深 圳 大 学 实 验 报 告

课程名称：操作系统

实验项目名称：综合实验 1——第一部分

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：谭舜泉

报告人：黎浩然 2018112061 冯海月 2018191116

实验时间：2021 年 3 月 9 日-2021 年 4 月 25 日

实验报告提交时间：2021 年 4 月 25 日

教务部制

实验目的与要求：

实验目的：

(1)、掌握计算机操作系统管理进程、处理机、存储器、文件系统的基本方法。

(2)、了解现代计算机操作系统的工作原理，具有初步分析、设计操作系统的能力。

(3)、通过阅读 xv6 操作系统代码，理解其是如何实现操作系统中的各种管理功能，在系统程序设计能力方面得到提升。

实验要求：

(1)、阅读“xv6 中文文档” 第 0 章：操作系统接口，回答以下问题：

a）xv6_rev7.pdf 中，8024-8026 行.

```
if(fork1() == 0)
```

为什么 fork1()返回值为 0 时才进入 if 语句内部？

```
runcmd(parsecmd(buf));
```

阅读 runcmd 的代码，其中：

$echo README 对应的 cmd->type 是哪个？

相应的，$ls; echo “hello world“ 对应的 cmd->type 是哪个？

而 ls | wc 对应的 cmd->type 是哪个？给出你的答案，并从代码中给出解释。

b）阅读 xv6_rev7.pdf 中 7930 行对应的 switch 分支及相关代码，请说明如何确保 rcmd->fd 为标准输入的呢？

c）sh.c 中第 7950 行至 7972 行：

第二、三个 if 语句中，管道的读端口和写端口都通过 close 语句关闭了，请问还怎么保证 pcmd->left 的输出进入管道的写端口，而 pcmd->right 的输入进入管道的读端口？

为什么在父进程这里，还需要有两个 close 语句？以及两个 wait 语句？

（2)、阅读“xv6 中文文档” 第 1 章：第一个进程，回答以下问题：

a） “xv6 的地址空间结构有一个缺点， 即无法使用超过 2GB 的物理 RAM”——请给出解释。为什么 xv6 的内存空间只能有 2GB?

b）(“xv6 中文文档”第 15 页）“这个映射就限制内核的指令+代码必须在 4mb 以内。 ”——请给出解释。为什么？

c）请问 initcode.S 所触发 exec 系统调用执行了哪个程序，而那个程序又是实现什么功能的呢？

（3)、阅读“xv6 中文文档”附录 A/B：PC 硬件及引导加载器，回答以下问题：

阅读 bootasm.S，查找资料，回答以下问题：

a）为什么主引导记录要存放在 0x7C00 开始的内存地址？（提示：这是历史遗留问题）

b）bootasm.S 第 21 行，“# Physical address line A20 is tied to zero…”这是著名的 Gate-A20，请介绍一下为什么要设定 Gate-A20。

c）bootasm.S 第 21 行-第 38 行，这是一段让人一头雾水的代码，请查找资料，解释一下这段代码为何和 enable A20 有关。

（参考 https://www.win.tue.nl/~aeb/linux/kbd/A20.html）

说明：

（1）本次实验课作业满分为 100 分，占总成绩的比例（待定）。

（2）本次实验课作业截至时间 2021 年 4 月 25 日（周日）23:59。

（3）报告正文：请在指定位置填写，本次实验不需要单独提交源程序文件。

（4）个人信息：WORD 文件名中的“姓名”、“学号”，请改为你的姓名和学号；实验报告的首页，请准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“实验报告提交时间”等信息。

（5）提交方式：请在 BLACKBOARD 平台中按时提交；延迟提交不得分。

（6）发现抄袭（包括复制&粘贴整句话、整张图），该次作业记零分。

（7）期末考试阶段补交无效。

(1)、阅读“xv6 中文文档” 第 0 章：操作系统接口，回答以下问题：

a）xv6_rev7.pdf 中，8024-8026 行.

if(fork1() == 0)

为什么 fork1()返回值为 0 时才进入 if 语句内部？

runcmd(parsecmd(buf));

阅读 runcmd 的代码，其中：

$echo README 对应的 cmd->type 是哪个？

相应的，$ls; echo “hello world“ 对应的 cmd->type 是哪个？

而 ls | wc 对应的 cmd->type 是哪个？给出你的答案，并从代码中给出解释。

在 if(fork1() == 0)中为什么 fork1()返回值为 0 时才进入 if 语句内部：

fork() == 0 表示创建一个进程并在该进程中执行 runcmd(parsecmd(buf));那么为什么要在一个新创建的进程中执行 runcmd(parsecmd(buf));呢

以单行命令执行为例，观察 runcmd 函数的实现：

```
56  // Execute cmd.  Never returns.
57  void
58  runcmd(struct cmd *cmd)
59  {
60    int p[2];
61    struct backcmd *bcmd;
62    struct execcmd *ecmd;
63    struct listcmd *lcmd;
64    struct pipecmd *pcmd;
65    struct redircmd *rcmd;
66
67    if(cmd == 0)
68      exit();
69
70    switch(cmd->type){
71    default:
72      panic("runcmd");
73
74    case EXEC:
75      ecmd = (struct execcmd*)cmd;
76      if(ecmd->argv[0] == 0)
77        exit();
78      exec(ecmd->argv[0], ecmd->argv);
79      printf(2, "exec %s failed\n", ecmd->argv[0]);
80      break;
```

在 EXEC 中，runcmd 最终调用 exec()来执行在 shell 中指定的程序，看看 exec 的实现：

```
10  int
11  exec(char *path, char **argv)
12  {
13    char *s, *last;
14    int i, off;
15    uint argc, sz, sp, ustack[3+MAXARG+1];
16    struct elfhdr elf;
17    struct inode *ip;
18    struct proghdr ph;
19    pde_t *pgdir, *oldpgdir;
20
21    if((ip = namei(path)) == 0)
22      return -1;
23    ilock(ip);
24    pgdir = 0;
25
26    // Check ELF header
27    if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
28      goto bad;
29    if(elf.magic != ELF_MAGIC)
30      goto bad;
31
32    if((pgdir = setupkvm()) == 0)
33      goto bad;
34
35    // Load program into memory.
36    sz = 0;
37    for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
38      if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
39        goto bad;
40      if(ph.type != ELF_PROG_LOAD)
41        continue;
42      if(ph.memsz < ph.filesz)
43        goto bad;
44      if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
45        goto bad;
46      if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
47        goto bad;
48    }
49    iunlockput(ip);
50    ip = 0;
```

通过代码及知识容易知道，exec 会将可执行文件加载到当前进程的虚拟内存中，替换掉或重新设置当前进程映像在虚拟内存中的内容。Exec 会在确认该调用可以成功后才释放掉原来的内存映像。一旦新的程序映像被成功加载，exec 会进行最后的一些设置：

```
85    // Commit to the user image.
86    oldpgdir = proc->pgdir;
87    proc->pgdir = pgdir;
88    proc->sz = sz;
89    proc->tf->eip = elf.entry;  // main
90    proc->tf->esp = sp;
91    switchuvm(proc);
92    freevm(oldpgdir);
93    return 0;
```

其中就包括当前进程程序计数器 eip 的设置为新的程序的入口处以及栈指针 esp 的设置。然后用 switchuvm 切换到该进程的页表，标记该进程为 RUNNING。switchuvm 同时还设置好任务状态段 SEG_TSS ，让硬件在进程的内核栈中执行系统调用与中断。然后调用 freeman 释放掉就掉内存映像并返回 0。

由于该进程的 Trap Frame 中的 eip 被设置为新的程序的入口函数。

值得注意的是：switchuvm 切换到该进程的页表，然 CPU 开始执行该进程（即进程自身）。CPU 会从 main 函数开始继续执行该进程。所以，91-92 行的代码实际上不会被执行。

也就是说 exec 在如果正常执行新的程序，将不会再返回。因此我们必须在新的进程中调用 exec。也就是为什么要在 fork1()返回值为 0 时才进入 if 语句内部。

在控制台输入一行 command，shell 是如何确定命令的类型呢？

Shell 的 main 函数如下：

```
int main(void)
{
  static char buf[100];
  int fd;

  // Assumes three file descriptors open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Clumsy but will have to do for now.
      // Chdir has no effect on the parent if run in the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    wait();
  }
  exit();
}
```

输入一行命令。Shell 先创建一个子进程，然后首先调用 parsecmd 解析该行命令，再用 runcmd 执行该行命令。

```
328  struct cmd*
329  parsecmd(char *s)
330  {
331    char *es;
332    struct cmd *cmd;
333
334    es = s + strlen(s);
335    cmd = parseline(&s, es);
336    peek(&s, es, "");
337    if(s != es){
338      printf(2, "leftovers: %s\n", s);
339      panic("syntax");
340    }
341    nulterminate(cmd);
342    return cmd;
343  }
```

Parsecmd 调用 parseline 解析这一行命令

首先我们来看看有哪些命令类型：

针对 cmdtype，在 sh.c 有如下定义。command 有 5 种类型（分别对应结构体）：EXEC（execcmd），REDIR（redircmd），PIPE（pipecmd），LIST（listcmd），BACK（backcmd）。

```
// Parsed command representation
#define EXEC  1
#define REDIR 2
#define PIPE  3
#define LIST  4
#define BACK  5

#define MAXARGS 10

struct cmd {
  int type;
};

struct execcmd {
  int type;
  char *argv[MAXARGS];
  char *eargv[MAXARGS];
};

struct redircmd {
  int type;
  struct cmd *cmd;
  char *file;
  char *efile;
  int mode;
  int fd;
};

struct pipecmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};

struct listcmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};

struct backcmd {
  int type;
  struct cmd *cmd;
};
```

$echo README 对应的 cmd->type 是哪个？

首先，查看 parsecmd 的定义。

```
struct cmd*
parsecmd(char *s)
{
  char *es;
  struct cmd *cmd;

  es = s + strlen(s);
  cmd = parseline(&s, es);
  peek(&s, es, "");
  if(s != es){
    printf(2, "leftovers: %s\n", s);
    panic("syntax");
  }
  nulterminate(cmd);
  return cmd;
}
```

从控制台得到一个字符串 buf，命令行只接收 echo 和 README 两个参数，对照 sh.c 中结构体的定义，可以发现结构体 execcmd 最符合格式要求。

但具体代码怎么反映这个过程呢？

阅读代码发现，parseline 函数逐层嵌套以下几个函数。在 parseexec 中，由于没有找到"("，执行 execcmd()，初始化 type 为 EXEC。

```
struct cmd*
parseline(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parsepipe(ps, es);
  while(peek(ps, es, "&")){
    gettoken(ps, es, 0, 0);
    cmd = backcmd(cmd);
  }
  if(peek(ps, es, ";")){
    gettoken(ps, es, 0, 0);
    cmd = listcmd(cmd, parseline(ps, es));
  }
  return cmd;
}
```

```
struct cmd*
parsepipe(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parseexec(ps, es);
  if(peek(ps, es, "|")){
    gettoken(ps, es, 0, 0);
    cmd = pipecmd(cmd, parsepipe(ps, es));
  }
  return cmd;
}
```

```
struct cmd*
parseexec(char **ps, char *es)
{
  char *q, *eq;
  int tok, argc;
  struct execcmd *cmd;
  struct cmd *ret;

  if(peek(ps, es, "("))
    return parseblock(ps, es);

  ret = execcmd();
  cmd = (struct execcmd*)ret;

  argc = 0;
  ret = parseredirs(ret, ps, es);
  while(!peek(ps, es, "|)&;")){
    if((tok=gettoken(ps, es, &q, &eq)) == 0)
      break;
    if(tok != 'a')
      panic("syntax");
    cmd->argv[argc] = q;
    cmd->eargv[argc] = eq;
    argc++;
    if(argc >= MAXARGS)
      panic("too many args");
    ret = parseredirs(ret, ps, es);
  }
  cmd->argv[argc] = 0;
  cmd->eargv[argc] = 0;
  return ret;
}
```

```
struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es)
{
  int tok;
  char *q, *eq;

  while(peek(ps, es, "<>")){
    tok = gettoken(ps, es, 0, 0);
    if(gettoken(ps, es, &q, &eq) != 'a')
      panic("missing file for redirection");
    switch(tok){
    case '<':
      cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
      break;
    case '>':
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    case '+':  // >>
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    }
  }
  return cmd;
}
```

图表 1

parseline()对这行命令进行解析：首先调用 parsepipe()去分析命令是否为管道类型；在 parsepipe()中开始又会去调用 parseexec()；在 parseexec()的内部，会调用 parseblock()检查命令中是否有“（）”，然后调用 execcmd 继续进行解析…

这里的命令没有其它特殊的字符，所以主要是在 parseexec 中的 execcmd 会将 cmd->type 设置为 EXEC。如下：

```
struct cmd*
execcmd(void)
{
  struct execcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = EXEC;
  return (struct cmd*)cmd;
}
```

相应的，$ls; echo “hello world“ 对应的 cmd->type 是哪个？

继续沿用上面的**图表 1**：在对$ls; echo “hello world 进行解析时，也会进入到 parseexec()中执行 cmd->type=EXEC 语句。不过，最终的调用 trace 会返回到 parseline()，然后在 if 语句中发现了“；”，然后进入到 listcmd()函数。

按照前面的分析：由于命令是可以嵌套的，所以最终的 parsecmd 返回的 cmd-type 是最外层的类型。最终会执行 parseline()中的 cmd = listcmd(cmd, parseline(ps, es));，初始化 cmd 为 listcmd。

```
struct cmd*
listcmd(struct cmd *left, struct cmd *right)
{
  struct listcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = LIST;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}
```

而 ls | wc 对应的 cmd->type 是哪个？

继续沿用上面的**图表 1**，在对 ls | wc 执行解析时会进入到 parseexec()中执行 cmd->type=EXEC 语句。然后调用 trace 会返回到 parseline()中匹配到“|”字符，随后进入到 pipecmd()

如$echo README 所示，回退到 parsepipe 后找到|进行 pipecmd 初始化：cmd = pipecmd(cmd, parsepipe(ps, es));。

```
struct cmd*
pipecmd(struct cmd *left, struct cmd *right)
{
  struct pipecmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = PIPE;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}
```

b）阅读 xv6_rev7.pdf 中 7930 行对应的 switch 分支及相关代码，请说明如何确保 rcmd->fd 为标准输入的呢？

查看该处的 switch 分支代码：

```
7930    case REDIR:
7931      rcmd = (struct redircmd*)cmd;
7932      close(rcmd->fd);
7933      if(open(rcmd->file, rcmd->mode) < 0){
7934        printf(2, "open %s failed\n", rcmd->file);
7935        exit();
7936      }
7937      runcmd(rcmd->cmd);
7938      break;
```

可见，当 cmd->type == REDIR 时，该处的代码会被执行。根据(a)中的分析，我们很容易追溯到 parseredirs()函数中。

```
375  struct cmd*
376  parseredirs(struct cmd *cmd, char **ps, char *es)
377  {
378    int tok;
379    char *q, *eq;
380
381    while(peek(ps, es, "<>")){
382      tok = gettoken(ps, es, 0, 0);
383      if(gettoken(ps, es, &q, &eq) != 'a')
384        panic("missing file for redirection");
385      switch(tok){
386      case '<':
387        cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
388        break;
389      case '>':
390        cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
391        break;
392      case '+':  // >>
393        cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
394        break;
395      }
396    }
397    return cmd;
```

查看一下 gettoken 函数：

```
266  int
267  gettoken(char **ps, char *es, char **q, char **eq)
268  {
269    char *s;
270    int ret;
271
272    s = *ps;
273    while(s < es && strchr(whitespace, *s))
274      s++;
275    if(q)
276      *q = s;
277    ret = *s;
278    switch(*s){
279    case 0:
280      break;
281    case '|':
282    case '(':
283    case ')':
284    case ';':
285    case '&':
286    case '<':
287      s++;
288      break;
289    case '>':
290      s++;
291      if(*s == '>'){
292        ret = '+';
293        s++;
294      }
295      break;
296    default:
297      ret = 'a';
298      while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
299        s++;
300      break;
301    }
302    if(eq)
303      *eq = s;
304
305    while(s < es && strchr(whitespace, *s))
306      s++;
307    *ps = s;
308    return ret;
309  }
```

分析可知，xv6 支持三种类型的重定向

1. ‘<’ 重定向标准输入
2. ‘>’ 重定向标准输出，复写
3. ‘>>’ 重定向标准输出，追加

在 parseredirs()中我们可以看到，parseredirs()匹配到第一个‘<’或‘>’后就尝试解析重定向类型及文件，然后退出循环。因此，xv6 也不支持重定向标准错误。

因此，要确保 rcmd->fd 为标准输入：命令中包含符号是‘<’（此时 redircmd 会将返回的 cmd->fd 设置为 0，则为标准输入），并且后面跟着合适的文件名用于重定向

输入重定向就是改变输入的方向，不再使用键盘作为命令输入的来源，而是使用文件作为命令的输入。

表3：Bash 支持的输出重定向符号

符号	说明
command <file	将 file 文件中的内容作为 command 的输入。
command <<END	从标准输入（键盘）中读取数据，直到遇见分界符 END 才停止（分界符可以是任意的字符串，用户自己定义）。
command <file1 >file2	将 file1 作为 command 的输入，并将 command 的处理结果输出到 file2。

和输出重定向类似，输入重定向的完整写法是 `fd<file`，其中 fd 表示文件描述符，如果不写，默认为 0，也就是标准输入文件。

c）sh.c 中第 7950 行至 7972 行：

第二、三个 if 语句中，管道的读端口和写端口都通过 close 语句关闭了，请问还怎么保证 pcmd->left 的输出进入管道的写端口，而 pcmd->right 的输入进入管道的读端口？为什么在父进程这里，还需要有两个 close 语句？以及两个 wait 语句？

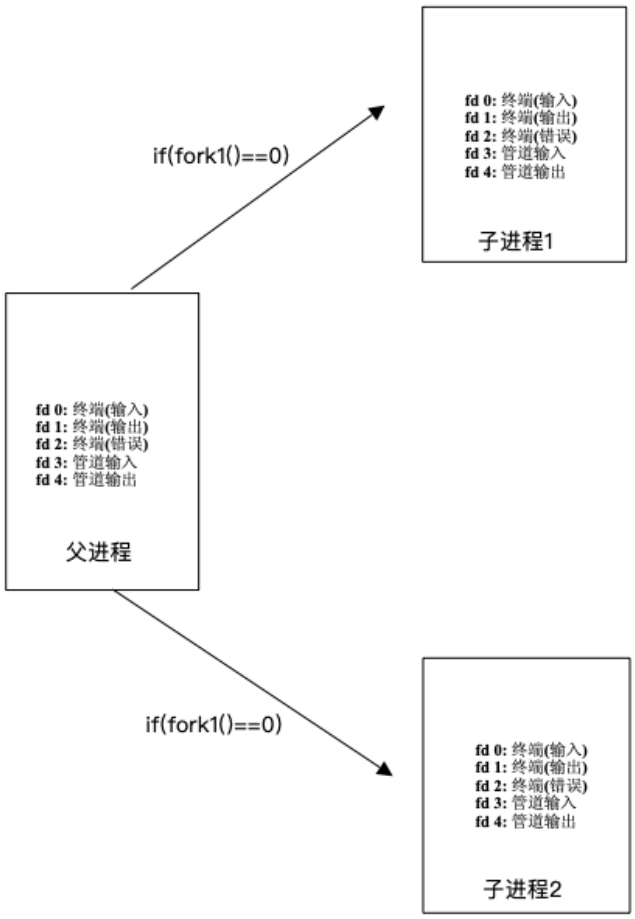
该处的代码如下：

```
7950   case PIPE:
7951     pcmd = (struct pipecmd*)cmd;
7952     if(pipe(p) < 0)
7953       panic("pipe");
7954     if(fork1() == 0){
7955       close(1);
7956       dup(p[1]);
7957       close(p[0]);
7958       close(p[1]);
7959       runcmd(pcmd->left);
7960     }
7961     if(fork1() == 0){
7962       close(0);
7963       dup(p[0]);
7964       close(p[0]);
7965       close(p[1]);
7966       runcmd(pcmd->right);
7967     }
7968     close(p[0]);
7969     close(p[1]);
7970     wait();
7971     wait();
7972     break;
```

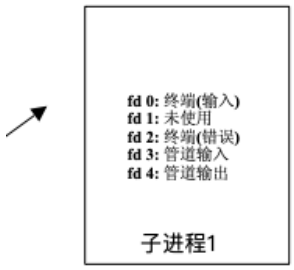
pipe()调用创建一个管道，参数是一个数组。我们知道在 C 语言中数组是按指针传递的。所以如果 pipe(p)成功执行，那么 p[0]里面存放的是管道输入的文件描述符，p[1]里面存放的是管道输出的文件描述符。

fork1()会生成一个进程，在两个 if 语句里面执行的分别是 shell 生成的两个子进程。

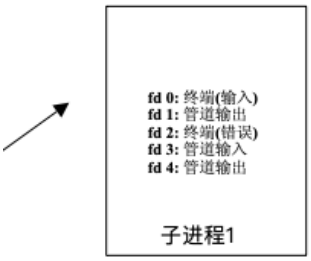
fork1()生成的子进程会继承父进程所有打开的文件以及对应的文件描述符。



第二个 if 语句中，也就是第一个子进程中：close(1)关闭了标准输出后，文件描述符 1 变为可用。



然后 dup(p[1])会将 p[1]指向的文件，也就是管道的输出端再分配一个最小的未被使用的文件描述符，也就是 1



解释一下：类似于 UNIX，xv6 也有一个文件数组表示打开的文件。我们可以在 xv6 中找到以下结构的定义：

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

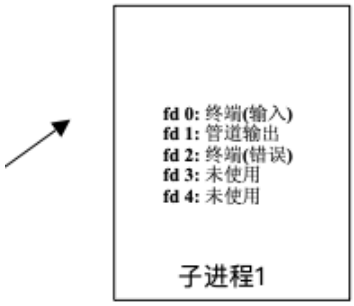
每一个使用文件描述符对应于一个 file 结构体。我们可以注意到结构体中的 ref 表示与当前打开文件关联的文件描述符的个数。ref 不为 0，文件就不会关闭。

通过 dup 分配的文件描述符会指向参数中的文件描述符的同一个结构体。也就是此时的文件描述符 1 和 p[1]都指向同一个打开的文件(file 结构体)，并且 ref = 2

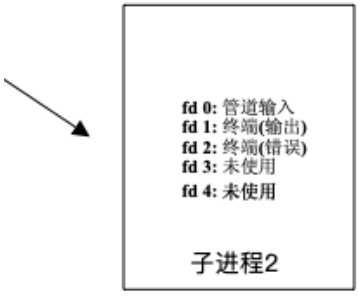
```
// Close file f.  (Decrement ref count, close when reaches 0.)
void
fileclose(struct file *f)
{
  struct file ff;

  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
  if(--f->ref > 0){
    release(&ftable.lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_NONE;
  release(&ftable.lock);

  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE){
    begin_trans();
    iput(ff.ip);
    commit_trans();
  }
}
```

尝试使用文件描述符去关闭一个文件的时候，首先会将对应的 file 结构体的 ref 减 1，只有当 ref==0 时，才会真正关闭文件，释放结构体。

所以，当在第一个子进程中调用 close(p[1])后，管道的输出仍然未被关闭，只是对应的 file 的结构的 ref=1



同理，子进程 2 最终会如下



由于 stdin 始终被宏定义为 0，stdout 始终被宏定义为 1。所以，当第一个进程试图往标准输出写内容的时候，它实际上写到了管道的输出端；而当第二个进程试图从标准输入读取内容的时候，它实际上读取到的是管道的输入端。

因此，此时的两个子进程就能通过管道通讯了，而其代码不需要做任何改变，因为 stdin 始终被宏定义为 0，stdout 始终被宏定义为 1。

为什么在父进程这里，还需要有两个 close 语句？以及两个 wait 语句？

父进程的 fork1()返回值大于 0，没有进入两个 if 语句，所以管道处于开放状态，需要两个 close()语句来将其读端和写端关闭。因为父进程不需要再使用管道，管道用于两个子进程的通讯。

两个 wait 语句等待两个 fork1()子进程的返回。

子进程经过 exec 后，新开的进程覆盖掉原来的进程映像，原调用进程的数据段、代码段和堆栈段被取代，在执行完之后，原调用进程的内容除了进程号以及一些资源限制设置外，其他全部被新的进程替换了。

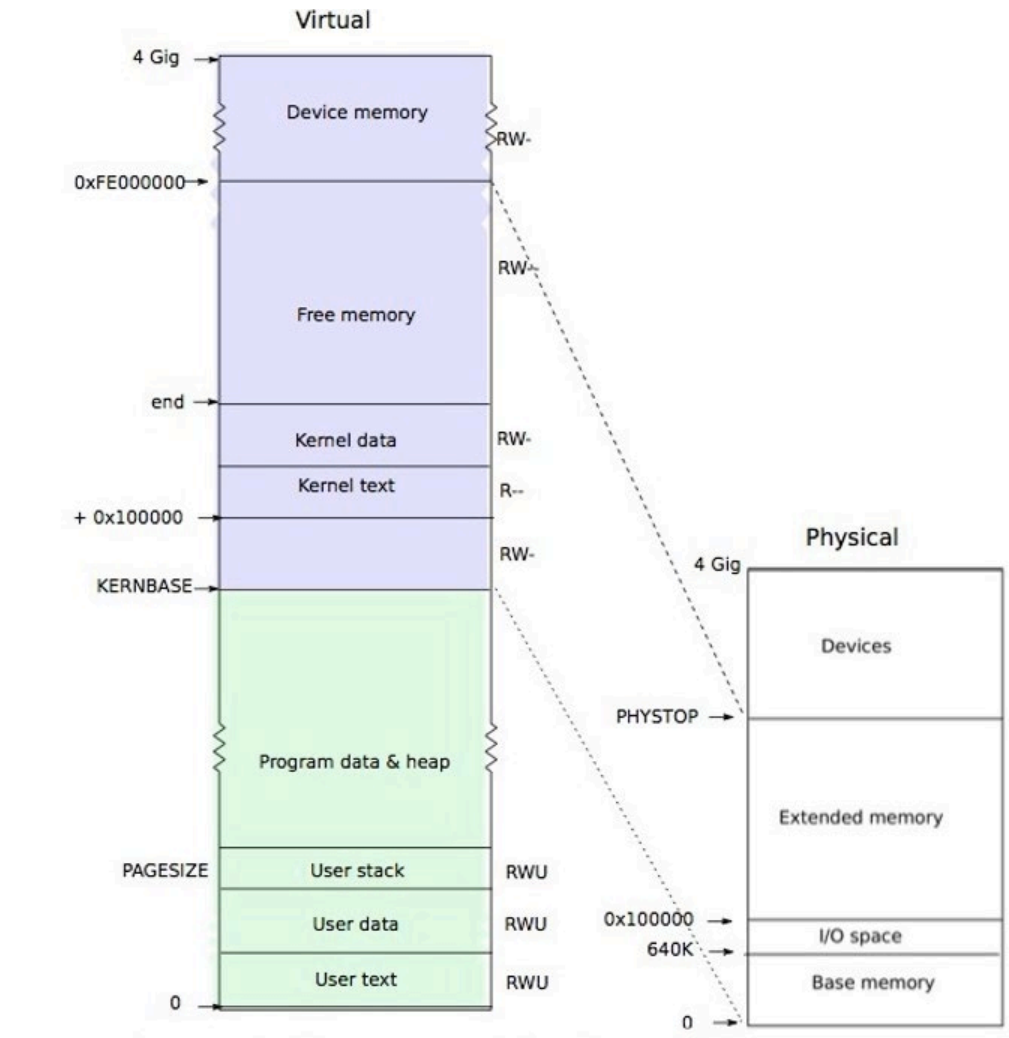
使用 wait()以回收完成运行后的子进程，防止它们成为孤儿进程。

（2）、阅读“xv6 中文文档” 第 1 章：第一个进程，回答以下问题：

a） “xv6 的地址空间结构有一个缺点， 即无法使用超过 2GB 的物理 RAM”——请给出解释。为什么 xv6 的内存空间只能有 2GB?

每个进程都有自己的页表，xv6 会在进程切换时通知分页硬件切换页表。

Xv6 在每个进程的页表中都包含了内核运行所需要的所有映射。这些映射都是将虚拟地址中的 KERNBASE(0x80000000) – (KERNBASE + PHYSTOP)映射物理地址 0-(PHYSTOP) 中。如下图所示。



Layout of a virtual address space and the physical address space.

在 xv6 中，PHYSTOP 被定义为最大的物理地址。由于 xv6 采用的是 32 位地址空间，所以 PHYSTOP <= (2^32 – 0x80000000)，即最大为 2GB 左右。

```
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses
```

这样的映射安排可以对内核代码和数据的操作变得很方便，因为采用的是直接映射。缺点就是 PHYSTOP 的大小被限制在 2GB。在上面的映射图中，物理地址中只有 0-PHYSTOP 能被使用。

b）（“xv6 中文文档”第 15 页）“这个映射就限制内核的指令+代码必须在 4mb 以内。 ”——请给出解释。为什么？

首先定位到 main.c 文件，查看 entry 中的页表定义 entrypgdir。

```
// Boot page table used in entry.S and entryother.S.
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.
// Use PTE_PS in page directory entry to enable 4Mbyte pages.
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};

//PAGEBREAK!
// Blank page.
```

页边界对齐

Entrypgdir 实际上是一个二级页表，NPDENTRIES 宏定义为 1024，该页表有 1024 项，二级页表的每个项能记录 1024 个页号，每个页面有 4K，所以页表项大小为 4M。使用二级结构使得页目录可以忽略那些没有任何映射的页表页，节省了存储空间。

```
120  // Page directory and page table constants.
121  #define NPDENTRIES      1024    // # directory entries per page directory
122  #define NPTENTRIES      1024    // # PTEs per page table
```

PDXSHIFT 被宏定义为 22，KERNBASE >> PDXSHIFT = 512。

```
127  #define PDXSHIFT        22      // offset of PDX in a linear address
```

在 Entrypgdir 的定义中，只设置了两个页表项，分别是页表项 0 和页表项 KERNBASE >> PDXSHIFT = 512。页表项 0 将虚拟地址 0:0x400000 映射到物理地址 0:0x400000，最后这个页表项是会被移除的。页表项 512 将虚拟地址的 KERNBASE:KERNBASE+0x400000 映射到物理地址 0:0x400000。这个页表项将在 entry 的代码结束后被使用；它将内核指令和内核数据应该出现的高虚拟地址处映射到了 boot loader 实际将它们载入的低物理地址处。而该页表项的大小被限制在 4MB，因此内核指令+代码必须在 4MB 以内。

值得一提的是，entrypgdir 中虽然有 1024 项，但是它们使用了 xv6 中只在初始页表中使用的超级页（见 entrypgdir 的定义）。 数组的初始化设置了 1024 条 PDE（页表项）中的 2 条，即 0 号和 512 号，而其他的 PDE 均为 0。xv6 设置了这两条 PDE 中的 PTE_PS 位，标记它们为“超级页”。

```
#define PTE_PS          0x080   // Page Size
```

```
// Map VA's [0, 4MB) to PA's [0, 4MB)
[0] = (0) | PTE_P | PTE_W | PTE_PS,
// Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
[KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
```

以下 entry 代码的汇编解析中第一段表明，内核通过设置 %cr4 中的 CP_PSE（？）（Page Size Extension） 位来通知分页硬件允许使用超级页。

```
# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax                      // 使用大页模式
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3                      // 设置CR3页表寄存器
  # Turn on paging.
  movl    %cr0, %eax                      // 启动分页
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.             // 设置堆栈指针
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump.
  mov $main, %eax                         // 跳到kernel的main
  jmp *%eax
```

c）请问 initcode.S 所触发 exec 系统调用执行了哪个程序，而那个程序又是实现什么功能的呢？

执行了 init 程序，源码如下图（init.c）所示。

init 程序的功能（见下图右）：创建一个新的控制台设备文件，然后把它作为描述符 0，1，2 打开。接下来它将不断循环，开启控制台 shell，处理没有父进程的僵尸进程，直到 shell 退出，然后再反复。

```
# Initial process execs /init.

#include "syscall.h"
#include "traps.h"


# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

```
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

（3）、阅读“xv6 中文文档”附录 A/B：PC 硬件及引导加载器，回答以下问题：

阅读 bootasm.S，查找资料，回答以下问题：

a）为什么主引导记录要存放在 0x7C00 开始的内存地址？（提示：这是历史遗留问题）

b）bootasm.S 第 21 行，“# Physical address line A20 is tied to zero…”这是著名的 Gate-A20，请介绍一下为什么要设定 Gate-A20。

c）bootasm.S 第 21 行-第 38 行，这是一段让人一头雾水的代码，请查找资料，解释一下这段代码为何和 enable A20 有关。

（参考 https://www.win.tue.nl/~aeb/linux/kbd/A20.html）

a) 首先，什么是主引导记录？

计算机启动的整个过程可描述如下。

1. 通电
2. 读取ROM里面的BIOS，用来检查硬件
3. 硬件检查通过
4. **BIOS根据指定的顺序，检查引导设备的第一个扇区（即主引导记录），加载在内存地址 0x7C00**
5. 主引导记录把操作权交给操作系统

硬件自检完成后，BIOS 把控制权转交给下一阶段的启动程序。按照“启动顺序”，BIOS 把控制权转交给排在第一位的储存设备。这时，计算机读取该设备的第一个扇区（最前面的 512 个字节）。如果最后两个字节是 0x55 和 0xAA，表明这个设备可以用于启动；如果不是，表明设备不能用于启动，控制权被转交给"启动顺序"中的下一个设备。

这最前面的 512 个字节，就叫做"主引导记录"（Master boot record，缩写为 MBR）。MBR 的主要作用是，告诉计算机到硬盘的哪一个位置去找操作系统。

主引导记录由三个部分组成：

（1） 第1-446字节：调用操作系统的机器码。

（2） 第447-510字节：分区表（Partition table）。

（3） 第511-512字节：主引导记录签名（0x55和0xAA）。

其中，第二部分"分区表"的作用，是将硬盘分成若干个区。

通过观测，我们发现，0x7c00 = 32KB-1024B。

0x7C00 这个地址来自 Intel 的第一代个人电脑芯片 8088，以后的 CPU 为了保持兼容，一直使用这个地址。1981 年推出的 IBM PC 5150 上运行的操作系统 DOS 1.0 需要的内存最少是 32KB。内存地址从 0x0000 开始编号，32KB 的内存就是 0x0000～0x7FFF。有如下原因。

BIOS developer team decided 0x7C00 because:

1. They wanted to leave as much room as possible for the OS to load itself within the 32KiB.

   他们想要在32KB的限制内，为操作系统的加载留下更多的空间。

2. 8086/8088 used 0x0 - 0x3FF for interrupts vector, and BIOS data area was after it.

   8086/8088芯片本身需要占用0x0 - 0x3FF作为中断向量，紧随其后就是BIOS数据区。

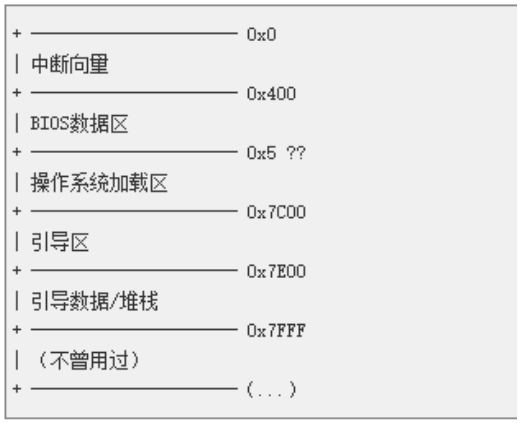
3. The boot sector was 512 bytes, and stack/data area for boot program needed more 512 bytes.

   引导扇区需要512字节，它所使用的栈和数据区需要额外的512字节。

4. So, 0x7C00, the last 1024B of 32KiB was chosen.

   因此，32KB尾部的1024B被选为主引导扇区。

加载并启动操作系统后，在重启电源之前将永远不会使用引导扇区，操作系统和应用程序可以自由使用 32KiB 的最后 1024B。因此，启动后的内存布局应当如下。



由此可知，主引导记录存放在 0x7C00 开始的内存地址。

b) 由于历史的原因，早期的电脑只有 1MB 内存空间，软件所能使用的内存容量最大为 1MB。后来，由于技术的进步，软件要求使用更多的内存，因此需要系统能够提供更大的内存空间。A20 信号的出现就是用来解决这个问题。

早期的电脑 CPU，比如 8088 的 CPU 只有 20 条地址线，因此最大的寻址空间为 2^20(B)=1(MB)。当地址的位数超过 20 位时，高位会被自动抹除。例如地址：
0x10ff11 = 1 0000 1111 1111 0001 0001 会被截断位 0000 1111 1111 0001 0001 = 0xff11。

后来在 80286CPU 推出时，它有 24 条地址线。80286CPU 应该是对 8088CPU 100%兼容的。然而 Intel 竟没有在 real mode 模式下对 24 为的地址进行高位截断。从而导致很多依赖于高位地址阶段的 8088 程序无法在 80286CPU 正常运行。后来 IBM 决定在主板总线加入一个开关，用于开启/禁用 0x100000 地址位。这称为 Gate-A20

c）bootasm.S 第 21 行-第 38 行，这是一段让人一头雾水的代码，请查找资料，解释一下这段代码为何和 enable A20 有关。

这段代码如下：

```
21    # Physical address line A20 is tied to zero so that the first PCs
22    # with 2 MB would run software that assumed 1 MB.  Undo that.
23  seta20.1:
24    inb     $0x64,%al               # Wait for not busy
25    testb   $0x2,%al
26    jnz     seta20.1
27
28    movb    $0xd1,%al               # 0xd1 -> port 0x64
29    outb    %al,$0x64
30
31  seta20.2:
32    inb     $0x64,%al               # Wait for not busy
33    testb   $0x2,%al
34    jnz     seta20.2
35
36    movb    $0xdf,%al               # 0xdf -> port 0x60
37    outb    %al,$0x60
38
```

IBM 发明了一种启用/禁止 0x100000 地址位的开关。在 8042 键盘控制器碰巧有一个备用引脚，因此可以利用该备用引脚控制与门以禁止高地址位。该信号称为 A20，当该信号为 0 时，会清除所有地址的高位。

首先将 0xd1 写入端口 0x64，然后将所需输出的端口值写到端口 0x60。

端口 0x64 的 1 号比特位表示输入缓冲区的状态，0 为空，1 为满。控制端口把状态送到寄存器中。当缓冲区为空时。再将 0xdf 送到 0x60 端口即可 enable A20；将 0xdd 送到 0x60 端口即可 disable A20。

++

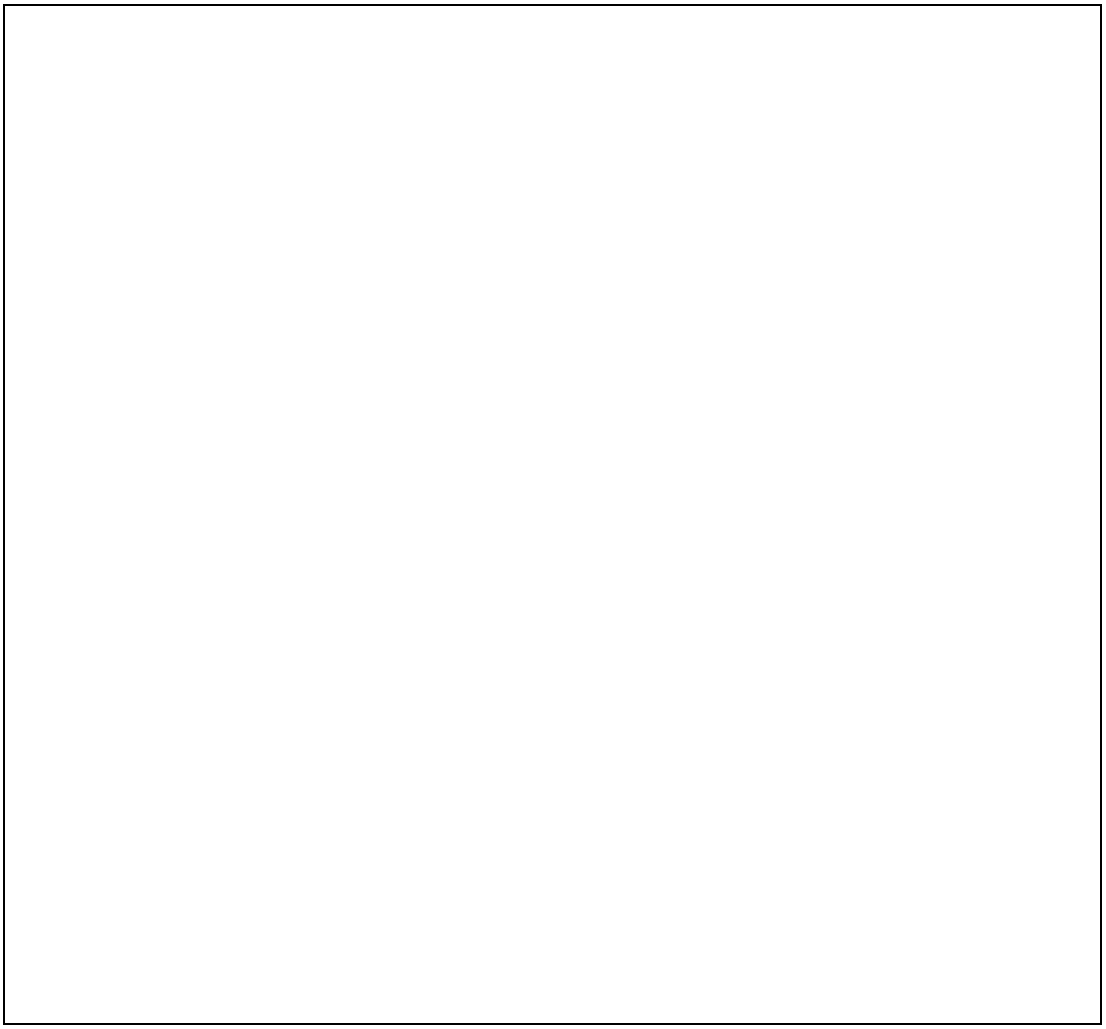
其他（例如感想、建议等等）。

通过本次实验，我们对 xv6 操作系统有了更深入的认识。

通过分析 runcmd 和 parsecmd 函数的实现、命令的类型以及这些函数处理并执行命令的具体实现，我们对 shell 程序的实现有了更加深入的认识。其中还包括了 shell 如何构建子进程、替换内存映像来执行命令程序。同时，我们对管道命令和重定向命令有了更加深入的认识。

同时，我们了解到了 xv6 在实现上的一些限制。例如，使用的物理内存无法超过 2GB，内核的代码+数据无法超过 4MB 等。并且加深了对 exec 函数的实现的理解。

最后，我们学习了引导程序相关的知识。以及操作系统发展史上的一些历史遗留问题及后来的解决方法。



指导教师批阅意见：   成绩评定：   指导教师签字：谭舜泉 2021 年 5 月 2 日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。