考试科目名称 操作系统原理与实践 II （A 卷）

考试方式：闭卷 考试日期：2009 年 月 日 教师：

系（专业）： 年级： 班级：

学号： 姓名： 成绩：

题号	一	二	三	四	五	六
分数						

得分 一、名词解释（本题满分 15 分）

1、 idle 进程

idle 进程不进入就绪队列，仅当就绪队列为空时 idle 进程才会被调度，每个 cpu 上都有一个 idle 进程。

2、 永久内核映射

允许内核建立高端页框（通过 alloc_page() 获得高端内存对应的 page）到内核地址空间的长期映射，内核定义从 PKMAP_BASE 到 FIXADDR_START 的线性地址空间用于用于映射高端内存。

3、 转换后援缓冲 TLB

是一种专用于页表缓存的高速存储部件，主要任务是对页表本身实行二级缓存，缓存页表中的最活跃内容。

4、 内核通用链表

是一种通过 struct list_head 数据结构定义的双向链表，包含两个指向 list_head 结构的指针 prev 和 next。但与传统的双链表结构模型不同， list_head 不是在链表结构中包含数据，而是在数据结构中包含链表节点。

5、 物理地址扩展

属于内存的物理地址扩展，允许将最多 64 GB 的物理内存用作常规的 4 KB 页面，并扩展内核能使用的位数以将物理内存地址从 32 扩展到 36。

得分 二、计算题（本题满分 15 分）

某基于 i386 体系结构的逻辑地址的段标志符（即段选择子）为 0x001F，其段内偏移为 0x00000003，CR3 的地址信息为 0x00102000：

A） 试根据下述 GDT 及 LDT 信息，计算出线性地址。

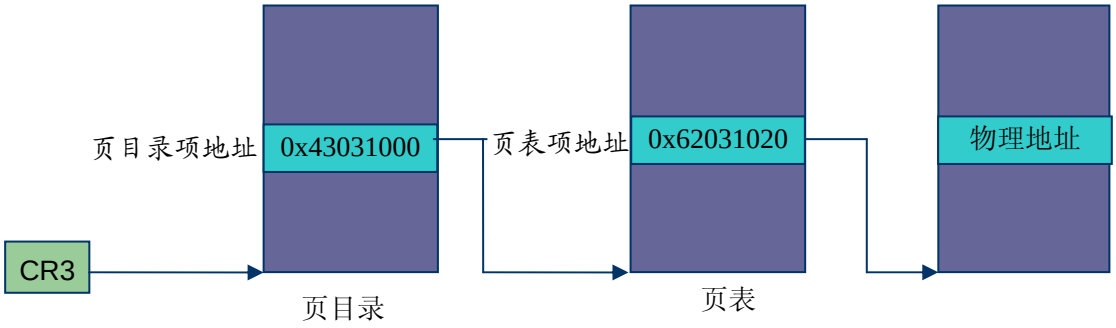
B）试结合下图中的页目录及页表信息，计算出与该线性地址对应的页目录项地址、页表项地址及物理地址。

INDEX	基地址
0	NULL
1	0x32021500
2	0x32021501
3	0x32021502
4	0x32021503
5	0x32021504
6	0x32021505

GDT 中各段描述符的基地址信息

INDEX	基地址
0	NULL
1	0x51021401
2	0x51021402
3	0x51021403
4	0x51021404
5	0x51021405
6	0x51021406

LDT 中各段描述符的基地址信息



线性地址：0x51021406

页目录地址：0x00102510

页表地址：0x43031042

物理地址：0x62031426

得分	

三、简答题（本题满分 30 分）

1、 试述 Linux 2.4 与 Linux 2.6 进程系统堆栈结构的特点及主要差别，并说明在 Linux 系统

中如何区别内核线程与普通进程。

Linux 2.4 与 Linux 2.6 进程系统堆栈结构的特点：每个进程都要单独分配一个系统堆栈，包含内核态的进程堆栈及进程描述符信息。每个系统堆栈占两个页框占据连续两个页框（8192 字节），且第一个页框起始地址为 2^{13} 的倍数。

Linux 2.4 与 Linux 2.6 进程系统堆栈结构主要差别：Linux 2.4 的内核堆栈内嵌 task_struct 结构，描述进程状态；Linux 2.6 的内核堆栈栈底用 thread_info 取代 task_struct，有效节省了内核堆栈空间开销。与此同时，Linux 2.6 采用 slab 分配器为每个进程分配 task_struct 对象。

内核线程与普通进程的区分主要通过进程描述符中的 mm 和 active_mm 来实现。 对普通进程而言，这两个字段存放相同的指针。对内核线程而言，由于不拥有任何内核描述符，因此，mm 字段总为 NULL，当内核线程运行时，active_mm 被初始化为前一个运行进程的 active_mm 的值。

2、 试述 Linux 2.4 与 Linux 2.6 进程调度体系结构的主要差别，并论述 Linux 2.6 调度策略中的负载均衡机制。

1）Linux2.4 所有就绪进程存在一个以 runqueue_head 为表头的全局进程队列中，调度器从中选取最适合调度的进程投入运行。整个队列由一个读/写自旋锁保护，多个处理器可以并行访问，但写操作必须互斥访问。 时间片重算算法是在所有的进程都用尽它们的时间片以后才重新计算。

2）Linux 2.6 的每个处理器有两个数组，活动就绪进程队列数组和不活跃就绪进程队列数组。如果一个进程消耗完了它的“时间片”，就进入不活跃就绪进程数组的相应队列的队尾。当所有的进程都“耗尽”了它的“时间片”后，交换活跃与不活跃就绪进程队列数组，因此，不需要任何其他的开销。

主要差别：Linux 2.4 的调度体系基于共享全局队列，其算法属于 O(n)，开销是线性增长的。而 Linux 2.6 每个处理器都有独立的就绪进程队列，各个处理器可以并行运行调度程序来挑选进程运行，不同处理器上的进程可以完全并行地休眠、唤醒和上下文切换。

Linux 2.6 采用相对集中的负载平衡方案，分为“推”和“拉”两类操作。

“拉”操作：当某个 CPU 负载过轻、而另一 CPU 负载较重时，系统会从重载 CPU 上“拉”进程过来。“拉”的负载平衡操作实现在 load_balance() 函数中，有两种调用方式：

1）“忙平衡”：当前 CPU 不空闲。

2）“空闲平衡”：当前 CPU 空闲。

“推”操作通过 migration_thread()完成。在系统启动时自动加载（每个 CPU 一个），并将自己设为 SCHED_FIFO 的实时进程。该线程定期检查 migration_queue 中是否有请求等待处理，若没有，就在 TASK_INTERRUPTIBLE 中休眠，直至被唤醒后再次检查。

3、 试述软中断、tasklet 及工作队列三种下半部分处理机制在设计思想、技术特点及使用场景等方面的主要区别。

软中断是一组静态定义的下半部接口，共有 32 个，必须在编译阶段静态注册。软中断可以在所有处理器上同时执行（即使两个类型相同的软中断），一个软中断不会抢占另一个软中断。在一个软中断处理程序运行的时候，当前处理器上的软中断被禁止。

tasklet 是一种基于软中断实现的、动态创建的下半部机制。与软中断不同，类型相同的

tasklet 不能同时执行，但两种不同类型的 tasklet 可以在不同处理器上同时执行。实际上，tasklet 由 HI_SOFTIRQ 及 TASKLET_SOFTIRQ 两类软中断代表组成，前者的优先级高于后者。

工作队列将工作推后，交由一个内核线程去执行，此时下半部分在进程上下文执行。因此，通过工作队列执行的代码可以占尽进程上下文的所有优势。

与软中断/tasklet 相比，工作队列支持睡眠，而软中断/tasklet 不支持。

4、 试述 Linux 系统中基于 IRQ 共享的中断处理实现机制。

Linux 通过 struct irq_desc_t 及 struct irqaction 持 IRQ 线共享。其中 irq_desc_t 用于描述中断源，其成员 action 为指针结构，该成员可将共享同一个中断号的多个设备连接起来。Irqaction 是一个链表结构，包含内核接收到特定 IRQ 后应采取的操作，所有共享一个 irq 的链接在一个 action 表。其成员 dev_id 用来记录由制造厂商定义的设备 ID，从而实现对不同中断设备的识别。

5、 试述 Linux 内存管理中 Slab 分配器的实现机制。

slab 分配器将内存区看成对象，并将对象按照类型分组成不同的高速缓存，每个高速缓存都是同种类型内存对象的一种“储备”。每个 slab 由一个或多个连续的页框组成。每个 slab 有三种状态：1）全满状态。全满意味着 slab 中的对象全部已被分配出去；2）半满状态，半满介于两者之间；3）全空状态。全空意味着 slab 中的对象全部是可用的。

slab 对象分配优先从半满的 slab 满足请求；如果没有半满状态 slab，则从全空的 slab 中取一个对象满足请求；如果没有空的 slab，则向伙伴系统申请页面生成一个新的 slab。

Linux 的 slab 分配器的高速缓存包括普通高速缓存和专用高速缓存两种类型。其中普通高速缓存共 2 组 26 个（一组用于 DMA 分配，另一组用于常规分配），由 slab 分配器自己使用，可根据大小分配内存。专用高速缓存(special cache)由内核其余部分使用，可根据类型分配。

得分	

四、程序分析题（本题满分 12 分）

__do_softirq()核心代码

```
#define MAX_SOFTIRQ_RESTART 10
asmlinkage void __do_softirq(void){
  struct softirq_action *h;
  __u32 pending;
  int max_restart = MAX_SOFTIRQ_RESTART;
  int cpu;
  pending = local_softirq_pending();
  account_system_vtime(current);
// 屏蔽其他软中断，使每个CPU上只能同时运行一个软中断
  __local_bh_disable((unsigned long)__builtin_return_address(0));
  trace_softirq_enter();
  // 针对 SMP 得到当前正在处理的 CPU
  cpu = smp_processor_id();
restart:
  set_softirq_pending(0);
  // 此时才开中断运行，以前运行状态一直是关中断运行
 //这时当前处理软中断才可能被硬件中断抢占。
  local_irq_enable();
  // 以下代码可被硬件中断抢占，但无法注册相应软中断
  h = softirq_vec;

  do {
    if (pending & 1) {
      h->action(h);
      rcu_bh_qsctr_inc(cpu);
    }
    h++;
    pending >>= 1;
  } while (pending);
  // 以下代码执行过程中硬件中断无法抢占。
  local_irq_disable();

  // 因为刚才开中断执行过程中可能多次被硬件中断抢占，每抢占一次就有可
 // 能注册一个软中断，所以要再重新取一次所有的软中断。
  pending = local_softirq_pending();
  if (pending && --max_restart)
    goto restart;
  // 如果以上步骤重复了 10 次后还有 pending 的软中断的话，调用 ksoftirq
  if (pending)
    wakeup_softirqd();
  trace_softirq_exit();
  account_system_vtime(current);
  _local_bh_enable();
}
```

ksoftirqd 核心代码

```
static int ksoftirqd(void * __bind_cpu){
  set_user_nice(current, 19);
  // 设置当前进程不允许被挂起
  current->flags |= PF_NOFREEZE;
  set_current_state(TASK_INTERRUPTIBLE);
  while (!kthread_should_stop()) {
  //若可处理，在此处理期间内禁止当前进程被抢占
    preempt_disable();
 // 首先判断系统当前没有需要处理的 pending 状态的软中断
    if (!local_softirq_pending()) {
      // 没有的话在主动放弃 CPU 前先要允许抢占
      preempt_enable_no_resched();
       //将当前进程放入睡眠队列，并切换新的进程执行
      schedule();
      // 再次被调度时，将从调用这个函数的下一条语句开始执行
      preempt_disable();
    }
    __set_current_state(TASK_RUNNING);

    while (local_softirq_pending()) {
      if (cpu_is_offline((long)__bind_cpu))
    // 如果当前被关联的 CPU 无法继续处理则跳转
    // 到 wait_to_die 标记出，等待结束并退出。
        goto wait_to_die;
      do_softirq();
      // 允许当前进程被抢占。
      preempt_enable_no_resched();
      // 该函数有可能间接的调用 schedule() 来切换当前进程
      cond_resched();
      preempt_disable();
    }
    preempt_enable();
    set_current_state(TASK_INTERRUPTIBLE);
  }
  // 如果将会停止则设置当前进程为运行状态后直接返回。
  __set_current_state(TASK_RUNNING);
  return 0;
wait_to_die:
   preempt_enable();
   set_current_state(TASK_INTERRUPTIBLE);
   while (!kthread_should_stop()) {
     schedule();
     set_current_state(TASK_INTERRUPTIBLE);
   }
   __set_current_state(TASK_RUNNING);
   return 0;
}
```

__do_softirq()：固定每次的循环次数，然后就返回；其余挂起的软中断在内核线程

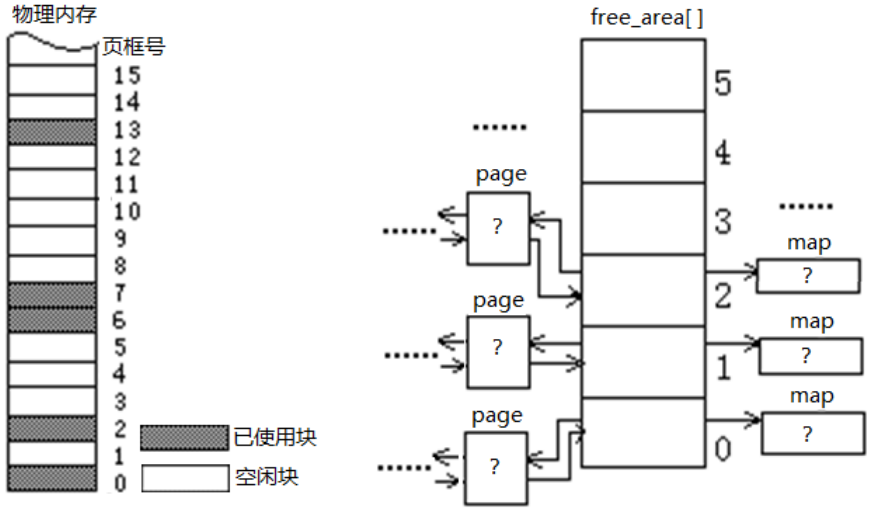
ksoftirqd 中执行。

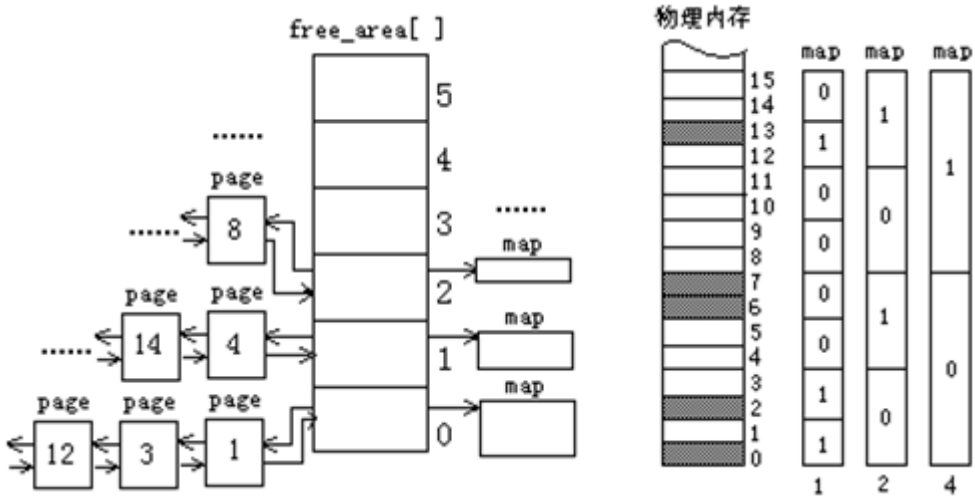
ksoftirqd 内核线程：被唤醒时，检查 local_softirq_pending 中的软中断并在必要时调用 do_softirq()。如果没有挂起的软中断，把当前进程状态设置为休眠，随后若当前进程需要，就调用 cond_resched()函数实现进程切换。

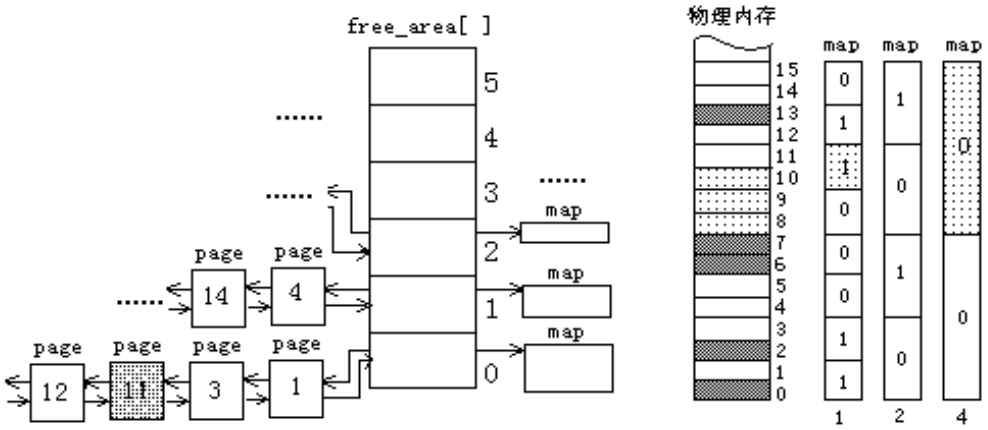
得分	

五、分析题（本题满分 13 分）

设某系统的物理内存由 16 个物理页框组成，并基于伙伴系统实施空闲页框管理。试根据系统当前物理内存使用情况（见左图），给出伙伴管理数据结构 free_area[]（右图）中空闲页框链表（page）结构及伙伴位图（map）的当前状态。若某用户申请三块连续空闲页框，该请求能否得到满足？若能得到满足，请给出请求响应后 free_area[]中 page 和 map 的状态图。







8、9、10页面分配后的示意

得分	

六、程序设计题（本题满分 15 分）

试采用多线程编程技术解决基于有界缓冲区（设缓冲区大小为 N）的生产者/消费者问题。其中消费者/生产者均基于轮询(round-robin)方式从缓冲区读/写入缓冲区。当缓冲区满时，生产者被阻塞；当缓冲区为空时，消费者被阻塞。

```
#include<stdio.h>
#include<pthread.h>
#define BUFSIZE 1000

struct prodcons {
  int buffer[BUFSIZE];
  pthread_mutex_t lock;   //互斥LOCK
  int readpos , writepos;
  pthread_cond_t notempty; //缓冲区非空条件判断
  pthread_cond_t notfull;  //缓冲区未满条件判断
};

void init(struct prodcons * b){
  pthread_mutex_init(&b->lock,NULL);
  pthread_cond_init(&b->notempty,NULL);
  pthread_cond_init(&b->notfull,NULL);
  b->readpos=0;
  b->writepos=0;
}
```

```
void put(struct prodcons* b,int data){
  //如果互斥锁mutex已经被上锁则挂起生产者线程,并返回
  pthread_mutex_lock(&b->lock);
  //等待缓冲区未满
  if((b->writepos + 1) % BUFSIZE == b->readpos){
    //缓冲区满,生产者将被挂起,直至重新被唤醒
    pthread_cond_wait(&b->notfull, &b->lock) ;
  }
  //写数据,并移动指针
  b->buffer[b->writepos]=data;
  b->writepos++;
  if(b->writepos >= BUFSIZE)
    b->writepos=0;
  //设置缓冲区非空的条件变量
  pthread_cond_signal(&b->notempty);
  pthread_mutex_unlock(&b->lock);
}

int get(struct prodcons *b){
  int data;
  pthread_mutex_lock(&b->lock);
  if(b->writepos == b->readpos){
    //等待缓冲区非空
    pthread_cond _wait(&b->notempty, &b->lock);
  }
  //读数据,移动读指针
  data = b->buffer[b->readpos];
  b->readpos++;
  if(b->readpos >= BUFSIZE)
    b->readpos=0;
  //设置缓冲区未满的条件变量
  pthread_cond_signal(&b->notfull);
  pthread_mutex_unlock(&b->lock);
  return data;
}
```

```c
#define OVER (-1)
struct prodcons buffer;
void *producer(void *data){
  int n;
  for(n = 0; n < 10000; n++){
    printf("%d \n", n) ;
    put(&buffer, n);
  }
  put(&buffer, OVER);
  return NULL;
}
void *consumer(void * data){
  int d;
  while(1){
    d = get(&buffer);
    if(d == OVER)
      break;
    printf("%d\n", d);
  }
  return NULL;
}

int main(void){
  pthread_t th_a, th_b;
  void *retval;
  init(&buffer);
  pthread_create(&th_a, NULL, producer, 0);
  pthread_create(&th_b, NULL, consumer, 0);
  pthread_join(th_a, &retval);
  pthread_join(th_b, &retval);
  return 0;
}
```