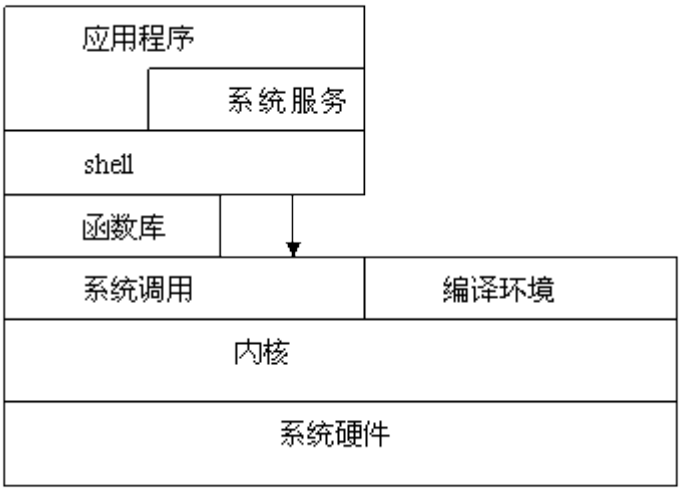
# Chpt1 概述

2013年1月7日
15:26

**1、关于操作系统概念结构**




**2、list_for_each和list_for_each_safe对比**

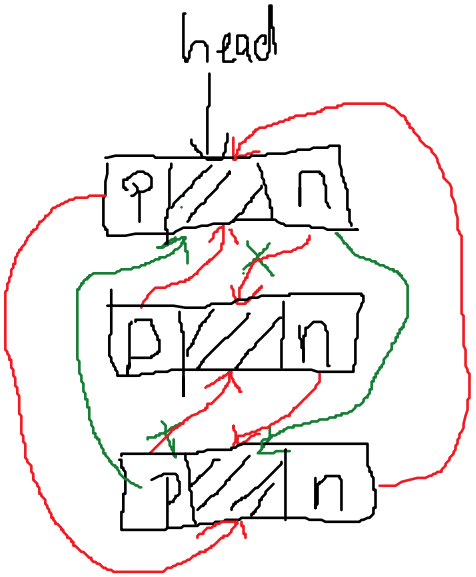
```
#define list_for_each(pos, head) \
     for (pos = (head)->next; prefetch(pos->next), pos != (head);
     pos = pos->next)
#define list_for_each_safe(pos, n, head) \
     for (pos = (head)->next, n = pos->next; pos != (head); \
          pos = n, n = pos->next)
```

（1）从定义可以看出，list_for_each_safe比list_for_each多一个变量n用来暂存将要被删除的节点指针，始终等于pos->next。因为在遍历链表过程中pos指针可能发生变化，因此用n暂存将要访问的下一个元素比较安全。比如，在遍历过程中使用list_del函数

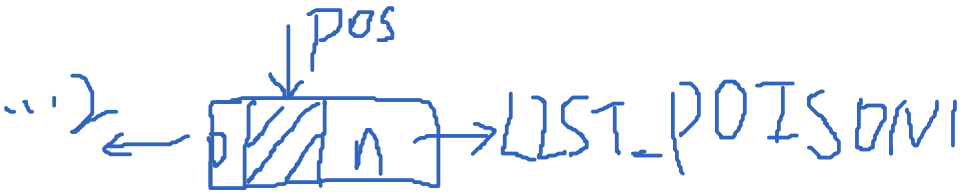
```
static inline void list_del(struct list_head *entry)
{
     __list_del(entry->prev, entry->next);
     entry->next = LIST_POISON1;
     entry->prev = LIST_POISON2;
}
```

entry形参传递pos变量，调用该函数之后pos状态如右图所示。

（2）为确保安全，总是优先使用list_for_each_safe




含3个元素的循环链表，
绿线表示删除操作




调用list_del之后pos指针状态

**3、list其他函数分析**

（1）

```
static inline void list_replace(struct list_head *old,
                struct list_head *new)
```

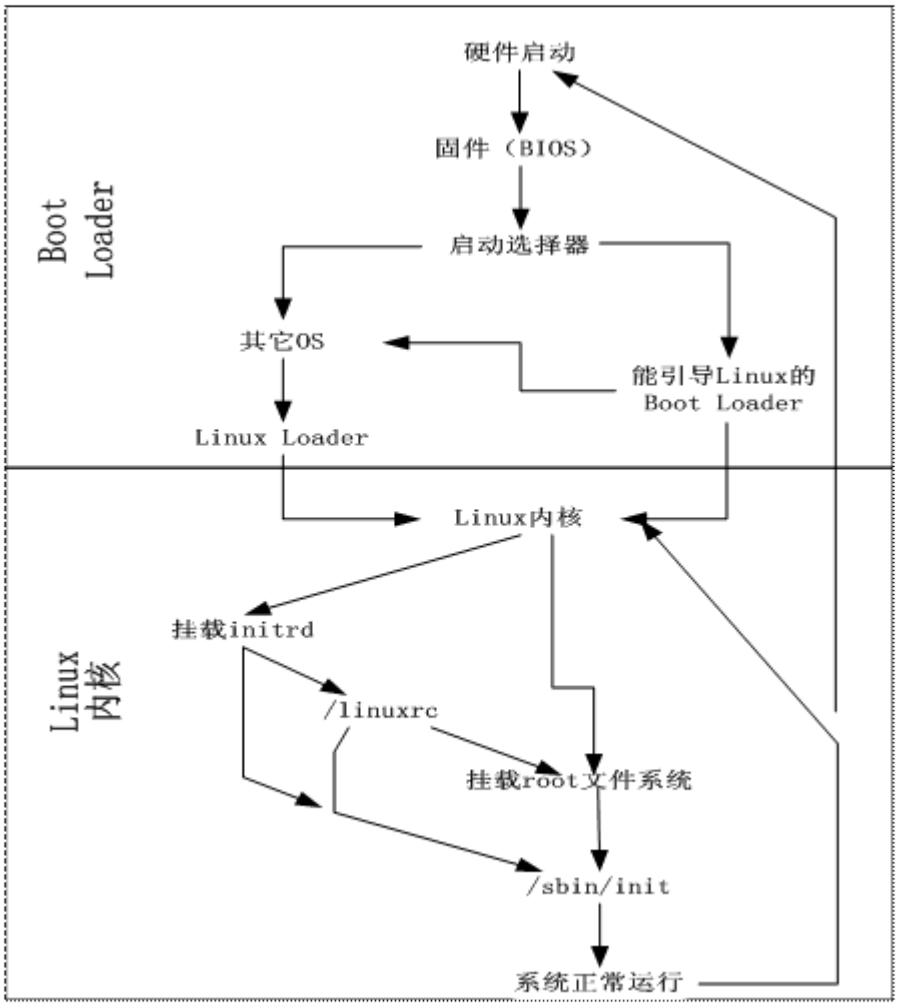
用new节点替换old节点，但替换后old节点还没释放，需要单独使用kfree释放。

（2）

**4、Linux操作系统启动流程分析**

系统启动指从计算机上电到显示器显示用户登录提示界面的整个过程。

过程主要分为2个阶段：载入内核和初始化运行环境。




源文档 <http://www.kerneltravel.net/journal/i/04.htm>

在分析之前先区别2个不同的概念：引导加载程序与引导程序。引导加载程序作用是将硬盘中的引导程序读到系统内存中然后将控制权交给引导程序，即BIOS。引导程序主要任务是将内核从硬盘上读到内存中，然后跳到内核的入口点去运行，即BootLoader。

（1）载入内核——将内核载入内存，并将控制权转交给内核的阶段

这一阶段是引导加载程序工作的过程，是BootLoader的主战场。i386平台上使用的BootLoader为GRUB或LILO，ARM平台上常用有Uboot。

BootLoader的职责：判断要载入的内核，这可以要求用户选择；载入内核和内核需要使用的相关数据，比如initrd或参数信息；为内核准备运行环境，比如让CPU进入特权模式；让内核投入运行。

（2）初始化运行环境

——检测硬件，初始化硬件；

——创建第一个进程（初始进程INIT_TASK，其实就是Idle进程）；

——由INIT_TASK创建新进程，初始化总线、网卡等外设和启动内核后台线程；

——设置文件格式，初始化文件系统，安装root文件系统（root文件系统作用是为应用程序提供环境）；

——打开/dev/console设备，重定向stdin/stdout/stderr到控制台，使用execve()系统调用加载执行init程序，至此系统进入用户态。

扩展几点：

（1）系统上电或复位后，所有CPU通常都由CPU制造商预先设定的地址开始执行，我们的BootLoader（PC上则是BIOS）则要放在该位置处。

（2）从启动过程中分析，启动系统包含BootLoader、BootLoader参数、内核映像和根文件系统4部分。在固态存储设备中的4者典型分布如下图。


屏幕剪辑的捕获时间: 2013/1/8 17:30

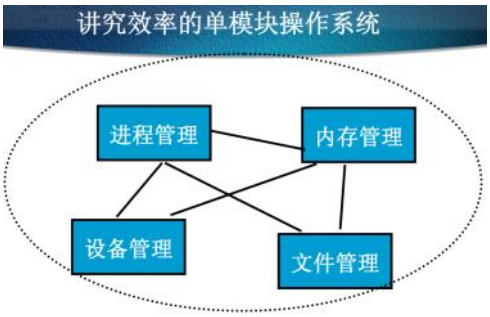
（3）启动流程图中的initrd

initrd可以看作是RAM映像，包含了一些非操作系统启动需要但非必须的模块。initrd主要用来把系统的启动划分为两个阶段：初始启动的内核只需保留最精简的驱动程序最小集，此后，在启动必须加载附加的模块时，从initrd中加载。

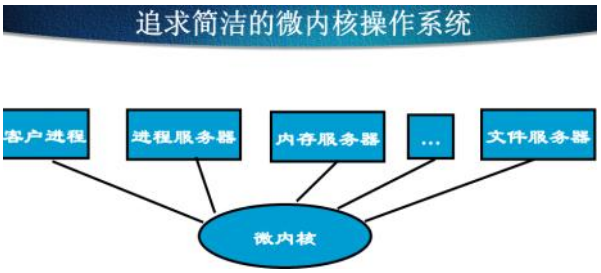
# 第1章习题作业

2013年1月6日
13:39

**习题1**
操作系统定义：计算机中的系统软件，有效合理的方式管理计算机的软硬件资源，使用户能灵活方便使用计算机。
（1）从不同的角度，操作系统展现的功能不同。“操作系统总是把繁琐留给自己，简单留给用户”。从设计者的角度，操作系统必须包含5大功能：进程管理（CPU）、内存管理、设备管理、文件系统、操作系统接口。
（2）操作系统演变经历了单批到处理，多批到处理到分时处理。
（3）从程序设计思想的角度可将操作系统分为单内核操作系统和微内核操作系统，Linux为单内核操作系统。单内核操作系统能实现模块间的任意调用，采用面向过程的设计思想；而微内核操作系统采用面向对象的设计思想，通过通信方式进行模块和内核的交互，如下图。




屏幕剪辑的捕获时间: 2013/1/6 13:55




屏幕剪辑的捕获时间: 2013/1/6 13:56
（4）操作系统组成：内核+系统程序（编译环境+API+AUI）

**习题2**
对操作系统的发展从2方面来说：
（1）技术发展：比如程序设计理论的发展、硬件技术的提升等
（2）用户需求及市场：比如实时系统的需求、自由开源网络环境的推动以及各操作系统设计厂商间的竞争推动等

**习题3**
硬件性价比比较低的时候，操作系统设计追求低成本高性能；
硬件性价比比较高以后，操作系统设计不再考虑硬件成本需求，着力在充分利用硬件的性能上；
当计算机被普及以后，操作系统的设计开始追求交互的便捷性

**习题4**
程序设计理论约束着操作系统的设计，从结构化设计到对象化设计，操作系统总是最后使用程序设计理论的软件之一；
操作系统是否要完全对象化至今处在徘徊期，完全面向对象就意味着相对过程设计要降低效率、扩充移植维护性能
人机交互设计的宗旨是为用户服务

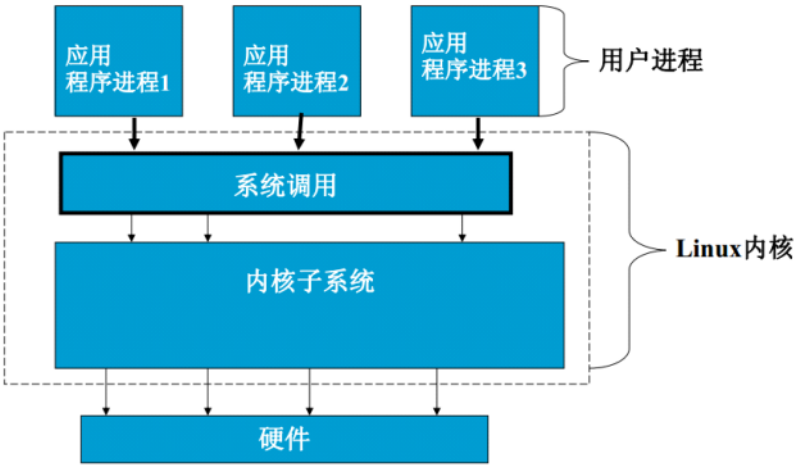
**习题5**
Linux伴随着开源网络文化诞生而逐渐发展壮大。开源的意义十分重大。

**习题6**
POSIX：Portable Operating System Interface 可移植操作系统接口
当一个操作系统满足POSIX标准时，在该系统上写的应用程序都能在任何UNIX系统中运行。

**习题7**
GNU：GNU is Not UNIX是自由软件基金会的一个项目，该项目的目标是开发一个自由的UNIX版本。
Linux 的开发使用核很多GNU工具，Linux上用于实现POSIX.2标准的工具几乎都是GNU项目开发的

**习题8**
Linux开发模式为：由广泛分布于世界各地的软件爱好者，以互联网为纽带，通过BBS，新闻组及电子邮件方式同时参与软件开发。
这种开发模式的优势在于凭兴趣开发，热情和创造性高；缺点是鱼龙混杂的环境下很难处理好开发过程中出现的问题。

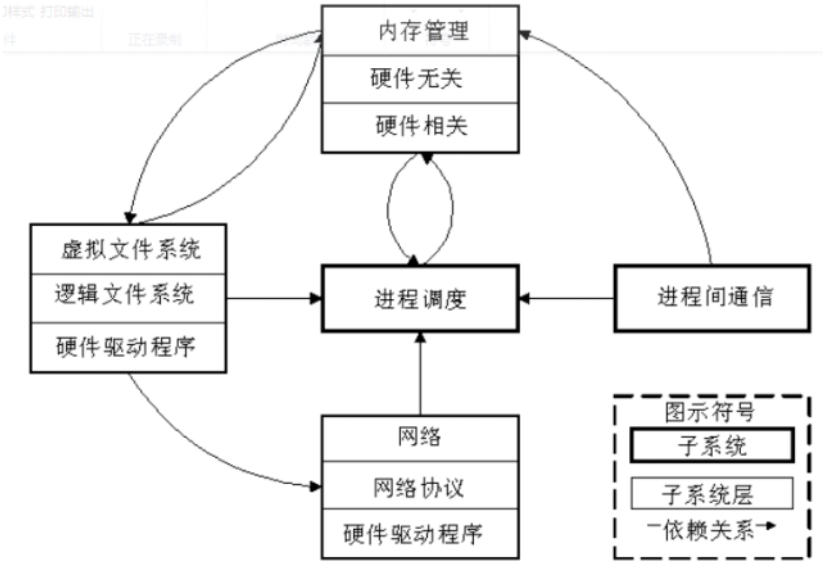
**习题9**
Linux系统的组成如下图，Linux内核介于硬件和用户应用程序之间，是跨接用户程序和硬件的桥梁，也是用户应用程序的环境基础。Linux内核包括内核子系统和系统调用2大模块。




屏幕剪辑的捕获时间: 2013/1/6 14:34

**习题10**

LInux内核子系统由进程调度、内存管理、虚拟文件系统、网络接口和进程间通信5个子系统组成，如下图。
设备管理功能通过虚拟文件系统实现。




屏幕剪辑的捕获时间: 2013/1/6 14:38

**习题11**

截止到2013年01月03日，正在开发的内核版本为3.8-rc2，最新稳定版本为3.7.1版。

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| mainline: | **3.8-rc2** | 2013-01-03 | [Full Source] | [Patch] | [View Patch] | | [Gitweb] | |
| stable: | **3.7.1** | 2012-12-17 | [Full Source] | [Patch] | [View Patch] | | [Gitweb] | [Changelog] |
| mainline: | **3.7** | 2012-12-11 | [Full Source] | [Patch] | [View Patch] | | [Gitweb] | |
| stable: | **3.6.11 (EOL)** | 2012-12-17 | [Full Source] | [Patch] | [View Patch] | [View Inc.] | [Gitweb] | [Changelog] |
| stable: | **3.5.7 (EOL)** | 2012-10-12 | [Full Source] | [Patch] | [View Patch] | [View Inc.] | [Gitweb] | [Changelog] |
| stable: | **3.4.24** | 2012-12-17 | [Full Source] | [Patch] | [View Patch] | [View Inc.] | [Gitweb] | [Changelog] |
| stable: | **3.2.36** | 2013-01-03 | [Full Source] | [Patch] | [View Patch] | [View Inc.] | [Gitweb] | [Changelog] |
| stable: | **3.0.57** | 2012-12-17 | [Full Source] | [Patch] | [View Patch] | [View Inc.] | [Gitweb] | [Changelog] |
| stable: | **2.6.34.13** | 2012-08-20 | [Full Source] | [Patch] | [View Patch] | [View Inc.] | [Gitweb] | [Changelog] |
| stable: | **2.6.32.60** | 2012-10-07 | [Full Source] | [Patch] | [View Patch] | [View Inc.] | [Gitweb] | [Changelog] |
| linux-next: | **next-20130104** | 2013-01-04 | | | | | [Gitweb] | |

源文档 <http://www.kernel.org/>

**习题12**

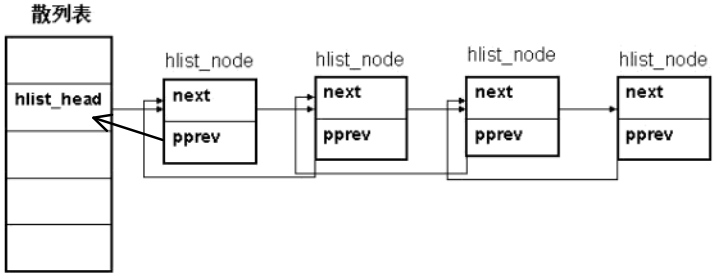
浏览Linux内核源代码网站http://lxr.linux.no/
linux/kernel目录下包含了进程调度、CPU管理、时钟、信号量等通用的与硬件体系结构无关的代码，全部为.c文件，与体系结构有关的代码在linux/arch/目录下。
有关文件还有待详细分析。

**习题13**

list.h文件中Hash表数据结构如下：

```
struct hlist_head {
    struct hlist_node *first;
};
struct hlist_node {
```

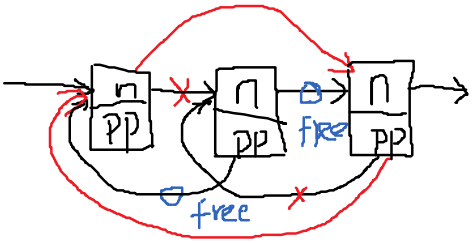
```
    struct hlist_node *next, **pprev;
};
```
用于Hash表的链表与循环链表不同，Hash表不是循环，而是有“头”无“尾”的；
Hash表节点的指针域也不同，**pprev为二级指针。
Linux中Hash链表用于在分离链接法中避免冲突问题。




源文档 <http://www.cnblogs.com/westfly/archive/2011/05/30/hlist_linux.html>

（1）Hash表初始化
包括hlist_head结构的初始化和hlist_node的初始化，
```
#define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)              // list_head初始化
static inline void INIT_HLIST_NODE(struct hlist_node *h)       // list_node初始化
{
    h->next = NULL;
    h->pprev = NULL;
}
```
全都初始化为NULL，NULL在Stddef.h中定义为((void *)0)。
（2）添加hlist_node节点
```
static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
{
    struct hlist_node *first = h->first;
    n->next = first;
    if (first)
        first->pprev = &n->next;
    h->first = n;
    n->pprev = &h->first;
}
```
将新节点插入到hlist_head和和第一个hlist_node之间。
还有2个添加节点的函数，其中的*next必须满足!=NULL，
```
// 将n添加到next前
static inline void hlist_add_before(struct hlist_node *n,
                    struct hlist_node *next)
// 将n添加到next后
static inline void hlist_add_after(struct hlist_node *n,
                    struct hlist_node *next)
```

（3）删除节点
```
static inline void __hlist_del(struct hlist_node *n)
{
    struct hlist_node *next = n->next;
    struct hlist_node **pprev = n->pprev;
    *pprev = next;
    if (next)
        next->pprev = pprev;
}
static inline void hlist_del(struct hlist_node *n)
{
    __hlist_del(n);
    n->next = LIST_POISON1;
    n->pprev = LIST_POISON2;
}
```
参考右图。注意删除函数中没有对hlist_node *n = NULL的情况进行判断。
可以只调用__hlist_del函数而自己使用kfree释放删除的节点吗？




删除hlist_node节点示意图

（4）遍历
```
// 获取节点的起始地址，从而可以访问节点中的域
#define hlist_entry(ptr, type, member) container_of(ptr,type,member)

// 2种遍历方法，可以与list_head的遍历对比
#define hlist_for_each(pos, head) \
    for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
         pos = pos->next)
#define hlist_for_each_safe(pos, n, head) \
    for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
         pos = n)
```
说明：
```
#define container_of(ptr, type, member) ({                \
    const typeof(((type *)0)->member) *__mptr = (ptr); \
    (type *)((char *)__mptr - offsetof(type, member)); })
```
黄色：获取ptr在内存中的绝对地址
青色：获取member域在type结构中的偏移地址

（5）hlist_head赋值
```
static inline void hlist_move_list(struct hlist_head *old,
                    struct hlist_head *new)
{
    new->first = old->first;
    if (new->first)
        new->first->pprev = &new->first;
    old->first = NULL;
```

```
}
```
赋值new=old，关键在于添加从第一个hlist_node->pprev到hlist头的指针。

（6）判断节点是否已经添加到Hash链表中

```
static inline int hlist_unhashed(const struct hlist_node *h)
{
    return !h->pprev;
}
```
从这里我们可以看到pprev指针设计的用途。

（7）Linux内核Hash表使用实例程序

```
// fun   : use hlist, refrence to BOOK_CLJ
// data  : 2013.01.07
// author: xhzuoxin

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>  // include kmalloc()
#include <linux/list.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("xhzuoxin");

#define N_HEADS              (10)
#define N_NODES              (20)

// heads
struct hashtbl
{
    int n;
    struct hlist_head *h;
};
// nodes
struct hnode
{
    int data;
    struct hlist_node nd;
};

// global variables
struct hashtbl head;

static int __init hlist_init(void)
{
    struct hnode *node;  // extra variable
    struct hlist_node *pos;
    struct hnode *p;
    int i = 0;
    int j = 0;

    // init head
    printk("hlist head init... \n");
    head.n = N_HEADS;
    head.h = (struct hlist_head*)kmalloc(head.n * sizeof(struct hlist_head),
    GFP_KERNEL);              // alloc space for head.h
    if (NULL == head.h)
    {
        return -1;
    }
    for(i=0; i<head.n; i++)
    {
        INIT_HLIST_HEAD(&head.h[i]);
    }

    // create N_NODES nodes
    for(i=0; i<N_NODES; i++)
    {
        node = (struct hnode *)kmalloc(sizeof(struct hnode), GFP_KERNEL);
        node->data = i + 1;
        j = (node->data) % 10;
        hlist_add_head(&node->nd, &head.h[j]);  // add node after head
        printk("Node %d has added to head[%d]...\n", node->data, j);
    }

    // list for each
    for(i=0; i<head.n; i++)    // for each head
    {
        printk("head[%d]\n", i);
        hlist_for_each(pos, &head.h[i])
        {
            p = hlist_entry(pos, struct hnode, nd);
            printk("    access %d...\n", p->data);
        }
    }

    return 0;
}

static void __exit hlist_exit(void)
{
    struct hlist_node *pos, *n;
    struct hnode *p;
    int i = 0;

    // del for each
    for(i=0; i<head.n; i++)    // for each head
        hlist_for_each_safe(pos, n, &head.h[i])
        {
```

```
head[2]
    access 12...
    access 2...
head[3]
    access 13...
    access 3...
head[4]
    access 14...
    access 4...
head[5]
    access 15...
    access 5...
head[6]
    access 16...
    access 6...
head[7]
    access 17...
    access 7...
head[8]
    access 18...
    access 8...
head[9]
    access 19...
    access 9...
[root@localhost chpt1]# _
```

屏幕剪辑的捕获时间: 2013/1/8 10:31

Hash表init程序执行部分结果

```
            int tmp = 0;

            if(pos != NULL)
            {
                hlist_del(pos);
                p = hlist_entry(pos, struct hnode, nd);
                tmp = p->data;
                kfree(p);
                printk("Node %2d has removed from hlist...\n", tmp);
            }
        }

    printk("Remove all.\n");
}

module_init(hlist_init);
module_exit(hlist_exit);
```

程序说明：
init程序中首先建立一个N_HEADS=10大小的动态数组，数组元素类型为struct hlist_head。
在使用求余散列函数将1-20共N_NODES=20个数插入到散列表中。
exit程序中逐个删除每个元素。

Makefile文件如下：

```
obj-m:=hlist.o
CURRENT_PATH:=$(shell pwd)
LINUX_KERNEL:=$(shell uname -r)
LINUX_KERNEL_PATH:=/usr/src/kernels/$(LINUX_KERNEL)-i686

all:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) modules
clean:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) clean
```

注意：LINUX_KERNEL_PATH指内核所在路径，不同的Linux发布版可能不同，此处使用的是REHL5.1系统。

# Chpt2 内存寻址

2013年1月6日
15:32

BIOS启动流程分析：
BIOS中包含POST（加电自检）、开机菜单设置、加载引导扇区、BIOS中断几部分程序。
（1）开机后，BIOS首先检查内存和硬件设备，检测完成后会在显示器上显示PCI设备清单；
（2）启动菜单中，用户可以设置各种启动参数（联想笔记本按F2进入）；
（3）接着就是从启动设备中装载引导扇区（第0扇区）程序到内存，载入位置固定在0000:7C00（对应有效地址7C000）处，执行完该载入过程则进入扇区1的boot.s（如下）代码执行；
（4）引导扇区被载入时会频繁的调用BIOS的磁盘、串口、屏幕、键盘等终端；
BIOS要做的工作就是找到启动分区所在设备。

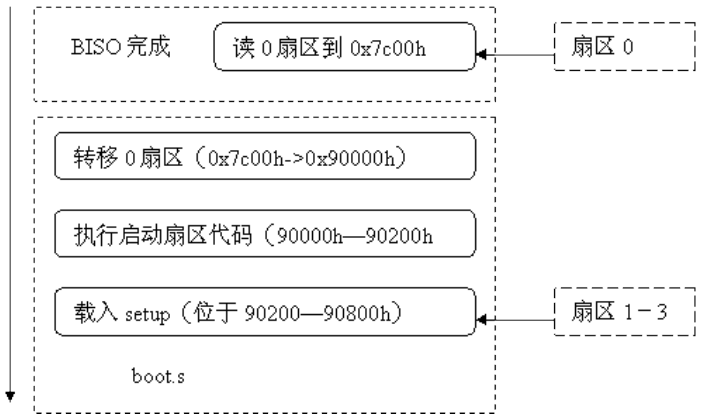
现在到启动扇区了，启动扇区安排在磁盘（硬盘/软盘）的第0扇区，大小为512Bytes,启动扇区的标志是扇区结尾为0x55和0xAA这2个字节——当检测到这两个字节时就会跳转到0000:7C00处执行。

BootLoader功能参见Chpt1。
对SagaLinux中boot.s（负责载入setup.s程序到内存）代码的分析如下：

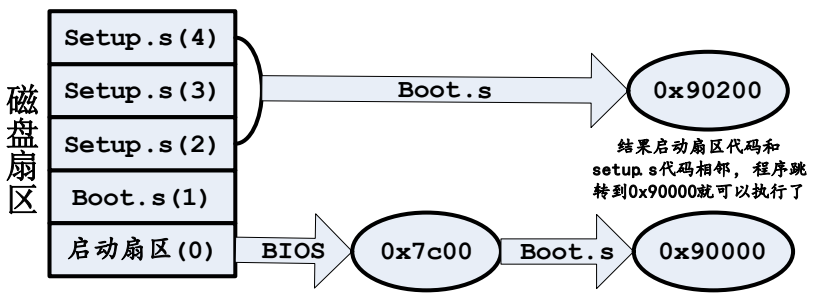
```
; 该段程序功能：载入BootLoader程序（这里是setup.s程序功能）
; 编译后刚好占512Bytes，占用第1个扇区
; 从第2（2~4）个扇区开始存放setup.s代码（setup.s完成内核载入和一些环境的初始化初始化）
[BITS 16]
[ORG 0]
; 之前BIOS完成了读取0扇区到0x7C00位置，0x7C00位置为一个固定的默认位置
jmp start

BOOTSEG EQU 0x07c0
INITSEG EQU 0x9000
SETUP EQU 0x9020
SYSSEG  EQU 0x1000

bootmsg          db    'Loading',0
dot         db    '.',0

; (1)载入启动扇区数据
start:            ; 在0x7c00所在的段内执行，将启动扇区0x7c00~0x7e00处的代码装载到0x90000~0x90200处，并跳转到0x90000处执行
    mov   [bootdrive], dl; dl->[0]
    mov   ax, BOOTSEG
    mov   ds, ax; ds=ax=0x07c0
    mov   ax, INITSEG
    mov   es, ax; es=ax=0x900
    xor   di, di; di=0
    xor   si, si; si=0
    mov   cx, 0x0200                  ; Bootsector is 512 bytes.
    cld
    rep
    movsb
    jmp   INITSEG:go    ; 跳转到0x90000处执行

; (2)进行相关初始化，载入第二阶段的程序
go:                ; 位于0x9000所在的段内
    mov   ax, INITSEG
    mov   ds, ax; ds=0x9000
    mov   es, ax; es=0x9000
    cli   ; TI(FLAG)=0 清中断，防止
    mov   ss, ax; 初始化堆栈所在的段为ss=0x9000
    mov   sp, 0xeeee; 初始化堆栈地址为sp=0xeeee
    sti   ; TI(FLAG)=1 开中断

    mov   si, bootmsg; si=bootmsg的偏移地址
    call  write_message; 在显示器上显示bootmsg


    mov   ax, INITSEG
    mov   es, ax; es=0x9000
    mov   bx, 0x0200;
    call  reset_drive
    call  start_loading; 装载磁盘中的startup.s代码到内存中


    jmp   INITSEG: 0x0200            ; setup loaded at 0x90200


reset_drive:

    push  ax
    mov   ah, 0
    int   0x13 ; 调用BIOS中断重置磁盘驱动器
    pop   ax
        ret

start_loading:                                ; 读取扇区号为0x02~0x04的3个扇区数据，放入起始地址为0x90200的内存中
    mov   ah, 0x02             ; ah = read function at int 13h 读扇区中断功能号
```

```
    mov   al, 0x03              ; al = the number of sectors to read
        mov ch, 0                ; ch = track number 柱面号
    mov   cl, 0x02              ; cl = starting sector 预读起始扇区
        mov dl, [bootdrive]             ; dl = drive number
        mov dh, 0                ; dh = head number磁头号

read_one_sector:                            ; 读一个扇区的过程
    push  ax
    mov   al, 0x1
    mov   ah, 0x2
    int   0x13                      ; 启动读扇区中断
    push  bx
      mov   si, dot
    call  write_message             ; 输出读扇区信息到显示器
    pop   bx
    pop   ax

    inc   cl                    ; increment sector value
    dec   al                    ; decrement the number of sectors to read
    add   bx, 0x200             ; increment the offset 每个扇区大小为512字节(0x200)
    cmp   al, 0                 ; check al if all the sectors read
    je    load_finished               ; 完成扇区读取则跳转到结束
    call  reset_drive
    jmp   read_one_sector           ; read next sector

load_finished:                              ; 结束
    call    reset_drive
    ret

; ------------------------------------------------------------

write_message:
    lodsb           ; DS:[SI] is read to al
    cmp   al, 0x0
    jz    end_message
    mov   ah, 0x0E    ; teletype Mode
    mov   bx, 0007    ; white on black attribute
    int   0x10
    jmp   write_message

end_message:
    ret

; ------------------------------------------------------------

    bootdrive db 0

    times 510-($-$$) db 0           ;;;; 剩余空间用0填充，使代码刚好512个字节,这时为了方便从第2个扇区存放setup.s代码
    dw 0xAA55                       ;;;; 结束标志0xAA55
```

boot.s代码流程图如下

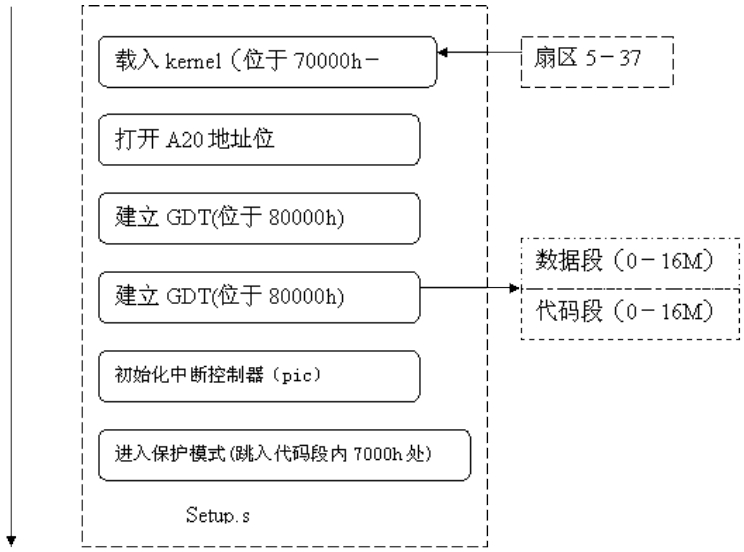



从代码搬移的角度整理启动过程为：




setup.s的代码流程图如下，下面将更加详细分析setup.s源代码

setup.s的代码流程图如下，下面将更加详细分析setup.s源代码



# 第2章习题作业

2013年1月10日
13:55

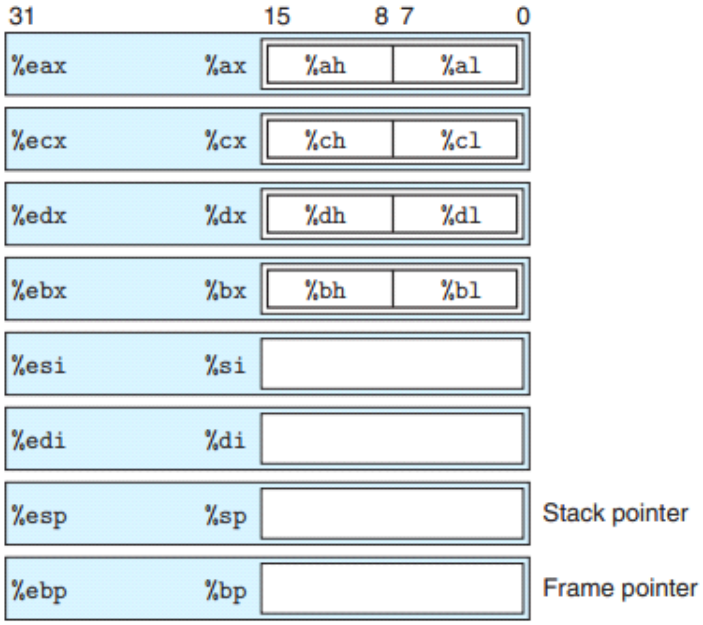
**习题1**

Intel处理器在从4位、8位、16位到32位演变过程中，16位处理器8086提出的分段概念和32位处理器80386中提出的保护模式概念起着决定性作用。在演变过程中，尽量保持硬件体系的向前兼容，同时又引入新技术解决新问题，比如引入分段的概念是为了解决16位处理器访存地址范围在0~64K的缺陷，比如在保持原有段寄存器为16位的基础上建立保护模式，使段范围可以达到4GB。

**习题2**

在80x86中，可供一般用户访问的寄存器：

（1）8个通用寄存器，如下图（可以按8位、16位和32位方式分别访问）；




（2）4个16位的段寄存器（CS/DS/SS/ES）

（3）EFLAGS和EIP寄存器

供操作系统使用的寄存器：

（1）4个控制寄存器；

（1）4个系统地址寄存器；

（2）8个调试寄存器；

（3）2个测试寄存器。

**习题3**

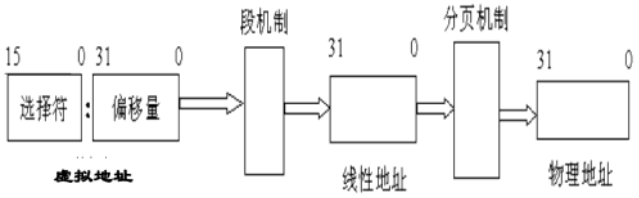
物理地址：硬件内存上对应的地址

虚地址：编程（高级语言、汇编语言）时所使用的地址，一般用“选择符：偏移量”表示

线性地址：将虚地址经过MMU进行段机制转换后的地址，线性地址连续不分段（0~4GB）

**习题4**

保护模式下，虚拟地址到物理地址的转换过程




屏幕剪辑的捕获时间: 2013/1/11 16:00

上图中，实现段机制和分页机制的模块叫做MMU（内存管理单元）。

**习题5**

```
/* C语言描述段描述符 */
struct Segment
{
    char index;       // 段索引值，一般保存在段寄存器中
    int base_addr;    // 基地址
    int limit;        // 界限
    char attribute;   // 段属性
}
```

更为详细的为

```
Struct Segment
{
    Char limit1;
    Char limit2;
    Char base1;
    Char base2;
    Char base3;
    Char base4;
    Char cfg;        // 存取权字节
    Char limit3;     // G D 0 0 19~16段界限
    Char base5;
}
```

**习题6**

x86中段寄存器叫做段选择符：因为x86中段寄存器用来保存了段描述符表中的索引值

**习题7**

保护模式：保护物理内存的访问。访问内存时不能直接从段寄存器中获得段的其实地址，而需要经过额外的转换和检查。
保护模式的实现：通过分页机制及CR0~CR3控制寄存器的设置

**习题8**

在x86上，设计者要求必须使用段机制，而Linux为了可移植性，巧妙的绕过段机制，将段机制和分页机制合并。
主要设定如下完成：
设定段的基地址为0，段的界限设为4GB，这时段地址就等于偏移地址等于线性地址。

Linux必须创建4个段描述符——内核代码段和数据段，用户代码段和数据段。
这4个段的定义子啊在Linux内核代码的arch/x86/include/asm/segment.h中

```
#define __KERNEL_CS    (GDT_ENTRY_KERNEL_CS * 8)
#define __KERNEL_DS    (GDT_ENTRY_KERNEL_DS * 8)
#define __USER_DS      (GDT_ENTRY_DEFAULT_USER_DS* 8 + 3)
#define __USER_CS      (GDT_ENTRY_DEFAULT_USER_CS* 8 + 3)
```

又有

```
#define GDT_ENTRY_KERNEL_CS 2
#define GDT_ENTRY_KERNEL_DS 3
#define GDT_ENTRY_DEFAULT_USER_DS 5
#define GDT_ENTRY_DEFAULT_USER_CS 6
```

由此，可以简化为

```
#define __KERNEL_CS    (0x10)
#define __KERNEL_DS    (0x18)
#define __USER_DS      (0x2B)
#define __USER_CS      (0x23)
```

以上4个段都放在GDT中，GDT定义如下

```
#define GDT_ENTRY(flags, base, limit)                \
     ((((base)  & 0xff000000ULL) << (56-24)) |     \
      (((flags) & 0x0000f0ffULL) << 40) |          \
      (((limit) & 0x000f0000ULL) << (48-16)) |     \
      (((base)  & 0x00ffffffULL) << 16) |          \
      (((limit) & 0x0000ffffULL)))
```

GDT是全局描述符表，具有描述符表的3项（基地址，界限，属性值）。

**习题9**

页的大小一部分由硬件设计者决定，另外一部分给操作系统设计者留有选择的权限。比如，硬件设计时可以设定为只允许2种类型页大小（4KB和4MB），通过寄存器标志由操作系统设计者确定到底使用4KB还是4MB。
也太小会导致页目录占用空间过大，页过大会导致访问速度降低。

**习题10**
对32位线性地址空间使用2级页表：因为Linux中页大小设置为4KB，则共有4GB/4KB=1M个页表项，1个页表项占用4个字节，共需要4MB的空间存储页表项，这显然在一页面中是存不下的，因此，采用2级页表的方式，即对4MB的页表项再次分页，4MB/4KB=1k，因此页目录项共占用1kx4=4k物理内存空间，刚好能在一个页面内容纳。

**习题11**
Linux内核代码中关于初始化页表的代码在arch/x86/kernel/head.s中。

**习题12**
页目录和页表的项数都为1k=1024项，刚好对应二进制10bits。由于存储每一项（页目录，页表）都需要占用4Bytes，所以将页表或页目录存储刚好需要4KB，占用一个物理页面。
如果页目录或页表占用位数>10则说明分页结构不合理，导致在单独的一个物理页面存不下页目录或页表。

**习题13**
页表项各个位的定义由操作系统设计者决定（？），通过表项结构中的R/W位和U/S位为页表或页提供硬件保护机制。

在线性地址到物理地址转换过程中，页表项提供页目录或页面的起始地址，结合线性地址结构中的偏移量，就可以获得物理地址。

习题14参见习题13.

**习题16**
页面高速缓存作用：加快访问速度。
缓存置换策略：随机算法、FIFO算法、近期最少（LRU）使用算法，在页面高速缓存中可以使用近期最少使用算法。

**习题17**
Linux使用三级分页是为了保持可移植性，因为许多处理器都采用64位结构，这时使用二级分页方式已经不再适合。
Linux主要使用分页而避开分段机制也是为了保持可移植性，因为许多处理器不支持分段。

**习题18**
参考教材和http://www.tldp.org/HOWTO/Assembly-HOWTO/assemblers.html

**习题19**
以下分析的2个文件都在arch/x86/include/asm目录下
（1）分析system.h中read_cr3()函数

```
#define read_cr3()      (native_read_cr3())             // read_cr3()即为native_read_cr3()
static inline unsigned long native_read_cr3(void)
{
    unsigned long val;      // 定义4字节变量
    asm volatile("mov %%cr3,%0\n\t" : "=r" (val), "=m" (__force_order));  // 将cr3寄存器
    // 的值读取到val变量中和__force_order内存单元中
    return val;      // 返回val变量的值，及cr3寄存器的值
}
```

（2）分析string_32.h中memecpy()函数

```
#define memcpy(t, f, n)                         \
    (__builtin_constant_p((n))          \
     ? __constant_memcpy((t), (f), (n))      \
     : __memcpy((t), (f), (n)))
```

__builtin_constant_p 是编译器gcc内置函数，用于判断一个值是否为编译时常量，如果是常数，函数返回1 ，否则返回0。因此，上面宏定义功能为：当n为常数时，调用__constant_memcpy，否则调用__memcpy。
[1]__memcpy函数

```
static __always_inline void *__memcpy(void *to, const void *from, size_t n)
{
    int d0, d1, d2;
    asm volatile("rep ; movsl\n\t"
             "movl %4,%%ecx\n\t"
             "andl $3,%%ecx\n\t"
             "jz 1f\n\t"
             "rep ; movsb\n\t"
```

```
                "1:"
                : "=&c" (d0), "=&D" (d1), "=&S" (d2)
                : "0" (n / 4), "g" (n), "1" ((long)to), "2" ((long)from)
                : "memory");
    return to;
}
```

该段代码整理如下：

```
    Rep;  循环移动字符串操作（ESI->EDI），移动次数通过下面%ecx设定
    movl n/4, %ecx; 循环移动n/4个字
    andl $3, %ecx;
    jz 1f; 如果%ecx<4，即剩下的字符串常小于1个字，则跳转到标号1:处
    rep
1:
```

汇编代码中设定to绑定到d1变量，而d1变量又绑定到EDI寄存器，因此to绑定到EDI寄存器；同理,from绑定到ESI寄存器。

[2]__constant_memcpy

```
static __always_inline void *__constant_memcpy(void *to, const void *from,
                                size_t n)
{
    long esi, edi;
    if (!n)
        return to;

    switch (n) {          // n<8字节时直接使用C语言复制
    case 1:
        *(char *)to = *(char *)from;
        return to;
    case 2:
        *(short *)to = *(short *)from;
        return to;
    case 4:
        *(int *)to = *(int *)from;
        return to;
    case 3:
        *(short *)to = *(short *)from;
        *((char *)to + 2) = *((char *)from + 2);
        return to;
    case 5:
        *(int *)to = *(int *)from;
        *((char *)to + 4) = *((char *)from + 4);
        return to;
    case 6:
        *(int *)to = *(int *)from;
        *((short *)to + 2) = *((short *)from + 2);
        return to;
    case 8:
        *(int *)to = *(int *)from;
        *((int *)to + 1) = *((int *)from + 1);
        return to;
    }

    esi = (long)from;       // 便于后面将from绑定到ESI寄存器
    edi = (long)to;         // 便于后面捡to绑定到EDI寄存器
    if (n >= 5 * 4) {
        /* large block: use rep prefix */
        int ecx;
        asm volatile("rep ; movsl"       // 这段汇编参考对__memcpy函数的分析
                  : "=&c" (ecx), "=&D" (edi), "=&S" (esi)
                  : "0" (n / 4), "1" (edi), "2" (esi)
                  : "memory"
        );
    } else {
        /* small block: don't clobber ecx + smaller code */
        if (n >= 4 * 4)
            asm volatile("movsl"
                      : "=&D"(edi), "=&S"(esi)
                      : "0"(edi), "1"(esi)
```

```
                    : "memory");
        if (n >= 3 * 4)
            asm volatile("movsl"
                    : "=&D"(edi), "=&S"(esi)
                    : "0"(edi), "1"(esi)
                    : "memory");
        if (n >= 2 * 4)
            asm volatile("movsl"
                    : "=&D"(edi), "=&S"(esi)
                    : "0"(edi), "1"(esi)
                    : "memory");
        if (n >= 1 * 4)
            asm volatile("movsl"
                    : "=&D"(edi), "=&S"(esi)
                    : "0"(edi), "1"(esi)
                    : "memory");
    }
    switch (n % 4) {    // 当字节数不够1个字的情况
        /* tail */
    case 0:
        return to;
    case 1:
        asm volatile("movsb"          // 移动一个字节
                : "=&D"(edi), "=&S"(esi)
                : "0"(edi), "1"(esi)
                : "memory");
        return to;
    case 2:
        asm volatile("movsw"          // 移动一个半字
                : "=&D"(edi), "=&S"(esi)
                : "0"(edi), "1"(esi)
                : "memory");
        return to;
    default:
        asm volatile("movsw\n\tmovsb"  // 移动一个半字+一个字节
                : "=&D"(edi), "=&S"(esi)
                : "0"(edi), "1"(esi)
                : "memory");
        return to;
    }
}
```

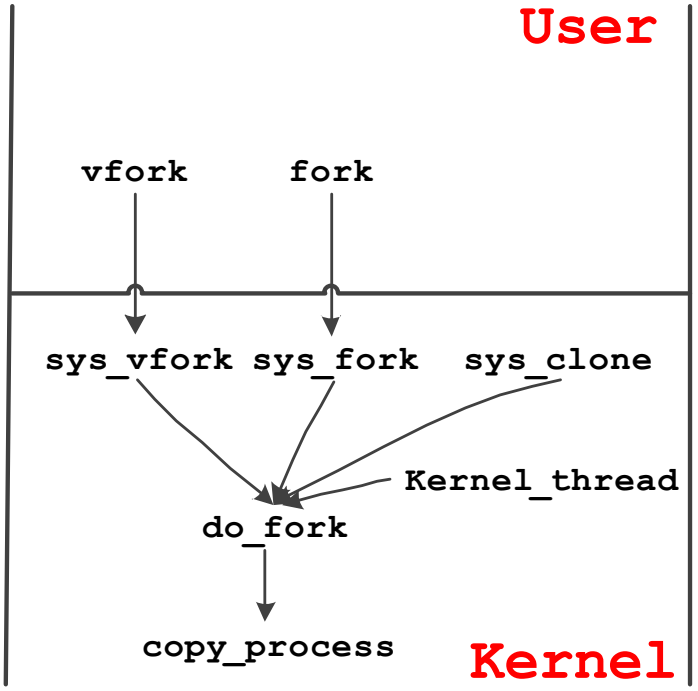
编写该段程序的作者竭力想提高字符串复制的效率，采用分类讨论的思想，程序看起来代码比较多，但实现的功能却很简单。字符串长度小于8时直接使用C语言复制，当>8字节时，则以1个字（4字节）为单位进行复制。

# Chpt3 进程

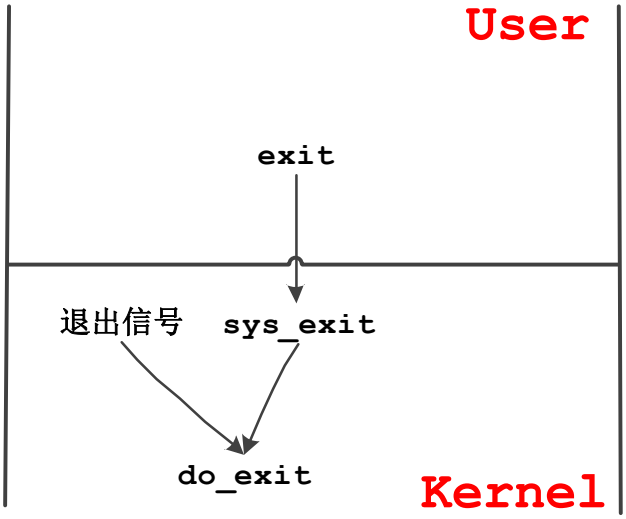
2013年1月7日
15:27

## 1、进程创建与销毁

创建进程




销毁进程




## 2、进程控制系统调用

Linux内核有一个提供系统调用的头文件：unistd.h

fork 创建一个新进程

clone 按指定条件创建子进程

execve 运行可执行文件

exit 中止进程

_exit 立即中止当前进程

getdtablesize 进程所能打开的最大文件数
getpgid 获取指定进程组标识号
setpgid 设置指定进程组标志号
getpgrp 获取当前进程组标识号
setpgrp 设置当前进程组标志号
getpid 获取进程标识号
getppid 获取父进程标识号
getpriority 获取调度优先级
setpriority 设置调度优先级
modify_ldt 读写进程的本地描述表
nanosleep 使进程睡眠指定的时间
nice 改变分时进程的优先级
pause 挂起进程，等待信号
personality 设置进程运行域
prctl 对进程进行特定操作
ptrace 进程跟踪
sched_get_priority_max 取得静态优先级的上限
sched_get_priority_min 取得静态优先级的下限
sched_getparam 取得进程的调度参数
sched_getscheduler 取得指定进程的调度策略
sched_rr_get_interval 取得按RR算法调度的实时进程的时间片长度
sched_setparam 设置进程的调度参数
sched_setscheduler 设置指定进程的调度策略和参数
sched_yield 进程主动让出处理器,并将自己等候调度队列队尾
vfork 创建一个子进程，以供执行新程序，常与execve等同时使用
wait 等待子进程终止
wait3 参见wait
waitpid 等待指定子进程终止
wait4 参见waitpid
capget 获取进程权限
capset 设置进程权限
getsid 获取会晤标识号
setsid 设置会晤标识号

3、关于进程调度的讨论
学习进程调度应该注意：调度策略、调度时机和调度步骤。
（1）调度策略有

- SCHED_NORMAL(SCHED_OTHER):普通的分时进程
- SCHED_FIFO :先入先出的实时进程
- SCHED_RR: 时间片轮转的实时进程
- SCHED_BATCH:批处理进程
- SCHED_IDLE: 只在系统空闲时才能够被调度执行的进程

屏幕剪辑的捕获时间: 2013/1/25 10:37

调度类（调度程序模块）封装了调度策略，将调度策略进行模块化。

- CFS 调度类（在 kernel/sched_fair.c 中实现）用于以下调度策略: SCHED_NORMAL、SCHED_BATCH 和 SCHED_IDLE。
- 实时调度类（在 kernel/sched_rt.c 中实现）用于 SCHED_RR 和 SCHED_FIFO 策略。

屏幕剪辑的捕获时间: 2013/1/25 10:38

（2）调度时机指什么时候调用schedule函数，教材中给出了4种调度时机：进程状态切换时、当前进程的时间片用完时、设备驱动程序运行时、从内核态返回用户态时。

（3）调度步骤：

Schedule函数工作流程如下:

1). 清理当前运行中的进程;

2). 选择下一个要运行的进程;

（pick_next_task 分析)

3). 设置新进程的运行环境;

4). 进程上下文切换。

4、小结

（1）进程是抽象的概念，是对程序执行过程的抽象

（2）进程控制块(PCB)是对进程这一抽象概念的计算机描述

（3）进程的组织方式有：进程树、进程链表、哈希表、运行队列和等待队列

（4）Linux采用时间片流转的优先级调度方式

（5）进程采用“写时复制”技术创建新进程

（6）进程从诞生到死亡经历了4个系统调用——fork、exec、exit、wait

（7）Linux的优先级及调度策略都可根据程序员的需要进行修改

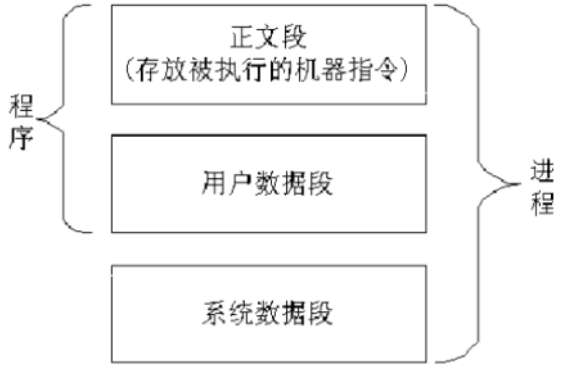
# 第3章习题作业

2013年1月23日
16:12

注：所有分析程序均来源于linux-2.6.32.60

**习题1**

程序是存放在磁盘上的一系列代码和数据的可执行映像，是一个静止的实体；
进程是执行中的程序，是动态的实体。
如下图，正因为进程是动态的，所以进程有系统数据段（其中包含PCB信息）。




进程的4要素：
（1）有一段可供执行的代码段；
（2）有专用的内核空间堆栈；
（3）在内核中有一个进程控制块结构(PCB)；
（4）有独立的用户空间

在Linux中，线程与进程是一样的，都通过task_struct结构体描述。有用户空间的线程叫用户线程，无用户空间的线程叫内核线程，不管用户线程还是内核线程都有内核空间。

**习题2**

进程是操作系统设计者为解决程序“并行”执行问题抽象出的逻辑概念，CPU个数有限，许多程序都想单独占用CPU，但CPU并没有分身术，为避免互不相让的程序之间厮打起来，因此引入了进程的概念。通过对进程的调度，合理的在多个进程中分配CPU，达到模拟并行的效果。

**习题3**

进程控制块（PCB）就是记录进程生命周期的人生档案。
PCB使用task_struct结构体表示，在include/linux/sched.h中定义。

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped 进程的状态，使用volatile以便实时的更新state */
    void *stack;            /* 进程堆栈指针 */
    atomic_t usage;         /* 计数器，计算进程使用的时间片长度 */
    unsigned int flags;     /* per process flags, defined below 预处理标志，取值也在sched.h中有定义 */
    unsigned int ptrace;

    int lock_depth;         /* BKL lock depth */

#ifdef CONFIG_SMP
#ifdef __ARCH_WANT_UNLOCKED_CTXSW
    int oncpu;
#endif
#endif

    int prio, static_prio, normal_prio;
    /*
     * prio:优先级，在0~MAX_PRIO(140)-1间取值，其中0~MAX_RT_PRIO(100)-1是实时进程，其它值属于非实时进程，prio越大优先级越小；与教材
     *      goodness计算结果值相同
     * static_prio:静态优先级，同教材中的nice值，取值为-20~19
     */
    unsigned int rt_priority;   /* 实时进程优先级 */
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;  /* rt->time_slice表示分配给进程的时间片，缺省值与static_prio有关；
                                 * rt->run_list为就绪队列
                                 */

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier: */
    struct hlist_head preempt_notifiers;
#endif

    /*
     * fpu_counter contains the number of consecutive context switches
     * that the FPU is used. If this is over a threshold, the lazy fpu
     * saving becomes unlazy to save the trap. This is an unsigned char
     * so that after 256 times the counter wraps and the behavior turns
     * lazy again; this to deal with bursty apps that only use FPU for
     * a short time
     */
    unsigned char fpu_counter;
#ifdef CONFIG_BLK_DEV_IO_TRACE
    unsigned int btrace_seq;
```

```
#endif

    unsigned int policy;  /* 调度策略: SCHED_FIFO,SCHED_RR,SCHED_OTHER */
    cpumask_t cpus_allowed;

#ifdef CONFIG_TREE_PREEMPT_RCU
    int rcu_read_lock_nesting;
    char rcu_read_unlock_special;
    struct rcu_node *rcu_blocked_node;
    struct list_head rcu_node_entry;
#endif /* #ifdef CONFIG_TREE_PREEMPT_RCU */

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
    struct sched_info sched_info;
#endif

    struct list_head tasks;  /* 进程链表 */
    struct plist_node pushable_tasks;  /* 进程hash表 */

    struct mm_struct *mm, *active_mm;  /* mm为用户空间描述指针，内核线程mm=NULL */

/* task state */
    int exit_state;    /* 与进程退出有关的状态 */
    int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */
    /* ??? */
    unsigned int personality;
    unsigned did_exec:1;
    unsigned in_execve:1;  /* Tell the LSMs that the process is doing an
                    * execve */
    unsigned in_iowait:1;


    /* Revert to default priority/policy when forking */
    unsigned sched_reset_on_fork:1;

    pid_t pid;         /* 进程的身份证号 */
    pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
    /* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
#endif

    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->real_parent->pid)
     */
    struct task_struct *real_parent; /* real parent process */
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
    /*
     * children/sibling forms the list of my natural children
     */
    struct list_head children;   /* list of my children */
    struct list_head sibling;    /* linkage in my parent's children list */
    struct task_struct *group_leader; /* threadgroup leader */

    /*
     * ptraced is the list of tasks this task is using ptrace on.
     * This includes both natural children and PTRACE_ATTACH targets.
     * p->ptrace_entry is p's link on the p->parent->ptraced list.
     */
    struct list_head ptraced;
    struct list_head ptrace_entry;

    /*
     * This is the tracer handle for the ptrace BTS extension.
     * This field actually belongs to the ptracer task.
     */
    struct bts_context *bts;

    /* PID/PID hash table linkage. */
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;

    struct completion *vfork_done;          /* for vfork() */
    int __user *set_child_tid;        /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid;      /* CLONE_CHILD_CLEARTID */

    cputime_t utime, stime, utimescaled, stimescaled;
    cputime_t gtime;
    cputime_t prev_utime, prev_stime;
    unsigned long nvcsw, nivcsw; /* context switch counts */
    struct timespec start_time;       /* monotonic time */
    struct timespec real_start_time;  /* boot based time */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
    unsigned long min_flt, maj_flt;

    struct task_cputime cputime_expires;
    struct list_head cpu_timers[3];
```

```
/* process credentials */
    const struct cred *real_cred;      /* objective and real subjective task
                      * credentials (COW) */
    const struct cred *cred;     /* effective (overridable) subjective task
                      * credentials (COW) */
    struct mutex cred_guard_mutex;    /* guard against foreign influences on
                      * credential calculations
                      * (notably. ptrace) */
    struct cred *replacement_session_keyring; /* for KEYCTL_SESSION_TO_PARENT */

    char comm[TASK_COMM_LEN]; /* executable name excluding path              /* 进程名称 */
                      - access with [gs]et_task_comm (which lock
                        it with task_lock())
                      - initialized normally by setup_new_exec */
/* file system info */
    int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
    struct sysv_sem sysvsem;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
    unsigned long last_switch_count;
#endif
/* CPU-specific state of this task */
    struct thread_struct thread;
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespaces */
    struct nsproxy *nsproxy;
/* signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    sigset_t saved_sigmask;     /* restored if set_restore_sigmask() was used */
    struct sigpending pending;

    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);
    void *notifier_data;
    sigset_t *notifier_mask;
    struct audit_context *audit_context;
#ifdef CONFIG_AUDITSYSCALL
    uid_t loginuid;
    unsigned int sessionid;
#endif
    seccomp_t seccomp;

/* Thread group tracking */
     u32 parent_exec_id;
     u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings, mems_allowed,
 * mempolicy */
    spinlock_t alloc_lock;

#ifdef CONFIG_GENERIC_HARDIRQS
    /* IRQ handler threads */
    struct irqaction *irqaction;
#endif

    /* Protection of the PI data structures: */
    spinlock_t pi_lock;

#ifdef CONFIG_RT_MUTEXES
    /* PI waiters blocked on a rt_mutex held by this task */
    struct plist_head pi_waiters;
    /* Deadlock detection and priority inheritance handling */
    struct rt_mutex_waiter *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
    /* mutex deadlock detection */
    struct mutex_waiter *blocked_on;
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
    unsigned int irq_events;
    int hardirqs_enabled;
    unsigned long hardirq_enable_ip;
    unsigned int hardirq_enable_event;
    unsigned long hardirq_disable_ip;
    unsigned int hardirq_disable_event;
    int softirqs_enabled;
    unsigned long softirq_disable_ip;
    unsigned int softirq_disable_event;
    unsigned long softirq_enable_ip;
    unsigned int softirq_enable_event;
    int hardirq_context;
    int softirq_context;
#endif
```

```
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH 48UL
    u64 curr_chain_key;
    int lockdep_depth;
    unsigned int lockdep_recursion;
    struct held_lock held_locks[MAX_LOCK_DEPTH];
    gfp_t lockdep_reclaim_gfp;
#endif

/* journalling filesystem info */
    void *journal_info;

/* stacked block device info */
    struct bio *bio_list, **bio_tail;

/* VM state */
    struct reclaim_state *reclaim_state;

    struct backing_dev_info *backing_dev_info;

    struct io_context *io_context;

    unsigned long ptrace_message;
    siginfo_t *last_siginfo; /* For ptrace use.  */
    struct task_io_accounting ioac;
#if defined(CONFIG_TASK_XACCT)
    u64 acct_rss_mem1;     /* accumulated rss usage */
    u64 acct_vm_mem1;/* accumulated virtual memory usage */
    cputime_t acct_timexpd;      /* stime + utime since last update */
#endif
#ifdef CONFIG_CPUSETS
    nodemask_t mems_allowed;     /* Protected by alloc_lock */
    int cpuset_mem_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
    /* Control Group info protected by css_set_lock */
    struct css_set *cgroups;
    /* cg_list protected by css_set_lock and tsk->alloc_lock */
    struct list_head cg_list;
#endif
#ifdef CONFIG_FUTEX
    struct robust_list_head __user *robust_list;
#ifdef CONFIG_COMPAT
    struct compat_robust_list_head __user *compat_robust_list;
#endif
    struct list_head pi_state_list;
    struct futex_pi_state *pi_state_cache;
#endif
#ifdef CONFIG_PERF_EVENTS
    struct perf_event_context *perf_event_ctxp;
    struct mutex perf_event_mutex;
    struct list_head perf_event_list;
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *mempolicy; /* Protected by alloc_lock */
    short il_next;
#endif
    atomic_t fs_excl;/* holding fs exclusive resources */
    struct rcu_head rcu;

    /*
     * cache last used pipe for splice
     */
    struct pipe_inode_info *splice_pipe;
#ifdef     CONFIG_TASK_DELAY_ACCT
    struct task_delay_info *delays;
#endif
#ifdef CONFIG_FAULT_INJECTION
    int make_it_fail;
#endif
    struct prop_local_single dirties;
#ifdef CONFIG_LATENCYTOP
    int latency_record_count;
    struct latency_record latency_record[LT_SAVECOUNT];
#endif
    /*
     * time slack values; these are used to round up poll() and
     * select() etc timeout values. These are in nanoseconds.
     */
    unsigned long timer_slack_ns;
    unsigned long default_timer_slack_ns;

    struct list_head *scm_work_list;
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
    /* Index of current stored adress in ret_stack */
    int curr_ret_stack;
    /* Stack of return addresses for return function tracing */
    struct ftrace_ret_stack     *ret_stack;
    /* time stamp for last schedule */
    unsigned long long ftrace_timestamp;
    /*
     * Number of functions that haven't been traced
     * because of depth overrun.
```

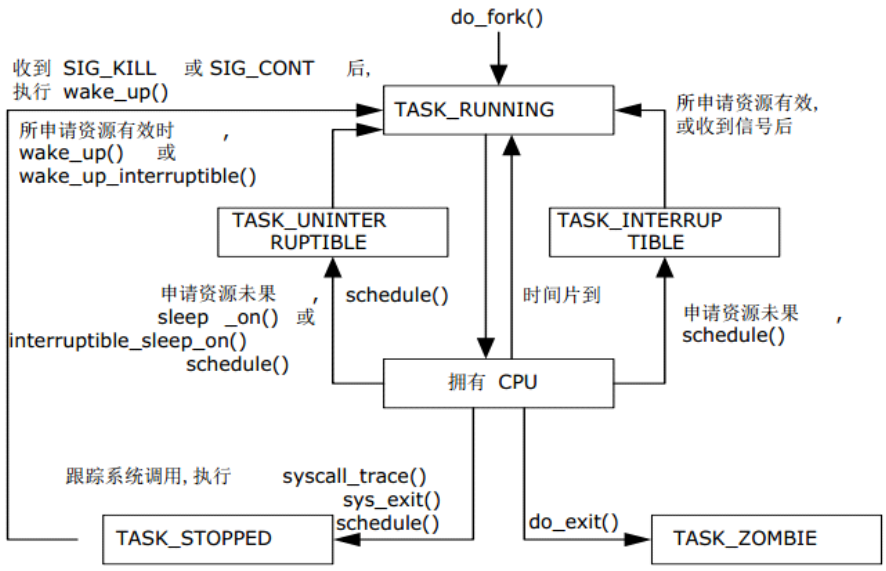
```
     */
    atomic_t trace_overrun;
    /* Pause for the tracing */
    atomic_t tracing_graph_pause;
#endif
#ifdef CONFIG_TRACING
    /* state flags for use by tracers */
    unsigned long trace;
    /* bitmask of trace recursion */
    unsigned long trace_recursion;
#endif /* CONFIG_TRACING */
};
```

**习题4**

由task_struct知道，有2个表示进程状态的域——state和exit_state。

```
#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_ZOMBIE          16
#define EXIT_DEAD      32
/* in tsk->state again */
#define TASK_DEAD      64
#define TASK_WAKEKILL        128
#define TASK_WAKING          256

/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE        (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED         (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED          (TASK_WAKEKILL | __TASK_TRACED)

/* Convenience macros for the sake of wake_up */
#define TASK_NORMAL          (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
#define TASK_ALL       (TASK_NORMAL | __TASK_STOPPED | __TASK_TRACED
```

下图显示了5种主要状态之间的切换




**Linux 进程的状态转换**

屏幕剪辑的捕获时间: 2013/1/23 19:47

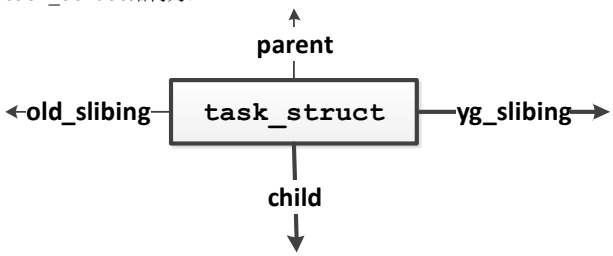
**习题5**

```
/* ================================================================================
 *
 *       Filename:  mytask.c
 *    Description:  自己写一个进程，模拟从进程树中插入和删除进程的过程
 *
 *        Version:  1.0
 *        Created:  2013/1/23 19:51:08
 *       Compiler:  gcc
 *         Author:  xhzuoxin ( QQ:1126804077 )
 *
 * ================================================================================ */

#include <stdio.h>
#include <stdlib.h>

#ifndef true
#define true                (1)
#endif
#ifndef false
#define false               (0)
```

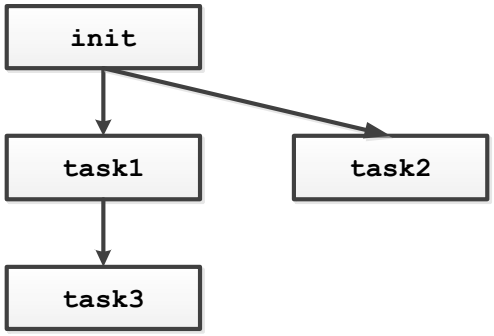
```
#endif

typedef int pid_t;
struct task_struct {
    volatile long state;
    pid_t pid;
    struct task_struct *parent, *child, *old_slibing, *yg_slibing;
};
typedef struct task_struct task_tree;

/* states */
#define TASK_RUNING          (0x00)
#define TASK_SLEEP           (0x01)
#define TASK_STOPPED         (0x02)
#define TASK_ZOMBIE          (0x04)
#define TASK_DEAD            (0x08)

int g_pid = 0;

int init_task(struct task_struct **task)
{
    (*task)->state = TASK_RUNING;
    (*task)->pid = g_pid++;
    (*task)->parent = NULL;
    (*task)->child = NULL;
    (*task)->old_slibing = NULL;
    (*task)->yg_slibing = NULL;

    return true;
}

/*
 * flag = 1: insert ins-task as current-task's yg_slibing node
 * flag = 0: insert ins-task as current-task's child node
 * */
int insert_task(struct task_struct **current_task,
        struct task_struct **ins_task,
        int flag)
{
    switch (flag) {
    case 0:
        if ( NULL != (*current_task)->child ) {
               (*ins_task)->old_slibing = (*current_task)->child;
               (*current_task)->child->yg_slibing = *ins_task;
        }
           (*current_task)->child = *ins_task;
           (*ins_task)->child = NULL;
           (*ins_task)->yg_slibing = NULL;
           (*ins_task)->parent = *current_task;
          break;
    case 1:
           /* waiting for adding */
          break;
    default:break;
    }

    return true;
}

int del_task(struct task_struct **dtask)
{
    if ( (*dtask)->child != NULL ) {       /* 被删除的task有孩子 */
        (*dtask)->child->parent = (*dtask)->parent;
        if ( (*dtask)->parent->child == *dtask ) {
            (*dtask)->parent->child = (*dtask)->child;
        }
        if ( NULL != (*dtask)->child->old_slibing ) {
            (*dtask)->child->child = (*dtask)->child->old_slibing;
            (*dtask)->child->old_slibing->yg_slibing = NULL;
        }
        (*dtask)->child->old_slibing = (*dtask)->old_slibing;
        if ( NULL != (*dtask)->old_slibing ) {
            (*dtask)->old_slibing->yg_slibing = (*dtask)->child;
        }
        (*dtask)->child->yg_slibing = (*dtask)->yg_slibing;
        if ( NULL != (*dtask)->yg_slibing ) {
            (*dtask)->yg_slibing->old_slibing = (*dtask)->child;
        }
    } else {                               /* 被删节点没孩子 */
        if ( (NULL != (*dtask)->yg_slibing) && (NULL != (*dtask)->old_slibing) ) {
            (*dtask)->yg_slibing->old_slibing = (*dtask)->old_slibing;
            (*dtask)->old_slibing->yg_slibing = (*dtask)->yg_slibing;
        } else if ( NULL != (*dtask)->yg_slibing ) {
            (*dtask)->yg_slibing->old_slibing = NULL;
        } else if ( NULL != (*dtask)->old_slibing ) {
            (*dtask)->parent->child = NULL;
        }
    }

    return true;
}

int main(void)
```

```
{
    struct task_struct *init = NULL;
    struct task_struct *task1 = NULL;
    struct task_struct *task2 = NULL;
    struct task_struct *task3 = NULL;

    /* init task */
    if ( NULL == (init=(struct task_struct *)malloc(sizeof(struct task_struct))) ) {
        printf("error when alloc space!\n");
        exit(1);
    }
    init_task(&init);

    /* 2th task */
    if ( NULL == (task1=(struct task_struct *)malloc(sizeof(struct task_struct))) ) {
        printf("error when alloc space!\n");
        exit(1);
    } else {
        init_task(&task1);
    }
    insert_task(&init, &task1, 0); /* insert task1 as init's child */
    /* 3th task */
    if ( NULL == (task2=(struct task_struct *)malloc(sizeof(struct task_struct))) ) {
        printf("error when alloc space!\n");
        exit(1);
    } else {
        init_task(&task2);
    }
    insert_task(&init, &task2, 0); /* insert task2 as init's child */
    /* 4th task */
    if ( NULL == (task3=(struct task_struct *)malloc(sizeof(struct task_struct))) ) {
        printf("error when alloc space!\n");
        exit(1);
    } else {
        init_task(&task3);
    }
    insert_task(&task1, &task3, 0); /* insert task3 as task1's child */

    /* del task1 */
    del_task(&task1);
    free(task1);
    printf("delete task ok!\n");
    printf("init->child=pid:%d\n", init->child->pid);
    printf("task2->old_slibing=pid:%d\n", task2->old_slibing->pid);

    return 0;
}
```

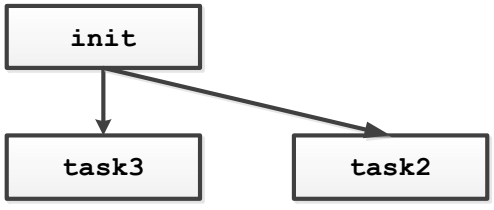
task_struct结构为：




插入task1~3后：



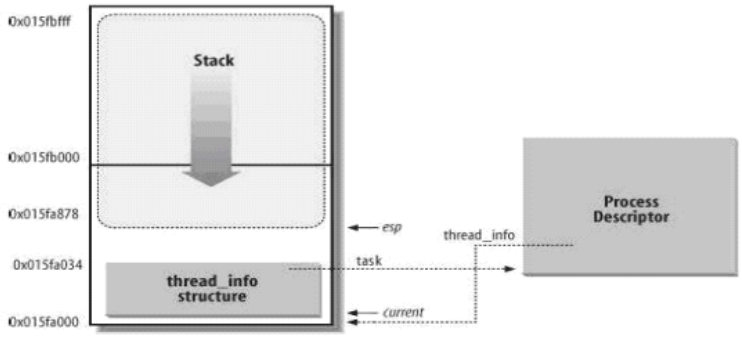

删除task1后：




运行结果为：

```
delete task ok!
init->child=pid:2
task2->old_slibing=pid:3
```

**习题6**




**可以是4K字节(1个页面)也可以是8K字节(2个页面)。**



若PCB堆栈占用8KB，则每个task_struct的低13bits为0，设tmp=0xffffe000
若PCB堆栈占用4KB，则每个task_struct的低12bits为0，设tmp=0xfffff000
使用3条汇编语句可获得PCB首地址：

```
movl tmp, %ecx
andl %esp, %ecx
mov %ecx, current
```

**习题7**

PCB的组织方式包括：
（1）进程树：描述进程之间的亲属关系
（2）进程链表：便于对成百上千个进程进程管理——插入、删除等操作
（3）Hash表：根据PID能够高效快速地查找到对应的PCB
（4）就绪队列：在进行进程调度时，不用检索整个链表，减小调度时间
（5）等待队列：当有多个事件同时到达需要处理时（比如中断、进程同步、定时等），等待队列显得极为有用

**习题8**

打印进程的PID及进程名称：

```
#include <linux/sched.h>
#include <linux/list.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init print_pid(void)
{
    struct task_struct *task,*p;
    struct list_head *pos;
    int count = 0;

    printk("<5>Hello World enter begin:\n");
    task = &init_task;
    list_for_each(pos, &task->tasks)
    {
        p = list_entry(pos, struct task_struct, tasks);
        count++;
        printk("<5>%d----->%s\n", p->pid, p->comm);
    }
    printk("<5>total peocesses:<1> %d\n", count);

    return 0;
}

static void __exit print_end(void)
{

}

module_init(print_pid);
module_exit(print_end);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("xhzuoxin");
```

**习题9**

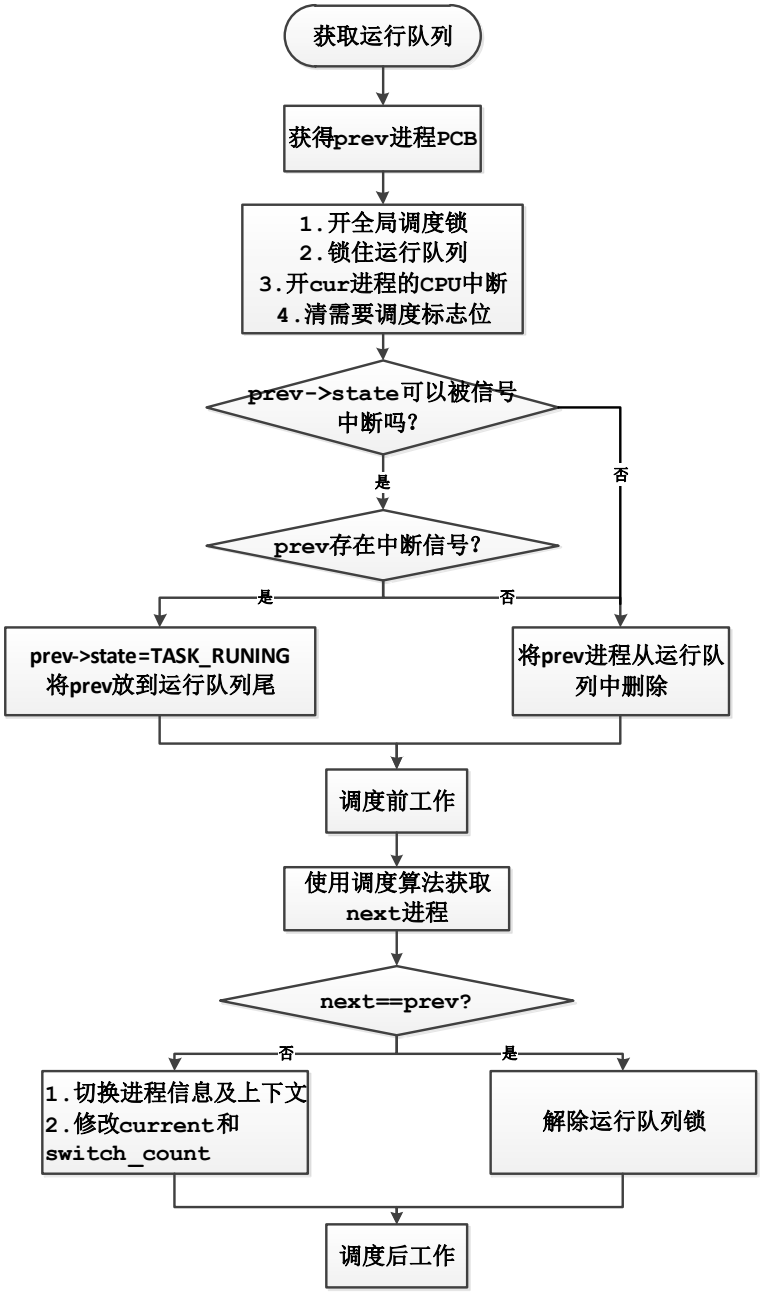
调度的实质是资源的分配，因此调度应该在特定环境的约束下使资源分配达到最优。
好的调度算法应该考虑：
（1）公平：保证每个进程得到合理的CPU时间
（2）高效：使CPU总是保持忙碌状态，即总有有利的进程在CPU上运行
（3）响应时间：使交互用户的响应时间尽可能短
（4）周转时间：使批处理用户等待输出的时间尽可能短
（5）吞吐量：单位时间处理的进程数尽可能多
“鱼和熊掌，不可兼得”，不同的操作系统为满足不同的需求（实时性、交互性好）需要在上面5条原则当中进行取舍。

注：利用《经济与管理》课程中“生产与成本理论”对调度优化进行分析。

**习题10**

schedule代码基于linux-2.6.32版本。

```
/*
 * schedule() is the main scheduler function.
 */
asmlinkage void __sched schedule(void)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;

need_resched:
    preempt_disable();
    cpu = smp_processor_id();          /* 获取当前使用CPU的ID */
    rq = cpu_rq(cpu);                  /* 获取当前CPU的运行队列(run_queue) */
    rcu_sched_qs(cpu);
    prev = rq->curr;                   /* 让prev指向当前进程 */
    switch_count = &prev->nivcsw;      /* 读取上下文切换（进程调度）计数器 */

    release_kernel_lock(prev);         /* 解开全局内核锁，即允许调度，并开当前使用cpu的中断 */
need_resched_nonpreemptible:

    schedule_debug(prev);              /* 统计各种调度时间 */

    if (sched_feat(HRTICK))            /* 如果定时器没有禁用，则禁用定时器 */
        hrtick_clear(rq);

    spin_lock_irq(&rq->lock);          /* 锁住运行队列同时关中断 */
    update_rq_clock(rq);               /* 更新运行队列的时钟 */
    clear_tsk_need_resched(prev);      /* 清需要调度标志位 */

    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {  /* 检查prev进程的状态 */
        if (unlikely(signal_pending_state(prev->state, prev)))  /* prev存在没有处理的信号则置为TASK_RUNING */
            prev->state = TASK_RUNNING;
        else
            deactivate_task(rq, prev, 1);                       /* 否则将prev从运行队列中删除 */
        switch_count = &prev->nvcsw;
    }

    pre_schedule(rq, prev);            /* 调度前准备 */

    if (unlikely(!rq->nr_running))
        idle_balance(cpu, rq);         /* CPU空闲则运行idle_task */

    put_prev_task(rq, prev);           /* 将退出的进程放到运行队列尾 */
    next = pick_next_task(rq);         /* 从运行队列中获取要运行的进程 */

    if (likely(prev != next)) {        /* 将要运行的进程与原进程不同 */
        sched_info_switch(prev, next);
        perf_event_task_sched_out(prev, next, cpu);

        rq->nr_switches++;
        rq->curr = next;               /* 修改运行队列当前进程指针变量为next */
        ++*switch_count;               /* 调度计数器加1 */

        context_switch(rq, prev, next); /* unlocks the rq */   /* 进程调度，切换进程环境 */
        /*
         * the context switch might have flipped the stack from under
         * us, hence refresh the local variables.
         */
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
    } else                             /* 将要运行的进程与原进程相同 */
        spin_unlock_irq(&rq->lock);

    post_schedule(rq);                 /* 进程调度结束后处理 */

    if (unlikely(reacquire_kernel_lock(current) < 0))
        goto need_resched_nonpreemptible;

    preempt_enable_no_resched();       /* 动态优先级更新 */
    if (need_resched())
        goto need_resched;
}
EXPORT_SYMBOL(schedule);      /* 导出调度函数 */
```




schedule()函数流程图

**习题11**

写时复制技术：指在创建新进程时没有把全部的父进程资源给子进程复制一份，而是将这些内容设置为只读状态，当父进程或子进程试图修改某些内容时，内核才在修改前将要修改的部分拷贝出来。

当父子进程共享的资源越多时，写时复制技术越有优势。

**习题12**

下面为do_fork()函数的代码：

```
/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long do_fork(unsigned long clone_flags,
          unsigned long stack_start,
          struct pt_regs *regs,
          unsigned long stack_size,
          int __user *parent_tidptr,
          int __user *child_tidptr)
{
    struct task_struct *p;
    int trace = 0;
    long nr;

    /*
     * Do some preliminary argument and permissions checking before we
     * actually start allocating stuff
     */
    if (clone_flags & CLONE_NEWUSER) {
        if (clone_flags & CLONE_THREAD)
            return -EINVAL;               /* 参数不合理 */
```

```
        /* hopefully this check will go away when userns support is
         * complete
         */
        if (!capable(CAP_SYS_ADMIN) || !capable(CAP_SETUID) ||
                !capable(CAP_SETGID))      /* 确保有系统管理员、设置UID、设置GID的权限 */
            return -EPERM;
    }

    /*
     * We hope to recycle these flags after 2.6.26
     */
    if (unlikely(clone_flags & CLONE_STOPPED)) {
        static int __read_mostly count = 100;

        if (count > 0 && printk_ratelimit()) {
            char comm[TASK_COMM_LEN];

            count--;
            printk(KERN_INFO "fork(): process `%s' used deprecated "
                    "clone flags 0x%lx\n",
                get_task_comm(comm, current),       /* 获得父进程的名称 */
                clone_flags & CLONE_STOPPED);
        }
    }

    /*
     * When called from kernel_thread, don't do user tracing stuff.
     */
    if (likely(user_mode(regs)))
        trace = tracehook_prepare_clone(clone_flags);       /* 准备复制父进程信息 */

    p = copy_process(clone_flags, stack_start, regs, stack_size,
             child_tidptr, NULL, trace);                    /* 复制父进程的PCB到子进程 */
    /*
     * Do this prior waking up the new thread - the thread pointer
     * might get invalid after that point, if the thread exits quickly.
     */
    if (!IS_ERR(p)) {
        struct completion vfork;

        trace_sched_process_fork(current, p);

        nr = task_pid_vnr(p);

        if (clone_flags & CLONE_PARENT_SETTID)
            put_user(nr, parent_tidptr);

        if (clone_flags & CLONE_VFORK) {
            p->vfork_done = &vfork;
            init_completion(&vfork);
        }

        audit_finish_fork(p);
        tracehook_report_clone(regs, clone_flags, nr, p);

        /*
         * We set PF_STARTING at creation in case tracing wants to
         * use this to distinguish a fully live task from one that
         * hasn't gotten to tracehook_report_clone() yet.  Now we
         * clear it and set the child going.
         */
        p->flags &= ~PF_STARTING;

        if (unlikely(clone_flags & CLONE_STOPPED)) {
            /*
             * We'll start up with an immediate SIGSTOP.
             */
            sigaddset(&p->pending.signal, SIGSTOP);
            set_tsk_thread_flag(p, TIF_SIGPENDING);
            __set_task_state(p, TASK_STOPPED);
        } else {
            wake_up_new_task(p, clone_flags);
        }

        tracehook_report_clone_complete(trace, regs,
                        clone_flags, nr, p);

        if (clone_flags & CLONE_VFORK) {
            freezer_do_not_count();
            wait_for_completion(&vfork);
            freezer_count();
            tracehook_report_vfork_done(p, nr);
        }
    } else {
        nr = PTR_ERR(p);
    }
    return nr;
}
```

do_fork函数具体细节还有待分析……

**习题13**

进程0是idle进程，在内核初始化工作的start_kernel()函数中从无到有的创建。
当就绪队列中没有其他进程时，进程0被调度运行。

**习题14**
init内核线程和init进程是不同的概念，init进程是用户态下的第一个进程，[init线程+使用execve()函数载入init(/sbin/init)可执行程序]才形成init进程。

**习题15**
```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    pid_t pc, pr;

    pc = fork();

    if(pc < 0)
    {
        printf("error occured!\n");
    }
    else if(pc == 0)  /* child process */
    {
        printf("This is child process with pid=%d\n", getpid());
        sleep(10);        /* sleep 10s */
    }
    else /* parent process */
    {
        pr = wait(NULL);     /* wait for child process */
        printf("catched a child process with pid=%d\n", pr);
    }

    return 0;
}
```

**习题16**
```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int status = 0;
int main(void)
{
    pid_t pc, pr;

    pc = fork();

    if (pc < 0) {
        printf("error occured!\n");
    } else if (pc == 0) {         /* child */
        printf("I'm child, pid=%d\n", getpid());
        sleep(1);          /* sleep 3s */
        exit(15);
    } else {
        printf("I'm parent, pid=%d;", getpid());
        pr = wait(&status);
        printf("child pid=%d, status=%d\n", pr, status/0x100);
    }

    return 0;
}
```

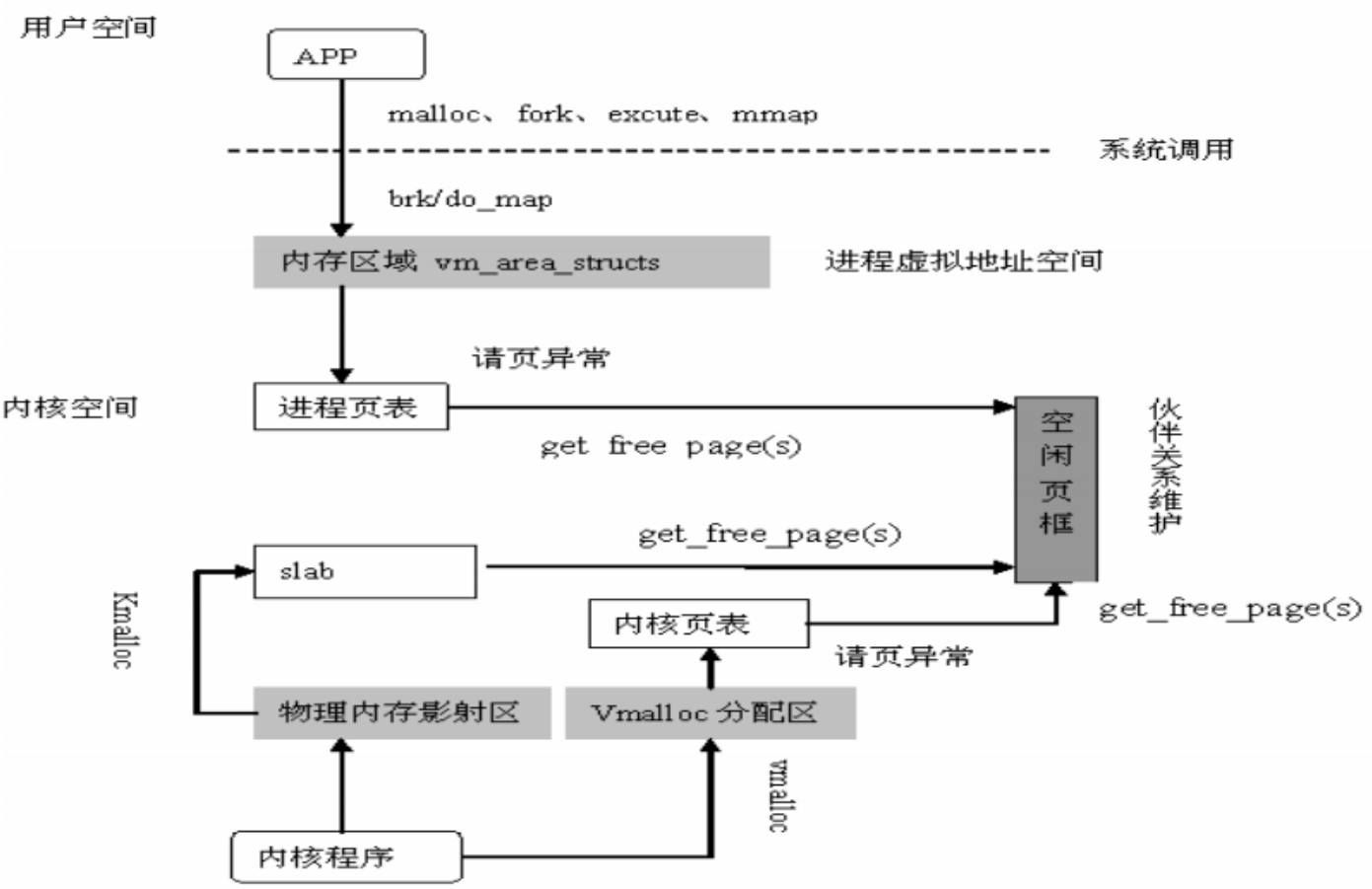
注：这里的wait接收到的status值是exit返回status值的256(0x100)倍。

**习题17**
Refrence 《UNIX环境高级编程》 相关章节。

# Chpt4 内存管理

2013年1月25日
10:43

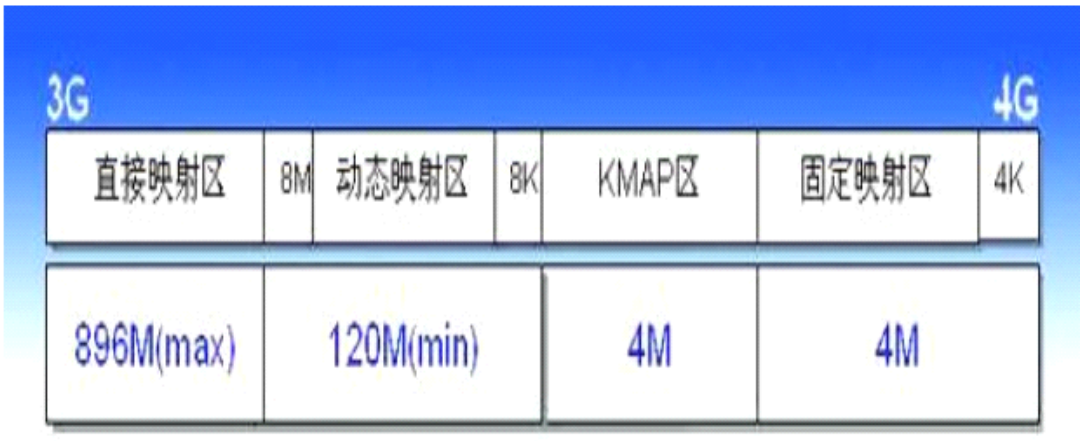
1、空间分配的调用关系




屏幕剪辑的捕获时间: 2013/2/2 16:17

从图中可以看出，最终对物理页框的分配都是通过内核中的
__get_free_page(s)。
malloc用于用户空间的分配，而kmalloc和vmalloc则用于内核空间的分配。

下图为内核空间的分布：




屏幕剪辑的捕获时间: 2013/2/2 16:24

kmalloc从896M(max)中（该区域与物理内存线性映射）分配内存。

vmalloc从动态映射区分配内存，特点是线性空间连续但物理空间不一定连续。
KMAP为永久内存映射区，只有4M，使用kmap将分配的高端内存映射到该区域。
固定映射区和尾端的4K隔离带一般用作特殊用途。

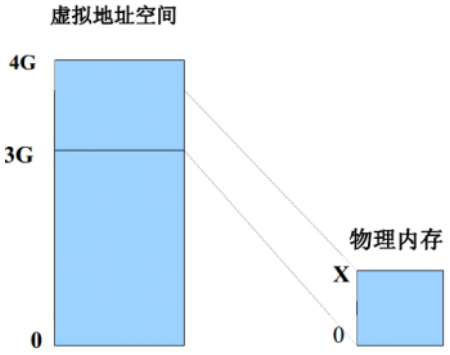
# 第4章习题作业

2013年1月25日
15:03

**习题1**
在进程的角度而言，有的数据是可以共享的，有的却是私有的，为了在虚拟内存区域中区分“共享”和“私有”的进程信息，从而将虚拟内存区域分为“内核空间”和“用户空间”。

**习题2**
共享的“内核空间”已经分配了1GB虚拟内存，而“用户空间”是进程私有的，因此为每个进程分配3GB的“用户空间”不会造成虚拟内存空间重叠。

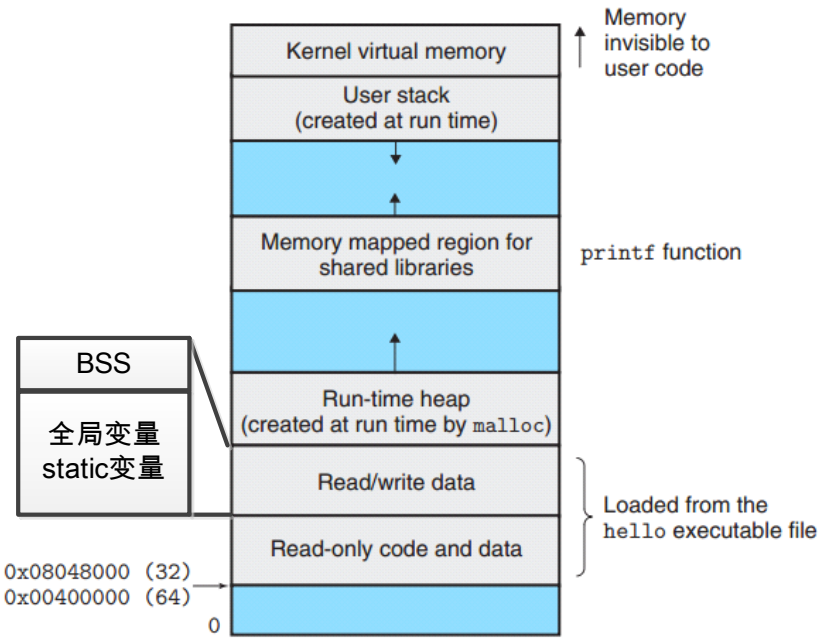
**习题3**
内核空间存放内核映像。
就内核空间而言，虚拟地址到物理地址的转换只差一个偏移量(PAGE_OFFSET=0xC0000000)，如下图所示，可知内核空间的虚拟地址到物理地址映射是线性映射的关系。




屏幕剪辑的捕获时间: 2013/1/25 15:14

**习题4**
内核映像：指内核代码和数据。
内核映像存放在内核空间0x100000为起始的地址处，映射到物理内存地址为0xC0100000。

**习题5**
用户空间的分布见下图（注：最上面为内核空间）：




从中可知，调用malloc分配的内存在堆空间中。用户空间的每个划分区域可以叫做虚存区。

**习题6**

```
struct mm_struct {
    struct vm_area_struct * mmap;       /* list of VMAs */
    struct rb_root mm_rb;              /* 使用红黑树组织VMA时的红黑树的根 */
    struct vm_area_struct * mmap_cache; /* last find_vma result */
    unsigned long (*get_unmapped_area) (struct file *filp,
```

```
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
    void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
    unsigned long mmap_base;            /* base of mmap area */
    unsigned long task_size;            /* size of task vm space */
    unsigned long cached_hole_size;     /* if non-zero, the largest hole below free_area_cache */
    unsigned long free_area_cache;            /* first hole of size cached_hole_size or larger */
    pgd_t * pgd;                /* 进程的页目录基地址 */
    atomic_t mm_users;                  /* How many users with user space? */
    atomic_t mm_count;                  /* How many references to "struct mm_struct" (users count as 1) */
    int map_count;                      /* number of VMAs */
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;         /* Protects page tables and some counters */

    struct list_head mmlist;            /* List of maybe swapped mm's.      These are globally strung
                         * together off init_mm.mmlist, and are protected
                         * by mmlist_lock
                         */

    /* Special counters, in some configurations protected by the
     * page_table_lock, in other configurations by being atomic.
     */
    mm_counter_t _file_rss;
    mm_counter_t _anon_rss;

    unsigned long hiwater_rss;    /* High-watermark of RSS usage */
    unsigned long hiwater_vm;     /* High-water virtual memory usage */

    unsigned long total_vm, locked_vm, shared_vm, exec_vm;
    unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;

    unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */

    struct linux_binfmt *binfmt;

    cpumask_t cpu_vm_mask;

    /* Architecture-specific MM context */
    mm_context_t context;

    /* Swap token stuff */
    /*
     * Last value of global fault stamp as seen by this process.
     * In other words, this value gives an indication of how long
     * it has been since this task got the token.
     * Look at mm/thrash.c
     */
    unsigned int faultstamp;
    unsigned int token_priority;
    unsigned int last_interval;

    unsigned long flags; /* Must use atomic bitops to access the bits */

    struct core_state *core_state; /* coredumping support */
#ifdef CONFIG_AIO
    spinlock_t        ioctx_lock;
    struct hlist_head ioctx_list;
#endif
#ifdef CONFIG_MM_OWNER
    /*
     * "owner" points to a task that is regarded as the canonical
     * user/owner of this mm. All of the following must be true in
     * order for it to be changed:
     *
     * current == mm->owner
     * current->mm != mm
     * new_owner->mm == mm
     * new_owner->alloc_lock is held
     */
    struct task_struct *owner;
#endif

#ifdef CONFIG_PROC_FS
    /* store ref to file /proc/<pid>/exe symlink points to */
    struct file *exe_file;
    unsigned long num_exe_file_vmas;
#endif
#ifdef CONFIG_MMU_NOTIFIER
    struct mmu_notifier_mm *mmu_notifier_mm;
#endif
};
```

从源代码中可以看出，mm_struct通过双向链表进行组织，而每个进程虚拟地址空间的虚存区(VMA)可以通过双向链表或者红黑树进行组织，在mm_struct中包含了链表头和红黑树的根。

**习题7**

```
/*
 * This struct defines a memory VMM memory area. There is one of these
 * per VM-area/task.  A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */
```

```
struct vm_area_struct {
     struct mm_struct * vm_mm;     /* The address space we belong to. */
     unsigned long vm_start;       /* Our start address within vm_mm. */
     unsigned long vm_end;         /* The first byte after our end address
                          within vm_mm. */

     /* linked list of VM areas per task, sorted by address */
     struct vm_area_struct *vm_next, *vm_prev;   /* 双向链表组织VMAs */

     pgprot_t vm_page_prot;        /* Access permissions of this VMA. */
     unsigned long vm_flags;       /* Flags, see mm.h. */

     struct rb_node vm_rb;      /* 红黑树节点——红黑树方式组织 */

     /*
      * For areas with an address space and backing store,
      * linkage into the address_space->i_mmap prio tree, or
      * linkage to the list of like vmas hanging off its node, or
      * linkage of vma in the address_space->i_mmap_nonlinear list.
      */
     union {
         struct {
              struct list_head list;
              void *parent;     /* aligns with prio_tree_node parent */
              struct vm_area_struct *head;
         } vm_set;

         struct raw_prio_tree_node prio_tree_node;
     } shared;

     /*
      * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
      * list, after a COW of one of the file pages.  A MAP_SHARED vma
      * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
      * or brk vma (with NULL file) can only be in an anon_vma list.
      */
     struct list_head anon_vma_node;     /* Serialized by anon_vma->lock */
     struct anon_vma *anon_vma;    /* Serialized by page_table_lock */

     /* Function pointers to deal with this struct. */
     const struct vm_operations_struct *vm_ops;

     /* Information about our backing store: */
     unsigned long vm_pgoff;       /* Offset (within vm_file) in PAGE_SIZE
                          units, *not* PAGE_CACHE_SIZE */
     struct file * vm_file;        /* File we map to (can be NULL). */
     void * vm_private_data;       /* was vm_pte (shared mem) */
     unsigned long vm_truncate_count;/* truncate_count or restart_addr */

#ifndef CONFIG_MMU
     struct vm_region *vm_region;  /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
     struct mempolicy *vm_policy;  /* NUMA policy for the VMA */
#endif
};
```

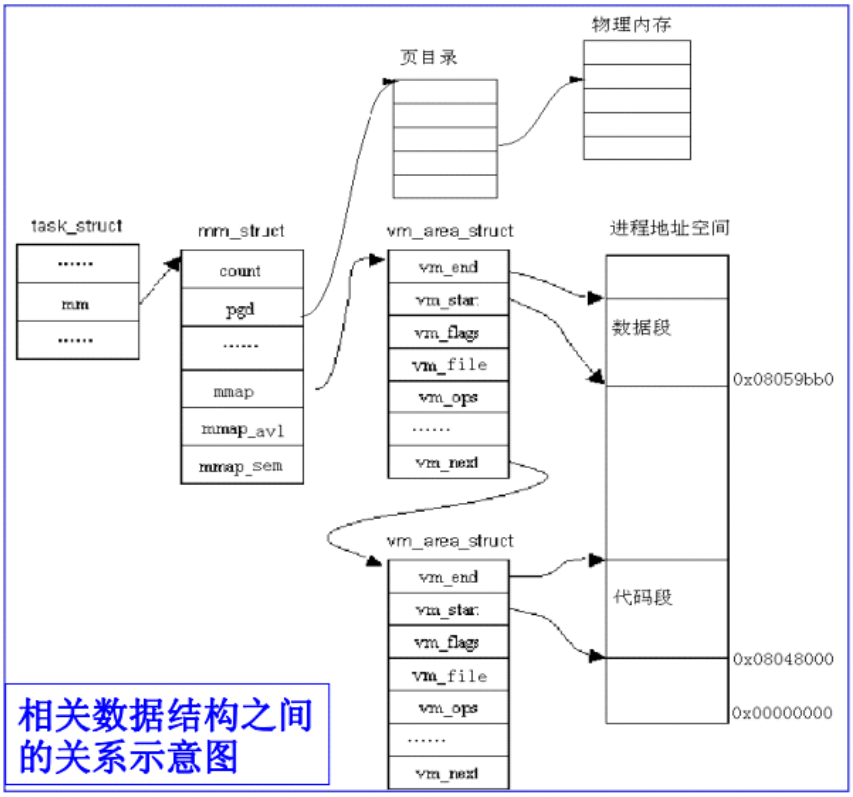
每个进程虚拟地址空间的虚存区(VMA)可以通过双向链表或者红黑树进行组织，在mm_struct中包含了链表头和红黑树的根。

**习题8**

把进程的用户地址空间划分为一个个区间：便于对进程的不同属性信息进行不同的管理或处理。这与我们生活中所说的“大事化小，小事化了”有相同之处。

注：对于复杂的问题，我们常常将其分割成多个模块处理，分割后模块的复杂度与分割的模块数成近似二次曲线的关系。

**习题9**




屏幕剪辑的捕获时间: 2013/1/26 19:20

task_struct结构中的mm域指向进程用户空间（mm_struct描述）。
mm_struct中的pgd域指向页目录的基地址，实际寻址时将装入CR3寄存器的高20bits。
mm_struct中的mmap或mm_rb域指向VMA的双向链表组织的表头或红黑树的树根。
vm_area_struct中的vm_start和vm_end分别存储了每个VMA的起始地址和结束地址（相对于进程地址空间首地址的绝对地址）。

**习题10**
进程何时创建用户空间：当创建一个新的进程时，子进程拷贝或共享父进程的用户空间。在子进程创建用户空间时，只拷贝部分父进程用户空间信息，而大部分信息先设置为只读的共享状态，当子进程或父进程需要向共享区间写入时子进程才将该共享区拷贝一份，这就是所谓的“写时复制”技术。

**习题11**
将可执行映像链接到进程用户空间的方法叫做“虚存映射”。也就是将磁盘文件映射到进程用户空间，这样把对磁盘文件的访问转化为对虚存区的访问。
虚存映射主要有2种：私有映射和共享映射。还有一种映射与文件无关则叫匿名映射，比如堆栈的映射。

**习题12**

```
// exam.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
     int i;
     unsigned char *buf;

     buf = (char *)malloc(sizeof(char)*1024);
     if (buf == NULL) {
          printf("alloc space error!\n");
          return -1;
     }
     printf("my pis is : %d\n", getpid());
     for (i=0; i<60; i++) {
          sleep(60);
     }

     return 0;
}
```

后台执行该程序./exam&

```
[monkeyzx@localhost chpt4]$ ./exam&
[1] 4908
[monkeyzx@localhost chpt4]$ my pis is : 4908
```

屏幕剪辑的捕获时间: 2013/1/29 19:38

查看exam进程的虚存区信息：cat /proc/4908/maps

屏幕剪辑的捕获时间: 2013/1/29 19:39

```
[monkeyzx@localhost chpt4]$ cat /proc/4908/maps
0036b000-00384000 r-xp 00000000 fd:00 951952     /lib/ld-2.5.so
00384000-00385000 r-xp 00019000 fd:00 951952     /lib/ld-2.5.so
00385000-00386000 rwxp 0001a000 fd:00 951952     /lib/ld-2.5.so
00388000-004c2000 r-xp 00000000 fd:00 952029     /lib/libc-2.5.so
004c2000-004c4000 r-xp 00139000 fd:00 952029     /lib/libc-2.5.so
004c4000-004c5000 rwxp 0013b000 fd:00 952029     /lib/libc-2.5.so
004c5000-004c8000 rwxp 004c5000 00:00 0
0076c000-0076d000 r-xp 0076c000 00:00 0          [vdso]
08048000-08049000 r-xp 00000000 00:15 2584       /mnt/hgfs/Linux/clj_linux/chpt4/exam
08049000-0804a000 rw-p 00000000 00:15 2584       /mnt/hgfs/Linux/clj_linux/chpt4/exam
099ea000-09a0b000 rw-p 099ea000 00:00 0
b7f51000-b7f53000 rw-p b7f51000 00:00 0
b7f60000-b7f61000 rw-p b7f60000 00:00 0
bfbbf000-bfbd5000 rw-p bfbbf000 00:00 0          [stack]
```

屏幕剪辑的捕获时间: 2013/1/29 19:40

从地址0x0848000开始的虚地址映射到exam程序代码区、exam数据区，0xbfbbf000开始是堆栈区。ld和libc库都各自含有代码区数据区和BSS区。

**问题：**[vdso]表示什么，在exam程序的代码数据区之后堆栈区之前的地址区域表示什么？？

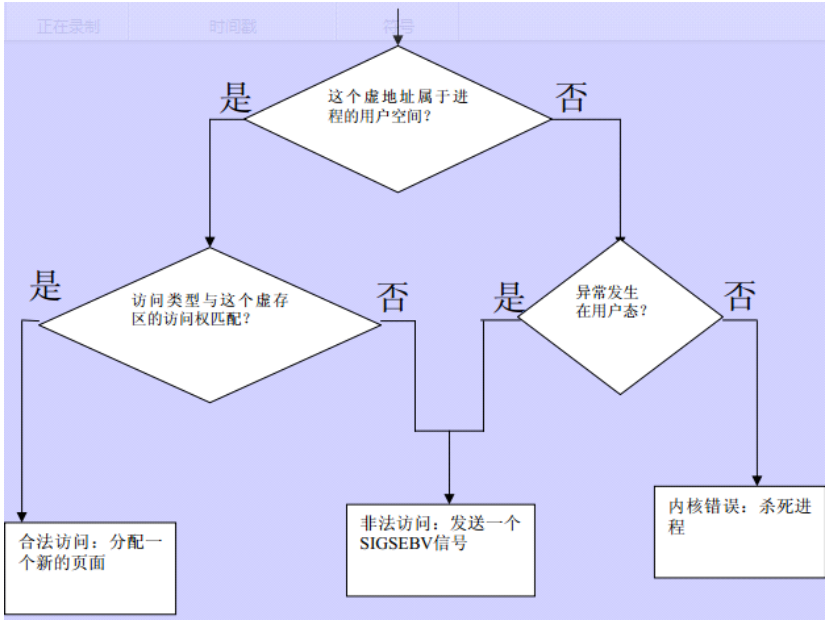
**习题13**

mmap()系统调用功能：在用户空间创建一个新的虚存区，能将文件映射到虚存区。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>

int copy_bytes = 6;  //要拷贝的字节数

int copy_file(int src_fd, int dst_fd)
{
     char *src_buf;
     char *dst_buf;
     int i = 0;

     src_buf = (char *)mmap(0, copy_bytes, /*PROT_READ*/0x1, /*MAP_PRIVATE*/0x02, src_fd, 0);
     //dst_buf = (char *)mmap(0x1000, copy_bytes, /*PROT_WRITE*/0x2, /*MAP_PRIVATE*/0x02, dst_fd, 0);
     //不能写入dst_buf，不知道为什么，所以下面使用write系统调用写入文件

     // 经典的代码写入函数
     while (i=write(dst_fd, src_buf, copy_bytes)) {
          if ((i == -1) && (errno==EINTR)) {
               break;
          } else if (i == copy_bytes) {
               break;
          } else if (i > 0) {
               copy_bytes -= i;
               src_buf += i;
          }
     }
     printf("copyed:\n");
     for (i=0; i<copy_bytes; i++) {
          //dst_buf[i] = src_buf[i];  //这里不能写入dst_buf
          printf("%c", src_buf[i]);
     }
     printf("\n");

     return 1;
}

int main(int argc, char *argv[])
{
     int src_fd = 0;
     int dst_fd = 0;

     if (argc != 3) {
               printf("usage: ./copy_file [src-file] [dst_file]\n");
               return -1;
     }
     /* open source file */
     if (-1 == (src_fd=open(argv[1], O_RDONLY))) {
          fprintf(stderr, "Open %s error:%s\n", argv[1], strerror(errno));
          exit(1);
     }
     /* open/create target file */
     if (-1 == (dst_fd=open(argv[2], O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR))) {
          fprintf(stderr, "Open %s error:%s\n", argv[2], strerror(errno));
          close(src_fd);
          exit(1);
```

```
    }

    copy_file(src_fd, dst_fd);

    close(src_fd);
    close(dst_fd);

    return 0;
}
```

**习题14**

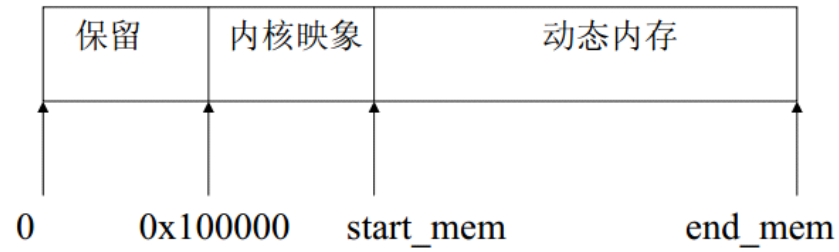



屏幕剪辑的捕获时间: 2013/2/1 20:50

上图为缺页异常处理程序的总体流程图。
（1）do_page_fault()首先读取引起缺页异常的虚地址，若虚地址在进程的用户空间，则虚地址有效，否则无效；
（2）对于无效的虚地址，判断异常发生时的系统所处状态而决定错误处理方法——缺页异常肯定发生在内核态；
（3）对于有效的虚地址，Linux需要判断页的访问权限——写访问、可写、可读、可执行，还要对物理页存在进行判断。

**习题15**
“请求调页”是一种动态的内存分配技术，将页面的分配推迟到不能再推迟为止。

**习题16**
系统启动后物理内存的布局为

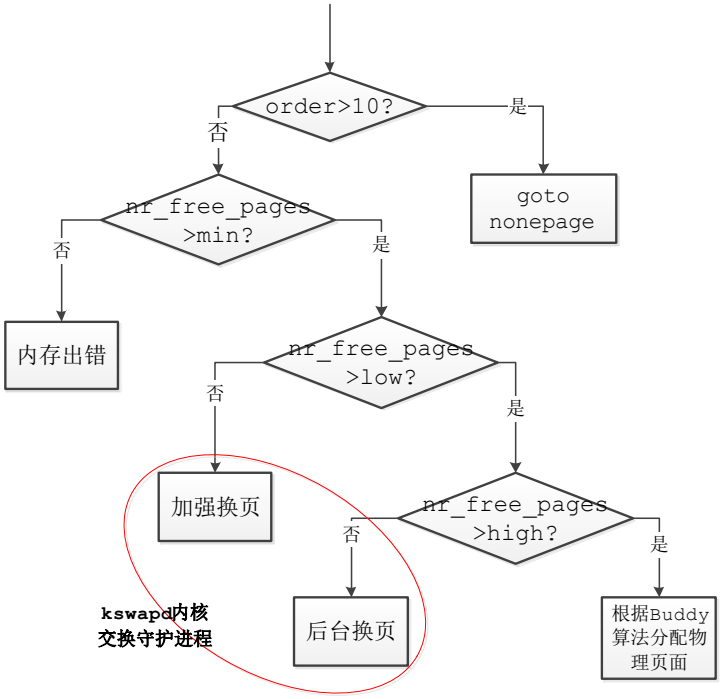



屏幕剪辑的捕获时间: 2013/2/1 21:25

（1）开头1M区间用来存放与系统硬件相关的代码或数据
（2）从0x100000~start_mem存放Linux内核映像
（3）从start_mem到end_mem存放用户的程序和数据

**习题17**
伙伴算法为什么能减少碎片：
（1）在分配时，根据需要，分配与所需内存大小最相近稍大的块，这样碎片最少；
（2）在回收时，检查相邻的伙伴块，实时对释放的伙伴块进行合并；
综合以上原因，伙伴算法的关键是对不同大小的连续空闲页面进行了分区（块）的管理。

**习题18**




上图为分配物理页面的流程图，代码上主要通过__get_free_pages()实现。

**习题19**

由于Buddy算法每次分配的物理内存比较大，为方便分配小块的内存区域，引入slab分配机制。

**习题20**

（1）slab专用缓冲区分配举例

下面为fork_init函数中的一段代码，用于创建task_struct_cachep缓冲区，

```
/* create a slab on which task_structs can be allocated */
task_struct_cachep =
     kmem_cache_create("task_struct", sizeof(struct task_struct),
          ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_NOTRACK, NULL);
```

在fork.c中有在task_struct_cachep创建对象的宏定义，

```
# define alloc_task_struct()  kmem_cache_alloc(task_struct_cachep, GFP_KERNEL)
# define free_task_struct(tsk) kmem_cache_free(task_struct_cachep, (tsk))
```

宏定义将在do_fork()中的dup_task_struct调用。

（2）slab通用缓冲区分配举例

"参见第一章习题13"

**习题21**

内核的非连续空间位于动态映射区。

**习题22**

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/vmalloc.h>

unsigned long pagemem;
unsigned char *kmallocmem;
unsigned char *vmallocmem;

MODULE_LICENSE("GPL");
MODULE_AUTHOR("xhzuoxin");

static int __init init_mmshow(void)
{
     pagemem = __get_free_page(GFP_KERNEL);
     if (!pagemem) {
          return -1;
     } else {
          printk(KERN_INFO"pagemem=0x%lx\n", pagemem);
     }

     kmallocmem = (char *)kmalloc(100, GFP_KERNEL);
     if (!kmallocmem) {
          free_page(pagemem);
          return -1;
     } else {
          printk(KERN_INFO"kmallocmem=0x%p\n", kmallocmem);
     }

     vmallocmem = (char *)vmalloc(1000000);
     if (!vmallocmem) {
```

```
        free_page(pagemem);
        kfree(kmallocmem);
        return -1;
    } else {
        printk(KERN_INFO"vmallocmem=0x%p\n", vmallocmem);
    }

    return 0;
}

static void __exit cleanup_mmshow(void)
{
    if (!pagemem) free_page(pagemem);
    if (!kmallocmem) kfree(kmallocmem);
    if (!vmallocmem) vfree(vmallocmem);
}

module_init(init_mmshow);
module_exit(cleanup_mmshow);
```

Makefile文件为：

```
obj-m:=alloc.o
CURRENT_PATH:=$(shell pwd)
LINUX_KERNEL:=$(shell uname -r)
LINUX_KERNEL_PATH:=/usr/src/kernels/$(LINUX_KERNEL)-i686

all:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) modules
clean:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) clean
```

编译模块，make
安装模块，insmod alloc.ko
查看安装信息，tail -n 3 /var/log/message

```
[root@localhost chpt4]# tail -n 3 /var/log/messages
Feb  2 19:02:53 localhost kernel: pagemem=0xcb2e3000
Feb  2 19:02:53 localhost kernel: kmallocmem=0xf708ccc0
Feb  2 19:02:53 localhost kernel: vmallocmem=0xf8d8a000
```

屏幕剪辑的捕获时间: 2013/2/2 19:08

从日志信息中可知3种分配的虚拟内存首地址。
（1）首先，分配的空间都是内核空间（3GB~4GB）
（2）vmalloc分配的地址位置在kmalloc分配的地址之后，可以尝试安装多次都是这种结果，下图为另一次结果

```
[root@localhost chpt4]# tail -n 3 /var/log/messages
Feb  2 19:11:35 localhost kernel: pagemem=0xca061000
Feb  2 19:11:35 localhost kernel: kmallocmem=0xf77de7c0
Feb  2 19:11:35 localhost kernel: vmallocmem=0xf8e80000
```

屏幕剪辑的捕获时间: 2013/2/2 19:12

**习题26**

# Chpt5 中断和异常

2013年2月3日
19:47

## 1、eg5_1源代码：中断编程

```
/*
 *  计算2次中断的时间间隔
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>

static int irq;
static char *interface;
static int count = 0;

module_param(interface, charp, 0644);
module_param(irq, int, 0644);

static irqreturn_t intr_handler(int irq, void *dev_id)
{
     static long interval = 0;

     if (count > 0) {
          interval = jiffies - interval;
          printk("The interval betwwen two interrupts is %ld\n", interval);
     } else {
          /* do nothing */
     }

     interval = jiffies;
     count++;

     return IRQ_NONE;
}

static int __init intr_init(void)
{
     if (request_irq(irq, &intr_handler, IRQF_SHARED, interface, &irq)) {
          printk(KERN_ERR"Fail to register IRQ%d\n", irq);
          return -EIO;
     }
     printk("%s Request on IRQ %d succeeded\n", interface, irq);

     return 0;
}

static void __exit intr_exit(void)
{
     printk("The %d interrupts happened on irq %d", count, irq);
     free_irq(irq, &irq);
     printk("Freeing IRQ %d\n", irq);
}

module_init(intr_init);
module_exit(intr_exit);

MODULE_LICENSE("GPL");
```

## 2、eg5_2源代码：小任务使用

```
#include <linux/module.h>
#include <linux/init.h>
```

```
#include <linux/fs.h>
#include <linux/kdev_t.h>
#include <linux/cdev.h>
#include <linux/kernel.h>
#include <linux/interrupt.h>

static struct tasklet_struct my_tasklet;

static void tasklet_handler(unsigned long data)
{
    printk(KERN_ALERT"tasklet_handler is running.\n");
}

static int __init test_init(void)
{
    tasklet_init(&my_tasklet, tasklet_handler, 0);
    tasklet_schedule(&my_tasklet);

    return 0;
}

static void __exit test_exit(void)
{
    tasklet_kill(&my_tasklet);
    printk(KERN_ALERT"test_exit running.\n");
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
```

3、eg5_3源代码：工作队列的使用

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/workqueue.h>

static struct workqueue_struct *queue = NULL;
static struct work_struct work;

static void work_handler(struct work_struct *data)
{
    printk(KERN_ALERT"work handler function.\n");
}

static int __init test_init(void)
{
    queue = create_singlethread_workqueue("helloworld");
    if (!queue) return -1;

    INIT_WORK(&work, work_handler, (void *)0);
    schedule_work(&work);

    return 0;
}

static void __exit test_exit(void)
{
    destroy_workqueue(queue);
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
```

# 第5章习题作业

2013年2月3日
19:48

**习题1**

中断:最初指外部中断（硬件中断），是避免实时查询I/O设备而提出的“需要即服务”的机制。
异常：是随着中断的概念扩展出来的，也叫内部中断，是为了解决机器运行时内部随机事件和编程方便提出的。

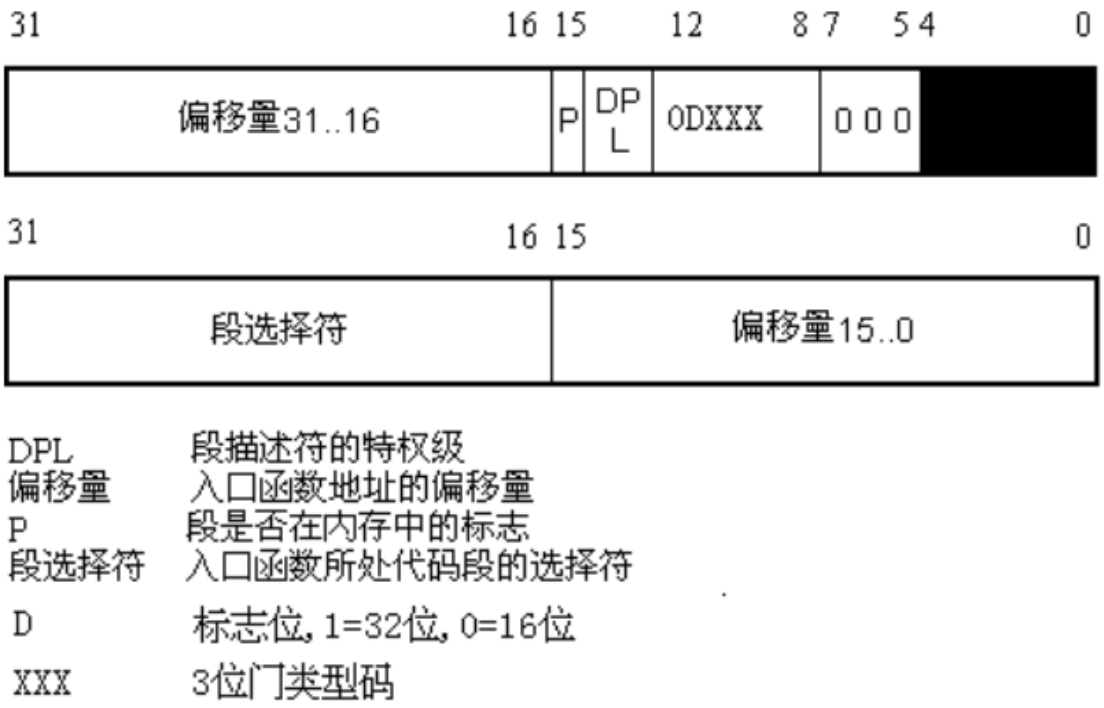
| 中断 | 异常 |
|---|---|
| 外部硬件 | 内部随机事件、编程方便 |
| 可选择是否屏蔽 | 一般不让屏蔽 |

**习题2**

中断向量：中断源的编号，通过中断向量可以计算出中断处理程序的入口地址。 **（中断向量定义有待商榷？？）**
（1）0~31：用于异常和非屏蔽中断
（2）32~47：分配给I/O设备引起的中断，一般为可屏蔽中断
（3）48~255：标识软中断，其中的0x80号用来实现系统调用

**习题3**

中断描述表（IDT）：保护模式下的中断向量表，其中的每一个表项叫做“门描述符”，其格式如下




屏幕剪辑的捕获时间: 2013/2/3 20:19

**习题4**

门描述符类型

| | xxx值 | DPL | 特点 |
|---|---|---|---|
| 中断门 | 110 | 0 | 清IF标志位关中断 |
| 陷阱门 | 111 | 0 | 不清IF标志位 |
| 系统门 | 111 | 3 | 系统调用通过系统门进入 |

**习题5**

CALL指令与INT指令

| | 格式 | 返回指令 | 处理对象 | 压栈reg |
|---|---|---|---|---|
| CALL | CALL 过程名 | RET | 过程 | EIP |
| INT | INT 中断号 | IRET | 中断处理程序 | EFLAG/CS/EIP |

**习题6**

IDT的初始化：

（1）系统初始化阶段初始化可编程中断控制器8259A,将IDT的起始地址装入IDTR寄存器中

（2）使用setup_idt()初始化IDT的256个表项

**习题7**

插入中断门：set_intr_gate

插入陷阱门：set_trap_gate

插入系统门：set_system_gate

**习题8**

内核处理异常：中断处理程序

**习题9**

中断和异常的硬件处理流程：

（1）确定所发生异常的向量i(0~255)

（2）通过ITR寄存器IDT表，读取IDT中的第i表项

（3）进行“段”级和“门”级检查

（4）检查是否发生特权级变化

**习题10**

中断处理程序与中断向量一一对应；

中断服务程序（ISR）针对不同的IO设备（服务）而言。

比如，网卡和图形卡共享0x01号中断线，则

```
void IRQ0x01_interrput()  /* 中断处理程序，中断号为0x01 */
{
    if (FLAGS == 0) {    /* 网卡中断服务程序 */
        ...
    } else if (FLAGS == 1) {  /* 图形卡中断服务程序 */
        ...
    } else {  /* 其它共享0x01中断线的设备 */
        ...
    }
}
```

**习题19**

（注：linux-2.6.32.60）

系统“滴答”时钟中断处理程序：主要完成3项工作

```
void do_timer(unsigned long ticks)
{
    jiffies_64 += ticks; /* 不管32位还是64位机都有jiffies=jiffies_64 */
    update_wall_time();  /* 更新墙上时间 */
    calc_global_load();  /* 计算负载值 */
}
```

该版本内核中将到期定时器的工作单独处理，没有使用“滴答”中断。

更新墙上时间程序：来源timer.c

```
789 /**
790  * update_wall_time - Uses the current clocksource to increment the wall time
791  *
792  * Called from the timer interrupt, must hold a write on xtime_lock.
793  */
794 void update_wall_time(void)
795 {
796     struct clocksource *clock;
797     cycle_t offset;
```

```
798     u64 nsecs;
799
800     /* Make sure we're fully resumed: */
801     if (unlikely(timekeeping_suspended))
802         return;
803
804     clock = timekeeper.clock;
805 #ifdef CONFIG_GENERIC_TIME
806     offset = (clock->read(clock) - clock->cycle_last) & clock->mask;
807 #else
808     offset = timekeeper.cycle_interval;
809 #endif
810     /* Check if there's really nothing to do */
811     if (offset < timekeeper.cycle_interval)
812         return;
813
814     timekeeper.xtime_nsec = (s64)xtime.tv_nsec << timekeeper.shift;
815
816     /* normally this loop will run just once, however in the
817      * case of lost or late ticks, it will accumulate correctly.
818      */
819     while (offset >= timekeeper.cycle_interval) {
820         u64 nsecps = (u64)NSEC_PER_SEC << timekeeper.shift;
821
822         /* accumulate one interval */
823         offset -= timekeeper.cycle_interval;
824         clock->cycle_last += timekeeper.cycle_interval;
825
826         timekeeper.xtime_nsec += timekeeper.xtime_interval;
827         if (timekeeper.xtime_nsec >= nsecps) {
828             int leap;
829             timekeeper.xtime_nsec -= nsecps;
830             xtime.tv_sec++;
831             leap = second_overflow(xtime.tv_sec);
832             xtime.tv_sec += leap;
833             wall_to_monotonic.tv_sec -= leap;
834             if (leap)
835                 clock_was_set_delayed();
836         }
837
838         raw_time.tv_nsec += timekeeper.raw_interval;
839         if (raw_time.tv_nsec >= NSEC_PER_SEC) {
840             raw_time.tv_nsec -= NSEC_PER_SEC;
841             raw_time.tv_sec++;
842         }
843
844         /* accumulate error between NTP and clock interval */
845         timekeeper.ntp_error += tick_length;
846         timekeeper.ntp_error -=
847             (timekeeper.xtime_interval + timekeeper.xtime_remainder) <<
848                     timekeeper.ntp_error_shift;
849     }
850
851     /* correct the clock when NTP error is too big */
852     timekeeping_adjust(offset);
853
854     /*
855      * Since in the loop above, we accumulate any amount of time
856      * in xtime_nsec over a second into xtime.tv_sec, its possible for
857      * xtime_nsec to be fairly small after the loop. Further, if we're
858      * slightly speeding the clocksource up in timekeeping_adjust(),
859      * its possible the required corrective factor to xtime_nsec could
860      * cause it to underflow.
861      *
862      * Now, we cannot simply roll the accumulated second back, since
863      * the NTP subsystem has been notified via second_overflow. So
864      * instead we push xtime_nsec forward by the amount we underflowed,
865      * and add that amount into the error.
866      *
867      * We'll correct this error next time through this function, when
868      * xtime_nsec is not as small.
869      */
870     if (unlikely((s64)timekeeper.xtime_nsec < 0)) {
871         s64 neg = -(s64)timekeeper.xtime_nsec;
872         timekeeper.xtime_nsec = 0;
873         timekeeper.ntp_error += neg << timekeeper.ntp_error_shift;
874     }
```

```
875
876     /* store full nanoseconds into xtime after rounding it up and
877      * add the remainder to the error difference.
878      */
879     xtime.tv_nsec = ((s64) timekeeper.xtime_nsec >> timekeeper.shift) + 1;
880     timekeeper.xtime_nsec -= (s64) xtime.tv_nsec << timekeeper.shift;
881     timekeeper.ntp_error += timekeeper.xtime_nsec <<
882                 timekeeper.ntp_error_shift;
883
884     nsecs = clocksource_cyc2ns(offset, timekeeper.mult, timekeeper.shift);
885     update_xtime_cache(nsecs);
886
887     timekeeping_update(false);
888 }
```

计算负载值：

```
 3145 /*
 3146  * calc_load - update the avenrun load estimates 10 ticks after the
 3147  * CPUs have updated calc_load_tasks.
 3148  */
 3149 void calc_global_load(void)
 3150 {
 3151     unsigned long upd = calc_load_update + 10;
 3152     long active;
 3153
 3154     if (time_before(jiffies, upd))
 3155         return;
 3156
 3157     active = atomic_long_read(&calc_load_tasks);
 3158     active = active > 0 ? active * FIXED_1 : 0;
 3159
 3160     avenrun[0] = calc_load(avenrun[0], EXP_1, active);
 3161     avenrun[1] = calc_load(avenrun[1], EXP_5, active);
 3162     avenrun[2] = calc_load(avenrun[2], EXP_15, active);
 3163
 3164     calc_load_update += LOAD_FREQ;
 3165 }
```

**习题20**

时钟中断的使用程序：

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/interrupt.h>
#include <linux/timer.h>
#include <asm/param.h>  // defined HZ

struct timer_list timer;
static unsigned long timeout;  /* delay times(unit:s) */

module_param(timeout, ulong, 0644);

static void timer_exec(unsigned long data)
{
    printk("<1> Is on time!\n");
}

static int __init timer_init(void)
{
    init_timer(&timer);
    timer.expires = jiffies + timeout*HZ;  /* 1s*/
    timer.data = 0;
    timer.function = timer_exec;

    add_timer(&timer);

    return 0;
}

static void __exit timer_exit(void)
{
    //del_timer(&timer);
    printk("<1>del module timer_test ok!\n");
}

module_init(timer_init);
```

```
module_exit(timer_exit);

MODULE_LICENSE("GPL");
```

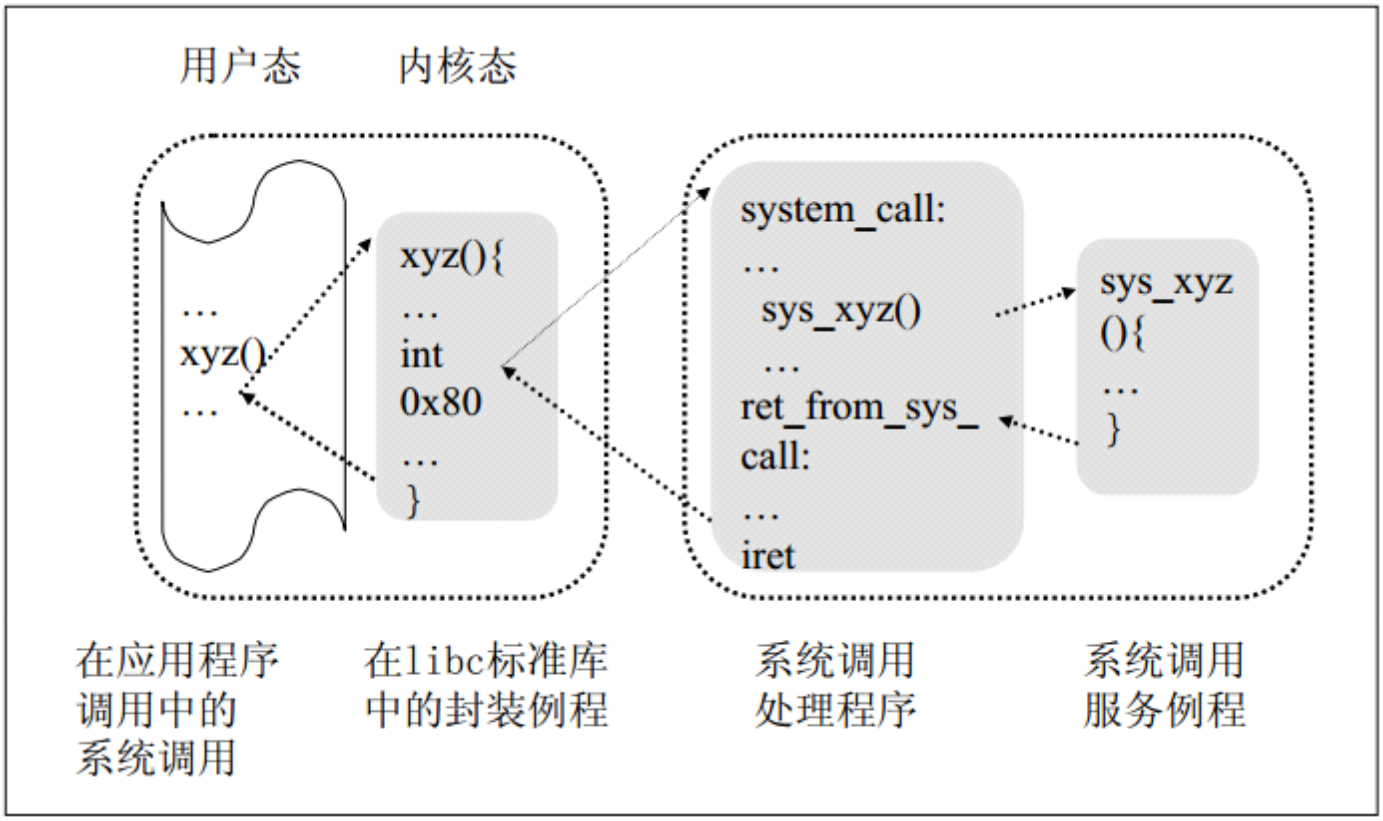
安装insmod timer_test.ko timeout=2，则在2s定时器时间到时控制台输出Is on time.

# Chpt6 系统调用

2013年2月6日
10:44

1、从中断处理的过程去理解系统调用的过程
比如中断处理程序与system_call()的对应，中断服务程序与系统调用服务例程的对应等

2、系统调用的过程




屏幕剪辑的捕获时间: 2013/2/6 11:55

# 第6章习题作业

2013年2月6日
10:44

**习题1**

系统调用：逻辑上讲，是实现内核空间和用户空间程序交互的接口。
为什么提供系统调用？因为在保护模式下，用户空间的程序无法访问内核空间，必须提供一种在用户空间程序访问内核空间的接口实现。
正如UNIX的格言，“提供机制而不是策略”，规范化的系统调用便于系统的移植。

**习题2**

API：Linux系统的API遵循POSIX标准，是应用程序编程接口，不同的API函数（malloc,free,calloc）可能调用了同一个系统调用（brk）；
内核函数：每一个系统调用都对应一个内核函数(sys_)，相对于系统调用叫服务例程
系统命令：通过调用系统调用实现，可能调用多个系统调用
系统调用：相对于内核函数叫封装例程

**习题3**

因为在系统调用过程中，虽然是不同的系统调用，但有许多工作完全一样（比如，保存寄存器、检查系统调用号、调用结束时恢复堆栈等），系统调用处理程序（system_call）完成的就是这些共通的功能。system_call中跳转到具体的中断服务程序（服务例程）中去执行。

**习题4**

在system_call中为什么保存进程PCB到%ebx？为了在system_call退出时完成到进程空间的切换。

**习题5**

system_call源代码如下：（在entry.s文件中）

```
517 ENTRY(system_call)
518     RING0_INT_FRAME         # can't unwind into user space anyway
519     pushl %eax          # save orig_eax
520     CFI_ADJUST_CFA_OFFSET 4
521     SAVE_ALL
522     GET_THREAD_INFO(%ebp)
523                     # system call tracing in operation / emulation
524     testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
525     jnz syscall_trace_entry
526     cmpl $(nr_syscalls), %eax   # 系统调用号有效性检查
527     jae syscall_badsys
528 syscall_call:
529     call *sys_call_table(,%eax,4)   # 跳转到服务例程
530     movl %eax,PT_EAX(%esp)      # store the return value
531 syscall_exit:
532     LOCKDEP_SYS_EXIT
533     DISABLE_INTERRUPTS(CLBR_ANY)    # make sure we don't miss an interrupt
534                     # setting need_resched or sigpending
535                     # between sampling and the iret
536     TRACE_IRQS_OFF
537     movl TI_flags(%ebp), %ecx
538     testl $_TIF_ALLWORK_MASK, %ecx  # current->work
539     jne syscall_exit_work
540
541 restore_all:
542     TRACE_IRQS_IRET
543 restore_all_notrace:
544     movl PT_EFLAGS(%esp), %eax  # mix EFLAGS, SS and CS
545     # Warning: PT_OLDSS(%esp) contains the wrong/random values if we
546     # are returning to the kernel.
547     # See comments in process.c:copy_thread() for details.
548     movb PT_OLDSS(%esp), %ah
549     movb PT_CS(%esp), %al
550     andl $(X86_EFLAGS_VM | (SEGMENT_TI_MASK << 8) | SEGMENT_RPL_MASK), %eax
551     cmpl $((SEGMENT_LDT << 8) | USER_RPL), %eax
552     CFI_REMEMBER_STATE
```

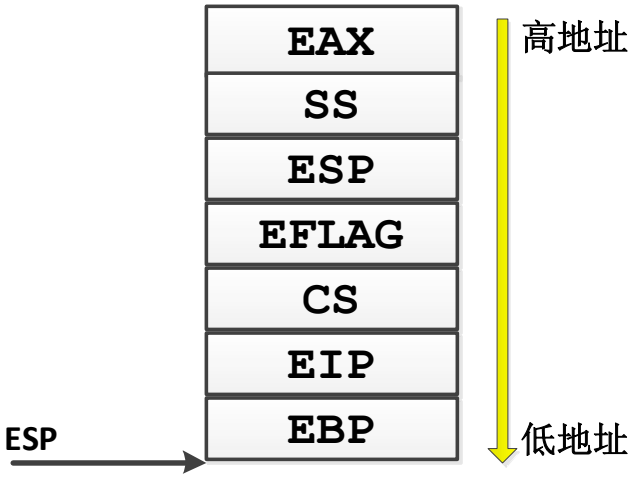
```
553     je ldt_ss           # returning to user-space with LDT SS
554 restore_nocheck:
555     RESTORE_REGS 4          # skip orig_eax/error_code
556     CFI_ADJUST_CFA_OFFSET -4
557 irq_return:
558     INTERRUPT_RETURN
559 .section .fixup,"ax"
560 ENTRY(iret_exc)
561     pushl $0            # no error code
562     pushl $do_iret_error
563     jmp error_code
564 .previous
565 .section __ex_table,"a"
566     .align 4
567     .long irq_return,iret_exc
568 .previous
569
570     CFI_RESTORE_STATE
571 ldt_ss:
572     larl PT_OLDSS(%esp), %eax
573     jnz restore_nocheck
574     testl $0x00400000, %eax     # returning to 32bit stack?
575     jnz restore_nocheck     # allright, normal return
576
577 #ifdef CONFIG_PARAVIRT
578     /*
579      * The kernel can't run on a non-flat stack if paravirt mode
580      * is active.  Rather than try to fixup the high bits of
581      * ESP, bypass this code entirely.  This may break DOSemu
582      * and/or Wine support in a paravirt VM, although the option
583      * is still available to implement the setting of the high
584      * 16-bits in the INTERRUPT_RETURN paravirt-op.
585      */
586     cmpl $0, pv_info+PARAVIRT_enabled
587     jne restore_nocheck
588 #endif
589
590 /*
591  * Setup and switch to ESPFIX stack
592  *
593  * We're returning to userspace with a 16 bit stack. The CPU will not
594  * restore the high word of ESP for us on executing iret... This is an
595  * "official" bug of all the x86-compatible CPUs, which we can work
596  * around to make dosemu and wine happy. We do this by preloading the
597  * high word of ESP with the high word of the userspace ESP while
598  * compensating for the offset by changing to the ESPFIX segment with
599  * a base address that matches for the difference.
600  */
601     mov %esp, %edx          /* load kernel esp */
602     mov PT_OLDESP(%esp), %eax   /* load userspace esp */
603     mov %dx, %ax            /* eax: new kernel esp */
604     sub %eax, %edx          /* offset (low word is 0) */
605     PER_CPU(gdt_page, %ebx)
606     shr $16, %edx
607     mov %dl, GDT_ENTRY_ESPFIX_SS * 8 + 4(%ebx) /* bits 16..23 */
608     mov %dh, GDT_ENTRY_ESPFIX_SS * 8 + 7(%ebx) /* bits 24..31 */
609     pushl $__ESPFIX_SS
610     CFI_ADJUST_CFA_OFFSET 4
611     push %eax           /* new kernel esp */
612     CFI_ADJUST_CFA_OFFSET 4
613     /* Disable interrupts, but do not irqtrace this section: we
614      * will soon execute iret and the tracer was already set to
615      * the irqstate after the iret */
616     DISABLE_INTERRUPTS(CLBR_EAX)
617     lss (%esp), %esp        /* switch to espfix segment */
618     CFI_ADJUST_CFA_OFFSET -8
619     jmp restore_nocheck
620     CFI_ENDPROC
621 ENDPROC(system_call)
```

**习题6**

如果说0x80是系统调用处理程序（system_call）的标识，则系统调用号是系统调用服务程序的标识。

**习题7**

系统调用进入内核后内核栈的示意图如下，（注：x86为满递减堆栈）




EAX：系统调用号
SS：用户栈段地址
ESP：用户栈顶指针
EFLAG：标志寄存器
CS：代码段寄存器
EIP：用户空间系统调用下一条指令地址
EBP：用户进程PCB

**习题8**

内容：使用kdb调试（跟踪）系统调用（read/write）

**习题9**

系统调用的封装：使用了__syscall0~__syscall5共6个宏定义对内核函数进程封装。

**习题10**

- 添加系统调用的步骤：
  - 添加系统调用号。系统调用号在unistd.h文件中定义，以“__NR_”开头
  - 在系统调用表（在entry.S中）中添加自己的服务例程sys_mysyscall
  - 实现系统调用服务例程：把sys_mysyscall加在kernel目录下的系统调用文件sys.c中
  - 重新编译内核
  - 编写用户态程序

屏幕剪辑的捕获时间: 2013/2/6 11:56

# 备注

2013年2月4日
21:47

1、内核源码在/usr/src/目录下，编写内核模块时常常要include [源代码根目录]/include/linux/目录下的相关头文件