哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

xv6 项目分析报告

学生班级：18 级 5 班

学生学号：180110518

学生姓名：胡智胜

2020 年 12 月

目录

简介

本报告是针对 MIT 操作系统课程 6.s081/Fall 2019 的 xv6 系统实验 Lab1 至 Lab10 的分析说明报告。在每个实验的设计方法中一般首先会介绍完成该实验所要了解的相关原理知识，接着会介绍完成该实验的代码的思路及流程。报告的最后给出了我的 xv6 项目完整代码的 GitHub 地址，以及我对学校 OS 实验的改进建议。报告中的内容仅是我个人总结和理解，如有任何错误的地方，还望多多理解。

Lab1 Utilities

1.1 内容分析

第一个实验要求我们实现 sleep, ping-pong, primes, find, xargs 五个指令，这五个指令都是作为工作在 user space 中的程序执行。sleep 使系统睡眠指定 timer interrupt 数，ping-pong 使两个进程之间通过两个 pipe 实现互相通信，primes 要求通过管道 pipe 实现质数筛选，find 要求在某一路径中查找某个文件，xargs 则要求实现 Linux 系统中 xargs 的**部分功能**，即将本行的参数同下一行参数合并起来执行 xargs 后面所跟的指令。

1.2 设计方法

1.2.1 sleep

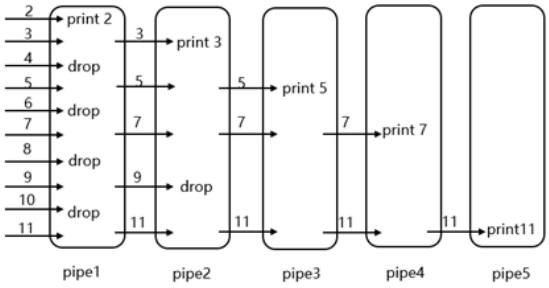
sleep 比较容易完成，只需从命令行参数中获取到 sleep 后面跟着的数字，调用系统函数 sleep(x)即可。sleep 的实现是 kernel/sysproc.c 中的 sys_sleep 函数，该函数使用到了 kernel/trap.c 中记录时钟中断数的 ticks 全局变量, 时钟中断的介绍在 xv6 book 中 4.4 章节，大致讲时钟中断是由 RISC-V CPU 自带的硬件产生, 其中断处理程序运行 RISC-V 的 machine-mode 中，且由于时钟中断每隔一段时间就会产生，为了不让时钟中断破坏 xv6 系统执行的一些关键操作，xv6 为时钟中断设置了另外一套寄存器供其使用，并且让时钟中断来产生软件中断报告给 xv6，以此来保证 kernel 的关键操作不被时钟中断打断。最后每当 xv6 处理一次时钟中断，kernel/trap.c 中的 ticks 就会加 1。

1.2.2 ping-pong

完成 ping-pong 需要掌握 pipe 管道的原理和功能。在本实验中首先应在一个进程中创建两个 pipe，这两个 pipe 相当于都是单向的管道，一个 parent_fd 管道用来父进程发送消息给子进程，一个 child_fd 管道用来子进程发送消息给父进程；创建完两个管道后调用 fork 函数分裂为父进程和子进程，接着父进程向 parent_fd 写入"ping",子进程读取 parent_fd 并打印相应信息，接着子进程向 child_fd 写入"pong"，父进程读取 child_fd 并打印相应信息。
要注意的点就是 pipe 在 read 的时候只有当所有的 write 端都被关闭时才会停止 read，因此我们在使用 pipe 时最好将不需要用到的端先关闭了（特别是写入端），如果一个进程退出时没有关闭一个 pipe 的写入端，就可能发生另一个进程一直在 pipe 读取端等待信息的情况。xv6 的 exit()函数(kernel/proc.c 中的 exit()函数)会将该进程的所有已打开的文件描述 fd 都关闭掉，因此我们只要保证进程在结束时调用一次 exit()函数就可以自动关闭已打开的 pipe 的写入端了。

1.2.3 primes

primes 就是 ping-pong 的进阶问题，还需要了解质数筛选算法，以及 dup()函数的作用，当初我在做该题时因为不清楚 dup 函数的作用导致半天都没想出来此题的思路。



该题的算法思想如上图，pipe 就相当于不同层之间的通道，每层负责将从上一层得到的数字的所有整数倍值都给筛选掉，只将那些不是整数倍的数字再写入给自己的下一层，例如一开始将所有数字 2~35 输入给第一层，第一层拿到的第一个数字是 2 因此将所有 2 的整数倍数字都淘汰掉，这样第二层拿到的第一个数字是 3…以此类推，最后每一层得到的第一个数字组合起来的集合就是所有质数。

1.2.4 find

find 功能实现首先需要了解的是部分文件系统中的知识——dirent 与 stat。文件系统中文件夹也是被实现为单一的文件，其 type 为 T_DIR，而一般的文件为 T_FILE；xv6 中的文件由 inode 及其附属数据代表，类型为文件夹的文件中的数据由 dirent 目录项组成，代表该文件夹中的各个文件，dirent 由一个文件的 inode 号和其文件名组成（inode 号相当于是磁盘中某个 inode 结构体所在 inode 数组中的下标）。而 stat 一般由 fstat 函数（kernel/fs.c 中 stati 函数）从某个已打开的文件怎获取，stat 中存储了该文件 inode 中的相应信息（dev, ino, type, nlink, size），其中 type 就是 T_FILE,T_DIR,T_DEV 中的一种（在 find 中不考虑 T_DEV）。

了解了以上知识后实现 find 功能就思路很清晰，首先先获取到要搜索的路径名和文件名，通过 open 函数打开该路径（文件夹文件）且获得打开文件的文件描述符 fd，再通过 fstat 获取该路径文件的文件信息 stat，再从该路径文件中循环读取 dirent，每读取到一个 dirent，就将当前路径 path 和该 dirent 中存储的文件名拼接到一起组成子文件的绝对路径，通过该路径读取该子文件的 stat 信息（代码中调用 user/ulib.c 中的 stat 函数获取 stat，stat 与 fstat 的唯一区别，fstat 要求传入的是文件描述符（文件已打开），stat 只要求传入文件的路径即可），再判断该子文件的类型，如果子文件是普通 T_FILE，则打印出该文件的完整路径，如果是文件夹文件 T_DIR，则递归调用 find 函数，其中传入的 path 就是该子文件的完整路径。

1.2.5 xargs

xargs 功能与实际 Linux 系统中的 xargs 功能不完全相同，xv6 实验中的 xargs 只需要将第一行的指令参数与下一行指令参数拼接在一起再 exec 执行指令即可，而 Linux 系统中的 xargs 功能更加复杂，例如 Linux 中 xargs 可将多行参数合并为一行执行，而本实验的 xargs 只支

持两行。比如 xargs echo 12 (换行) 34 (换行) 56，真正的 xargs 应该是输出 12 34 56 但本实验的要求是将第一行附加在其余行前面执行，则结果就变成了 12 34 (换行) 12 56

实现本实验中的 xargs 功能首先是将第一行参数保存下来，具体做法是通过 malloc 申请 32 个 512 字节的空间用于保存各行字符串，再设置一个大小为 32 的 char 型指针数组 n_params 用于保存第一行的各个参数，

```
// n_params 是(从第二行开始)每一行与第一行进行拼接成的参数列表
    char *n_params[MAXARG];
    // 先将第一行的参数复制到 n_params
    for (int i = 1; i < argc; i++)
    {
        n_params[i - 1] = argv[i];
    }
```

接着从输入的第二行开始，对第 n 行:
将第 n 行的字符串进行切分得到第 n 行的各个参数，将这些参数附加到 n_params 中末端（即拼接在第一行参数的尾部），最后调用 fork 创建出子进程，在子进程中调用 exec 系统函数执行由“第一行的命令及参数”与“第 n 行参数“拼接在一起的命令参数 n_params。

```
if (fork() == 0)
{
    exec(command, n_params);
    exit();
}
```

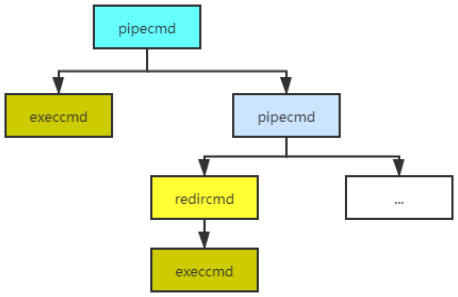
1.3 实验结果

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/xv6-riscv-fall19$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
sleep, no arguments: OK (1.5s)
sleep, returns: OK (0.9s)
sleep, makes syscall: OK (1.0s)
pingpong: OK (1.1s)
primes: OK (1.0s)
find, in current directory: OK (1.2s)
find, recursive: OK (1.3s)
xargs: OK (1.0s)
Score: 100/100
```

Lab2 Shell

2.1 内容分析

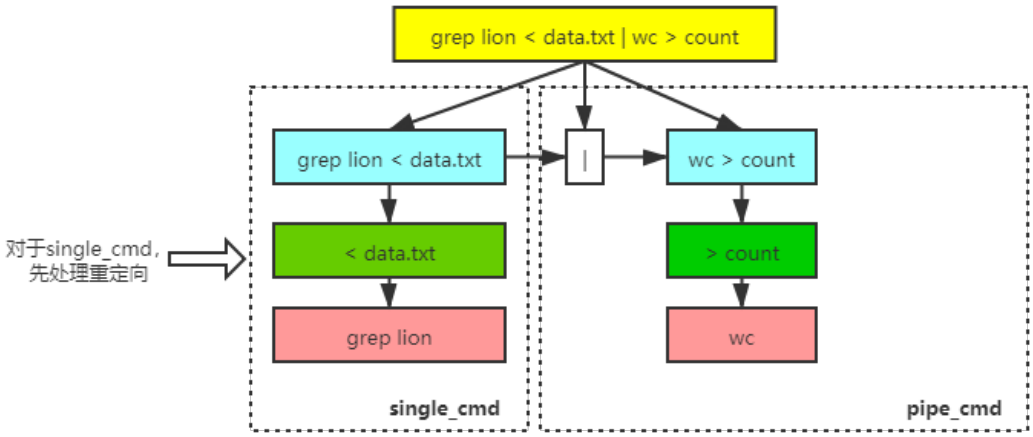
第 2 个实验要求完成一个基础的 shell 功能，限制是不能使用 malloc 函数。xv6 自身的 sh 功能是通过构造几种 cmd 结构体组成的“cmd 树“来实现的，例如下图:



xv6 的 sh 实现主要思想是首先通过 parseline(user/sh.c)函数将用户输入的一行参数解析为 cmd 树，然后再调用 rumcmd 函数去递归下降式地执行该生成的 cmd 树，本实验的代码可以借鉴上述思想。

2.2 设计方法

借鉴 kernel/sh.c 中的思想，不过我们不可使用 malloc 函数，意味着原本 sh.c 中的构造 cmd 树的思想不再适用，此时我们又可参考编译原理构造语法树生成中间代码的思想，我们既可以等待语法树构造完毕后再递归下降式地遍历语法树生成中间代码，也可以在构造语法树的同时就将中间代码生成出来。因此，我们可以像 kernel/sh.c 中构造 cmd 树的同时去执行相应指令即可，而不必等待 cmd 树构造完毕后再去执行该 cmd 树。同时也要注意到本实验并不要求我们实现 sh 的所有功能，只需要实现一般的命令执行、输入输出重定向、pipe 管道即可。
设计思想如下：



将输入的一串字符串看作是 "single_cmd | single_cmd | ..." 即多个 single_cmd 通过 pipe 连接起来，因此把不包含 pipe 的 cmd 看作是单个 cmd，把包含 pipe 的 cmd 看作是多个“单个 cmd”的组合体；将 pipe_cmd 看作是 "single_cmd | pipe_cmd"以此递归下去，函数每次处理一个 single_cmd 即可；
对于 single_cmd，将其看作一个 command 与 params 以及其它重定向语句组合而成，即 single_cmd = command + params + redir；那些重定向语句只对当前进程有效，并且重定向语句要先处理，之后再调用 exec 执行 command 与 params；在处理重定向语句时，找到重定向语句并执行，先找'>','<','>>'三个标识，找到后再取得跟在这些符号后面的文件名，需要说明的是，在本实验中'>>'与'>'效果相同(sh.c 中也将'>>'与'>'处理的语句相同)。

2.3 算法分析

在主函数中：

1. 在 user/nsh.c 中首先打开标准输入 0，标准输出 1，标准错误输出 2；
2. 循环读取用户输入的一行命令到缓冲区 buf 中（若读取失败则退出循环）：
    1) 将 buf 中最后一个有效位置上的换行符'\n'丢弃(设置为 0)；
    2) 调用 fork 创建子进程，在子进程中：
        a) 切分 buf，将 buf 字符串中各个符号按空格切分出来并存储到 char 指针数组 args 中（例如将"A < B | C"切分为"A","<","B","|","C"）
        b) 用切分出来的参数数组调用 run_pipe_cmd
    3) 调用 wait(0)等待子进程的完成，回到第 2.步
3. 主函数结束

在 run_pipe_cmd(int argc, char **args)中：

1. 找到第一个'|'字符在 args 中的下标为 i；
2. 如果没得到字符'|'（说明没有 pipe），则调用 run_sing_cmd(argc, args)即按照 single_cmd 去处理该 cmd；
3. 否则，将 args[i] = 0 即清除掉'|'，（之所以清除掉这个，是因为 exec 要求最后一个参数的字符串指针的下一个指针需要是 0）：
    1) 创建 pipe 管道 p；
    2) 调用 fork 分裂为父子进程，在子进程中：
        a) close 标准输出 1；
        b) 调用 dup(p[1])，将管道 p 的写入端置为子进程的标准输出；
        c) 关闭原来的 p[0]和 p[1]；
        d) 调用 run_sing_cmd(i, args)；
    3) 在父进程中：
        a) close 标准输入 0；
        b) 调用 dup(p[0])，将管道 p 的输出端置为父进程的标准输入；
        c) 关闭原来的 p[0]和 p[1]；
        d) 调用 run_pipe_cmd (argc - i - 1, args + i + 1)；
4. run_pipe_cmd 函数结束；

在 run_single_cmd(int argc, char **args)中：

1. 循环执行 try_to_redir(argc, args)，直到 try_to_redir 返回负数；
2. 接着再找到第一个不为 0 的 args 中元素的下标（即不为空指针的元素）
3. 调用 exec 执行指令；

在 try_to_redir(int argc, char **args)中首先在字符串指针数组 args 中找到'>','<','>>'三种字符其中一种，若未找到则返回-1，找到第一个时就停止继续查找；若是'>'符，则函数关闭标准输出 1，再 open 跟在'>'符后面的文件；若是'<'符，则函数关闭标准输入 0，再 open 跟在'<'符后面的文件；若是'>>'则和'>'处理相同；接着返回 1 代表此次处理成功。

2.4 实验结果

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/xv6-riscv-fall19$ ./grade-lab-sh
make: 'kernel/kernel' is up to date.
running nsh tests: (4.2s)
  simple echo: OK
  simple grep: OK
  two commands: OK
  output redirection: OK
  input redirection: OK
  both redirections: OK
  simple pipe: OK
  pipe and redirects: OK
  lots of commands: OK
Score: 100/100
huhu@huhu-virtual-machine:~/HITSZ2020_OS/xv6-riscv-fall19$
```

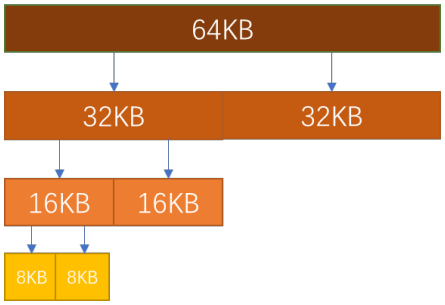
Lab3 Allocator

3.1 内容分析

实验的要求有两个：一是解决 xv6 原有的 file 结构体静态分配内存空间（使用了固定大小的数组）导致最大同时打开的文件数量被限制为 NFILE 的问题；二是 xv6 已经有的 buddy allocator 的 sz_info 结构体中，为每个块都指定了 1 位的 alloc 空间，但其实可以每一对伙伴之间用 1 位 alloc，这样就可以使原来的 alloc 占用的空间缩小为二分之一的问题。

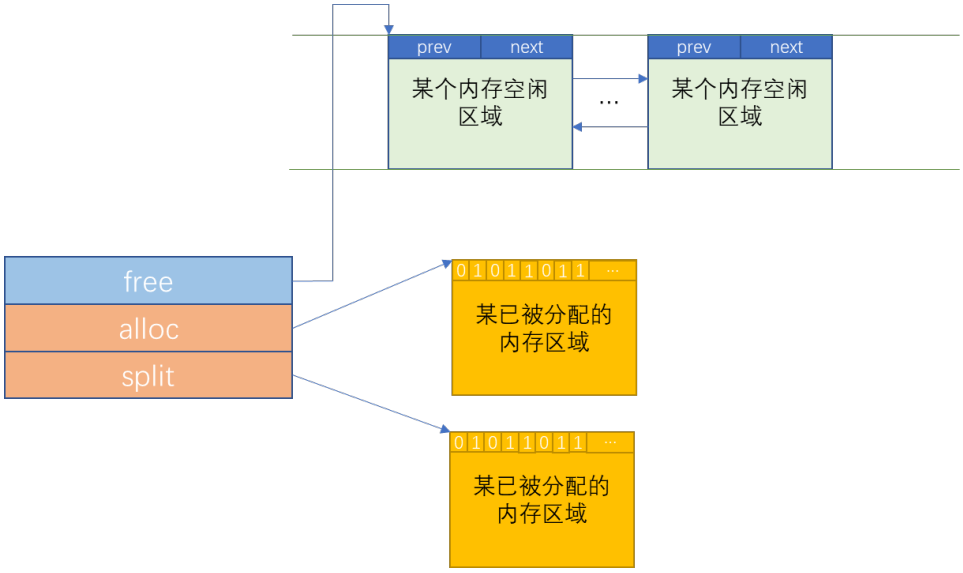
3.2 设计方法

在伙伴系统中，空闲空间首先从概念上被看成大小为2^N的大空间。当有一个内存分配请求时，空闲空间被递归地一分为二，直到刚好可以满足请求的大小（再一分为二就无法满足）。这时，请求的块被返回给用户。例如下例，一个 64KB 大小的空闲空间被切分以便提供 7KB 的块。



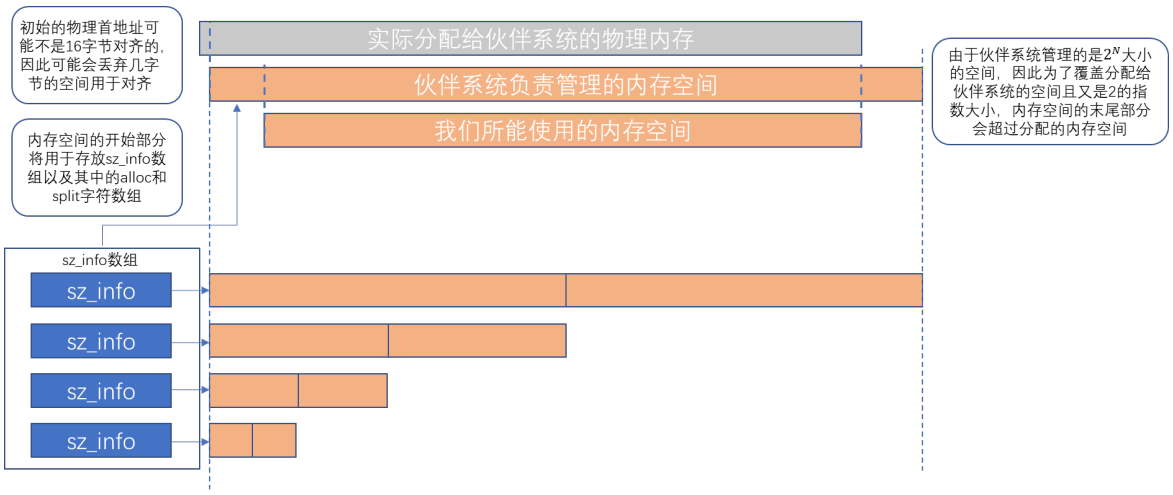
伙伴系统的最大功能在于块被释放的时候时，如果将这个 8KB 的块归还给空闲列表，分配沉痼会检查其“伙伴”是否空闲，如果是，就合并这两块。这个递归合并过程继续上溯。直到合并整个内存区域，或者某一个块的伙伴还未被释放。

在 xv6 中，伙伴系统是由一系列 sz_info 构成的，每个 sz_info 的结构如下：



每个 sz_info 负责管理**型号为 k**（型号为 k 的块的空间大小为2^k*16KB，k 从 0 开始一直到一个最大值 MAXSIZE）的所有块的信息， 其中 free 是一个双向链表，其中每个节点代表了该型号 k 的空闲块的物理地址，alloc 和 split 则都各自指向一片空间，其中数据的各位代表着其管理的某个型号 k 的块的状态，例如：第 i 个型号 k 的块已经被分配了，而且被分割 split 了，则 alloc 和 split 所指向的空间的第 i 位就为 1。这里第 i 个型号 k 的块，其实就是该块的首地址减去 buddy allocator 管理的空间首地址 bd_base 再整除型号 k 的块大小的商。

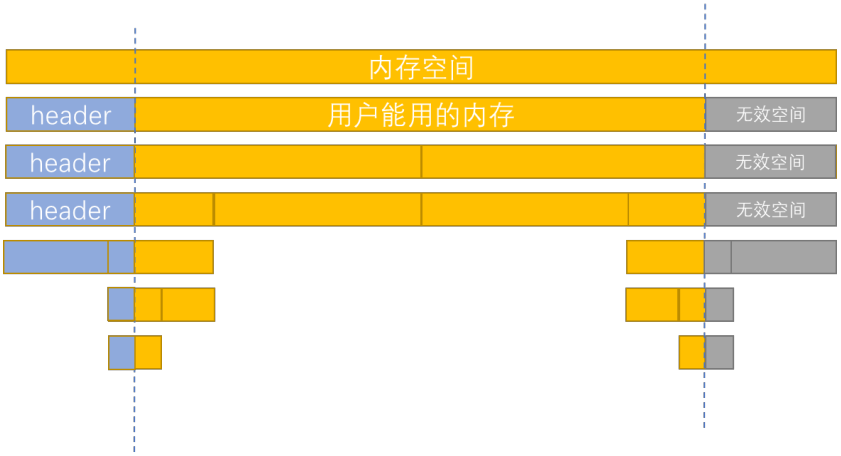
在知道单个 sz_info 的代表的含义后，就能很清晰地明白 xv6 系统中整个 buddy allocator 的结构，如下图：



buddy allocator 中的 sz_info 数组中第 i 个 sz_info 就负责管理所有型号为 i 的块，任何一个块被标记为 split 后就会在下层 sz_info 中被进一步管理，而 xv6 中允许的最小块大小为 16B 即型号 0 的块。

在 xv6 的 buddy allocator 初始化的时候系统分配的物理内存并不一定是 16 字节对齐或者刚好是2^N大小，因此在初始化的时候会将伙伴系统的内存首地址向后移动到第一个十六字节对齐的位置作为首地址 bd_base，若不是2^N大小则在初始化的时候会将内存大小扩展到第一个2^N满足$2^N \geq$ *系统分配的内存*，而系统允许的最大块的型号就为 N-1（即最大的块大小为

2^{N-1}*16B)。
同时 buddy allocator 在完成内存空间的初始化后，还会将前面部分的空间作为 sz_info 数组以及每个 sz_info 中 alloc 和 free 字符数组的存储区域，为了让这部分区域不被使用，伙伴系统在初始化时就将这部分区域在各个层次的 sz_info 同时标记为已分配和已切分的状态（kernel/buddy.c 中 bd_mark_data_structures 函数）；
而在伙伴系统管理的内存空间的末尾部分，由于之前的内存空间初始化时末尾部分可能会超过实际分配的内存，因此也需要将这部分的内存空间在伙伴系统中 sz_info 的各个层次上同时标记为已分配和已切分的状态（kernel/buddy.c 中 bd_mark_unavailable 函数）。
在其余内存空间都配置好后，伙伴系统就会开始初始化用户所能使用的内存空间（kernel/buddy.c 中 bd_initfree 函数），对于这段内存空间，其最坏的情况发生在空间的开始和空间的末尾，因为在空间内部的区域 buddy allocator 只需分配型号最大的块即可，而在空间的两端存在空间碎片就需要不断细化进入下层 sz_info（越下层的 sz_info 管理的块就越小），直到到达能管理这些空间碎片的 sz_info 为止。例如下图：



3.3 算法分析

要解决第一个问题比较简单，就是在 kernel/file.c 中删除掉原来 ftable 中设置的 file 数组，并在 filealloc 函数中将原来的“寻找一个空闲的 file 作为将要返回的 file”改为“调用 bd_malloc 函数分配一个 sizeof(struct file)的空间 f，并将该 f 所指向的内存初始为全 0，最后将该 f 作为 filealloc 函数的返回值”。在修改完 filealloc 函数后还需修改 fileclose 函数，就是在 fileclose 函数中“将传入的 file 结构体中的 ref 置为 0，type 置为 FD_NONE”的后面添加“调用 bd_free 函数对 file 结构体的内存空间进行释放”的操作即可。

要解决第二个问题，首先要改变原来的“为每个块都分配一位 alloc”的情况中使用的 bit_isset，bit_set，bit_clear 三个函数，在这里我新建了三个函数 bit_isset_alloc，bit_set_alloc，bit_clear_alloc，代表着针对 sz_info 中的 alloc 而新建的三函数。原来的 bit_isset，bit_set，bit_clear 函数的作用分别是在指定的内存空间中判断某位是否为 1，设置某位为 1，设置某位为 0；而新建的 bit_isset_alloc，bit_set_alloc，bit_clear_alloc 函数的作用则是将传入的第 index 位看作是之前的第 index/2 位，即假如想要设置第 12 位为 1，则 bit_set_alloc 将会设置指定内存空间中的第 6 位为 1。

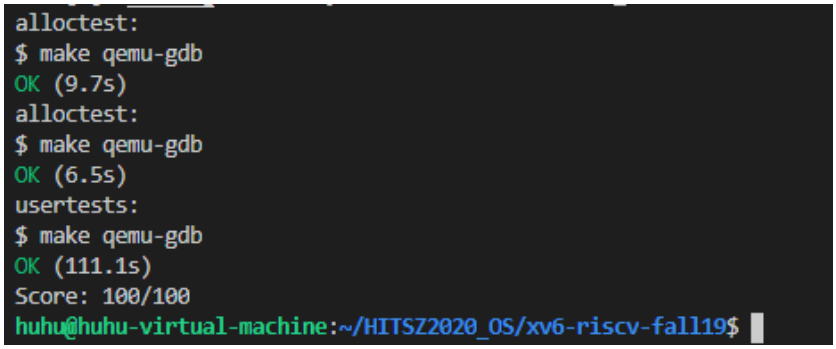
接下来的工作便是将之前 buddy allocator 中涉及 alloc 的地方都需要进行修改，首先是 bd_print 中有一处涉及打印 alloc 数组，由于 alloc 大小减半，因此需要将之前调用

bd_print_vector 中的代表 alloc 长度的值 NBLK(k)减半，改为 NBLK(k)/2 即可。
其次是将 bd_malloc 函数中对 alloc 调用 bit_set 的代码全修改为调用 bit_set_alloc, 将 bd_free 函数中对 alloc 调用 bit_clear, bit_isset, bit_set 的代码改为调用 bit_clear_alloc, bit_isset_alloc, bit_set_alloc，将 bd_mark 中对 alloc 调用 bit_set 的代码改为调用 bit_set_alloc。

在将所有对 alloc 的操作改为用相应的 bit_isset_alloc，bit_set_alloc，bit_clear_alloc 函数后，还需修改的就是 bd_initfree_pair 函数。bd_initfree_pair 函数被用于当一对“伙伴”其中一块被占用而另一块空闲，需要将空闲的块压入型号 k 的 sz_info 的 free 链表的情况。未经修改的 xv6 伙伴系统在该函数中是通过获取到两块对应的 alloc 值，如果这对“伙伴”的 alloc 值不相同则代表一块被占用而另一块空闲，因此再判断两块的 alloc 值，将 alloc 为 0 的块压入到对应 sz_info 中的 free 表。
修改后的 bd_initfree_pair 只需要判断一对“伙伴”的对应的 alloc 值是否为 1，之所以只需要判断是否为 1, 是由于修改后的 alloc 值代表的是一对“伙伴”之前对应的两个 alloc 值的异或。在判断出一对“伙伴”存在需要 free 的情况后，还需判断出哪个伙伴是空闲的。这个问题解决办法是给 bd_initfree_pair 传入要 free 的内存空间区域[left, right]，这样我们便可以判断一对“伙伴”中哪个块对应的内存空间完全包含于[left, right]之中就应该被压入到 sz_info 的 free 链表。

最后，还需在 bd_init 中将之前对 alloc 数组分配的大小由 sizeof(char)* ROUNDUP(NBLK(k), 8)/8，改为 sizeof(char)* ROUNDUP(NBLK(k), 16)/16。16 个块对应一个 char，因此需要将块数量按照 16 对齐，再按照 16 个块 1 个 char 去分配空间。

3.4 实验结果

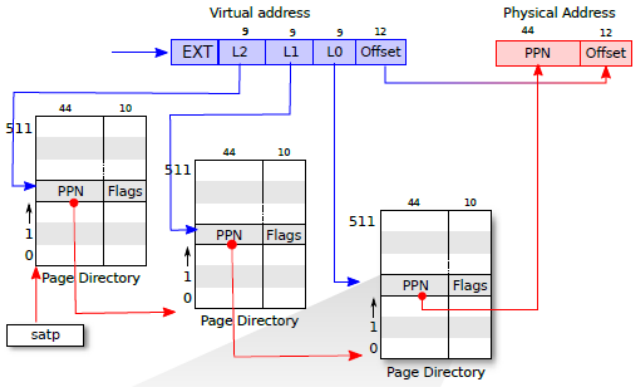


Lab4 Lazy Page Allocation

4.1 内容分析

该实验要求实现 Lazy Allocation 功能，就是在进程申请内存的时候操作系统并不立马为其分配内存空间，而是等到进程真正要使用这片内存区域的时候才为其分配空间。通过这种“吝

嗇”的分配来让操作系统能腾出更多的内存运行更多的进程。
当然本实验第一个要求是实现一个打印页表的功能 vm_print，这个函数只要了解了 PTE 的功能以及原理，同时阅读了 xv6 book 中第三章关于 xv6 的 PTE 实际实现就可以很快完成。具体来说，在 xv6 中使用了三级页表的结构，例如一个虚拟地址 va，共 64 位，前 25 位没有使用（作为保留位），后 12 位作为页内偏移量（即 xv6 中一页也是 4KB），剩下的 27 位就是三级页表所要用的三个 VPN 了，例如下图中 L2,L1,L0 就分别代表了在各自页表中的下标（取自 xv6 book）：



本实验的第二个要求就是实现 Lazy Allocation，最后所需写的代码较少，不过要想通过所有 usertests 的话最后还应针对下图的六个要求分别做修改才行。

Now you have the basics working, fix your code so that all of usertests passes:

- Handle negative sbrk() arguments.
- Kill a process if it page-faults on a virtual memory address higher than any allo
- Handle fork() correctly.
- Handle the case in which a process passes a valid address from sbrk() to a syste
- Handle out-of-memory correctly: if kalloc() fails in the page fault handler, kill th
- Handle faults on the invalid page below the stack.

Lazy Allocation 主要的实现在 kernel/trap.c 中，在 trap 中需要我们正确处理因为缺页造成的 trap。要求我们在缺页处理中才进行真正的物理内存分配(kalloc())并插入相应 PTE 到该进程的页表中(mappages())。

4.2 设计方法

对于第一个设计 vmprint 打印出页表内容就不详细介绍其实现方法，大体做法只要按照 PTE 的格式打印出相应值即可，如果遇到 PTE_V 为 1，但是 PTE_X, PTE_W, PTE_R 标志都为 0 的话就代表该 PTE 指向的是一个下级页表而不是实际数据页，对于这种指向下级页表的 PTE，我们需要递归调用 vmprint 来进入该下级页表继续打印。当然其实我们也可以记录当前页表的层级（因为 xv6 是采用的三级页表结构，因此除了第三级页表，前两级页表指向的都是下一级页表的页）。
对于 Lazy Allocation，我们首先需要将 kernel/sysproc.c 中的 sys_sbrk 函数中的 growproc(n) 函数调用删除，与之代替的是直接增加当前进程的内存大小(myproc()->sz)，并返回内存增加之前的大小。
接着需要在 kernel/trap.c 中实现其核心功能。首先在 RISCV 的 scause 寄存器组来存放的是造成当前 trap 的原因，因此我们应在 usertrap 中判断 scause 的值，若 scause 值为 13 或 15，就代表是缺页错误，此时就需要内核为其分配真正的物理内存。通过 kalloc 函数申请新的一页，再将该页的内容初始化为 0。接着需要获取 stval 寄存器中的值，该寄存器中存放了此次造成此次缺页错误的将要访问的虚拟地址 va，获取到该虚拟地址后再调用

PGROUNDDOWN 宏定义将其转换为所在页的首地址，接着我们通过调用 mappages 将 va 指向新分配的物理内存首地址，并将这个映射保存为 PTE 插入到该进程的页表中。
理想情况下，通过在 kernel/trap.c 中对缺页错误的处理我们应该能够正确运行 xv6 了，可要想完成实验，还需修改以下几处地方（分别对应上图中的五点要求）：
第一条要求：handle negative sbrk() arguments.
对于该条要求，只需在 sys_sbrk 中判断当 n 为负数时，正确调用 uvmdealloc 回收内存即可；

第二条要求：Kill a process if it page-faults on a virtual memory address higher than any allocated with sbrk().
对于该条要求，需要在 kernel/trap.c 中处理缺页错误时判断一下虚拟地址 va 是否超过了当前内存拥有的合法内存大小 p->sz，如果超出了这个范围，则应直接 exit(-1)结束该进程；

第三条要求：Handle fork() correctly.
对于该条要求，需要在 kernel/vm.c 中的 uvmcopy 函数中将之前的两个报告页面不错在的 panic 给注释掉，改为 continue；即在 fork 中发现父进程的某个内存缺页，则子进程不用去管，因为如果子进程要访问这些页面，则到时候自然会报缺页错误交给我们已经设计好的 usertrap 函数处理。

第四条要求：Handle the case in which a process passes a valid address from sbrk() to a system call such as read or write, but the memory for that address has not yet been allocated.
对于该条要求，需要在 kernel/vm.c 的 walkaddr 函数中判断下 walk 返回的 pte，如果 pte 为 0 或者 pte 的有效位 PTE_V 为 0，则代表此时要访问的虚拟地址还未分配物理内存，因此需要我们做类似之前在 trap.c 中的 usertrap 函数的操作，即 kalloc 申请新页，之后再调用 mappages 创建 va 到新页的 pte 并插入到当前进程的页表中。

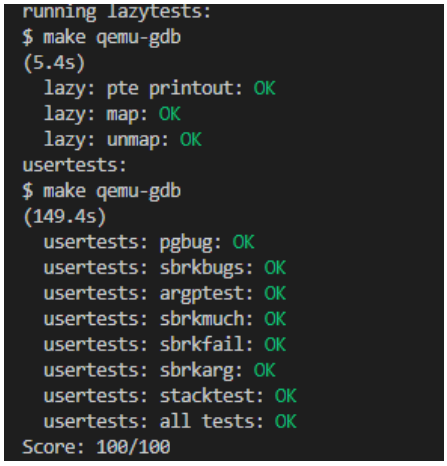
第五条要求：Handle out-of-memory correctly: if kalloc() fails in the page fault handler, kill the current process.
对于该条要求，就是在我们用 kalloc 申请新的物理页内存时，如果 kalloc 返回空指针，即代表申请内存失败，需要直接调用 exit(-1)结束当前进程。

第六条要求：Handle faults on the invalid page below the stack.
对于满足该条要求，我上网查了下别人的做法，最后的操作是在解决缺页错误时，需要判断下虚拟地址 va 是否小于当前进程的 trapframe 中的 sp 指针，xv6 中栈是向下增长的，即 sp 所代表的栈顶实际是栈中地址最小的，因此如果 va 要访问的虚拟地址低于 sp 则需要 exit(-1)结束进程。（对于这第六条的操作我也不太确定，因为根据 xv6 book 中在栈的下面是 data 段和 text 段，如果用户想要访问 data 段的时候不就 va 也小于 sp 了？）

完成以上六个要求的修改后，代码就应该能通过 usertests。

4.3 实验结果

这个实验的 make grade 中第一个测试是关于 vmprint 的，这个测试完全是硬编码检测（就是比较两个字符串是否完全相同，我反复检查了很久才发现是空格的位置不对），因此需要我们格外仔细。最后的结果如下图：



Lab5 Copy on Write Fork for xv6

5.1 内容分析

本实验中要在 xv6 系统中实现 copy on write 的功能，就是当 fork 函数被调用时，系统并不会立即为子进程分配新的物理内存且复制父进程的物理内存中的数据，而是让子进程先暂时就使用和父进程相同的物理内存空间，只不过这部分物理空间全部被标记为“只读”状态，当子进程或父进程需要写入物理内存时才将要写入的物理内存页新复制一份，并将相应的虚拟地址重新映射到该新复制的物理页上，并组成新的 PTE 插入到进程的页表中。

5.2 设计方法

实现 Copy on write（写时复制）功能，就得首先考虑物理内存的多重映射问题，之前的 xv6 的每页物理内存都最多只映射到某一个进程的某一页虚拟页，因此当该进程释放掉该虚拟页时就应该释放掉该物理页。然而当实现写时复制的功能后就会存在分布在不同的进程的多个虚拟页映射同一个物理页的问题，这样的话就应当在最后一个映射到该物理页的虚拟页被释放掉后才能将该物理页释放掉，因此需要为每个物理页记录当前有多少个虚拟页映射到它，当映射数量降为 0 时才应当将该物理页真正地释放掉。

kernel/kalloc.c 是 xv6 内核管理物理内存的文件，该文件中的 kmem 上记录了当前物理内存的使用情况。因此为了实现上述记录功能，在 kmem 中新建一个 uint* ref_count 指针来代表各个物理页的引用数量，例如 ref_count[100]就代表了第 100 页的引用数量。因此我们需要在 kinit()初始化函数中完成对该部分区域的划分和创建，之前的 kmem 管理的物理内存从 end 到 PHYSTOP，添加了 ref_count 后，kem 管理的物理内存范围就变成了从 end+pa_offset 到 PHYSTOP，其中 pa_offset 是计算 PHYSTOP 到 end 一共有多少物理页并且每个物理页占 sizeof(uint)的 ref_count 空间，因此

$$pa\_offset = (((PHYSTOP - (uint64)end) >> 12) + 1) * sizeof(uint)$$

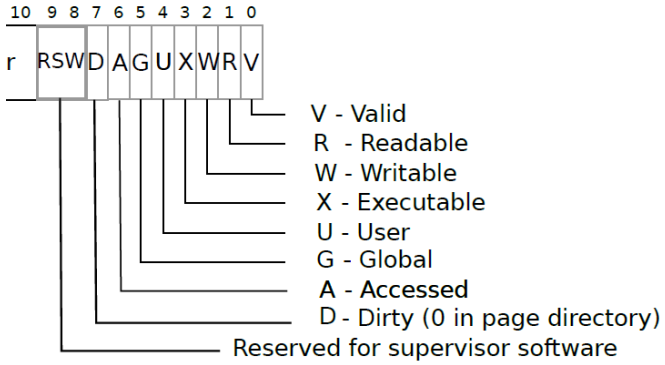
在将存储 ref_count 的内存区域创建出来后，我们还应修改 kalloc.c 中的其它函数以实现 ref_count 的记录引用数功能。

首先是 kfree 函数的修改，之前的 kfree 函数是将传入的物理地址的页添加到 kmem 的 freelist 中，代表该页是空闲的已经可以重新利用。为了实现引用数功能，将原来的 kfree 函数改名为 k_real_free，而新建一个 kfree 函数。新 kfree 函数同样接收一个物理地址 pa，只不过会先计算该物理地址所在页的物理页框号 PPN，再将 ref_count[PPN]减 1，代表该物理页的引用数减 1，之后再判断引用数是否为 0，若为 0 则调用 k_real_free 将该物理地址真正地 free 掉。

新建一个 klink 函数，该函数接收一个物理地址 pa，作用是将该 pa 对应物理页的引用数加 1。并且应在 kalloc 函数中调用一次 klink 函数，代表对新创建的页的引用数加 1。

至此，我们完成了对 xv6 内存管理系统的修改，现在的 xv6 内存管理系统便可支持多个虚拟页映射至同一个物理页的情况，并且当没有虚拟页映射物理页时才将该物理页释放掉。

接下来就是对 fork 过程中涉及到内存的部分的修改，以及页写入错误的处理。首先将 PTE 中未使用的标志位——第 8 位作为 PTE_COW，PTE_COW 若为 1 代表该物理页是被多个虚拟页映射，正处于写时复制的状态。



fork 创建子进程的过程中会调用 uvmcopy 函数(kernel/vm.c)来将父进程的物理内存以及页表复制到子进程。之前的 uvmcopy 函数，对于父进程的每一物理页，都会为子进程分配(kalloc)一个新的物理页，并调用 memmov 将父进程的物理页中的数据复制到子进程的新物理页中，之后再调用 mappages 将新的物理页以及该虚拟地址插入到子进程的页表中，且该新物理页的标志位和父进程的对应物理页保持不变。而在新的 uvmcopy 函数中，不再为子进程分配新物理页，转而代之的是首先将该物理页在父进程页表中的 PTE 项的 PTE_W 清 0，PTE_COW 置为 1，接着直接将父进程的物理页和对应的虚拟地址插入到子进程的页表中，PTE 的 flags 保持和父进程的相同。

copyout 函数(kernel/vm.c)也需要修改，copyout 函数功能是将内核物理内存中（kernel memory)的某个地址 src 的数据传输到某个进程的某个虚拟地址 dstva（user memory）。之前的 copyout 函数做法是首先找到该虚拟地址的虚拟页首地址 va0，再通过 walkaddr 函数在该进程的页表中查找到 va0 对应的物理地址 pa0，再调用 memmov 从 src 传输数据到 pa0 即可。修改后的 copyout 函数应首先取出该 va0 的页表项 pte，再判断页表项的 PTE_COW 是否为 1，若为 1 则代表该 va0 目前的物理页是写时复制状态，需要先将该物理页复制一份再写入。因此在这里 copyout 会首先调用 kalloc 申请新物理页 mem，再将 pa0 中的数据复制到 mem，再将 va0 映射到 mem 上，并创建 PTE_W=1 且 PTE_COW=0,其余标志位同旧物

理页对应 PTE 标志相同的 PTE 插入到进程的页表中。之后便可进行 copyout 操作。并按照上述循环，直到 copyout 将指定长度的数据传输完毕。

最后要修改的就是 usertrap 函数(kernel/trap.c)，在写入一个 PTE_W=0 的页时，会触发 scause=15 的中断，因此在 usertrap 判断 r_scause()是否为 15，若等于 15 就需要进行同 copyout 函数做的那样完成写时复制操作。

在完成上述功能的修改后，xv6 系统便具有了写时复制的功能，理论上在物理内存大小不变的情况能够执行更多次 fork 调用进行子进程的创建。

5.3 实验结果

```
running cowtest:
$ make qemu-gdb
(9.4s)
  simple: OK
  three: OK
  file: OK
usertests:
$ make qemu-gdb
OK (132.0s)
time: OK
Score: 100/100
huhu@huhu-virtual-machine:~/HITSZ2020_OS/xv6-riscv-fall19$
```

Lab6 user-level threads and alarm

6.1 内容分析

本实验要实现的任务有两个，第一个是线程切换，需要我们实现在 user 空间中创建多个线程并在线程之间正确切换的功能。第二个是实现系统定时函数的功能，就是给系统设定一个时间间隔 n 以及函数 fn，让系统每隔 n 个时钟中断就执行一次函数 fn，注意 fn 也是 user function，因此整个过程会经历 user ->kernel ->user(函数 fn) ->kernel ->user 的过程。

6.2 设计方法

6.2.1 uthread

要实现线程的切换首先要保证在进程切换时能够安全地处理寄存器的使用问题，首先需要了解 RISCV 寄存器的结构，如下图：

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

在 RISCV 中，通常会出现 a 调用 b 的情景，这时候 a 就是 Caller 调用方，b 就是 Callee 被调用方，在上图最右边一栏"Saver"代表了对应寄存器由谁来负责保存：

$Caller\ Saved:$ 该寄存器由调用方自己负责保存，被调用方可以修改这些寄存器而不用保护；

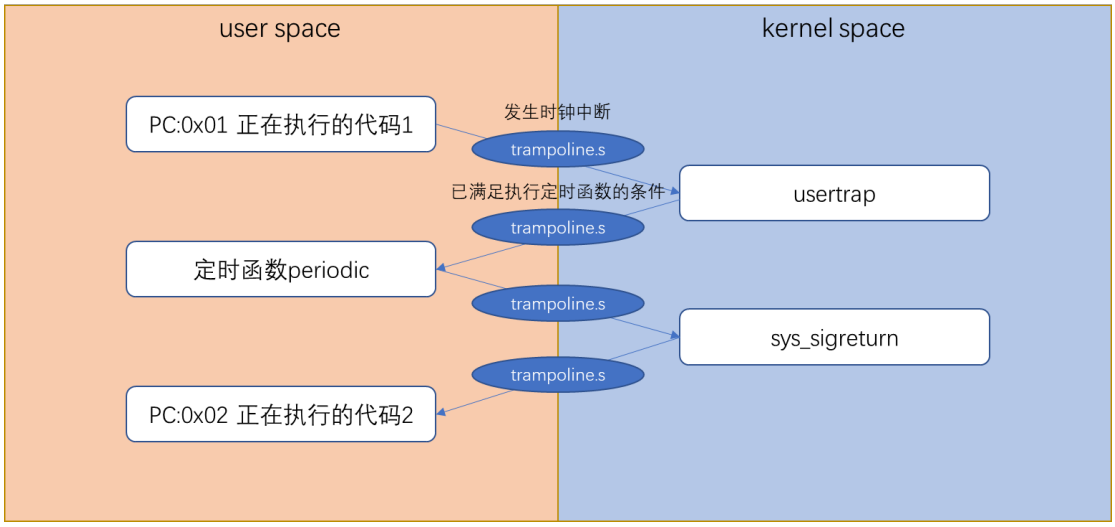
$Callee\ Saved:$ 该寄存器由被调用方负责保存，在退出被调用方时，被调用方需要保证这些寄存器的值没被更改；

通过上面的 RISCV 寄存器知识，我们就知道了在线程切换时应该要保存/恢复哪些寄存器，就是 s0-s11 和 sp，并且还应额外保存/恢复 ra 寄存器的值，这样才能使线程在切换工作完成后能正确地返回到自己应该执行的函数 func 的起始位置。
其实在 xv6 的进程切换的代码中已经有了现成的寄存器保护/恢复的代码，那就是 kernel/proc.h 和 kernel/switch.s，在这两个文件中向我们展示 xv6 在进程切换时是如何保存/恢复 ra, s0-s11 寄存器的，因此我们可以直接借鉴。

因此为了实现线程切换功能，只需要：
在 user/uthread.c 中创建新的结构体 context，其中存放有 14 个 uint64 代表着 14 个要保存的寄存器值，并在 thread 结构体中添加一个 context 属性。
在线程创建的函数 uthread_create 中，借鉴 kernel/proc.c 中的 allocproc 函数，对 context 字段初始化，并将其中的 ra 值设置为 func 函数指针，将 sp 值设置为该线程的 STACK 空间的栈底。
在线程切换的函数 thread_schedule 中，添加调用 thread_switch 的代码，传入的两个参数将会被保存在 a0 和 a1 寄存器，第一个参数是当前线程的 context，第二个参数是下一个线程的 context，代表从当前线程切换到下一个线程。
最后要完成 thread_switch.s 的汇编代码，该文件完成将 ra,sp,s0-s11 寄存器的值存储在当前线程的 context 中，并将下一个线程的 context 中的 ra,sp,s0-s11 的值取出到对应寄存器中。该部分代码直接借鉴 kernel/switch.s 即可。

6.2.2 alarm

第二个任务的代码量较少，重要的是理解整个 alarm 执行的过程，示意图如下：

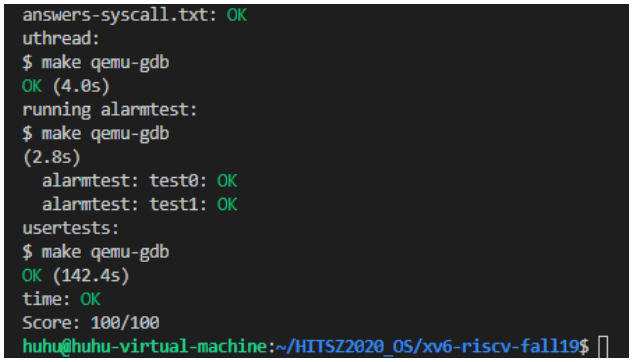


trampoline.s 完成用户模式和内核模式之间的转换。每次发生用户模式和内核模式转换的时候，trampoline.s 会将 user space 的相应的寄存器的值保存到当前进程的 trapframe 中，trapframe 还保存有四个 kernel 开头的内核寄存器的值，当切换到内核模式时候会将这四个寄存器加载到对应寄存器中，以此完成由用户模式到内核模式的切换，这里需要注意 satp 是保存一级页表首地址的寄存器，当 satp 切换后就代表使用的虚拟地址不再是 user space 了而是 kernel space（xv6 中 kernel 和 user 的虚拟地址空间是分开的，不像 Linux 是 kernel 和 user 共存于同一个虚拟地址空间）。从内核模式到用户模式的操作差不多，就是保存四个 kernel 开头的寄存器的值，再加载之前保存的 trapframe 中的用户寄存器的值到相应寄存器。

执行整个定时函数的过程如下，当一个时钟中断发生时，会由用户模式切换到内核模式并执行 usertrap，在 usertrap 中将一个时钟计数加 1，如果时钟计数满足设定的时钟间隔，在此次 usertrap 处理过程中就会保存当前的 trapframe 到 old_trapframe 中，并之后返回到用户模式中定时函数的首地址并执行定时函数，定时函数执行完毕后调用 sigreturn 进入内核模式，内核模式下 sys_sigreturn 将之前的保存的 old_trapframe 恢复到 trapframe 中并将时钟计数重置为 0 以等待下次再次执行定时函数，并随后返回到用户模式中上一次被中断的用户指令处，继续执行下一条用户指令。

按照上面的过程在 proc.h 的 struct proc 中增加一个 old_trapframe 的结构体，并且设置一个当前时钟计数 ticks 和时钟间隔 tick_interval，以及一个代表是否正在执行定时函数的标志位 inhandler(防止在执行定时函数时，被时钟中断导致进入新的定时函数)，同时还设置一个保存定时函数的首地址指针 handler。设置完相应结构体后，就可以按照上面描述的过程去修改 usertrap 函数和相应 sys_sigalarm 和 sys_sigreturn 函数即可。

6.3 实验结果



Lab7 locks

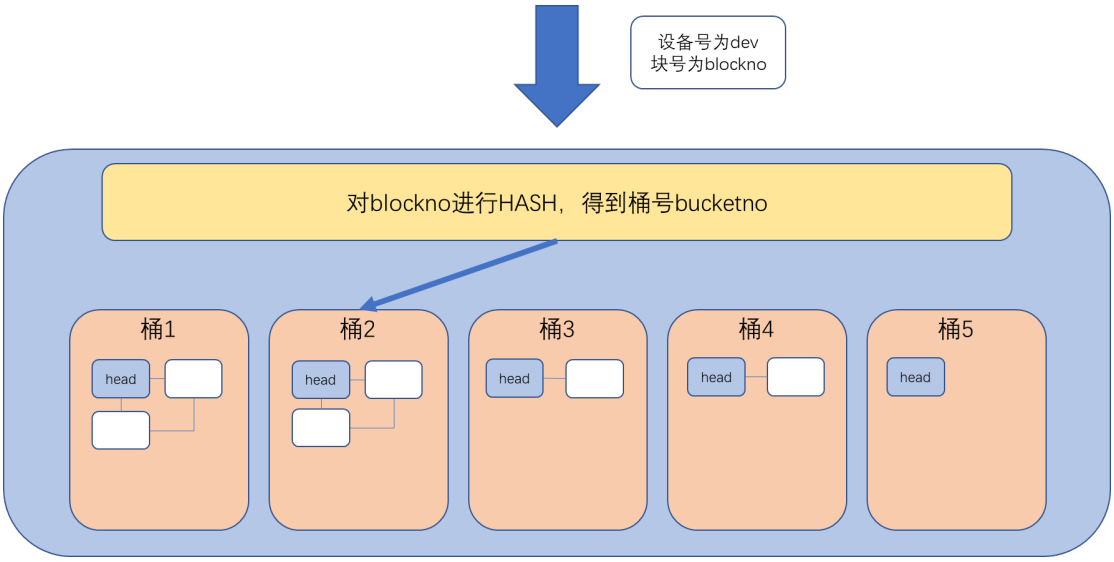
7.1 内容分析

本实验要求解决由于锁争用而造成 xv6 系统并行化程度不高的问题。主要有两处锁争用场景需要改善，一是 kernel 中内存管理的 kmem 同一时间只能供一个 cpu 使用，二是磁盘块缓冲区 buffer cache 同一时间也只能供一个进程使用。

7.2 设计方法

解决 kalloc 的锁争用，只需要为每个 cpu 都分配一个 kmem 即可，即每个 cpu 拥有一个自己的内存空闲页的链表 freelist，并且每个 freelist 对应一个 spinlock，即每个 cpu 的 kmem 对应一个 lock。当一个 cpu 的 freelist 没有空闲页的时候就到其它 cpu 的 freelist 中去获取。按照上面的思想去修改 kernel/kalloc.c 中的对应的代码即可通过 kalloctest。

kernel/bio.c 文件是 xv6 系统中管理硬盘块缓存的地方，其中 bcache 拥有一片缓存区域 buf 以及一个锁 lock，同时还需注意 bcache 将 buf 组织成一个双向循环链表，该链表有一个头结点 head 方便搜索 buf。按照之前的结构，多个进程同时想要使用 bcache 的话，必须抢用一个 lock，因此效率非常的低。为了提高并行程度，将 bcache 从单独一个链表(一个 head)改变为用多个链表(多个 head)，即多个 bucket。例如下图：



每当一个进程想要获取设备号为 dev，块号为 blockno 的磁盘块时，bcache 会将 blockno 进行哈希计算，计算出该块号对应的桶号，并申请该桶的 lock，获取到该桶的 lock 后就可在其中的双向链表中搜索，看 bcache 是否已经缓存了该块。如果缓存有该块（dev 和 blockno 都符合，则使块的引用数加 1，返回该块；如果在桶中没有找到，则就去寻找一个空闲的缓存块 buf 作为该磁盘块的缓存。如果自己桶中没有空闲的 buf（refcnt 等于 0），就去其它桶寻找空闲块，将找到的空闲块从原来的双向链表中分离出来，插入到自己桶双向链表的头部。

同时当用户想要释放掉一个块时，也是先根据 blockno 哈希计算出 bucketno 再到对应桶中搜索对应的块，并使该块 refcnt 减 1，如果 refcnt 减为 0 则将该空闲块从双向链表中取出再插入到头部。注意，搜索空闲块从 head->next 开始从前往后搜索，搜索是否已经存在的块则是从 head->prev 开始从后往前搜索。

7.3 实验结果

```
running kalloctest:
$ make qemu-gdb
(33.1s)
  kalloctest: test0: OK
  kalloctest: test1: OK
running bcachetest:
$ make qemu-gdb
(52.4s)
  bcachetest: test0: OK
  bcachetest: test1: OK
usertests:
$ make qemu-gdb
OK (143.4s)
time: OK
Score: 100/100
huhu@huhu-virtual-machine:~/HITSZ2020_OS/xv6-riscv-fall19$
```

Lab8 file system

8.1 内容分析

本实验要求有两个：一是增大 xv6 文件系统中的 file 最大大小限制；二是实现软连接 symbolic link 功能；
在操作系统中软连接和硬链接的区别主要是，硬链接不会创建 inode，只会将 inode 的 link 数加 1，软连接会新建一个 inode，相当于新建一个文件，这个文件的类别是 SYMLINK，类似于我们日常使用电脑中的快捷方式那样；并且硬链接只能是在同一个设备中创建链接，而软连接可以跨设备进行连接。

8.2 设计方法

8.2.1 Large files

在操作系统中，文件由 inode 及相关的数据块组成，例如下图（取自 xv6 book）

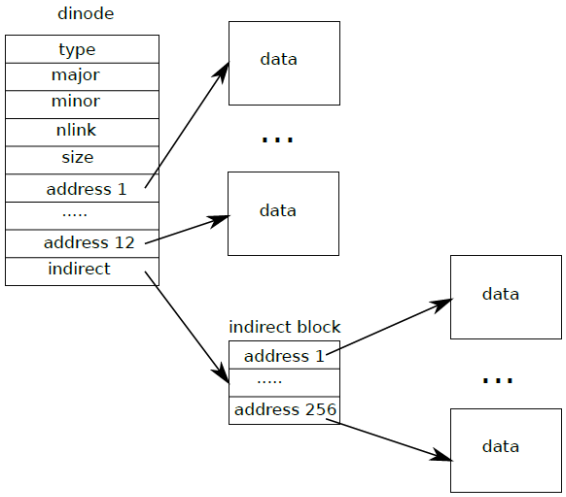


Figure 7.3: The representation of a file on disk.

在 xv6 中，dinode 是 inode 在硬盘中的结构体，inode 是 inode 中内存中的结构体。
对于第一个实验，我们要考虑的是 inode 中的 addrs 属性，addrs 记录了文件各个数据块的块号，其中最后一块数据块被 xv6 用作一个目录块，意思是其中存储了文件其余的数据块的块号。inode 中 addrs 长度为 13，前 12 块为直接数据块，第 13 块是目录块。而 xv6 一个磁盘块的大小 1KB，而一个块号占 4B，因此目录块中最多存储 256 个块号，在这种设计下，xv6 中一个文件最大大小为 12+256=268 块。

为了增大文件的最大容量，选择前 11 块作为直接数据块，第 12 块作为目录块，第 13 块作为一级目录块，一级目录块的含义是其中存储的 256 个块号指向的是 256 个二级目录块，因此在这种设计下，xv6 中一个文件最大大小就可以达到 11+256+256*256=65803

块。
在了解以上思路后，其实在文件需要修改的地方主要就两处，kernel/fs.c 的 bmap 函数和 itrunc 函数。当然首先应修改 kernel/fs.h 中的 NDIRECT 变量从 12 改为 11。

bmap 函数的功能是在给定文件 inode 的条件下，返回第 n 个数据块的块号。首先应在原来 bmap 函数的基础上增加对 n 在第 267 到第 65802 块（从第 0 块起）的判断，若是 267~65802 块，则首先应取出 addrs 中最后一块的块号，当然如果最后一块不存在就为其分配，调用 bread 函数从磁盘中读出该块的内容，再计算此数据块的所在二级目录块的位置，若二级目录块不存在也为其分配，调用 bread 读取内容，最终在二级目录块计算数据块的位置，最终读取出该块的块号，当然若数据块不存在也是需要为其分配。

itrunc 函数的功能是释放掉一个文件在磁盘中的所有数据，在实验中需要在 itrunc 函数中增加对二级目录块结构的释放，大致操作是如果一级目录块存在，就取出其内容再遍历其中的每个存在的二级目录块，再读取二级目录块内容，遍历其中每个存在的数据块块号，调用 bfree 释放掉该数据块，遍历结束后再 bfree 该二级目录块，当所有二级目录块遍历结束后就 bfree 掉一级目录块。

8.2.2 Symbolic links

首先需要在 xv6 系统中注册 symlink 系统调用函数（包括在 user/user.h, user/usys.pl, kernel/syscall.c 等一系列文件中添加有关 symlink 的代码）。并在 kernel/stat.h 中添加新的 inode 类型 T_SYMLINK，同时在 kernel/fcntl.h 中添加 O_NOFOLLOW 的 open flag 标志（用于在 open 文件时传入的 flag），O_NOFOLLOW 的作用是告诉 open 函数，如果将要打开的文件是 SYMLINK 类型，不需要进行递归打开，即不需要打开软连接所指向的文件，打开当前 SYMLINK 文件即可。

对于 sys_symlink 函数，该函数结束两个字符串 target 和 path（都是文件路径），其中 target 是 symlink 将要指向的文件所在路径，而 path 则是新创建的 SYMLINK 文件的所在路径。函数大体思路是，首先调用 create 函数从 path 中创建新文件 inode，接着调用 balloc 为新文件申请一个数据块，并将块号存入 addrs[0]中，接着将 target 字符串通过 memmov 存入该数据块中即可。

接着就是要修改 sys_open 函数，在 open 函数中需要对 inode 的 type 以及传入的 open flags 进行判断，如果 inode 类型是 SYMLINK 类型并且传入的 flags 中 O_NOFOLLOW 标志位为 0，代表着需要进行递归查找，找到软连接真正连接到的文件。

递归查找的函数为 find_true_inode，该函数接受一个未被 lock 的 inode，以及一个递归深度 depth，设置递归深度的作用防止死循环，当 depth 大于 10 的时候便直接返回 0，例如 A->B，B->A，则 open 就可能由于找不到不是 SYMLINK 类型的文件而无限递归下去。在 find_true_inode 中，首先读取当前 inode 的第一个数据块，（因为之前在 SYMLINK 类型文件的第一个数据块中存储了目标文件的所在路径 target），取出第一个数据块中的字符串数据，再调用 namei 函数从文件路径中读取出目标文件的 inode 块，接着需要从目标文件的 inode 块中判断文件类型，如果文件类型依然是 T_SYMLINK，则需要再次调用 find_true_inode 并

将 depth 加 1，如果文件类型不是 T_SYMLINK 类型就直接返回该文件的 inode。

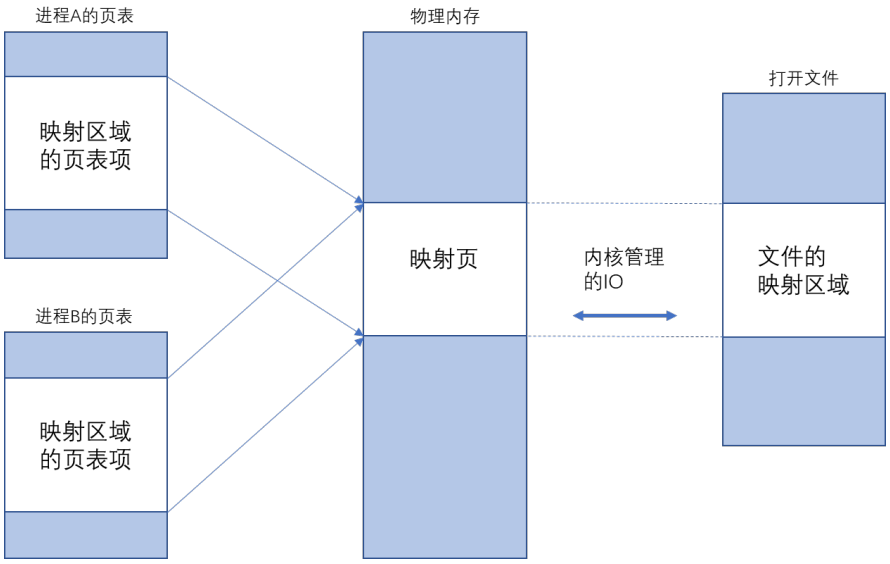
8.3 实验结果

```
running bigfile:
$ make qemu-gdb
OK (196.8s)
running symlinktest:
$ make qemu-gdb
(1.0s)
  symlinktest: symlinks: OK
  symlinktest: concurrent symlinks: OK
usertests:
$ make qemu-gdb
OK (270.2s)
time: OK
Score: 100/100
huhu@huhu-virtual-machine:~/HITSZ2020_OS/xv6-riscv-fall19$
```

在 grade-lab-fs 文件中，我将 bigfile 测试的 timeout（最大运行时间）从 180 调成了 300，可能是因为虚拟机配置的内核数不够导致运行时间超时？总之之前的 bigfile 测试未能通过，调整时间限制后便能通过全部测试。

Lab9 mmap

9.1 内容分析



本实验要求我们实现 UNIX 系统中的 mmap 内存映射功能，具体作用是将磁盘中的文件指定区间的数据直接映射到某个进程的虚拟空间中，并且允许同一个文件的指定数据空间映射到多个进程的虚拟地址，这些虚拟地址可以共享同一片物理内存也可以不必。

9.2 设计方法

在本实验中最后采用的设计是在 fork 时，父子进程的 VMA 映射使用的是同一片物理内存，也即是上图中的情况。在这种设计中就需要像之前 Lab5 Copy on Write 做的那样，为物理内存中的每页设置一个引用数 ref_count，每当有一个进程有该页的引用，就需要将 ref_count 加 1，而每次释放一个物理内存也只是让 ref_count 减 1，最后当 ref_count 为 0 时才真正 kfree 掉该物理页。当然这部分 kernel/kalloc.c 的内容不是本实验的重点，因此不详细介绍。

本实验重点是要实现两个系统调用函数 sys_mmap 和 sys_munmap，其中 sys_mmap 函数负责将指定文件 fd 全部内容映射到当前进程的某片虚拟地址空间中（本实验中 mmap 的参数 addr 和 offset 均为 0），代表着 sys_mmap 自己决定所要存放的虚拟地址空间区域，并且一次映射就是将整个文件的数据空间进行映射。sys_munmap 则是将给定虚拟地址 addr 以及长度 len 的虚拟空间的 VMA 映射给取消掉，并且如果某个文件的映射的空间 VMA 所占的虚拟页全部被 unmap 掉时，还需将 VMA 结构给 free 掉。下面介绍本实验中的 VMA 结构。

VMA(virtual memory area)结构就是负责记录一个进程的虚拟地址空间使用情况的结构体，在本实验中为每个进程设置了 16 个 VMA 结构体（kernel/proc.h struct proc），VMA 结构体各字段属性如下：

```
// VMA 结构体
struct VMA {
  int used; // used=1 代表已经占用
  uint64 addr;
  int length;
  int flags;
  int prot;
  int npages; // page 的数量,当 page 数量减为 0 时就需要 file->ref 减 1
  struct file* file;
};
```

used 代表该 VMA 结构体是否已被使用；addr 代表该 VMA 所记录的虚拟地址首地址；length 代表该 VMA 所用虚拟地址的长度；flags 是 MAP_SHARED 或 MAP_PRIVATE；prot 是 PROT_READ 或 PROT_WRITE；npages 是为该 VMA 分配的虚拟页数量，在 munmap 中每 unmap 一页（无论有没有为该页分配真正的物理页），npages 减 1，当 npages 为 0 时，就应关闭该文件（fileclose）；最后 file 是指向所映射文件的指针。

接下来介绍 sys_mmap 主要实现思路。首先获取到传入的 6 个参数，分别是不会用到的地址 addr，映射区域长度 length，读取写入权限 prot，共享/私有映射 flags，文件描述符 fd，不会用到的偏移量 offset。接着根据文件描述符在当前进程 proc 中获取到该文件的 file 结构体指针。接着，从当前进程的静态 VMA 数组中找到第一个 vma.used=0 的空闲 VMA，将该 VMA 的 used 设置为 1，并赋值其余 addr、length、flags 等字段属性。接着调用 filedup 将文件的引用数加 1。最后返回 vma 的虚拟首地址 addr。

这里重点强调关于 VMA 中 addr 属性如何赋值，即如何决定将文件空间映射到哪个虚拟地址上。在 xv6 中最大虚拟地址 MAXVA 为 1<<38，而为了避免和程序中一般会使用到的虚拟

地址冲突，将 VMA 所管理的虚拟地址首地址 VMASTART 设置为 1<<37，且设置每个 VMA 最大的虚拟地址空间大小 VMASIZE 为 1<<20，因此我们最大可以分割出 1<<17 个 VMA，但实际上在实验中为每个进程只分配了 16 个 VMA，因为 xv6 中允许同时打开的文件数量最大就是 16 个，因此 16 个 VMA 在本实验中已经足够。

接下来介绍 sys_munmap 主要实现思路。munmap 接受一个虚拟地址 addr 和长度 length，首先将 addr 减去上面介绍的 VMASTART 再除以 VMASIZE 就得到了对应的 VMA 在当前进程 VMA 数组中的下标，得到 VMA 后，便从 addr 所在虚拟页的首地址开始遍历每一虚拟页（直到长为 length 的虚拟空间全部被覆盖）：对于虚拟页 va，调用 walk 函数得到 PTE，接着将 VMA 中 npages 减 1 代表 unmap 了 1 页，接着判断 PTE 是否有效，如果有效则得到其中的物理地址 pa，如果 MAP_SHARED 在 VMA 中的 flags 被设置，那么就需要将该物理页写回磁盘（具体操作不介绍）。在上述遍历完成后，检查 VMA 的 npages 是否为 0，若为 0 则还需要调用 fileclose 函数将文件的引用数减 1。

在 mmap 和 munmap 函数实现完毕后，还需在 trap 中处理缺页错误，因为对于某个文件第一次被映射时，mmap 是不会立即分配物理内存空间的，需要在缺页处理中申请新物理页，并从文件中的指定位置读取一页数据到该新物理页，具体操作和在 Lab4 Lazy Allocation 中的 usertrap 函数相同，因此不详细介绍，只不过在本实验中 usertrap 多了一个调用 readi 从磁盘中读取 PGSIZE（1 页）的数据到新建立的物理页 mem 中而已。

要通过本实验全部测试，只剩下一些细枝末节的工作。例如还需要在 fork 创建子进程时将 VMA 映射所覆盖的物理内存的每一页的引用数 ref_count 加 1；还需要在 exit 时候将这些 ref_count 减 1，如果减为 0 则直接 free 掉对应物理页。

9.3 实验结果

```
running mmaptest:
$ make qemu-gdb
(3.9s)
  mmaptest: mmap_test: OK
  mmaptest: fork_test: OK
usertests:
$ make qemu-gdb
OK (35.9s)
time: OK
Score: 100/100
huhu@huhu-virtual-machine:~/HITSZ2020_OS/xv6-riscv-fall19$
```

Lab10 networking

10.1 内容分析

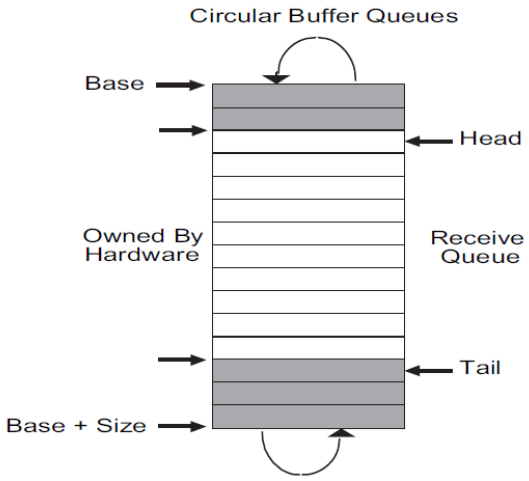
本实验主要要求有两个：一是完成 E1000 网卡驱动程序的接受和发送功能；二是完成 socket 套接字的接受 udp，读取，写入，关闭等功能。

本实验按照官网给的提示 hints 以及阅读 E1000 网卡驱动的 datasheet 中 3.2.3, 3.2.6, 3.3.3, 3.3.4 等章节基本上可以完成。

10.2 设计方法

10.2.1 Network device driver

要实现 E1000 网卡驱动程序中接受和发送，除了借鉴 Hints 之外还需了解以下 E1000 驱动知识：



在 E1000 驱动中，每接受或传送一个数据帧都是在一个循环缓冲队列中（所以有两条队列，接受缓冲队列和发送缓冲队列），也就是上图中的结构。该缓冲队列中每一条记录都是一个描述符 Descriptor（Receive Descriptor 和 Transmit Descriptor），描述符中记录了接受数据或发送数据所要用到的相关信息。该循环缓冲队列主要有两个指针 Head 和 Tail，其中 Head 和 Tail 之间的区域的描述符(白色区域)代表着还未被准备好的描述符（代表该描述符为空或者该描述符相关的数据帧正在发送中（on the flying)），而 Tail 到 Head 之间的灰色区域则是代表已经被硬件接收完毕或者已经准备好相关数据就等软件处理的描述符。

接受描述符如下图：

	63	48 47	40 39	32 31	16 15	0
0	Buffer Address [63:0]					
8	Special	Errors	Status	Packet Checksum (See Note)	Length	

Buffer Address 就是将接受的数据所要存放的物理内存地址，Status 字段则是存放了该数据帧以及该描述符的相关状态（例如该描述符是否已被使用或者该描述符是否已经被硬件处理好可以使用）。下图是 Status 字段详细结构图：

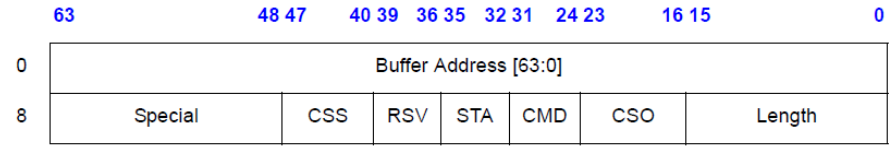
Table 3-2. Receive Status (RDESC.STATUS) Layout

7	6	5	4	3	2	1	0
PIF	IPCS	TCPCS	RSV	VP	IXSM	EOP	DD

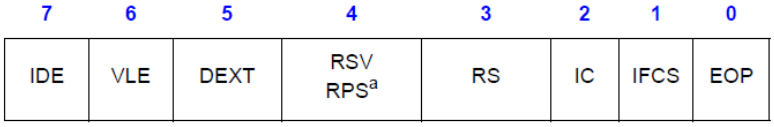
这里需要重点说明的是一个描述符只有当它的数据被硬件接收完毕后才应该调用 E1000 驱

动程序的 e1000_recv 函数来进行处理，而每当一个数据帧的数据被硬件接收完毕，即可以对其进行接收处理时，在 Receive Descriptor 中 Status 字段的 DD 标志位将会被置为 1。

发送描述符如下图：



Buffer Address 是已经准备好的数据物理内存首地址，在本实验中我们还需了解的字段是 STA 和 CMD，STA 就是 Status 字段，其中的 DD 标志位依然是待该描述符已经被硬件处理完毕就会被置为 1。CMD 是在发送过程中的一些参数设置，CMD 字段结构如下图：



在网卡驱动的 datasheet 中有每个标志位的详细说明，在本实验中我们只需要了解 RS 字段和 EOP 字段，RS 标志位是告诉硬件，当准备好该发送描述符后需要将 Status 字段中的 DD 标志位置为 1，由于我们实验中要用到 DD 标志位来判断每个发送描述符是否准备好，因此 RS 需要被置为 1；EOP 标志位是代表着该数据帧是该网络层数据包的最后一帧，在本实验中我们并未涉及到网络数据包的分组，因此传输的每帧都是数据包的最后一帧，因此 EOP 需要被置为 1。

还需要说明的是，在网卡驱动中为每个文件符还配置了一个缓冲区指针，例如下述代码：

```
#define RX_RING_SIZE 16
static struct rx_desc rx_ring[RX_RING_SIZE] __attribute__((aligned(16)));
static struct mbuf *rx_mbufs[RX_RING_SIZE];
```

接受缓冲队列 rx_ring 中每个文件描述符都对应一个 mbuf 的指针，指向数据 mbuf 的存储地址，在 mbuf 中我们可以给数据加上或删去各种协议的头字段，mbuf 的功能就是方便加载和删除各层协议的头字段的。

在了解完以上网卡驱动知识后，我们便可按照实验 Hints 的提示完成第一部分的实验。对于 e1000_recv 接受数据，大致思路就是，从接受缓冲队列中根据尾指针 Tail 加 1 再 mod 队列长度 RX_RING_SIZE 得到下一个要处理的数据帧的接收描述符 tail_desc，再判断该描述符中 DD 标志位是否为 1，（如果不为 1 则结束函数）如果为 1，再获取到该描述符的 mbuf，将该 mbuf 的长度调整为描述符中描述的数据长度，再调用 net_rx 交给上层协议去继续处理数据并接收即可。此时 mbuf 所代表的物理内存就代表已经被使用，因此我们还需为文件描述符分配新的接收内存区域，之后再将描述符中的 addr 地址指向新的内存区域，且描述符的 DD 字段清空为 0 即可。关于 e1000_recv 还需注意的是，在接收完一个数据帧后还需继续扫描接收缓冲队列 rx_ring，继续获取下一个文件描述符，重复执行上述操作即可（直到遇到的描述符 DD 字段为 0）。

对于 e1000_transmit 发送数据，大致思路和 e1000_rec 相似，这里不再赘述。

10.2.2 Network sockets

本实验中实现 socket 套接字功能需要完善四个函数的功能，在 kernel/sysnet.c 中的 sockrecvudp, sockclose, sockread, sockwrite，首先需要了解的是 socket 套接字的结构体 sock:

```
struct sock {
  struct sock *next; // the next socket in the list
  uint32 raddr;      // the remote IPv4 address
  uint16 lport;      // the local UDP port number
  uint16 rport;      // the remote UDP port number
  struct spinlock lock; // protects the rxq
  struct mbufq rxq;  // a queue of packets waiting to be received
};
```

在 xv6 中对于所有打开的 socket 都存储在单向链表 sockets 中，sock 结构体中 next 指针就是指向该链表中的下一个 socket。在 xv6 系统中对于 socket 也是作为文件进行表示的，其文件类型为 FD_SOCK，在 fileread 和 filewrite 中若判断出文件类型为 FD_SOCK 就需要调用 sockread 和 sockwrite。

在结构体 sock 中，raddar 代表目标 IP 地址，lport 代表该 socket 会话的本地端口，rport 代表目的主机上的端口，lock 是保护该 sock 的自旋锁，rxq 是 mbuf 的队列结构，其中每当 socket 接收到一个 packet 时，其数据 mbuf 就会被压入到队列 rxq 中等待被读取。

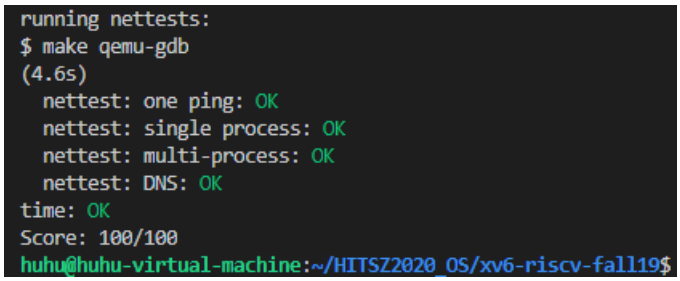
对于 sockrecvudp 函数，接收一个存放数据的 mbuf，以及远程主机 IP 地址 addr，本地主机上的端口 lport，远程主机上的端口 rport。首先应该去存放有所有打开的 socket 的单向队列 sockets 中寻找正确的 socket 来接收该数据包，即 addr, lport, rport 三个值都匹配上的 socket，找到正确的 socket 后调用 mbufq_pushtail 函数将该 mbuf 压入到 socket 的 mbuf 队列 rxq 中即可，最后还需调用 wakeup 函数唤醒可能因为执行 read 等待数据包到来而进入睡眠状态的 socket。

对于 sockclose 函数，大致思路就是将传入的参数 sock* si，在队列 sockets 中找到其父节点，让父节点的 next 指针指向 si 的 next。再对 si 的接收队列 rxq 中还未被读取的 mbuf 调用 mbuffree 释放掉内存空间即可。最后调用 kfree 释放掉该 sock 的内存空间。

对于 sockread 函数，主要就是读取传入的参数 sock*si 的接收队列 rxq 中的 mbuf，如果 rxq 队列为空就进入睡眠状态，如果非空，就取出 rxq 第一个 mbuf，将该 mbuf 所指向的数据读取到给定的虚拟地址上。

对于 sockwrite 函数，就是新建一个 mbuf，并在该 mbuf 中创建足够大的空间 n，再调用 copyin 函数从给定虚拟地址上读取 n 字节的数据到新建的 mbuf 中，最后调用 net_tx_udp 函数将该 mbuf 通过 socket 发送出去。

10.3 实验结果



GitHub 地址

本实验完整的 xv6 项目代码地址为：https://github.com/huzhisheng/xv6-riscv-fall19

OS 实验改进建议

对于之后搭建 HITSZ 的 OS 实验平台，我的建议是尽量给出如 xv6 book 一样的，对整个 HITSZ 的 OS 实验平台的架构以及实现细节进行介绍的中文参考书。在做 xv6 实验的过程中，我们需要去阅读 xv6 book，尽管其中的英文词汇或者语法不是很复杂，但往往给我一种在读后面的内容时就感觉把前面的内容差不多忘记了的感觉，这就要求我做一些花花绿绿的笔记来方便自己之后再次查阅，我想如果能有一本 xv6 中文参考书那么也会使实验做得更加快，比如我在做第 8 个 xv6 实验 fs 时，就花了大概两天去阅读 xv6 book 的 Chapter7 File System，然而写代码就不需要这么久。(只是个人建议)
同时我也建议在设置实验题目顺序的时候能合理考虑课程的教学进度，例如在我第一次实验中做实现 find 指令的时候，就因为还没学过文件系统，所以对 dirent 结构体，fstat 函数等相关概念非常陌生，导致当时做这个实验就感觉很困难。