 运行信号处理程序；结束时，调用sys_sigreturn()把上面保存的内核堆栈的内容再拷贝回内核堆栈；
- 然后正常返回用户态。





Linux 基本概念

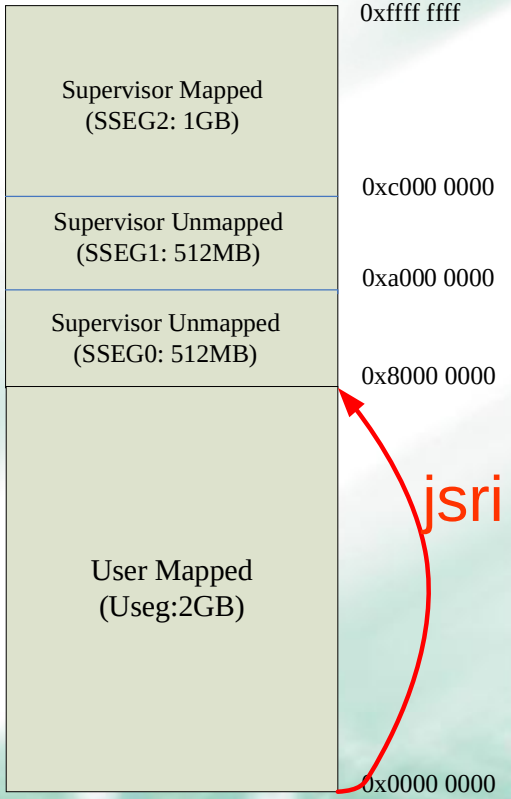
- 系统调用
- 内存管理
- 进程管理
- 虚拟文件系统（VFS）
- 信号机制
- 内核初始化过程

系统初始化三个阶段

- 第一阶段：CPU本身的初始化，例如虚拟地址模式的开启；
- 第二阶段：系统基础设施的初始化，例如内存管理和进程管理的建立和初始化；
- 第三阶段：根设备的安装和外部设备的初始化，载入init进程。

第一阶段（head.S部分）

- ckcore head.S 完成的主要工作：
  - 初始化PSR à
  - 使能cache à
  - 建立内核初始化需要的页面映射 à
  - 使能MMU à
  - 内核跳转到SSEG0段运行 à
  - 建立初始异常向量表 à
  - 初始化BSS段 à
  - 跳到start_kernel执行下一步初始化



第二阶段（start_kernel函数）

- 初始化系统的核心数据结构，主要包括：

setup_arch():执行与体系结构相关的设置

trap_init():设置各种异常入口地址

init_IRQ():初始化IRQ中断处理机制

sched_init():初始化每个处理器的可运行进程队列，设置系统初始化线程即0号线程。

softirq_init():对软中断子系统进行初始化

console_init():初始化控制台、显示器

init_modules():初始化kernel_module

fork_init():定义系统最大进程数

最后进入rest_init()函数并调用kernel_thread()创建init内核线程,进行系统配置。 init内核线程占用进程描述表的第一项，由它来创建其他进程完成系统初始化。

第三阶段（init线程）

- init内核线程首先调用do_basic_setup()来初始化外部设备及加载驱动程序。如： PCI总线初始化、网络初始化、文件系统初始化、加载文件系统。
- 初始化结束，打开/dev/console 设备重新定向控制台。
- 用系统调用execve执行用户态程序/sbin/init。
- 至此，linux的内核初始化工作完成。

参考资料

1. 《Linux内核源代码导读》陈香兰 著.
2. 《LINUX内核源代码情景分析》毛德操 胡希明 著.
3. 《c640 MMU Micro-Architecture》中天 著.
4. linux2.6.30.4 内核源代码.

Thank you !