Lab 1

知识点

这些是与本实验有关的原理课的知识点：

- 系统启动
- x86的中断和异常机制
- 用中断实现系统调用
- 中断描述符表

此外，本实验还涉及如下知识点：

- 硬盘驱动程序
- ELF文件格式

遗憾的是，如下知识点在原理课中很重要，但本次实验没有很好的对应：

无

练习1

使用`make "V="`命令得到如下生成ucore.img磁盘镜像的过程，具体的命令及它们的输出见`make_cmds.txt`。

1. 编译生成内核代码

命令为`gcc -I... -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -c kern/***.c -o obj/kern/***.o`。

该条命令：

1. `-I...`设定了引用文件查找目录
2. `-fno-builtin -nostdinc`关闭了内建的库
3. `-Wall`开启所有警告
4. `-ggdb -gstabs`添加调试信息
5. `-m32`生成32位代码
6. `-fno-stack-protector`不生成栈保护代码
7. 最后，把`kern`下的.c代码生成到`obj/kern`的.o文件

2. 链接生成内核映像

命令为`ld -m elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/***.o`。

该条命令：

1. `-m elf_i386`生成32位ELF映像
2. `-nostdlib`关闭了内建的库

3. `-T tools/kernel.ld`使用链接器脚本`tools/kernel.ld`，这个脚本描述了代码和数据在内存中的布局，以及设定了内核入口地址
4. `-o bin/kernel obj/***.o`把`obj`下的.o文件链接生成`bin/kernel`文件

3. 编译生成bootloader代码

命令为`gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -c boot/bootasm.S -o obj/boot/bootasm.o`。

命令参数的含义几乎与1.完全类似。

特别地：

- `-Os`表示对生成代码的大小进行优化，开启此选项的目的是满足启动扇区510字节代码的限制，若优化后仍超过大小，就需要手动编写汇编来优化了

4. 链接生成bootloader映像

命令为`ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/***.o -o obj/bootblock.o`。

该条命令：

1. `-m elf_i386`生成32位ELF映像
2. `-nostdlib`关闭了内建的库
3. `-N`设置代码段和数据段都可读可写，关闭动态链接
4. `-e start`指定入口点符号为`start`
5. `-Ttext 0x7C00`设置代码段起始地址为`0x7c00`
6. `obj/boot/***.o -o obj/bootblock.o`把`obj/boot/`下的.o文件链接生成`obj/bootblock.o`文件

5. 生成bootloader二进制代码

命令为`objcopy -S -O binary obj/bootblock.o obj/bootblock.out`。

该条命令：

1. `-S`不拷贝重定位信息和调试信息
2. `-O binary`拷贝二进制代码
3. `obj/bootblock.o obj/bootblock.out`将ELF格式的`obj/bootblock.o`文件中的代码段拷贝到`obj/bootblock.out`

6. 生成启动扇区

为生成启动扇区，先编译生成`tools/sign.c`工具。该工具检查`obj/bootblock.out`文件的大小是否超过510字节，然后利用这个文件生成启动扇区`bin/bootblock`。启动扇区的特点见下。

7. 初始化磁盘镜像文件

命令为`dd if=/dev/zero of=bin/ucore.img count=10000`。

该条命令：

1. `if=/dev/zero`从全零的一个设备文件读取
2. `of=bin/ucore.img`写入到`bin/ucore.img`
3. `count=10000`共10000个扇区（共5120000字节）

8. 将启动扇区写入镜像文件

命令为`dd if=bin/bootblock of=bin/ucore.img conv=notrunc`。

该条命令：

1. `if=bin/bootblock`从6.生成的启动扇区文件读取
2. `of=bin/ucore.img`写入到`bin/ucore.img`
3. `conv=notrunc`不将`bin/ucore.img`文件清空

9. 将内核映像写入镜像文件

命令为`dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc`。

该条命令：

1. `if=bin/kernel`从2.生成的内核映像文件读取
2. `of=bin/ucore.img`写入到`bin/ucore.img`
3. `seek=1`写入时，从第1扇区开始写入
4. `conv=notrunc`不将`bin/ucore.img`文件清空

启动扇区的特点

能够被BIOS识别的启动扇区有如下特点：

1. 符合扇区基本要求，大小512字节
2. 最后两个字节，即第510和511个字节，分别为`0x55 0xAA`；若表示为一个16位的字，则为`0xAA55`（小端序）

若是有分区信息，中间还将有分区表。

练习2

首先，将`gdbinit`脚本中如下两行**去掉**，使得运行`make debug`时不自动开始运行。

```
break kern_init
continue
```

然后运行`make debug`开始实验。

此时，QEMU停在`0xfffffff0`，使用`i r`命令得到，cs=`0xf000`，eip=`0xfff0`。GDB对于段机制的处理不是很好，所以需要手动指定物理地址来查看相应代码。

执行`x/i 0xffffff0`可以得到开机后执行的第一条指令：`ljmp $0x3630, $0xf000e05b`。指令很奇怪，这是由于没有设置指令集为16位，GDB默认是32位，导致反汇编错误，使用`set arch i8086`设置之后重新`x/i 0xffffff0`，得到正确结果：`ljmp $0xf000, $0xe05b`。

执行几次`si`，查看指令的执行情况，发现果真跳转到`0xfe05b`开始执行了。

执行`b *0x7c00`在`0x7c00`处设下断点，执行`c`使得QEMU继续运行，之后发现在`0x7c00`暂停了，断点有效。

执行`x/10i 0x7c00`查看反汇编，发现与`bootasm.S`一致，但与`bootblock.asm`不太一致。这是由于反汇编生成`bootblock.asm`时，使用的是32位指令集。

现在开始调试设置A20的代码。

执行`b *0x7c0a`在`0x7c0a`设下断点，然后使用`si`单步执行，期间用`layout`切换格局，可以同时查看汇编代码以及寄存器的值。

练习3

系统加电时，处理器处于实模式状态，为了让它进入保护模式使用完整的32位地址线以及实现更多的保护功能，需要进行一些准备，然后将CR0中PE位置位，最后处理器开始执行保护模式下32位代码。

详细操作描述如下：

1. 开启第20位地址线（A20）

由于历史原因，进入保护模式之前，需要开启第20位地址线（下面简称A20）来获得使用完整4G内存的能力。

`bootasm.S`中开始A20的方法是：

1. 等待直到8042不忙
2. 向8042控制端口写入写P2端口命令（`0xd1`）
3. 等待直到8042不忙
4. 向8042数据端口写入`0xdf`来打开A20

值得注意，随着时代的发展，开启A20的方法千变万化，这只是其中的一种。

2. 配置全局描述符表（GDT）

由于本实验弱化对于分段机制的使用，GDT及其基址和界限的描述可以硬编码在代码中。GDT中有三个描述符：空描述符、代码段描述符以及数据段描述符。然后，用`lgdt`载入GDT的基址和界限即可。注意，GDT需要4字节对齐。

3. 开启保护模式

将CR0的PE位（第0位）置1即可开启保护模式。注意，置1后，处理器并未立即开始执行32位代码。

4. 开始执行32位代码

使用远跳转指令`ljmp`，将代码段寄存器`cs`设置为新的值（代码段选择子），从而真正开始32位保护模式。

由于2.配置GDT时将代码段设置为恒等映射，同时未开启分页机制，逻辑地址和物理地址直接相等，跳转目标就是下一条指令的地址。

注意跳转后还应该将各个数据段寄存器也设置为新的值（数据段选择子）。

练习4

为了能够将内核映像加载到物理内存，首先需要读取扇区。

`bootmain.c`通过`inb`、`insl`、`outb`指令来和磁盘控制器进行通讯，使用最朴素的方法读取扇区（没有DMA或中断）。读取一个扇区分为如下几步：

1. 等待磁盘控制器就绪
2. 用`outb`指令设置要读取的数量、读取的LBA（这里使用LBA编址方式）
3. 用`outb`指令向控制端口写入读取扇区命令
4. 用`repne insl`指令读取相应数量的双字（128个双字即为512字节），由于有`repne`前缀，这一条指令实际上是一个循环

有了读扇区的基础，可以开始读取并解析ELF格式的内核映像。

ELF格式的内核映像有ELF头结构，里面标识了程序节的个数，每个程序节说明了该节在映像中的偏移、长度以及应加载到内存的位置，据此读取每一个程序节并存入正确的地址即可。

最后，ELF头结构里面有内核的入口点地址，是一个整数，可以通过将此整数转换为函数指针然后调用来跳转到内核。

练习5

这个练习提供的注释已经写得非常详细，根据它写出C代码即可。本练习的关键点在于C语言调用约定和栈。

某函数被调用后，初始化栈帧完毕之后的栈局势如下图：

```
|高地址方向|
-----------
| 第四参数 | ebp + 5 dword
-----------
| 第三参数 | ebp + 4 dword
-----------
| 第二参数 | ebp + 3 dword
-----------
| 第一参数 | ebp + 2 dword
-----------
| 返回地址 | ebp + 1 dword
-----------
|  旧ebp  | <==== ebp, esp + 0
-----------
|低地址方向|
```

此外，值得讨论的是，给`print_debuginfo`传参数的时候需要传入eip-1的值，这是因为eip指向了下一条指令，而程序希望处理当前指令。还需注意的是，`read_ebp`定义为必须内联，`read_eip`定义为不得内联，这都是为了能够获得ebp和eip的真实的值。

测试输出有意义的最后一行为：

```
ebp:0x00007bf8 eip:0x00007d6e args:0xc031fcfa 0xc08ed88e 0x64e4d08e
0xfa7502a8
```

这是第一个被调用的函数的信息，查看`bootasm.S`的第68~71行后发现实际上是调用了`bootmain`。

`0x00007bf8`是bootmain的栈帧基址；

`0x00007d6e`是这条语句后一条语句的起始地址，也是`bad`标签的地址：

```
((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

`args`本应为`bootmain`的前4个参数，而`bootmain`根本没有参数，所以这4个32位整数就是调用`bootmain`**之前**，栈顶的4个32位整数。注意到调用`bootmain`之前，esp被赋值为`0x7c00`，于是`0xc031fcfa`、`0xc08ed88e`、`0x64e4d08e`、`0xfa7502a8`实际上为位于`0x7c00~0x7c0f`的代码：

```
0xfa 0xfc 0x31 0xc0
0x8e 0xd8 0x8e 0xc0
0x8e 0xd0 0xe4 0x64
0xa8 0x02 0x75 0xfa
```

查看`bootblock.asm`验证了这一点。

经过与参考答案对比，我在ebp为0时没有停止循环，应修复。

练习6

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。

中断描述符表一个表项为8字节，把它看成一个64位整数。中断服务程序入口点的（代码）段选择子在31-16位。由于历史原因，中断服务程序入口点的段偏移量被拆散到两个不连续的地方：偏移量的15-0位在15-0位，31-16位在63-48位。

注意到`trap.h`中定义了`T_SYSCALL`以及`T_SWITCH_TOK`，它们是由用户态调用的软中断号，所以dpl应设置为3。

此外，我查阅了某文献得知`T_SYSCALL`还应设置`istrap`为1，否则在执行系统调用时无法及时响应中断，可能是个问题。

经过与参考答案对比：

1. 对于循环次数，我硬编码为256，参考答案的做法可以编译时计算，更优雅
2. 我直接使用内联汇编`lidt`，而答案使用封装好的`lidt`内联函数

3. 参考答案没有考虑`T_SYSCALL`

扩展练习1

内核态到用户态

这个子任务实现的重点在于给`iret`指令提供ss和esp。首先是存放地点的选择，我的实现没有对产生中断之前栈的情况做过多假设，所以不能用之前的栈空间，即不能直接修改tf指向的内容的ss和esp，因为此中断是在内核态的时候产生，处理器没有压入ss和esp（实际上，此时tf的ss和esp的位置分别对应`lab1_switch_to_user`的返回地址以及旧ebp）。我的办法是在当前栈上开设一个新的`trapframe`结构体（利用C语言局部变量即可），将旧的内容复制进去，然后进行如下设置。

需要设置各个段寄存器：cs设置为用户代码选择子，其他数据段寄存器设置为用户数据选择子。

需要注意的是esp的设置，esp需要设置为中断返回之后，`lab1_switch_to_user`函数能够正常返回。

为了用户能够使用`cprintf`进行输出，需要设置IOPL为用户态，修改tf的eflags就能够在中断返回的时候设置IOPL。

最后使用新的`trapframe`结构体进行中断返回。

用户态到内核态

由于中断在用户态触发时处理器会将栈切换到内核的另外一个栈，处理代码可以直接对原来的用户栈进行修改。在原来的用户栈上开设一个新的`trapframe`结构体，将旧的内容复制进去，然后进行和上面类似的设置。

注意不需要复制或设置ss和esp。同时，由于我的实现在中断返回之后不需要额外操作，不能为ss和esp预留空间，否则中断返回之后还需要恢复栈（手动弹出ss和esp）。

最后使用新的`trapframe`结构体进行中断返回。

经过与参考答案对比，我的实现没有对产生中断之前栈的情况做过多假设，在中断返回之后也不需要额外操作，所以`lab1_switch_to_user`以及`lab1_switch_to_kernel`的实现都只需要内联一条软中断指令`int`。这对扩展练习2的实现也有很大帮助。

扩展练习2

键盘中断是异步中断，由于我在实现扩展练习1时没有对产生中断之前栈的情况做过多假设，在中断返回之后也不需要额外操作，所以对于这个扩展练习，只需要在键盘中断处理程序中检测到0或3被按下时跳转到扩展练习1相应代码处执行即可。

Lab 2

知识点

这些是与本实验有关的原理课的知识点：

- 首次匹配
- x86分页机制
- 二级页表

此外，本实验还涉及如下知识点：

- 内核实现的通用链表

遗憾的是，如下知识点在原理课中很重要，但本次实验没有很好的对应：

- 最佳匹配
- 最差匹配
- 碎片整理技术
- 反置页表

练习1

实验代码已经提供了某种连续内存分配算法，为实现首次匹配，需要修改代码，维护`free_list`有如下性质：

1. 链表中的页面块均为空闲
2. 每个空闲页面块首个页面挂在链表中，对应的`Page`结构体有属性，为页面块的页面数；该页面块其余页面不直接挂在链表中，也没有属性，属性值的位置清零
3. 链表中页面块按照首地址增序排列

对于`default_alloc_pages`函数的修改

需要修改`default_alloc_pages`函数来维护上面的性质，但改动不大。

主要修改就是将新分裂出来的页面块（若有）设置好属性并插入正确的位置，而不是直接插入到`free_list`最后。

正确的位置实际上就是原页面块原来位置的后一项。实现上，可以先将新分裂出的页面块插入，再将分配出去的页面块移出。

对于`default_free_pages`函数的修改

还需要修改`default_free_pages`函数来维护上述性质，但改动不大。

在重新设置页面属性、合并相邻页面后，需要将该页面插入到链表正确的位置中，也就是将原来的`list_add(&free_list, &(base->page_link));`替换为一个循环，根据页面首地址找到正确的位置然后插入。

改进

这个分配算法还可能继续优化，优化可能有如下几点：

1. 释放时的合并操作，在查找与该空闲块相邻的下一个块时不需要循环遍历查找，只需要根据当前块的页面数计算出来即可，接着判断下一个页面块是否空闲，然后尝试合并
2. 释放时的合并操作，若是在所有页面块最后一个页面处也记录这个页面块的页面数，则在查找与该新释放出的空闲块相邻的上一个块时也不需要循环遍历查找，只需要根据上一块的页面数计算

出来即可，接着判断是否空闲，然后尝试合并
3. 在分配时考虑缓存的影响，尽量分配在缓存中的热页面

~~与参考答案对比，我认为我的实现对原来实验代码的修改较小，思路清晰，实现简洁。~~

经过修改参考答案，我的实现和参考答案一致了。

练习2

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。

这个练习主要是页目录项的填写，页目录项的字段见下。特别地，其中P、A、D位对于实现页替换算法有很大用处，W位对于实现写时复制有很大用处。

```
 31                12 11 9 8 7 6 5 4 3 2 1 0
+--------------------+----+-+-+-+-+-+-+-+-+-+
|    页表基址高20位    |忽略|G|S|0|A|C|T|U|W|P|
+--------------------+----+-+-+-+-+-+-+-+-+-+
```

说明：

- 页表基址：页表的**物理地址**，需要4KB对齐，所以只需填写高20位
- 忽略：内核可以记录自己需要的信息
- G：对于页目录，这一项不用
- S：页大小，1为4MB，0为4KB；ucore置为0
- A：在上次清零之后，该页是否被访问（读或者写）过，可用于页替换算法的实现
- C：置1则不缓存，否则缓存
- T：置1则缓存写穿，否则写回
- U：置0则页表中任何页面只能内核态访问，否则用户态可能可以访问，还取决于页表项的设置
- W：页面是否可写，但若CR0的WP位置为0，内核总可以写；写时复制的实现需要用到页表中的这一位
- P：页面是否存在于内存中

页表项的字段和页目录项十分类似：

```
 31                12 11 9 8 7 6 5 4 3 2 1 0
+--------------------+----+-+-+-+-+-+-+-+-+-+
|    页基地址高20位    |忽略|G|0|D|A|C|T|U|W|P|
+--------------------+----+-+-+-+-+-+-+-+-+-+
```

说明：

- 页基地址：页面的**物理地址**，需要4KB对齐，所以只需填写高20位
- 忽略：内核可以记录自己需要的信息
- G：这一页表项是否是全局项，若被置位，在CR3改变时，相应TLB不被清除，对于内核的页面映射，可以设置此位来优化内核访问内存的性能
- D：在上次清零之后，该页是否被写过，可用于页替换算法的实现
- U：置0则该页面只能内核态访问，否则用户态可以访问
- P：页面是否存在于内存中

- 其余位和页目录项含义相同

另外，当P位清零时，其余位都可供内核自由使用。这可以用来存放交换分区的一些信息，可用于页替换算法的实现。

如果ucore执行过程中访问内存，出现了页错误异常，处理器会将异常的线性地址保存在CR2寄存器中，然后查询IDT找到中断服务程序入口点。由于当前已在内核态，不涉及特权级的变化，处理器还会向当前栈中压入cs、eip和错误码。错误码记录了异常的一些标志，比如是读还是写操作触发了异常，是非法访问还是缺页触发了异常，这对于页替换算法和写时复制的实现都有用。最后，跳转至中断服务程序。这些事情中间还会做权限检查，不通过还会触发保护错误。这些事情执行的顺序可能与处理器实现有关。

与参考答案对比，我没有判断`alloc_page`失败的情况，应修复。

练习3

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。主要实现的就是页目录项的清除和页面引用计数的维护。另外，还需要清除TLB。

无论是页目录中的页目录项还是页表中的页表项，它们的高20位都是物理页号，以此作为索引，可以在`Page`结构体数组中找到对应物理页的元数据，正如`pte2page`宏实现的那样。反之，可以通过`Page`结构体的指针算出其在数组中的下标，然后算出物理页号，正如`page2pa`宏实现的那样，进而用来填写页目录项或者页表项。

若要实现虚拟地址与物理地址相等，需要将`KERNBASE`改为`0x00000000`，同时需要做一些其他辅助性的修改:

1. 链接脚本`kernel.ld`需要修改链接的虚拟地址为`0x00100000`
2. 清除页目录项`boot_pgdir[0] = 0;`赋值前需要检查`KERNBASE`，若为0则不清除
3. 断言`assert(boot_pgdir[0] == 0);`也需要检查`KERNBASE`，若为0则不执行断言
4. 其他测试代码使用的地址区域应避开`0x00000000 ~ 0x38000000`
5. 测试脚本`grade.sh`中的测例也需要相应修改

具体修改见GitHub lab2-zero分支。

与参考答案对比，我在减少和判断页面引用计数时将两个操作分开进行了，在某些情况下可能会有重复释放页面的问题，应修复。

Lab 3

知识点

这些是与本实验有关的原理课的知识点：

- 先进先出替换算法：原理课注重原理，本实验注重实现
- 扩展时钟替换算法：原理课注重原理，本实验注重实现

此外，本实验还涉及如下知识点：

无

遗憾的是，如下知识点在原理课中很重要，但本次实验没有很好的对应：

- 覆盖技术
- 其他局部页替换算法
- 全局页替换算法
- Belady现象
- 抖动和负载控制

练习1

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。

主要就是在页错误异常处理程序中，当`mm`结构体拥有此虚拟地址但页表中还没有映射时，调用`pgdir_alloc_page`来分配并映射一个物理页面。这其实是按需分配的实现。

页目录项和页表项的字段在Lab 2实验报告中已做说明，这里重点说明对实现页替换算法有重要作用的字段。

主要使用页表项的字段。页表项的字段和页目录项十分类似：

```
 31                12 11 9 8 7 6 5 4 3 2 1 0
+--------------------+----+-+-+-+-+-+-+-+-+-+
|    页基地址高20位    |忽略|G|0|D|A|C|T|U|W|P|
+--------------------+----+-+-+-+-+-+-+-+-+-+
```

与页替换算法有关的字段有：

- A：在上次清零之后，该页是否被访问（读或者写）过，可用于时钟替换算法和扩展时钟替换算法的实现
- D：在上次清零之后，该页是否被写过，可用于扩展时钟替换算法的实现
- P：页面是否存在于内存中，当此位清零时，其余位都可供内核自由使用。这可以用来存放交换分区的一些信息，可用于各种页替换算法的实现。

如果ucore的页错误处理程序执行过程中访问内存，又出现了页错误异常，由于这个异常不可以屏蔽，处理器会将异常的线性地址保存在CR2寄存器中，然后查询IDT找到中断服务程序入口点。由于当前已在内核态，不涉及特权级的变化，处理器还会向当前栈中压入cs、eip和错误码。错误码记录了异常的一些标志，比如是读还是写操作触发了异常，是非法访问还是缺页触发了异常，这对于页替换算法和写时复制的实现都有用。最后，跳转至中断服务程序。这些事情中间还会做权限检查，不通过还会触发保护错误。这些事情执行的顺序可能与处理器实现有关。

与参考答案对比，我缺少了对于出错情况检查的代码，应修复。

练习2

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。

在成功分配一个物理页面的时候将此页面插入物理页面链表尾；当需要选择一个页面被换出时，选择链表头的页面返回。这就实现了先进先出替换算法。

此外，若某页面已在页表中映射（页表项为非零值）但还是产生了缺页异常，说明此页面被换出了。此时，缺页异常处理程序需要选择一个页面换出，然后将触发缺页异常的正在被访问的页面换入并修改页表映射。还要设置好`Page`结构体的`pra_vaddr`，此字段说明了该物理页面对应的虚拟地址，在换出时用于计算该页面应写入交换分区的位置。

与参考答案对比，我缺少了对于出错情况检查的代码，应修复。

扩展时钟替换算法设计与实现见下。

扩展练习 - 扩展时钟替换算法

这个扩展练习的实现在GitHub lab3-challenge分支中，文件主要为`swap_extclk_dirty.c`。

算法的实现完全按照MOOC中对于扩展时钟替换算法的讲解，不过有以下实现上的细节。

我的设计中，这个替换算法在运行时有两个状态：启动状态和工作状态。算法主要维护两个数据：物理页面链表以及扩展时钟替换算法的时钟指针。

在初始化之后，算法处于启动状态，这时候算法对于系统可用的物理页面还不了解，当`map_swappable`被调用，且物理页面还不在物理页面链表中，说明有一个新的物理页面被分配，算法将其放入物理页面链表中。

`swap_out_victim`被第一次调用时，说明系统物理内存均已分配完毕，也说明物理页面链表中已包含了全部的物理页面，可以转入工作状态真正开始运行替换算法，选择页面换出了。

在选择页面换出时，算法以时钟指针为起始，不断遍历这个物理页面链表，到链表尾部则回到头部继续遍历。物理页面链表其实就是一个循环链表，方便了实现。对于每一个被遍历的物理页面，考察它对应的虚拟地址的页表项中AD两个位（AD两位说明见上）：

1. 若A位为0，D位也为0，则选择这个页面被换出，将时钟指针指向链表下一项，本次算法结束
2. 若A位为0，D位为1，则将此页面写入交换分区，写入成功后清除D位，并刷新TLB
3. 若A位为1，则将其清零并刷新TLB，不考虑D位的情况

刷新TLB是为了让页面再次被访问或被修改时，AD位能够再次被处理器置位。若不清除，处理器根据TLB的缓存会认为AD位已经被置位，而不修改内存中的值，算法在下次读取时将读到错误的值。这确实会导致性能的损失，但确保了算法和MOOC的讲解一致。

为了减少写交换分区的次数，优化性能，还需修改`swap_out`函数，在判断D位已被清零的情况下不再写入交换分区。

经测试，我的实现用MOOC上的测例运行（见`check_swap`），缺页次数、缺页发生的时刻以及每次缺页时算法执行情况都和MOOC上一致，说明实现大致正确。

同时，我也实现了不考虑D位的时钟替换算法，文件主要为`swap_extclk.c`。与考虑D位的区别在于，对于每一个被遍历的物理页面，只考察它对应的虚拟地址的页表项中A位（A位说明见上）：

1. 若A位为0，则选择这个页面被换出，将时钟指针指向链表下一项，本次算法结束
2. 若A位为1，则将其清零并刷新TLB

总之，算法在AD位都为0（对于不考虑D位的算法，则A位为0就够了）时会真正选择页面换出，通过查询页表可以得到AD位的具体信息，在发生页错误异常并且权限检查通过后会执行页面替换算法来进行换入和换出操作。

与参考答案对比，我认为我的方法思路清晰、实现简洁。

Lab 4

知识点

这些是与本实验有关的原理课的知识点：

- 进程状态模型
- 内核线程的控制

此外，本实验还涉及如下知识点：

- `proc`结构体及其`context`结构体的理解

遗憾的是，如下知识点在原理课中很重要，但本次实验没有很好的对应：

- 用户进程
- 挂起的进程

练习1

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。

要实现的就是`alloc_proc`，分配一块小内存作为进程控制块，然后对其初始化。初始化操作主要是将其他字段（使用`memset`）清零，然后对特定字段，如`pid`、`cr3`等进行初始化。由于本实验是内核线程，`cr3`可以初始化为当前`cr3`（利用`rcr3`）。

与参考答案对比，我的实现使用`memset`，而参考答案对每一个字段逐一初始化。参考答案的语义更清晰，而我的性能可能会稍好一些。另外，我没有初始化`pid`为-1，在某些情况下可能出问题，应修复。此外，参考答案直接使用`boot_cr3`而不需要利用`rcr3`。

对于`trapframe`结构体和`context`结构体作用的理解

首先，对于每一个任务（进程或者线程）而言，保存了两组寄存器上下文信息。

一个是该任务自身控制流的上下文，一般是在该任务通过某种方式（比如中断或者系统调用）主动或被动进入内核时保存的，存于`trapframe`结构体中，存于该任务的内核栈中，此任务的控制块存放一个指向它的指针。若任务是内核线程，任务的栈与其内核栈是一个，但这不是问题。

另外一个是该任务进入内核后，对应的内核控制流的上下文，存于`context`结构体中，嵌在任务的控制块中。

值得注意的是，为统一起见，内核线程也包含有这两个部分。

这两个结构体的使用的例子请见练习3，是关于进程切换的。

练习2

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可，主要实现`do_fork`。

对于`pid`的分配，内核在`do_fork`中通过调用`get_pid`分配一个新的`pid`然后赋值给新分配的进程。下面分析`get_pid`的算法。

算法主要使用两个变量：`next_safe`以及`last_pid`。注意到它们被声明为`static`，效果类似于全局变量。首先，算法初始化`next_safe`以及`last_pid`均为`MAX_PID`，这是合法`pid`的上界加一。

每次分配，算法首先将`last_pid`增加1（特别地，若达到或超过`MAX_PID`则回到1，并认为此时`last_pid`超过了`next_safe`），此时若是仍然小于当前`next_safe`，则立即分配当前`last_pid`，本次算法结束。否则，此时说明`last_pid`已到达甚至超过`next_safe`，需要重新计算`next_safe`了。

为了接下来描述方便，假设所有未被分配的`pid`组成一段一段区间，叫做空闲`pid`区间。

计算`next_safe`的主要过程是一个循环，循环内有两个操作"同时"进行：

1. 寻找`last_pid`的下一个合法值，即大于`last_pid`的最近的空闲区间的下界，存于`last_pid`。注意，若`last_pid`被更新，`next_safe`需要重新开始计算。
2. 寻找当前`last_pid`所在空闲`pid`区间的上界加一，即已被分配的`pid`中大于`last_pid`的最小的那一个，存于`next_safe`。

总之，`last_pid`维护了最新分配出去的`pid`，而`next_safe`维护了`last_pid`所在空闲`pid`区间的上界加一。

这样就保证了每一次`get_pid`分配出来的`pid`没有其他进程正在使用，而且分配效率很高。从这里也看出，需要先分配`pid`，然后再将进程插入链表，否则算法会混乱。

与参考答案对比，由于在分配`pid`、将进程插入链表和散列表时涉及到对共享变量的操作，参考答案用关中断的方法把共享变量保护了起来，而我没有这么做，应修复。另外，对于多核处理器，应该使用其他手段，因为关中断不能确保其他处理器的互斥访问了。

练习3

假设系统初始化已经完成，任务都已经就绪。当一个时钟中断来临，当前任务的状态被保存到`trapframe`结构体，中断服务程序中可能会调用`schedule`选择下一个占用处理器的任务，然后在`schedule`中调用`proc_run`来切换到下一个任务。本实验没有在时钟中断服务程序中进行调度，而是在`idle`内核线程中循环调用`schedule`，原理是一致的。`proc_run`首先设置当前任务指针为下一个任务，然后加载相应的内核栈、页表，最后调用`switch_to`。可以看出，实际上切换上下文的是`switch_to`。`switch_to`完成了保存当前所有寄存器，并加载下一个任务的寄存器的动作，最后跳转到下一个任务的断点。由于这是一个返回值为`void`的函数，所以不保存和加载`eax`也没有问题。事实上，根据调用约定，只需保存和加载应该由被调用者保存的寄存器。

这样的效果就是，对于同一个任务而言，从被调度走到被调度回来，它感觉到的只是发生了中断或系统调用，中间调用了`schedule`以及`switch_to`，然后奇迹般地一层层返回了（由于我们破坏了`eax`，返回值是不确定的），最后从中断返回到断点，就像发生了一个普通的中断。它对其余任务的存在毫不知情。

于是，创建一个新的任务也很简单，只需要在它的内核栈上构造出它刚刚被中断的场景，即`trapframe`结构体，以及在任务控制块中构造出它的内核控制流的上下文，也就是`context`结构体，在下一次`proc_run`函数选中这个任务的时候，这个任务就开始执行了。`trapframe`结构体中`eip`只需设置为任务的入口点，这样，从中断返回之后便会开始执行任务；而`context`结构体中`eip`的设置需要稍作考虑，应该设置为一段能够实现从中断返回任务控制流的代码的地址，这里设置为`forkret`的入口。

对于本次实验而言，内核创建了两个内核线程：`idle`以及`init`。

`idle`是一个循环，不断释放自己的时间片，内核在没有其他进程可以运行的情况下，会选择运行此进程。

本实验的`init`进程只是打印字符串然后退出，之后的`init`进程还会进行更多的动作。

`local_intr_save(intr_flag)`是关闭中断，并将原来的中断使能状态保存到变量`intr_flag`中。

`local_intr_restore(intr_flag)`则是根据变量`intr_flag`恢复中断使能。

这两条语句中间的部分不会发生中断，若是单处理器系统，就构成了临界区。因此，这两条语句保护了切换任务上下文的过程不被中断破坏。上面提到，对于多核处理器，应该使用其他手段，因为关中断不能确保其他处理器的互斥访问了。

Lab 5

知识点

这些是与本实验有关的原理课的知识点：

- 进程状态模型
- 用户进程

此外，本实验还涉及如下知识点：

- ELF文件格式
- 写时复制

遗憾的是，如下知识点在原理课中很重要，但本次实验没有很好的对应：

- 挂起的进程

练习0

首先，需要对之前的代码进行微小的修改：

1. `alloc_proc`加入对新增字段的初始化，由于我使用`memset`初始化，不需要修改
2. `do_fork`时设置对父进程的指针
3. `do_fork`时将进程插入进程链表、增加进程计数器的操作，替换为`set_links`函数，该函数不仅有上面两个操作，还设置了进程间的关系指针
4. `ticks % TICK_NUM == 0`时，将打印提示信息替换为将当前进程的`need_resched`置1，表明当前进程时间片已耗尽，需要运行调度器

5. `idt_init`中我已在Lab 1已经设置了系统调用的门，这个实验不需要修改了

练习1

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。主要就是修改`load_icode`，补上对`trapframe`结构体的初始化。初始化的核心是`eip`以及`esp`。`eip`需要设为ELF映像中定义的入口点，`esp`则设为用户栈顶，由于开启了分页，它只需要设置为一个常数，为用户栈顶的虚拟地址`USTACKTOP`。

当创建一个用户态进程并加载了应用程序，并且内核调度器选择此进程开始执行，过程为：

1. `proc_run`被调用，`current`被置为此进程控制块的指针，标志着此进程成为运行状态
2. 加载该进程的内核栈和页表
3. `switch_to`被调用，由于此进程的`context`的`eip`被置为`forkret`，`switch_to`最终跳转到`forkret`
4. `forkret`进而跳转到`__trapret`，`__trapret`开始的代码是进入用户态的关键
5. `__trapret`首先根据当前进程的`trapframe`结构体“恢复”（加载）内核新构造的中断上下文，最后使用`iret`指令同时完成跳转到用户进程入口点、切换到用户栈以及切换到用户态的操作

具体而言，处理器在执行`iret`指令时，当前栈的布局是这样的：

```
|  High Address  |
------------------
|   SS (ring3)   |
------------------
|       ESP      |
------------------
|     EFLAGS     |
------------------
|   CS (ring3)   |
------------------
|       EIP      |  <---- ESP, CS=ring0
------------------
|  Low  Address  |
```

当处理器发现栈中的`CS`为ring3，它在弹出`EFLAGS`之后还会弹出`SS`和`ESP`，并将它们赋给相应寄存器。

至此，处理器的`iret`指令就同时完成了跳转到用户进程入口点、切换到用户栈以及切换到用户态的操作。

我的实现和参考答案十分一致。

练习2

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。需要补充的地方是`copy_range`函数中拷贝内存和将页面映射信息填入页表的实现。

值得说明的是，我最初实现时手动填写了页表项，而没有调用`page_insert`，可能会造成引用计数维护错误，后来自己发现并修复了。

我的实现和参考答案十分一致。

写时复制的设计与实现请见下面的小节。

练习3

fork

`fork`创建了一个新的进程控制块，此时进程为UNINIT，然后将当前进程几乎所有信息复制到了新的进程。没有复制的信息包括内核栈和用于存放返回值的`eax`寄存器，它们都是新的。

在`wakeup_proc`被调用之后，进程状态变为RUNNABLE的就绪态。

exec

`exec`先将当前进程的内存布局（`mm`结构体）清除干净，然后调用`load_icode`填写目标ELF映像中说明的内存布局。注意，进程的状态不发生变化。

wait

这个系统调用涉及两个进程，当前进程及其子进程。`wait`首先检查当前进程是否有子进程，若无则直接返回错误。接着，检查是否有ZOMBIE状态的子进程，若有则直接释放子进程残存的资源，然后立即返回子进程的返回状态码，此时子进程已不存在。

若没有ZOMBIE状态的子进程，此进程变为SLEEPING状态，等待这样的子进程出现，再进行上面的操作。

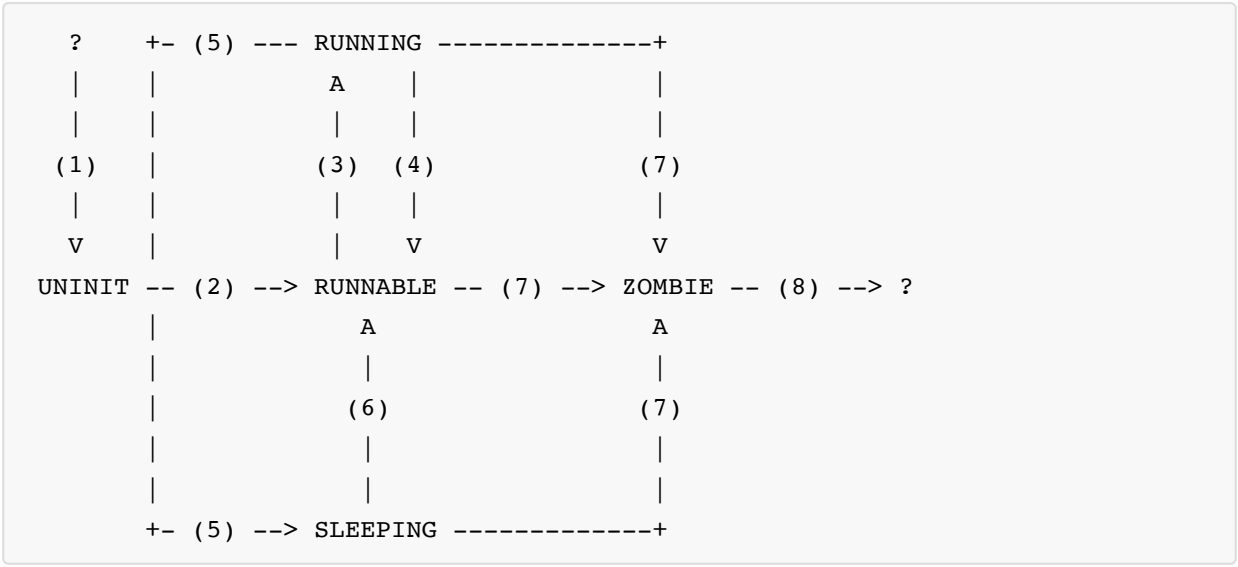
kill

这里的`kill`实现不是发信号，而是将指定进程的标记为设为EXITING，在下一次中断来临时，让进程调用`exit`来自己杀死自己，从而变成ZOMBIE状态。注意，`kill`并不直接改变进程的状态。

exit

这个系统调用涉及两个进程，当前进程及其父进程。`exit`会清除当前进程几乎所有资源，除了进程控制块以及内核栈，接着将其所有子进程的父进程置为init进程，然后将当前进程的状态变为ZOMBIE。若该进程有父进程，并且父进程正在等待子进程退出，则将父进程的状态从SLEEPING变为RUNNABLE态。

总结成状态转移图如图所示：



(1) `alloc_proc`

(2) `wakeup_proc`

(3) `proc_run`被调用时，作为`proc_run`的参数

(4) `proc_run`被调用时，作为`proc_run`的调用者

(5) 调用了`wait`（或`sleep`，上面没有涉及）

(6) 子进程调用了`exit`（或定时器超时，上面没有涉及）

(7) 调用了`exit`

(8) 父进程调用了`wait`

扩展练习 - 写时复制的设计与实现

我的写时复制实现主要修改两个函数。具体实现请见GitHub lab5-challenge分支。

对于`copy_range`，我将函数`share`参数的语义定义为是否用写时复制的方式实现内存的“复制”。为此，该函数需要在申请页面之前进行判断，若是开启写时复制，则直接使用已存在的页面；在把原来的内容复制到新页面的时候也需要进行判断，若是开启写时复制，就不需要复制了。然后，同样用`page_insert`完成页表映射和引用计数增加1的操作，但对于写时复制，还需要设置可写位（W位）为0。此外，对于写时复制，还需要将原来页面页表项的可写位（W位）清零并刷新TLB，因为这个页面已经变成共享的页面了。对于`dup_mmap`，需要将传入给`copy_range`的`share`参数变为1（代表开启写时复制）。

此时，对于共享页面的写操作会导致页错误异常，为此需要进一步修改页错误异常处理程序`do_pgfault`来最终实现写时复制。

`do_pgfault`已有的代码已经对页面访问权限的情况做了很详细的检查，若能执行到并且通过`(*ptep & PTE_P)`的判断，说明该进程有权限对该页面进行操作，而该页面又确实真正存在于物理内存中，但此时还是发生了页错误异常，表明页表中存储的权限和该进程实际拥有的权限不一致。这个不一致性告诉内核，这是一个需要写时复制的页面，它可能正在和其它进程共享（如果页面引用计数大于1）。此时，内核才需要真正分配一个新的页面，复制数据，然后修改这个进程的页表项，重新映射对应的虚拟地址，并维护页面的引用计数。这次映射的时候，可写位（W位）就可以设置为1

了。

考虑出错页面的引用计数恰好等于1的特殊情况。这个情况说明该页面曾经被共享过，但其他进程又对这个页面进行了写操作，分裂出去了。进一步，这说明该页面已经没有和其他进程共享，只需要修改页表项将此虚拟地址对应的可写位（W位）置1并刷新TLB即可。

至此，写时复制的实现就比较完整了。

另外，我的实现没有考虑挂起的情况。

Lab 6

知识点

这些是与本实验有关的原理课的知识点：

- 时间片轮转算法：不过原理课没有考虑实现上的细节，实验需要考虑

此外，本实验还涉及如下知识点：

- 调度器框架
- Stride Scheduling调度算法

遗憾的是，如下知识点在原理课中很重要，但本次实验没有很好的对应：

- 其他调度算法
- 实时调度
- 多处理器调度
- 优先级反置问题解决

练习0

首先，需要对之前的代码进行微小的修改：

1. `ticks % TICK_NUM == 0`时，不再将当前进程的`need_resched`置1，因为这是临时的调度策略
2. 每一次时钟中断来临时调用`sched_class_proc_tick`函数，为此还需要将这个函数变为非`static`并在头文件中加入声明

与参考答案相比，参考答案忘记在时钟中断调用`sched_class_proc_tick`了，这是严重的错误。

练习1

一般来说，调度器内部会维护一个数据结构，存放目前就绪的进程，基于这个设计，有一些接口。现在，总结调度器接口每个函数的作用或用法：

init

初始化调度器内部数据结构。同时清零进程个数计数器。

enqueue

将一个（新的）进程放入调度器内部数据结构。同时，将进程个数计数器增加1。

如果是基于时间片的调度算法，并且这个进程时间片用完，则需要分配新的时间片。

dequeue

将一个指定的进程从调度器内部数据结构中移除。同时，将进程个数计数器减少1。

pick_next

当前进程需要被重新调度，即需要选择一个新的进程运行时，此函数会被调用来从内部数据结构中选出下一个运行的进程。

这个函数可以认为是调度算法的核心。

proc_tick

每一次时钟中断，此函数都会被调用。

如果是基于时间片的调度算法，可以在此函数中减少当前进程的时间片。

如果当前进程时间片用完，则置当前进程为需要被重新调度状态。

一个例子

下面为简单，记Round Robin调度算法为RR调度器。

假设系统已经初始化好，当前进程剩余时间片为5。

此时，发生了时钟中断，`proc_tick`被调用，该进程时间片减少1后恢复运行，好像没有什么变化。

直到第5次发生时钟中断，`proc_tick`被调用，该进程时间片被减少到0，根据算法，当前进程的`need_resched`被设置为1，说明需要重新调度了。

中断处理程序发现`need_resched`被设置为1后，接下来会调用`schedule`，调度器真正开始发挥作用。

首先，内核会调用`enqueue`将当前进程放入RR调度器队列的尾部，放入时，RR调度器还会将这个进程的时间片增加为算法设定好的时间片最大值。然后，调用`pick_next`从RR调度器队列首部选择下一个占用处理器的进程，如果存在的话，接着还需要调用`dequeue`，将这个进程从RR调度器的队列中取出。如果失败，说明没有可以运行的进行了，内核就让`idle`进程占用处理器。注意，`pick_next`可能返回的进程就是刚刚调用`enqueue`时被放入RR调度器队列的那个进程，不过这不需要特殊处理。

最后，内核调用`proc_run`开始执行新的进程。关于`proc_run`的说明，请见Lab 4实验报告的练习3。

多级反馈队列调度算法

调度算法的设计，其实是接口函数的实现的设计。

算法需要为每个优先级维护一个队列，然后维护一个当前优先级的运行状态。同时，还需要为每个进程记录当前所在的优先级，初始化时，优先级为最高优先级。

init

初始化算法维护的所有数据结构。

enqueue

若该进程时间片为0，说明是新进程或者刚刚用完了时间片，那么不改变其优先级，并将其放入相应优先级的队列中。

否则，说明该进程没有用完时间片，将其优先级降低一级，然后将其放入相应优先级的队列中。

最后，将进程的时间片置为相应优先级应有的时间片。

dequeue

从相应优先级的队列中将此进程取出即可。

pick_next

首先，根据调度算法维护的当前优先级的运行状态，利用某种调度算法（比如RR调度），选择下一个要执行的进程的优先级。

然后，从相应优先级队列中，取出一个进程返回。

proc_tick

这是基于时间片的调度算法，函数的功能或用法同上面所述。

练习2

Stride Scheduling调度算法的实现实际上是接口的实现，把接口实现正确，调度算法就可以正常工作了。

算法实现参考MOOC和注释的讲解，Stride Scheduling调度算法与Round Robin调度算法唯一不同之处在于每一次选择的进程是当前`stride`值最小的那个而不是进程队列（可用链表实现）的头。这里，我选用斜堆实现。`run_queue`的`lab6_run_pool`存放斜堆堆顶的指针，这个调度算法实际上就是利用这个堆，每次选择`stride`值最小的进程。特别地，当堆中无进程时，`lab6_run_pool = NULL`。实际上也可以用链表实现，不过，这样查找`stride`值最小的进程并取出的复杂度为O(n)而不是O(logn)了。

与练习1类似，有如下接口的实现需要说明：

init

初始化堆，实际上就是赋值`lab6_run_pool = NULL`。

同时也要清零进程个数计数器。

enqueue

将新的进程插入斜堆，更新堆顶指针。将进程个数计数器增加1。

如果是基于时间片的调度算法，并且这个进程时间片用完，则需要分配新的时间片。

dequeue

将指定的进程从斜堆删除，更新堆顶指针。将进程个数计数器减少1。

pick_next

斜堆堆顶就是`stride`值最小的进程，根据Stride Scheduling调度算法，返回它即可。

特别地，对于Stride Scheduling调度算法，此时还需要增加进程的`stride`值，说明这个进程又执行了`pass = BIG_STRIDE / priority`。虽然此时进程还未执行，但是在之后调度到别的进程之前，一定会执行这么多的。

另外，时间片不需要在此函数维护，因为`enqueue`函数已经维护了。

proc_tick

与Round Robin调度算法一致，在此函数中减少当前进程的时间片。

如果当前进程时间片用完，则置当前进程为需要被重新调度状态。

BIG_STRIDE的选取

记`max_stride`为某时刻所有`stride`真实值的最大值，`min_stride`为某时刻所有`stride`真实值的最小值，`PASS_MAX`为每一次`stride`增量的最大值。

对于每一个进程而言，考虑到优先级是`priority`正整数，每一次`stride`的增量`pass`有这样的关系：

`pass = BIG_STRIDE / priority <= BIG_STRIDE`

所以，`PASS_MAX <= BIG_STRIDE`

首先，利用数学归纳法证明`max_stride - min_stride <= PASS_MAX`：

不妨假设进程数多于一个，否则结论立即成立。

基础

起初，所有进程的`stride`都为0，所以`max_stride = min_stride = 0`。

某个进程的`stride`被增加后，有`max_stride = pass <= PASS_MAX`，且仍有`min_stride = 0`。

于是`max_stride - min_stride = max_stride <= PASS_MAX`。

归纳

根据归纳假设，有`max_stride - min_stride <= PASS_MAX`，为了便于区分，加一个角标，改写为：

`max_stride0 - min_stride0 <= PASS_MAX`，0表示增加`stride`之前的状态

根据算法，原来`min_stride0`的进程需要被变为`min_stride0 + pass <= min_stride0 + PASS_MAX`

1 若`min_stride0 + pass <= max_stride0`，则`max_stride`不发生变化，而`min_stride`变大，所以

```
   max_stride - min_stride
 = max_stride0 - min_stride
<= max_stride0 - min_stride0
<= PASS_MAX
```

2 否则，若`min_stride0 + pass > max_stride0`，则`max_stride = min_stride0 + pass`，而`min_stride`变大，所以

```
   max_stride - min_stride
 = min_stride0 + pass - min_stride
<= min_stride0 + pass - min_stride0
 = pass
<= PASS_MAX
```

Q.E.D.

于是，`max_stride - min_stride <= PASS_MAX <= BIG_STRIDE`

接着，考虑`BIG_STRIDE`的选取。

由于算法将使用两进程`stride`的差值，并将结果转换为32位有符号数来进行大小判断，因此需要保证任意两个进程`stride`的差值在32位有符号数能够表示的范围之内，即`max_stride - min_stride <= 0x7fffffff`。

立即得到`BIG_STRIDE <= 0x7fffffff`。经过实际测试，`BIG_STRIDE`稍微超过一点（比如取为`0x80000000`），就会导致最后结果混乱。

我的实现与参考答案十分一致，不过没有实现链表的版本。

Lab 7

知识点

这些是与本实验有关的原理课的知识点：

- 禁用中断的同步方法
- 信号量：实现上与原理课的讲解有细微偏差
- 条件变量、管程：实现上与原理课讲解有很大不同
- 哲学家就餐问题

此外，本实验还涉及如下知识点：

无

遗憾的是，如下知识点在原理课中很重要，但本次实验没有很好的对应：

- 纯软件实现的同步方法
- 读者-写者问题
- 死锁
- 进程间通信

练习0

首先，需要对之前的代码进行微小的修改：在每一次时钟中断来临时改为调用`run_timer_list`，而不是调用`sched_class_proc_tick`。同时，恢复`sched_class_proc_tick`为`static`。

练习1

在实现信号量时，使用到了等待队列`wait_queue_t`。

等待队列实际上就是一个链表，存放了若干等待对象。等待对象`wait_t`实际上就是进程指针和一些标志组成的结构体。等待队列各种操作实际上就是对链表操作的包装。

初始化操作

一个信号量维护了其数值以及一个等待队列。

初始化时需要设置信号量的数值，以及初始化等待队列。

down（P）操作

首先判断数值是否大于0，大于0的话则直接减少1，然后立即返回。否则，将当前进程放入等待队列并运行调度器，调度到其他进程。当再次被调度回来的时候说明进程被唤醒，若等待对象的标志未被修改则进一步说明获得了信号量。等待对象的标志发生了变化，则说明是其他原因唤醒的。无论如何，进程会将自己从等待队列中移除（若还在队列中），本次操作完成。

注意，整个操作过程中要关中断来保护共享变量不被意外修改。

up（V）操作

这个操作和down操作在某种意义下是对偶的。

首先判断该信号量的等待队列是否为空，为空则将数值增加1然后立即返回。否则，说明有进程在等待此信号量，此时，选取等待队列队首的元素唤醒即可。

同样，整个操作过程中要关中断来保护共享变量不被意外修改。

从上面可以看出，这样的实现保持了信号量的数值大于等于0的性质，这与原理课不同，其他的实现也与原理课的讲解有很大区别。

为了给用户态提供信号量，一种可能的方法是直接复用内核线程的down、up操作的函数，把它们包装成系统调用供用户进程使用。此外，还要加入用户向内核申请、释放一个信号量对象的系统调用。需要注意的是，内核线程的down以及up操作使用信号量结构体的指针作为参数，但系统调用不能这样做，因为出于安全性考虑，用户进程不能持有或间接访问内核的指针。与Lab 8中的打开的文件类似，可为每一个进程维护一个信号量指针数组，由一个用户进程的多个线程共享，创建、down、up或释放操作时都使用数组的下标来进行访问。此外，内核还应做必要的参数检查。

练习2

这个练习提供的注释已经写得非常详细，比实际要写的代码还要多，根据它写出C代码即可。与参考答案对比，我的实现和参考答案十分一致。

这里的条件变量的实现基于信号量，经过分析，实现的是Hoare管程。

初始化操作

条件变量维护了一个等待者计数器和用于实现条件变量的信号量。条件变量所在的管程维护了一个互斥体`mutex`用于保护管程，以及一个`next`信号量和`next`计数器组合起来用于模拟一个进入管程的等待队列。

初始化时，将它们全部初始化。

wait操作

首先，会将等待者计数器增加1，记录这个线程正在等待的事实。然后，会利用`next`信号量和`next`计数器释放管程控制权，同时唤醒被阻塞的signal进程，如果没有被阻塞的signal进程（`next`计数器为0），则释放互斥体。

接着，真正开始利用条件变量的信号量等待signal进程发来通知。收到通知之后，将等待者计数器恢复（即减少1），结束等待操作。

signal操作

首先判断是否有等待者，若没有，则直接返回。否则，将`next`计数器增加1，表示当前线程会被等待者阻塞，然后利用条件变量的信号量完成对等待者的通知。之后利用`next`信号量等待重新获得管程控制权。最后，将`next`计数器恢复（即减少1），完成本次操作。

由于上面已经讨论了一个给用户态提供信号量的方法，要为用户态提供条件变量，可以直接把内核态条件变量的实现复制到用户态，只是将其中信号量的操作改为调用上述相应的系统调用。

能否不基于信号量机制来实现条件变量?

我认为可以直接使用等待队列实现，但是仍然需要用到某种锁机制，可以是互斥体，也可以是自旋锁。为简单，这里描述Hansen管程的设计。

初始化操作

初始化锁和等待队列。

wait操作

工作流程设计如下：

1. 保存中断标志并关中断
2. 释放锁
3. 将当前进程设置为睡眠状态并放入等待队列
4. 恢复中断
5. 运行调度器，切换到其他进程，等待切换回来
6. 保存中断标志并关中断
7. 将当前进程从等待队列中移除
8. 恢复中断
9. 重新获得锁，此时可能再次睡眠

注意，1、2顺序不能够互换，否则释放锁之后，另外一个进程可能在当前进程被放入等待队列之前执行signal操作，当前进程无法收到，从而导致其一直在等待队列中睡眠。

signal操作

工作流程设计如下：

1. 保存中断标志并关中断
2. 若等待队列不为空，则选择队列中第一个进程唤醒
3. 恢复中断
4. 可以运行调度器，切换到其他进程，从而尝试获得Hoare管程的效果

从上面可以看出，条件变量的wait操作与信号量的down操作十分类似，条件变量的signal操作与信号量的up操作十分类似。区别在于，条件变量在signal时，若等待队列为空会忽略本次操作，而信号量up操作时，若队列为空会把计数器加一，从而让之后的down操作直接通过。同时，条件变量还需要维护一个锁。

Lab 8

知识点

这些是与本实验有关的原理课的知识点：

- 管道
- 虚拟文件系统框架
- 文件描述符
- 目录等
- inode、打开的文件等结构体
- inode缓存

此外，本实验还涉及如下知识点：

- 简单文件系统
- ucore特定的文件系统架构

遗憾的是，如下知识点在原理课中很重要，但本次实验没有很好的对应：

- 其他进程间通信机制，例如信号、消息队列和共享内存
- RAID
- 磁盘调度算法
- 磁盘缓存
- IO

练习0

首先，需要对之前的代码进行微小的修改：

1. `alloc_proc`加入对新增字段的初始化，由于我使用`memset`初始化，不需要修改
2. `do_fork`中，使用`copy_files`复制进程打开的文件

练习1

这个练习需要实现SFS文件系统层的`sfs_io_nolock`函数。实现过程中，需要用到以下辅助函数或函数指针：

- `sfs_bmap_load_nolock`：根据相对于文件而言的块号，获得inode号。
- `sfs_buf_op`：带缓冲区的块操作，可以用来进行对齐或非对齐的块读写操作；但是，对齐的操作不应当用此函数，否则会有多余的拷贝操作。根据读写操作的不同，分别指向`sfs_wbuf`或`sfs_rbuf`。
- `sfs_block_op`：直接的块操作，可以用来进行对齐的块读写操作。根据读写操作的不同，分别指向`sfs_wblock`或`sfs_rblock`。

根据要实现的`sfs_io_nolock`函数的输入参数，可以计算出读写文件操作的开始和结束位置，它们可能是没有按照块大小对齐的，为此需要特殊处理。我的处理方案分两大类情况：

第一类情况

开始和结束位置中有一个值没有按照块大小对齐，并且开始和结束在同一块内。

对于这种情况，只须调用`sfs_bmap_load_nolock`得到块号，然后使用`sfs_buf_op`完成非对齐的操作。

第二类情况

不是上述情况的其他情况，例如开始和结束位置不在同一块。

可以将开始到结束的区间分为三段：

1. 首部非对齐的部分
2. 中间对齐的部分
3. 尾部非对齐的部分

对于首部非对齐的部分以及尾部非对齐的部分，处理方法类似于第一类情况。

对于中间对齐的部分，对于其中每一个块，调用`sfs_bmap_load_nolock`得到块号后直接调用`sfs_block_op`完成对齐的操作。注意，虽然`sfs_block_op`支持一次操作多个块，但这里对于文件的操作不能一次调用`sfs_block_op`完成所有块的操作，因为相对于文件连续的块不一定在下层存储设备上连续，即块号不一定连续，故对于每一个块需要重新调用`sfs_bmap_load_nolock`得到块号。

注意，实现时还需要考虑出错处理。

与参考答案对比，我的思路和参考答案一致，但是参考答案的实现更简洁明了。

管道机制的设计

创建

首先，是管道的创建，即`pipe`系统调用的实现。

为创建管道，可以先创建一个类型为管道的inode，操作虚表配置为如下说明的读写操作，同时要为管道准备好缓冲区。然后，创建两个“打开的文件”结构体（`file`结构体）和相应的文件描述符，一个设置为只读，另外一个设置为只写，都绑定到此inode。最后，把这两个文件描述符返回给用户。一种实现是用户提供一个数组，内核填写为两个文件描述符。

读

读的实现也就是inode的`read`函数的实现，记需要读的长度为`r`（代表read），缓冲区内的可用数据长度为`d`（代表data），则分为以下情况。

情况1 `r <= d`

从缓冲区移动长度为`r`的数据到用户空间，然后令`d = d - r`。返回值设定为`r`。

唤醒写者（若有）。

情况2 `r > d`且写者未关闭管道

从缓冲区移动长度为`d`的数据到用户空间，然后令`d = 0`、令`r = r - d`、唤醒写者（若有），然后将当前进程变为睡眠态，直到写者写入数据后被唤醒，被唤醒后再次开始循环。直到满足`r <= d`，接着按照情况1处理，但返回值设定为最初的`r`。循环途中若写者关闭了管道，则立即返回，返回值设定为当前已经读到的数据的长度。

情况3 `r > d`且写者关闭了管道

从缓冲区移动长度为`d`的数据到用户空间，然后令`d = 0`。返回值设定为`d`。

注意，读者关闭管道时要唤醒写者（若有）。

写

读的实现也就是inode的`write`函数的实现，记需要写的长度为`w`（代表write），缓冲区内的可用缓冲区长度为`b`（代表buffer），则分为以下情况。

情况1 `w <= b`且读者未关闭管道

从用户空间拷贝长度为`w`的数据到缓冲区，然后令`b = b - w`。返回值设定为`w`。

唤醒读者（若有）。

情况2 `w > b`且读者未关闭管道

从用户空间拷贝长度为`b`的数据到缓冲区，然后令`b = 0`、令`w = w - b`、唤醒读者（若有），然后将当前进程变为睡眠态，直到读者读出数据后被唤醒，被唤醒后再次开始循环。直到满足`w <= b`，接着按照情况1处理，但返回值设定为最初的`w`。循环途中若读者关闭了管道，则立即返回，返回值设定为当前已经复制到缓冲区的数据的长度。

情况3 读者关闭了管道

直接返回0。

注意，写者关闭管道时要唤醒读者（若有）。

约束

上面的实现其实隐含了如下几点特殊的约束条件：

- `b + d = 缓冲区大小`
- 不能同时有读者和写者处于睡眠，有则说明出现了死锁

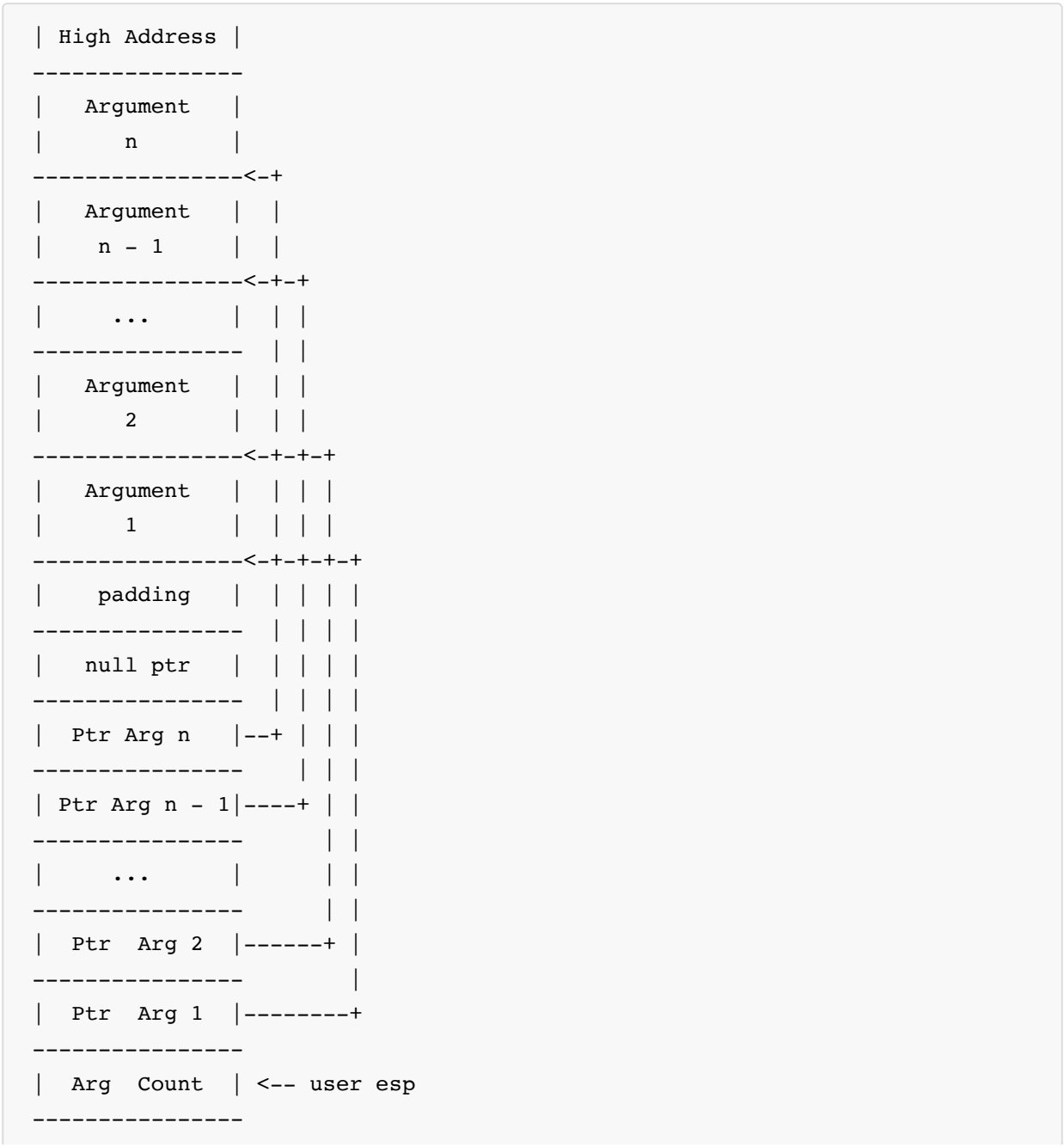
顺便一提，管道的实现和stdin设备的实现非常类似。

练习2

这个练习有两大部分：读取并解析ELF格式的可执行文件映像、在用户态栈上构建命令行参数。

为了实现`load_icode`来读取并解析ELF格式的可执行文件映像，主体逻辑部分可以复用Lab 7中的代码，只是把其中对于`binary`字节数组的拷贝操作改为调用`load_icode_read`进行读文件操作。

接着，需要在用户态栈上填充命令行参数，显然，用户代码的参数不能存放于内核的空间。通过阅读并分析`user/libs/initcode.S`以及`user/libs/umain.c`，可以知道在运行`user/libs/initcode.S`的代码之前，栈上需要有这样的布局：



```
| Low  Address |
```

其中，参数内容长度不确定，参数指针和参数个数变量均为4字节。最后一个参数指针的后面是一个标志结束的空指针，参数指针开始位置根据约定和性能考虑，需要4字节对齐，所以空指针后面有4字节对齐的填充（padding）。

实现时，只需要将内核空间的参数拷贝到用户栈上、压入填充、压入空指针、压入对应的参数指针，最后压入参数个数即可。此外，由于之前加载了新的页目录基址寄存器`cr3`，这里可以直接使用用户态的虚拟地址进行访问。

与参考答案对比，处理ELF部分我和答案实现完全一致，对于在用户态栈上填充命令行参数部分，我认为我的实现思路更清晰，代码风格更优美。

硬链接机制的设计

硬链接的设计非常简单，只需要创建时将目录项的名字设定为指定的名字，目录项的inode号设置为目标文件或目录等的inode号即可。此外，还需要增加目标inode的引用计数。

删除时，同样需要减少目标inode的引用计数，归零后，清除目标inode及其数据块。

其他操作均不需要变化，也不需要修改数据结构。

软链接机制的设计

软链接也叫做符号链接，实际上就是一类特殊类型的文件，文件的数据为指向的目标文件或目录等的路径。

创建时，只需将类型设置为符号链接，文件内容（数据）设置为目标路径字符串。

删除操作和普通文件实现一致。

此外，在各类`lookup`操作时要考虑软链接的作用，需要真正处理软链接的目标而不是软链接本身。

祝贺我通过自己的努力，完成了ucore OS lab1 - lab8!