内核线程与同步

1. 实验内容的简述。

本次实验分两个步骤：

part1 是对于 vfprintf ( ) 函数的实现，用 putchar_func 函数使其可以实现对字符串，int 型整数，无符号十六进制数和 char 型字符的输出。

part2 则是实现内核线程的管理，实现 pcb 创建，线程切换，最后可以通过三个生产者，四个消费者，五个临界区的信号量测试用例。

2.实验目标。

1) 通过本次实验深入了解 pcb 创建，线程切换的过程。

2) 理解通过 pv 操作封锁临界区，进程同步和维护资源计数。

3) 理解信号量的原子性是不可中断的过程，设计 lock unlock 来实现信号量 pv 操作的原子性。

4) 了解线程切换时堆栈指针的切换，以及一些计算机的内部操作。

3.完成实验涉及的额外准备知识。

1) 掌握 git 的使用，用 git 管理代码，并创建分支来测试自己的代码。

2) 了解 Intel IA32 体系结构、中断与 I/O 机制以及内核线程创建切换的实现和信号量 pv 操作以及 lock unlock 等相关知识。

3) 熟悉使用 linux 的内核链表 listhead，通过 listhead 来创建队列等。

4.实验设计。

1）实现 vfprintk（ ），首先将要输出的参数（int 型整数，无符号十六进制数）转化为字符串类型，再通过 putchar_func 函数使其一个字符一个字符的输出，进而实现对字符串，int 型整数，无符号十六进制数和 char 型字符的输出。

2）在实现生产者消费者问题之前，应该先实现几个进程的创建与切换，先申请 pcb 内存空间，在对其初始化，在加入到运行队列，几个进程在运行队列之后，再实现其调度的算法。

3）实现进程的唤醒和睡眠，实现进程加入运行队列和从运行队列中删除。

4）实现信号量的 pv 操作，创建三个生产者，四个消费者，五个临界区，看运行结果是否满足#1 ~ #4 的输出。

5.具体实现。

PRAT1：

1）printf 的实现：检测要输出的字符串中的%，检测到%后，在看其下一个字符。

%d：用 itoa 函数将其转化为字符串，再用 putchar_func 函数将其一个字符一个字符的输出；

%c：把单子符强制转化为 int 整形数，再传递给新定义的参数 char p 用 putchar_func 直接输出 p 对应的字符即可；

%s：字符串类型直接用 putchar_func 输出即可；

%x：用 utoa 将讲述据直接转化为显示成十六进制数的字符串，并在字符串前面加上 0x 便于输出时判断其为十六进制；

%%：只输出一个%。

2）其他情况%后面形式的不规范用 assert（0）报错。

PRAT2：

1）队列头节点的创建和初始化：用 listhead 定义两个全局的头节点，一个 run_queue，一个 wait_queue，分别为运行队列和等待队列的头节点，并用 list_init 对两个头节点进行初始化。

2）线程的创建：由于不好动态的为新建的 pcb 创建内存空间，所以定义了一个全局的 PCB 数组，然后穿键线程的时候，直接从数组中取存储空间，初始值设为 20。对于新创建的线程，直接加入到等待队列中去，而用(&wait_queue)->next 来指向等待队列的第一个 pcb 的 listhead 成员，所以等待队列是一个循环双向链表，而用(&wait_queue)->next 指针指向这个双向链表。

3）线程的唤醒：如果对应的 pcb 在等待队列中则，把要唤醒的线程的 pcb 从等待队列中删去并将其加入到运行队列中去即可，否则直接介入到运行队列中，而运行队列的实现和等待队列的实现是一样的，同样是 listhead 的双向循环链表，并用(&run_queue)->next 指向这个双向循环的运行队列。

4）线程的切换：运行队列中已经有三个线程，接下来就是切换。切换的原理用一句话表述，就是在中断、异常的驱动下，切换堆栈、寄存器和进程现场的行为。就是令每一个中断到来的时候，把寄存器现场信息正确地保存在当前运行进程的堆栈上。寄存器现场(TrapFrame)被保存到 current 指针指向线程所对应的堆栈上，而在中断返回时，返回 current 指针指向线程所对应的寄存器现场。汇编代码中 movl current_tf, %esp 来实现现场的保存。这样，在 irq_handle(tf)函数中进行 current->tf = tf;并且让 current 指针指向运行队列的下一个可运行队列（通过 list_entry()来实现），之后再将 current 指针指向的 tf 赋给 current_tf，这就实现了线程的切换。eip 是线程启动后执行第一条指令的地址，就是线程函数的开始位置，这样就可以看到三个函数切换的效果（分别是循环打印“a”，“bbbb”，”ccccccccc”)。

5）lock（），unlock（）的实现：其实用关中断就可以实现，但是为了实现 lock unlock 的嵌套，可以设置一个 count 全局变量，初始值为 0，当 lock 时，另 count++；unlock 时，count--；只有当 count 等于 0 时才执行打开中断，这样就解决了嵌套的问题。

6）sleep 的实现：只需要将当前的线程从运行队列中删除就可以了，然后再插入一个无害的外部中断即可，asm volatile("int $0x80");

这是就要修改 irq_handle ( ) ，当 irq==80 (十六进制) 时，对其进行中断处理，current 切换到下一个线程即可。注意 Sleep 一定要在 lock unlock 保护下执行。

7) 这时再将信号量的测试代码加入到程序中，就可以实现#1 ~ #4 同时右边严格递增的输出。

6.遇到的困难与解决方法。

遇到的困难：

1) 运行队列建立不正确，链表没有搭建成功。

2) 切换实现不正确。

3) 实现信号量是输出结果不满足要求。

解决的办法：

1) 用 printk 输出运行队列链表的前后节点的地址来判断链表搭建是否成功，同时仔细看 list.h 中各个函数的实现，了解其功能，在对 wakeup 进行修改，然后才找出错误，改正过来。

2) current 没有初始化，所以开始的时候，esp 指向的是系统的堆栈，而不是自己线程的堆栈，所以将 current 初始化为 NULL，再在 irqhandle 中判断 current 是否为空，如果是空，就让 current 指向运行队列中的第一个 pcb (此时运行队列只有一个线程) 。

3) sleep ( ) 实现有问题，只要用 list_del 从运行队列中删除正在运行的线程即可，而 asm volatile("int $0x80");放在了开中断后，所以运行出错。

7.总结

本次实验深切体会到了调试代码的痛苦，不断的用 printk ( ) ，来测试程序执行了哪一步，输出地址看链表建立是否和想象中的一样等，不停的修改，反思自己的逻辑是否有问题，但要始终坚信机器永远是正确的，所以要不停的修改，反复测试，才能找到问题所在的根源。但是通过调试代

码，对一些理论知识有了更深一层的了解，更加的熟悉了电脑内部如何工作，线程如何切换等问题，所以说本次实验的收获还是很大的。

参考文献：80386 手册，课程网站上面的讲解

十分感谢苏成大神对本次实验的指导。

回答问题：

1：启动分页。main.c 中的 entry 函数是整个操作系统代码的入口点。entry 所执行的第一件事就是
init_kvm，创建内核页表并且启动分页。注意到我们在链接内核时使用的编译选项：
$(LD) $(LDFLAGS) -e entry -Ttext 0xC0100000 -o kernel $(OBJS)
内核“认为”它处于内存的 0xc0100000 位置，但是我们在载入时，却是将内核载入到物理内存的
0x100000 位置。在正确的页表建立以前，任意对
0xc0100000 附近地址的访问都将引起非预期的结
果。实际上，编写 bootloader 和 init_kvm 时都应当非常小心地处理虚拟地址和物理地址。请仔细阅
读 bootloader 和 init_kvm 的代码，结合 objdump 的结果，解释内核启动分页的过程。此外，为什
么 kernel 在分页未启动时，试图用虚拟内存地址访问，但却不会产生任何错误？

答：

虚拟机内存大小 128MB，通过计算知道物理地址范围是 0x0000 0000 ~ 0x0800 0000。逻辑地址区间是 0xC000 0000 ~ 0xC800 0000。定义页表 kptab[]有 2^15 项，页目录 kpdir[]有 2^10 项，页框大小 4KB(PG_SIZE)。初始化时，将 128MB 内存分成 2^15 页，所有页的起始位置存入物理地址中，末两位为标记位，表示的是存在并可读写。建立页目录 kpdir[NR_PDE]，页目录元素个数即为每个页表中页表项的数量。初始化时设为全零，不存在不可读写。页目录首地址转换为物理地址 pdir，并将页表中每 NR_PTE(1024)项的物理地址存放进其中，标记为存在和可读写。由于页表共 2^15 项，每隔 1024 项是一个目录，因此共 32 个目录，页目录仅使用了前 32 项。CR3 是页目录基址寄存器，将

kpdir 映射的物理地址存入 CR3，作为页目录的物理地址，建立了分页机制。CR0 最高位是分页允许位，设为 CR0_PG(0x80000000)，启动分页。就此，发现 pdir[]下标 10 位。ptab[]对内存做了索引，每 0x1000 作一个索引。因此当内核使用一个虚拟地址时，根据高 10 位找到页表基地址，确定地址在页表中的位置。而后根据中间 10 位在该位置开始的 1024 个页表项中找到该页表项，确定在该地址内存中位置。最后根据低 12 位作为偏移量找到该地址在内存中的位置。在 kernel 分页未启动时，在 boot/main.c 中，定义了虚拟地址转换成物理地址，减去 0xC0000000，而非分页之后的含有二级页表的虚拟地址。需要使用地址时，执行 VA_TO_PA 操作，获取物理地址。因此虚拟内存访问没有任何错误。

2：理解 volatile。在 main.c 中有一段难以理解的代码：

```
init_kvm();
void(*volatile next)(void) = os_init;
asm volatile("addl %0, %%esp" : : ""(KOFFSET));
next();
panic("init code should never return");
```

细心的你会发现，即便将 volatile 关键字删除，我们的程序仍然能够平稳正确的运行。然而，这里的
volatile 却是不能忽略的。当我们创建 ring3 的用户进程以不同的内存映射访问时，删除 volatile 将
会引起整个操作系统的崩溃。
请使用 objdump 工具比较在有和没有 volatile 关键字时生成汇编代码的细微区别，并解释为什么现
在代码运行得很好，但是在位于内存低位的页表映射改变时，为什么会发生问题。

答：

通过 objdump 生成汇编代码，比较可以看出，在减去 volatile 时，汇编代码由

```
void
entry(void) {
  30:   55                      push   %ebp
  31:   89 e5                   mov    %esp,%ebp
  33:   83 ec 28                sub    $0x28,%esp
        init_kvm();
  36:   e8 fc ff ff ff          call   37 <entry+0x7>
        void(*volatile next)(void) = os_init;
  3b:   c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
        asm volatile("addl %0, %%esp" : : ""(KOFFSET));
  42:   81 c4 00 00 00 c0       add    $0xc0000000,%esp
        next();
  48:   8b 45 f4                mov    -0xc(%ebp),%eax
  4b:   ff d0                   call   *%eax
        panic("init code should never return");
  4d:   c7 04 24 18 00 00 00    movl   $0x18,(%esp)
  54:   e8 fc ff ff ff          call   55 <entry+0x25>
}
  59:   c9                      leave
  5a:   c3                      ret
```

变为：

```
void
entry(void) {
  30:   55                      push   %ebp
  31:   89 e5                   mov    %esp,%ebp
  33:   83 ec 08                sub    $0x8,%esp
        init_kvm();
  36:   e8 fc ff ff ff          call   37 <entry+0x7>
//      void(*volatile next)(void) = os_init;
//      asm volatile("addl %0, %%esp" : : ""(KOFFSET));
        void(*next)(void) = os_init;
        asm ("addl %0, %%esp" : : ""(KOFFSET));
  3b:   81 c4 00 00 00 c0       add    $0xc0000000,%esp
        next();
  41:   e8 fc ff ff ff          call   42 <entry+0x12>
```

可以看出，33 的 sub 指令中$0x28 变为$0x8，没有 volatile 关键字后，void (*next) (void) =os_init；这步操作并没有翻译出汇编语言代码，也就是 next 指针并没有重新赋予地址。同时也缺少了 mov　call*%eax 等指令。在有 volatile 关键字时，在用到这个变量时必须每次都从原始内存地址中读取这个变量的值，而不是使用保存在寄存器里的备份。void(*volatile next)(void) = os_init;代码执行时，将 0x0 存入-0xc (%ebp) 位置，在调用 next()函数时，从-0xc 位置重新取值赋

给%eax，调用%eax 位置函数即 os_init()。在去除 volatile 关键字的代码中，由于编译器优化，直接从寄存器中取值，不是从内存中取值，因此如果内存映射改变时，由于不影响寄存器的值，程序仍然调用寄存器所指向的内存区域，这样载进行访问使就很容易出错。