ucore实验报告

笔记本：	My Notebook		
创建时间：	2022/8/4 20:43	**更新时间：**	2022/8/6 13:21
作者：	赵凌珂		
URL：	https://blog.csdn.net/qq_51684393/article/details/125069000		

ucore实验：

lab1介绍了x86的启动，函数的调用以及中断处理等操作。
启动：
加电之后，CS和EIP结合形成了第一个地址，其他设备也会有对应的初始值。地址指向的是BIOS内存固件EPROM。然后BIOS工作，硬件初始化。加载磁盘或者硬盘的第一个主引导扇区（零号）。地址：0x7c00
bookloader是扇区里的其中一个代码，进一步加载ucore。
从实模式（16位）变到保护模式（32位），读取kernel等代码。跳转到ucore OS入口。
GDT段

中断：
IDT：中断描述符表（index对应中断号）
中断之后获得中断号，根据index查找对应的中断门，找到段选择址（GDT）。

lab2：物理内存的管理
特权级：用户态和内核态
内存物理空间
基于连续物理内存空间的动态内存分配与释放
建立页机制：
页目录项+页表项+物理地址

lab3：虚拟内存管理
给未被映射的地址映射上物理页

基于FIFO的页面替换算法：
当需要调用的页不在页表中时，并且在页表已满的情况下，会引发页面替换，此时需要进行换入和换出操作：
判断是否访问过，先对未访问过的页进行FIFO；
将最早被换入，且最近没有再被访问的页被换出；

ucore-lab1：

操作系统是一个软件，也需要通过某种机制加载并运行它。我们将通过另外一个简单的软件bootloader来完成这些工作。为此，我们需要完成一个能够切换到x86的保护模式并显示字符的bootloader，为启动操作系统ucore做准备。

这个bootloader可以切换到X86保护模式，能够读磁盘并加载ELF执行文件格式，并显示字符。

练习一：镜像文件ucore.img的生成

```
//初始化
+ cc kern/init/init.c
gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/init/init.c -o obj/kern/init/init.o
//读行
+ cc kern/libs/readline.c
gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/libs/readline.c -o obj/kern/libs/readline.o
//标准IO
+ cc kern/libs/stdio.c
gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/libs/stdio.c -o obj/kern/libs/stdio.o

+ cc kern/debug/kdebug.c
gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/debug/kdebug.c -o obj/kern/debug/kdebug.o
+ cc kern/debug/kmonitor.c
gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o
+ cc kern/debug/panic.c
gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/debug/panic.c -o obj/kern/debug/panic.o

//时钟控制
+ cc kern/driver/clock.c
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/driver/clock.c -o obj/kern/driver/clock.o
+ cc kern/driver/console.c
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/driver/console.c -o obj/kern/driver/console.o
+ cc kern/driver/intr.c
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/driver/intr.c -o obj/kern/driver/intr.o
+ cc kern/driver/picirq.c
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/driver/picirq.c -o obj/kern/driver/picirq.o
+ cc kern/trap/trap.c
gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/trap/trap.c -o obj/kern/trap/trap.o
+ cc kern/trap/trapentry.S
gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/trap/trapentry.S -o obj/kern/trap/trapentry.o
```

```
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/trap/vectors.S -o obj/kern/trap/vectors.o
+ cc kern/mm/pmm.c
gcc -Ikern/mm/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/mm/pmm.c -o obj/kern/mm/pmm.o
//格式化输出
+ cc libs/printfmt.c
gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector
-Ilibs/  -c libs/printfmt.c -o obj/libs/printfmt.o
//字符串相关
+ cc libs/string.c
gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector
-Ilibs/  -c libs/string.c -o obj/libs/string.o
//建立链接
+ ld bin/kernel
ld -m    elf_i386 -nostdlib -T tools/kernel.ld -o
bin/kernel  obj/kern/init/init.o obj/kern/libs/readline.o obj/kern/libs/stdio.o
obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/debug/panic.o
obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/intr.o
obj/kern/driver/picirq.o obj/kern/trap/trap.o obj/kern/trap/trapentry.o
obj/kern/trap/vectors.o obj/kern/mm/pmm.o  obj/libs/printfmt.o obj/libs/string.o


//构建bootblock
+ cc boot/bootasm.S
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector
-Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector
-Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
+ cc tools/sign.c
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
+ ld bin/bootblock
ld -m    elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o
obj/boot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 488 bytes
build 512 bytes boot sector: 'bin/bootblock' success!

//构建ucore.img
//使用dd创建空文件
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.151391 s, 33.8 MB/s
//使用dd将bin/bootlock写入ucore.img
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000233029 s, 2.2 MB/s
//使用dd将bin/kernel写入ucore.img的1个block之后，即bootlock之后
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
146+1 records in
146+1 records out
74923 bytes (75 kB) copied, 0.00185385 s, 40.4 MB/s
```

查看Makefile文件可知生成ucore.img文件的代码为：

```
# create ucore.img
UCOREIMG    := $(call totarget,ucore.img)

$(UCOREIMG): $(kernel) $(bootblock)
    $(V)dd if=/dev/zero of=$@ count=10000
    $(V)dd if=$(bootblock) of=$@ conv=notrunc
    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc
```

1.创建一个大小为 10000 字节的块
2.将 bootblock，kernel 拷贝过去。
3.通过 dd 命令将 bootblock 放到第一个 sector，将 kernel 放到第二个 sector 开始的区域。
bootblock 就是引导区，kernel 则是操作系统内核。

执行less tools/sign.c：

```
1 #include <stdio.h>
      2 #include <errno.h>
      3 #include <string.h>
      4 #include <sys/stat.h>
      5
      6 int
      7 main(int argc, char *argv[]) {
      8     struct stat st;
      9     if (argc != 3) {
     10         fprintf(stderr, "Usage: <input filename> <output filename>\n");
     11         return -1;
     12     }

            //读取文件
     13     if (stat(argv[1], &st) != 0) {
     14         fprintf(stderr, "Error opening file '%s': %s\n", argv[1],
strerror(errno));
     15         return -1;
     16     }

            //输出文件大小
     17     printf("'%s' size: %lld bytes\n", argv[1], (long long)st.st_size);


            //检查文件大小，不得超过510
     18     if (st.st_size > 510) {
     19         fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
     20         return -1;
     21     }

            //定义缓冲区
     22     char buf[512];
     23     memset(buf, 0, sizeof(buf));
     24     FILE *ifp = fopen(argv[1], "rb");
     25     int size = fread(buf, 1, st.st_size, ifp);
     26     if (size != st.st_size) {
     27         fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
     28         return -1;
     29     }
     30     fclose(ifp);

            //结束标志位
     31     buf[510] = 0x55;
     32     buf[511] = 0xAA;

            //写入文件
     33     FILE *ofp = fopen(argv[2], "wb+");
     34     size = fwrite(buf, 1, 512, ofp);

            //检查文件大小
     35     if (size != 512) {
     36         fprintf(stderr, "write '%s' error, size is %d.\n", argv[2],
size);
     37         return -1;
     38     }
     39     fclose(ofp);//释放文件
     40     printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);
```

```
    41     return 0;
    42 }
    43
```

由编码可以看出，硬盘主引导扇区的特征是：

1.磁盘主引导扇区只有 512 字节；
2.内容不超过 510 字节；
3.最后两个字节为 0x55和0xAA。

练习二：使用qemu调试：

1.单步跟踪BIOS的执行

1.修改 lab1/tools/gdbinit：

```
file bin/kernel
set architecture i8086
target remote :1234
break kern_init
continue
```

2.在 lab1目录下，执行make debug：

```
warning: A handler for the OS ABI "GNU/Linux" is not built into this
configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x0000fff0 in ?? ()
Breakpoint 1 at 0x100000: file kern/init/init.c, line 17.

Breakpoint 1, kern_init () at kern/init/init.c:17
(gdb)
```

这时gdb停在BIOS的第一条指令处：0xffff0

3 在看到gdb的调试界面(gdb)后，执行si命令，就可以看到BIOS在执行了；

4 此时的CS=0xf000, EIP=0xfff0；
执行x /2i 0xffff0：

```
(gdb) x /2i 0xffff0
   0xffff0:     ljmp   $0xf000,$0xe05b
   0xffff5:     xor    %dh,0x322f
```

进一步可以执行
x /10i 0xfe05b
可以看到后续的BIOS代码。

```
  0xfe05b:     cmpl   $0x0,%cs:0x65a4
   0xfe062:     jne    0xfd2b9
   0xfe066:     xor    %ax,%ax
```

```
   0xfe068:     mov    %ax,%ss
   0xfe06a:     mov    $0x7000,%esp
   0xfe070:     mov    $0xf3c4f,%edx
   0xfe076:     jmp    0xfd12a
   0xfe079:     push   %ebp
   0xfe07b:     push   %edi
   0xfe07d:     push   %esi
(gdb)
```

2.在初始化位置0x7c00设置实地址断点

1.执行less lab1init命令：

```
      1 file bin/kernel
      2 target remote :1234
      3 set architecture i8086
      4 b *0x7c00
      5 continue
      6 x /2i $pc
```

其中，b即break ,设置断点于0x7c00位置

2.执行make lab1-mon：

```
0x0000fff0 in ?? ()
warning: A handler for the OS ABI "GNU/Linux" is not built into this
configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
Breakpoint 1 at 0x7c00

Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:    cli
   0x7c01:    cld
(gdb)
```

可知，断点在0x7c00处，设置成功。

3.在下方的gdb中输入x/10i $pc查看之后的十条指令：

```
=> 0x7c00 <start>:      cli
   0x7c01 <start+1>:    cld
   0x7c02 <start+2>:    xor    %ax,%ax
   0x7c04 <start+4>:    mov    %ax,%ds
   0x7c06 <start+6>:    mov    %ax,%es
   0x7c08 <start+8>:    mov    %ax,%ss
   0x7c0a <seta20.1>:   in     $0x64,%al
   0x7c0c <seta20.1+2>: test   $0x2,%al
   0x7c0e <seta20.1+4>: jne    0x7c0a <seta20.1>
   0x7c10 <seta20.1+6>: mov    $0xd1,%al
```

与bootasm.S中代码进行比较：

```
# start address should be 0:7c00, in real mode, the beginning address of the
running bootloader
.globl start
start:
.code16                                             # Assemble for 16-bit mode
```

```
    cli                                             # Disable interrupts
    cld                                             # String operations
increment

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number zero
    movw %ax, %ds                                   # -> Data Segment
    movw %ax, %es                                   # -> Extra Segment
    movw %ax, %ss                                   # -> Stack Segment

    # Enable A20:
    #  For backwards compatibility with the earliest PCs, physical
    #  address line 20 is tied low, so that addresses higher than
    #  1MB wrap around to zero by default. This code undoes this.
seta20.1:
    inb $0x64, %al                                  # Wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.1
```

可见代码一致。

练习三：bootloader进入保护模式的过程

查看bootasm.S的代码：

```
#include <asm.h>
# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.set PROT_MODE_CSEG,        0x8                     # kernel code segment
selector
.set PROT_MODE_DSEG,        0x10                    # kernel data segment
selector
.set CR0_PE_ON,             0x1                     # protected mode enable flag

# start address should be 0:7c00, in real mode, the beginning address of the
running bootloader
.globl start

//实模式启动执行,告诉编译器使用16位模式编译,然后关中断,最后将段寄存器清零
start:
.code16                                             # Assemble for 16-bit mode
    cli                                             # Disable interrupts
    cld                                             # String operations increment

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number zero
    movw %ax, %ds                                   # -> Data Segment
    movw %ax, %es                                   # -> Extra Segment
    movw %ax, %ss                                   # -> Stack Segment

    # Enable A20:
    #  For backwards compatibility with the earliest PCs, physical
    #  address line 20 is tied low, so that addresses higher than
    #  1MB wrap around to zero by default. This code undoes this.

//在保护模式下,将A20的置于高电位,使得A20的32条地址线全部可用
seta20.1:
    inb $0x64, %al                                  # Wait for not busy(8042
input buffer empty).
    testb $0x2, %al
```

```
    jnz seta20.1

    movb $0xd1, %al                                 # 0xd1 -> port 0x64
    outb %al, $0x64                                 # 0xd1 means: write data to
8042's P2 port

seta20.2:
    inb $0x64, %al                                  # Wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al                                 # 0xdf -> port 0x60
    outb %al, $0x60                                 # 0xdf = 11011111, means set
P2's A20 bit(the 1 bit) to 1

    # Switch from real to protected mode, using a bootstrap GDT
    # and segment translation that makes virtual addresses
    # identical to physical addresses, so that the
    # effective memory map does not change during the switch.

//加载GDT表
    lgdt gdtdesc

//进入保护模式
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax
    movl %eax, %cr0
```

x86 引入了几个新的控制寄存器(Control Registers) cr0 cr1 … cr7 ，每个长 32 位。这其中的某些寄存器的某些位被用来控制 CPU 的工作模式，其中 cr0 的最低位，就是用来控制 CPU 是否处于保护模式 的。因为控制寄存器不能直接拿来运算，所以需要通过通用寄存器来进 行一次存取，设置 crO 最低位为 1 之后就已经进入保护模式。但是由于 现代 CPU 的一些特性（乱序执行和分支预测等），在转到保护模式之后 CPU 可能仍然在跑着实模式下的代码，这显然会造成一些问题。因此必 须强制 CPU 清空一次缓冲，最有效的方法就是进行一次 long jump 。

bootloader 从实模式进入保护模式的过程:

在开启 A20 之后，加载了 GDT 全局描述符表，它是被静态储存在引导 区中的，载入即可。接着，将 crO 寄存器的 bit O 置为 1，标志着从实 模式转换到保护模式。
由于我们无法直接或间接 mov 一个数据到 cs 寄存器中，而刚刚开启 保护模式时，cs 的影子寄存器还是实模式下的值，所以需要告诉 CPU 加载新的段信息。长跳转可以设置 cs 寄存器，CPU 发现了 crO 寄存器第 O位的值是 1，就会按 GDTR 的指示找到全局描述符表 GDT，然后根据索 引值把新的段描述符信息加载到 cs 影子寄存器，当然前提是进行了一 系列合法的检查。所以使用一个长跳转 limp $PROT_MODE CSEG, $protcseg 以更新 cs 基地址，至此 CPU 真正进入了保护模式，拥有了 32 位的处理能力。
进入保护模式后，设置 ds，es,fs，gs，ss 段寄存器，建立堆栈 (0~Ox7c00)，最后进入 bootmain 函数。

练习四：bootloader加载ELF格式的OS的过程

查看bootmain.c代码：

```
#include <defs.h>
#include <x86.h>
#include <elf.h>

//bootloader读取磁盘扇区的代码
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready//等待磁盘扇区就位
    waitdisk();

//发出读取磁盘的命令,下面指令联合制定了扇区号
    outb(0x1F2, 1);                         // count = 1
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);                      // cmd 0x20 - read sectors

    // wait for disk to be ready
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
}

static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;
    va -= offset % SECTSIZE;
    uint32_t secno = (offset / SECTSIZE) + 1;
    for (; va < end_va; va += SECTSIZE, secno ++) {
        readsect((void *)va, secno);
    }
}


//.判断ELF格式的文件并放入内存
void
bootmain(void) {

//判断是否为ELF格式的文件
    // read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

//判断加载的内存位置
    struct proghdr *ph, *eph;

//加入内存
    // load each program segment (ignores ph flags)
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph ++) {
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
    }

//跳入内核的入口并将控制权全部交给内核
    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

```
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}
```

读取硬盘扇区大致流程如下:
1. 等待磁盘准备好
2. 发出读取扇区的命令
3. 等待磁盘准备好
4. 把磁盘扇区数据读取到指定内存

加载 ELF 格式的 OS 的大致流程为:
1. 读取 ELF 的头部
2. 判断 ELF 文件是否合法
3. 找到 ELF 有关内存位置的描述表
4. 按照这个描述表将数据载入内存
5. 根据 ELF 头部储存的入口信息找到内核的入口并跳转

练习五：实现函数调用堆栈跟踪函数

根据kdebug.c的代码注释完成代码：

```
void
print_stackframe(void) {
     /* LAB1 YOUR CODE : STEP 1 */
     /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
      * (2) call read_eip() to get the value of eip. the type is (uint32_t);
      * (3) from 0 .. STACKFRAME_DEPTH
      *    (3.1) printf value of ebp, eip
      *    (3.2) (uint32_t)calling arguments [0..4] = the contents in address
(unit32_t)ebp +2 [0..4]
      *    (3.3) cprintf("\n");
      *    (3.4) call print_debuginfo(eip-1) to print the C calling function name
and line number, etc.
      *    (3.5) popup a calling stackframe
      *           NOTICE: the calling funciton's return addr eip  = ss:[ebp+4]
      *                   the calling funciton's ebp = ss:[ebp]
      */
    uint32_t ebp = read_ebp(), eip = read_eip();


    int i, j;
    for (i = 0; ebp != 0 && i < STACKFRAME_DEPTH; i ++) {
        cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        uint32_t *args = (uint32_t *)ebp + 2;
        for (j = 0; j < 4; j ++) {
            cprintf("0x%08x ", args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip - 1);
        eip = ((uint32_t *)ebp)[1];
        ebp = ((uint32_t *)ebp)[0];
    }
}
```

练习六：完善中断初始化和处

1.中断描述符表，打开mmu.h代码：

```
/* Gate descriptors for interrupts and traps */
struct gatedesc {
    unsigned gd_off_15_0 : 16;        // low 16 bits of offset in segment
    unsigned gd_ss : 16;            // segment selector
    unsigned gd_args : 5;            // # args, 0 for interrupt/trap gates
    unsigned gd_rsv1 : 3;            // reserved(should be zero I guess)
    unsigned gd_type : 4;            // type(STS_{TG,IG32,TG32})
    unsigned gd_s : 1;                // must be 0 (system)
    unsigned gd_dpl : 2;            // descriptor(meaning new) privilege level
    unsigned gd_p : 1;                // Present
    unsigned gd_off_31_16 : 16;        // high bits of offset in segment
};
```

中断向量表一个表项的大小为16+16+5+3+4+1+2+1+16=64bit，即8字节。其中0-15位和 48-63位分别为偏移量的低16位和高16 位，两者拼接得到段内偏移量，16-31 位gd_ss为段选择器。根据段选择子和 段内偏移地址就可以得出中断处理程序的地址。

2.完善trap.c中的函数idt_init：

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    // set for switch from user to kernel
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
    // load the IDT
    lidt(&idt_pd);
}
```

3.完善trap.c中的中断处理函数trap：

```
static void
trap_dispatch(struct trapframe *tf) {
    char c;

    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event using a
global variable (increase it), such as ticks in kern/driver/clock.c
         * (2) Every TICK_NUM cycle, you can print some info using a funciton,
such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
        ticks ++;
```

```
        if (ticks % TICK_NUM == 0) {
            print_ticks();
        }
        break;
    case IRQ_OFFSET + IRQ_COM1:
        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOU:
        if (tf->tf_cs != USER_CS) {
            switchk2u = *tf;
            switchk2u.tf_cs = USER_CS;
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;

            // set eflags, make sure ucore can use io under user mode.
            // if CPL > IOPL, then cpu will generate a general protection.
            switchk2u.tf_eflags |= FL_IOPL_MASK;

            // set temporary stack
            // then iret will jump to the right stack
            *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
        }
        break;
    case T_SWITCH_TOK:
        if (tf->tf_cs != KERNEL_CS) {
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
            switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct
trapframe) - 8));
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
        }
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    default:
        // in kernel, it must be a mistake
        if ((tf->tf_cs & 3) == 0) {
            print_trapframe(tf);
            panic("unexpected trap in kernel.\n");
        }
    }
}


/* *
* trap - handles or dispatches an exception/interrupt. if and when trap()
returns,
* the code in kern/trap/trapentry.S restores the old CPU state saved in the
* trapframe and then uses the iret instruction to return from the exception.
* */
void
trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
}
```

执行make qemu：

```
Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a6 args:0x00010094 0x00000000 0x00007b38 0x00100092
    kern/debug/kdebug.c:305: print_stackframe+21
ebp:0x00007b18 eip:0x00100c95 args:0x00000000 0x00000000 0x00000000 0x00007b88
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00000000 0x00007b60 0xffff0000 0x00007b64
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x00007b84 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x0010359c 0x00103580 0x0000136a 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00010094
    kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d67 --
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

---

ucore-lab2：

练习一：实现 first-fit 连续物理内存分配算法

1.打开list.h：

```
struct list_entry {
    struct list_entry *prev, *next;
};
//list_entry是双链表

typedef struct list_entry list_entry_t;
//重命名

static inline void list_init(list_entry_t *elm) __attribute__((always_inline));
//初始化
static inline void list_add(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
static inline void list_add_before(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
static inline void list_add_after(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
//添加一个新的项目
static inline void list_del(list_entry_t *listelm)
__attribute__((always_inline));
//从列表中删除一个项目
static inline void list_del_init(list_entry_t *listelm)
__attribute__((always_inline));
//从列表中删除并重新定义
static inline bool list_empty(list_entry_t *list) __attribute__((always_inline));
//判断列表是否为空
```

```
static inline list_entry_t *list_next(list_entry_t *listelm)
__attribute__((always_inline));
//前一项
static inline list_entry_t *list_prev(list_entry_t *listelm)
__attribute__((always_inline));
//后一项


static inline void __list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t
*next) __attribute__((always_inline));
static inline void __list_del(list_entry_t *prev, list_entry_t *next)
__attribute__((always_inline));
```

2.打开memlayout.h可以查看物理页Page结构：

```
/* *
* struct Page - Page descriptor structures. Each Page describes one
* physical page. In kern/mm/pmm.h, you can find lots of useful functions
* that convert Page to other data types, such as phyical address.
* */
struct Page {
    int ref;                        // page frame's reference counter
    uint32_t flags;                 // array of flags that describe the status of
the page frame
    unsigned int property;          // the num of free block, used in first fit
pm manager
    list_entry_t page_link;         // free list link
};
```

3.打开default_pmm.c编写first-fit算法：

1）查看default_init函数：

```
free_area_t free_area;

#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)

static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
```

将free_area.free_list(全局的空闲链表入口)定义为free_list
将free_area.nr_free(全局空闲页数量)定义为nr_free
在default_init中，先将free_list进行init初始化，然后将空闲页数量置为0。

2）查看 default_init_memmap 函数：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    list_add(&free_list, &(base->page_link));
}
```

3）查看default_alloc_pages 函数：

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            list_add(&free_list, &(p->page_link));
    }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

4）查看default_free_pages 函数：

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    list_add(&free_list, &(base->page_link));
}
```

5）修改函数：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);//当为假时中止程序
    struct Page *p = base;//声明一个base页
    for (; p != base + n; p ++) {//初始化
        assert(PageReserved(p));//检查
        p->flags = p->property = 0;//标记，property为0表示连续空页为0
        set_page_ref(p, 0);
    }
    base->property = n;//第一个页表base的property设置为n，表明有n个空闲块
    SetPageProperty(base);//标志位
    nr_free += n;//空闲页数目为n
    list_add_before(&free_list, &(base->page_link));//循环
}

static struct Page
default_alloc_pages(size_t n) {
    assert(n > 0);//n应该大于0
    if (n > nr_free) {
        return NULL;//无法分配
    }
    //有足够的空间分配
    struct Page *page = NULL;
    list_entry_t *le = &free_list;//声明链表头部
    while ((le = list_next(le)) != &free_list) {//遍历整个链表
        struct Page *p = le2page(le, page_link);//找到了合适的空闲空间
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        if (page->property > n) {//把多出的内存加到链表里
            struct Page *p = page + n;
            p->property = page->property - n;
            list_add_after(&free_list, &(p->page_link));
    }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}

static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {//初始化标记和ref
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;//将property改为n
    SetPageProperty(base);
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {检查能否合并
        p = le2page(le, page_link);
        le = list_next(le);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
```

```
    nr_free += n;
    for(le = list_next(&free_list); le != &free_list; le = list_next(le))
    {
    p = le2page(le, page_link);
    if (base + base->property <= p) {
        assert(base + base->property != p);
        break;
    }
}
list_add_before(le, &(base->page_link));
}

static size_t
default_nr_free_pages(void) {
    return nr_free;
}
```

练习二：实现寻找虚拟地址对应的页表项

补全get_pte函数：

```
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
   //pgdir：PDT的内核虚拟地址；la：需要映射的线性地址；creat：决定是否分配页面；
   pde_t *pdep = &pgdir[PDX(la)];//一级页表
    if (*pdep & PTE_P) {//如果存在,返回ptep
        pte_t *ptep = (pte_t *)KADDR(*pdep & ~0x0fff) + PTX(la);
        return ptep;
    }
    //不存在分配page
    struct Page *page;
    if (!create || (page = alloc_page()) == NULL) {//如果不存在，
            return NULL;
    }
    set_page_ref(page, 1);//查找页面，引用次数+1
    uintptr_t pa = page2pa(page);//线性地址
    memset(KADDR(pa), 0, PGSIZE);//转成虚拟地址并初始化为0
    *pdep = pa | PTE_U | PTE_W | PTE_P;//设置
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];//返回ptep
}
```

练习三：释放某虚地址所在的页并取消对应二级页表项的映射

补全page_remove_pte函数：

```
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_P) {//检查page是否存在
        struct Page *page = pte2page(*ptep);
        if (page_ref_dec(page) == 0) {//如果ref为0，则free
            free_page(page);
        }
        *ptep = 0;//清除第二个页表ptep
        tlb_invalidate(pgdir, la);//刷新tlb
    }
}
```

通过运行结果，我们可以看到ucore在显示其entry（入口地址）、etext（代码段截止处地址）、edata（数据段截止处地址）、和end（ucore截止处地址）

的值后，探测出计算机系统中的物理内存的布局（e820map下的显示内容）。接下来ucore会以页为最小分配单位实现一个简单的内存分配管理，完成二级页表的建立，进入分页模式，执行各种我们设置的检查，最后显示ucore建立好的二级页表内容，并在分页模式下响应时钟中断。

- 数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥？

pages的每一项与页表中的页目录项和页表项有对应，pages每一项对应一个物理页的信息。一个页目录项对应一个页表，一个页表项对应一个物理页。假设有N个物理页，pages的长度为N，而页目录项、页表项的前20位对应物理页编号。一个PDE对应1024个PTE，一个PTE对应1024个page。

- 如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事？ 鼓励通过编程来具体完成这个问题

物理地址和虚拟地址之间存在offset：

将KERNBASE改为0即可。

---

ucore-lab3：

练习一：给未被映射的地址映射上物理页

完成do_pgfault函数：

```
int do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    int ret = -E_INVAL;

// 获取pgfault的虚拟地址所在虚拟页
    struct vma_struct *vma = find_vma(mm, addr);
    pgfault_num++;

// 如果当前访问的虚拟地址不在已经分配的虚拟页中
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }

// 检测错误代码
    switch (error_code & 3) {
    default:
        // 写，同时存在物理页，则写时复制

    case 2:
        // 读，同时不存在物理页
        if (!(vma->vm_flags & VM_WRITE)) {
```

```
            cprintf("do_pgfault failed: error code flag = write AND not present,
but the addr's vma cannot write\n");
            goto failed;
        }
        break;

    case 1:
        // 读，同时存在物理页
        cprintf("do_pgfault failed: error code flag = read AND present\n");
        goto failed;

    case 0:
        // 写，同时不存在物理页面
        if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
            cprintf("do_pgfault failed: error code flag = read AND not present,
but the addr's vma cannot read or exec\n");
            goto failed;
        }
    }

// 设置页表条目的权限
    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= PTE_W;
    }
    addr = ROUNDDOWN(addr, PGSIZE);
    ret = -E_NO_MEM;
    pte_t *ptep=NULL;

// 查找当前虚拟地址的页表项
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }

// 如果这个页表项所对应的物理页不存在
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }

// 如果这个页表项所对应的物理页存在，但不在内存中
    else {
        // 如果swap已经初始化
        if(swap_init_ok) {
            struct Page *page=NULL;

            // 将目标数据加载到某块新的物理页中
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            // 将该物理页与对应的虚拟地址关联，同时设置页表
            page_insert(mm->pgdir, page, addr, perm);
            // 当前缺失的页已经加载回内存中，所以设置当前页为可swap
            swap_map_swappable(mm, addr, page, 1);
            page->pra_vaddr = addr;
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
   }
   ret = 0;
failed:
    return ret;
}
```

执行make qemu进行测试：

```
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ----------------------
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
```

- 请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中组成部分对ucore实现页替换算法的潜在用处。

分页机制的实现，确保了虚拟地址和物理地址之间的对应关系，一方面，通过查找虚拟地址是否存在于一二级页表中，可以容易发现该地址是否是合法的，同时可以通过修改映射关系即可实现页替换操作。另一方面，在实现页替换时涉及到换入换出：换入时需要将某个虚拟地址对应的磁盘的一页内容读入到内存中，换出时需要将某个虚拟页的内容写到磁盘中的某个位置。

- 如果ucore的缺页服务例程在执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情？

CPU会把产生异常的线性地址存储在CR2寄存器中，并且把表示页访问异常类型的值（简称页访问异常错误码，errorCode）保存在中断栈中。之后通过上述分析的trap–> trap_dispatch–>pgfault_handler–>do_pgfault调用关系，一步步做出处理。

练习二：基于FIFO的页面替换算法

完成map_swappable函数：

```
static int
_fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page   *page,
int swap_in)
{
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    list_entry_t *entry=&(page->pra_page_link);

    assert(entry != NULL && head != NULL);
    list_add(head, entry);
    return 0;
}
```

完成swap_out_victim函数：

```
static int
_fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)
{
     list_entry_t *head=(list_entry_t*) mm->sm_priv;
         assert(head != NULL);
     assert(in_tick==0);

     list_entry_t *le = head->prev;
     assert(head!=le);
     struct Page *p = le2page(le, pra_page_link);
     list_del(le);
     assert(p !=NULL);
     *ptr_page = p;

     return 0;
}
```

执行make qemu进行测试：

```
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
count is 7, total is 7
check_swap() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
```

1.需要被换出的页的特征是什么？
最早被换入，且最近没有再被访问的页

2.在ucore中如何判断具有这样特征的页？
通过判断是否访问过，对未访问过的物理页FIFO即可

3.何时进行换入和换出操作？
当需要调用的页不在页表中时，并且在页表已满的情况下，会引发PageFault，此时需要进行换入和换出操作

---

ucore-lab4：

练习一：分配并初始化一个进程控制块

alloc_proc函数如下：

```
static struct proc_struct * alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
```

```
    if (proc != NULL) {
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
    }
    return proc;
}
```

整个分配初始化函数的运行过程为：
1. 在堆上分配一块内存空间用来存放进程控制块
2. 初始化进程控制块内的各个参数
3. 返回分配的进程控制块

练习二：为新创建的内核线程分配资源

完成do_fork函数：

```
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;
    // 首先分配一个PCB
    if ((proc = alloc_proc()) == NULL)
        goto fork_out;
    // 设置子进程的父进程
    proc->parent = current;
    // 分配内核栈
    if (setup_kstack(proc) != 0)
        goto bad_fork_cleanup_proc;
    // 将所有虚拟页数据复制过去
    if (copy_mm(clone_flags, proc) != 0)
        goto bad_fork_cleanup_kstack;
    // 复制线程的状态
    copy_thread(proc, stack, tf);
    // 将子进程的PCB添加进hash list或者list
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        list_add(&proc_list, &(proc->list_link));
        nr_process ++;
    }
    local_intr_restore(intr_flag);
    // 设置新的子进程可执行
    wakeup_proc(proc);
    // 返回子进程的pid
    ret = proc->pid;

fork_out:
    return ret;
```

```
bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

请说明ucore是否做到给每个新fork的线程一个唯一的id？请说明你的分析和理由。

在函数get_pid中，如果静态成员last_pid小于next_safe，则当前分配的last_pid一定是安全的，即唯一的PID。
但如果last_pid大于等于next_safe，或者last_pid的值超过MAX_PID，则当前的last_pid就不一定是唯一的PID，此时就需要遍历proc_list，重新对last_pid和next_safe进行设置，为下一次的get_pid调用打下基础。
之所以在该函数中维护一个合法的PID的区间，是为了优化时间效率。如果简单的暴力搜索，则需要搜索大部分PID和所有的线程，这会使该算法的时间消耗很大，因此使用PID区间来优化算法。

get_pid代码如下：

```
// get_pid - alloc a unique pid for process
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;
        goto inside;
    }
    if (last_pid >= next_safe) {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        while ((le = list_next(le)) != list) {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid) {
                if (++ last_pid >= next_safe) {
                    if (last_pid >= MAX_PID)
                        last_pid = 1;
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid)
                next_safe = proc->pid;
        }
    }
    return last_pid;
}
```

练习三：理解 proc_run 函数

pron_run函数：

```
void proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            // 设置当前执行的进程
            current = proc;
            // 设置ring0的内核栈地址
            load_esp0(next->kstack + KSTACKSIZE);
            // 加载页目录表
            lcr3(next->cr3);
            // 切换上下文
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

具体执行过程如下：

1.让 current 指向 next 内核线程 initproc；
2.设置任务状态段 ts 中特权态 0 下的栈顶指针 esp0 为 next 内核线程 initproc 的内核栈的栈顶，即 next->kstack + KSTACKSIZE ；
3.设置 CR3 寄存器的值为 next 内核线程 initproc 的页目录表起始地址 next->cr3，这实际上是完成进程间的页表切换；
4.由 switch_to函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当 switch_to 函数执行完“ret”指令后，就切换到 initproc 执行了。
