操作系统

实 验 报 告

实验名称：shell 命令解释器的实现

学　　员：侯华玮　　学　　号：202102001015

培养类型：无军籍学员　　年　　级：21 级

专业：计算机科学与技术（天河拔尖班）　所属学院：计算机学院

指导教员：文艳军　　职　　称：教授

实 验 室：　　实验日期：2023.05.24

国防科技大学训练部制

操作系统 实验报告

一、实验目的和内容

实验目的

深入掌握系统调用的使用方法，学习如何调试测试内核程序。

实验内容

扩充版本 0 内核，在系统启动时直接提供命令解释器功能（不运行硬盘上的命令解释器），即不断接收用户输入的命令并正确给出反馈。要求该命令解释器既支持内部命令 `cd、sync、exit、cat`；也支持外部命令，即可以接收 `cp、vi` 等命令，然后执行硬盘上相应的可执行程序。

实验要求

提交实验报告和所有源码，实验报告应对命令解释器的详细设计和测试过程等进行说明。

二、操作方法与实验步骤

实验实现思路

在内核中实现一个函数(`myshell`)，作为命令解释器。这个函数应该从控制台读取行，将它们解析为命令和参数，然后执行适当的操作。

对于 `cd、sync、exit` 和 `cat` 等内置命令，需要编写执行这些操作的程序。例如，`cd` 命令将调用 `chdir()` 系统调用，`sync` 将调用 `sync()`，`exit` 将终止命令解释器，`cat` 将读取一个文件并将其内容写到控制台。

对于像 `cp` 和 `vi` 这样的外部命令，解释器函数需要从文件系统中加载相应的程序，创建一个新进程，并在新进程中启动程序，涉及到使用 `fork()` 和 `execve()` 系统调用。

修改内核的初始化代码（`init/main.c`），在内核被初始化后，在它试图从文件系统加载 `init` 进程之前，启动自己编写的命令解释器函数。

实验步骤

1. 编写命令解释器顶层封装函数 `myshell`
2. 编写命令读取函数 `read_line`

3. 编写命令执行函数 `run_command`
4. 编写内部命令函数 `mycd, mycat`
5. 编写外部命令函数封装 `exec_cp, exec_vi, exec_ls`

---

1 编写命令解释器顶层封装函数 `myshell`

- 初始化

  在程序开始时，首先设置当前工作目录为根目录，之后调用硬盘初始化 `setup` 函数，将控制台设备打开，然后将标准输出重定向到控制台设备，这意味着程序将在控制台上进行 IO 操作，包括输出提示信息和读取用户输入的命令。

- 进入命令循环

  程序进入一个无限循环，用于读取用户输入的命令并执行。循环体内，首先输出当前命令提示符（包括当前工作目录），之后调用 `read_line` 函数从控制台设备中读取用户输入的命令。接着，将用户输入的命令传递给 `run_command` 函数，该函数用于解析并执行用户输入的命令。

- `run_command` 函数

  该函数用于解析并执行用户输入的命令，其大致处理流程如下：

  - 分离参数：根据用户输入的命令将其拆分成参数，并存放到一个 `char` 指针数组中。
  - 处理内置命令：内置命令是 shell 程序支持的一些命令，例如 `cd、exit`，这里程序检查用户输入的命令是否是内置命令并执行相应的操作，例如改变当前工作目录或退出程序。
  - 运行外部程序：若用户输入的不是内置命令，则程序在 `PATH` 环境变量指定的路径中查找用户输入的程序名对应的可执行文件，并调用 `execve` 函数执行该文件。

```c
void myshell(void)
{
    char buf[CMD_LEN + 1];
    cwd[0] = '/';
    cwd[1] = '\0';
    setup((void *)&drive_info);
    (void)open("/dev/tty0", O_RDWR, 0);
    (void)dup(0);
    (void)dup(0);
    print_hello();
    while (1)
    {
        printf("myshell: %s $ ", cwd);
        read_line(buf, 256);
        run_command(buf);
    }
}
```

2 编写命令读取函数 `read_line`

使用系统调用 `read` 读取一行输入，并做简单处理。

```c
int read_line(char *buf, int maxlength)
{
  int i = 0;
  char c;
  while (i < maxlength - 1)
  {
    if (read(0, &c, 1) != 1)
    {
      printf("read error\n");
      return -1;
    }
    if (c == '\n')
    {
      buf[i] = '\0';
      break;
    }
    else
    {
      buf[i] = c;
      i++;
    }
  }
  return i;
}
```

3 编写命令执行函数 `run_command`

1. 解析命令

   代码首先计算命令字符串的长度，如果长度为 0，直接返回。

   - 如果命令以 "`cd`" 开头，调用 `mycd` 函数改变当前工作目录。
   - 如果命令是 "`sync`"，调用 `sync` 函数进行磁盘同步。
   - 如果命令是 "`pwd`"，输出当前工作目录。
   - 如果命令是 "`exit`"，调用 `_exit` 函数退出程序。
   - 如果命令以 "`cat`" 开头，调用 `mycat` 函数查看文件内容。

2. 处理外部命令

> 如果用户输入的不是内置命令，则进入 else 分支。程序通过 fork 函数创建一个子进程，在子进程中通过 execve 系统调用执行用户输入的命令。如果命令以 cp、vi 或 ls 开头，则调用相应的函数执行命令。如果用户输入的命令既不是内置命令也不属于上述三种情况，则输出 "command not found" 并返回 -1。

3. 等待子进程结束

> 如果创建子进程成功，则父进程调用 `wait` 函数等待子进程结束。`wait` 函数挂起当前进程，直到任一子进程结束，返回结束的子进程 ID。父进程在接收到子进程结束的信号后，调用 `printf` 输出一行提示信息，之后结束 `wait` 的循环，返回 1。

C

```c
int run_command(char *command)
{
  int res;
  int pid;
  int len = strlen(command);
  if (len == 0)
  {
    return 1;
  }
  else if (!strncmp(command, "cd", 2))
  {
    mycd(command + 3);
  }
  else if (!strcmp(command, "sync"))
  {
    sync();
  }
  else if (!strcmp(command, "pwd"))
  {
    printf("%s\n", cwd);
  }
  else if (!strcmp(command, "exit"))
  {
    _exit(0);
  }
  else if (!strncmp(command, "cat", 3))
  {
    mycat(command + 4);
  }
  else // outside command
  {
    if (!(pid = fork()))
    {
      if (!strncmp(command, "cp", 2))
      {
        exec_cp(command, len);
      }
      else if (!strncmp(command, "vi", 2))
      {
        exec_vi(command, len);
      }
```

```
      else if (!strcmp(command, "ls"))
      {
        exec_ls(command, len);
      }
      else
      {
        printf("command not found\n");
        return -1;
      }
      _exit(2);
    }
    if (pid > 0)
      while (pid != wait(NULL))
        ;
  }

  return 1;
}
```

---

4 编写内部命令函数 `mycd, mycat`

`mycd` 函数接收一个参数 `target_path`，为要切换到的目录路径。该函数的功能实现分为以下几个步骤：

1. 调用 `chdir` 函数切换目录

   函数首先调用 `chdir` 函数尝试切换目录，并输出切换结果，如果返回值为 -1，则说明切换失败，函数将输出 "chdir error" 并直接退出。

2. 处理相对路径和绝对路径

   - 如果输入的目录路径为绝对路径，函数将该路径直接赋值给 cwd 变量；
   - 如果输入的目录路径为相对路径，则需要根据当前路径计算出新路径。

   函数首先定义一个字符数组 res_path 用于存储新路径，初始化 abs 变量为 0，表示输入的目录路径为相对路径。之后使用 strtok 函数对目录路径字符串进行分割，每次取得一个相对路径字符串，判断是否为 `.` 或 `..`，分别对应当前目录和父目录。如果是当前目录，则不做处理；如果是父目录，则将 cwd 后缀 "/" 之前的字符串作为新的路径。否则，将该相对路径字符串加入到新路径中，并将 `abs` 变量设为 0，表示当前在相对路径下。重复该操作，直到所有路径分割完成。遍历完所有相对路径后，将新路径赋值给 `cwd` 变量。

C

```
void mycd(char *target_path)
{
  int res = chdir(target_path);
  int abs = 0;
  printf("chdir res: %d\n", res);
  if (res == -1)
  {
```

```
    printf("chdir error\n");
    return;
  }
  else
  {
    char res_path[NAME_LEN + 1];
    if (target_path[0] == '/')
    {
      strcpy(res_path, target_path);
      abs = 1;
    }

    char *relative_path;
    relative_path = strtok(target_path, "/");
    while (relative_path != NULL)
    {

      printf("relative_path: %s\n", relative_path);
      if (!strcmp(relative_path, "."))
      {
        printf("cwd: %s\n", cwd);
      }
      else if (!strcmp(relative_path, ".."))
      {
        if (strcmp(cwd, "/"))
        { // not top directory
          int i = strlen(cwd) - 1;
          while (cwd[i] != '/')
          {
            i--;
          }
          if (i == 0)
          {
            cwd[i + 1] = '\0';
          }
          else
          {
            cwd[i] = '\0';
          }
        }
        printf("cwd: %s\n", cwd);
      }
      else
      {
        if (abs == 1)
        {
          cwd[0] = '/';
          strcpy(cwd + 1, relative_path);
          abs = 0;
        }
        else
        {
          if (strcmp(cwd, "/"))
          {
            strcat(cwd, "/");
```

```
            }

            strcat(cwd, relative_path);
          }

          printf("cwd: %s\n", cwd);
        }
        relative_path = strtok(NULL, "/");
      }
    }
}
```

`mycat` 实现了在控制台输出指定文件内容的功能，函数接收一个参数 `filename`，为要显示内容的文件名。该函数的功能实现分为以下几个步骤：

1. 拼接文件路径

   函数首先将当前工作目录路径和要显示的文件名拼接成完整的文件路径，并存储在 file_path 字符数组中。

2. 打开文件

   函数通过 `open` 函数打开文件，并将文件描述符存储在 fd 变量中。若打开文件失败，函数将输出 "open error" 并返回 -1。

3. 读取文件内容

   函数通过 `read` 函数读取文件内容，每次最多读取 1024 字节，读取到的内容存储在 `file_buf` 字符数组中。若读取成功，则通过 `write` 函数将读取到的内容打印到控制台上。

4. 关闭文件

   函数在读取完文件后，通过 `close` 函数关闭文件。

C

```c
int mycat(char *filename)
{
  char file_path[NAME_LEN + 1];
  char file_buf[1024];
  strcpy(file_path, cwd);
  strcat(file_path, filename);
  printf("file_path: %s\n", file_path);
  int fd = open(file_path, O_RDONLY, 0);
  if (fd == -1)
  {
    printf("open error\n");
    return -1;
  }
  else
  {
    int len = 0;
```

```
    while ((len = read(fd, file_buf, 1024)) > 0)
    {
      write(1, file_buf, len);
    }
  }
  close(fd);
  return 0;
}
```

---

5 编写外部命令函数封装 `exec_cp, exec_vi, exec_ls`

`exec_cp, exec_vi, exec_ls` 三个函数都是使用 execve 函数调用外部命令进行操作，其中：

- `exec_cp` 函数实现的是文件复制功能，调用 `/usr/bin/cp` 命令；
- `exec_vi` 函数实现的是编辑文件功能，调用 `/bin/vi` 命令；
- `exec_ls` 函数实现的是显示目录内容功能，调用 `/usr/bin/ls` 命令。

它们的实现思路大致相同，具体流程概括如下：

1. 处理命令参数
2. 准备环境变量
3. 调用外部命令

C

```c
void exec_cp(char *command, int len)
{
  int i = 0;
  for (i = 3; i < len; i++)
  {
    if (command[i] == ' ')
    {
      command[i] = '\0';
      break;
    }
  }
  printf("f1: %s\n", command + 3);
  printf("f2: %s\n", command + i + 1);
  char *cp_argv[] = {"/usr/bin/cp", command + 3, command + i + 1, NULL};
  char *cp_envp[] = {"PATH=/usr/root", NULL};
  execve("/usr/bin/cp", cp_argv, cp_envp);
}

void exec_vi(char *command, int len)
{
  int i = 0;
  for (i = 3; i < len; i++)
  {
    if (command[i] == ' ')
    {
```

```
      command[i] = '\0';
      break;
    }
  }
  printf("filename: %s\n", command + 3);
  char *vi_argv[] = {"/bin/vi", command + 3, NULL};
  char *vi_envp[] = {"HOME=/usr/root", "TERM=console", NULL};
  execve("/bin/vi", vi_argv, vi_envp);
}

void exec_ls(char *command, int len)
{
  char *ls_argv[] = {"/usr/bin/ls", NULL};
  char *ls_envp[] = {"PATH=/usr/root", NULL};
  execve("/usr/bin/ls", ls_argv, ls_envp);
}
```

- `exec_cp` 函数的实现：

  通过遍历 `command` 字符串，查找第二个空格符号，将其替换为字符串结束符 `'\0'`，以将命令字符串分割为两个文件路径字符串。然后，使用 `printf` 函数输出这两个文件路径，以便在终端输出中查看相关日志。接下来，定义了两个数组变量 `cp_argv` 和 `cp_envp`，用于存储 `cp` 命令的参数和环境变量。然后使用 Linux 的 `execve` 系统调用执行 `cp` 命令，将 `cp_argv` 和 `cp_envp` 作为参数传递给 `execve` 函数。

- `exec_vi` 函数的实现：

  通过遍历 `command` 字符串，查找第二个空格符号，将其替换为字符串结束符 `'\0'`，以将命令字符串分割为两个文件路径字符串。然后，使用 `printf` 函数输出文件路径字符串，以便在终端输出中查看相关日志。接下来，定义了两个数组变量 `vi_argv` 和 `vi_envp`，并分别指定了 `vi` 命令的参数和环境变量。然后使用 Linux 的 `execve` 系统调用执行 `vi` 命令，将 `vi_argv` 和 `vi_envp` 作为参数传递给 `execve` 函数。

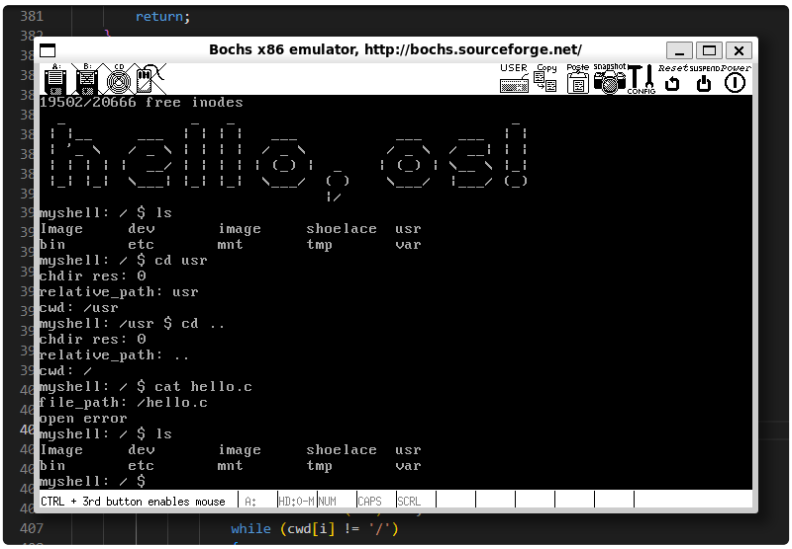
  需要注意的是，`vi` 命令所需的环境变量可能与本地环境设置有所不同，因此，在执行 `vi` 命令时，需要设置适当的环境变量以确保其正确执行。在此代码中，设置了 `HOME` 和 `TERM` 两个环境变量，分别指定了 `vi` 命令使用的主目录和终端类型。

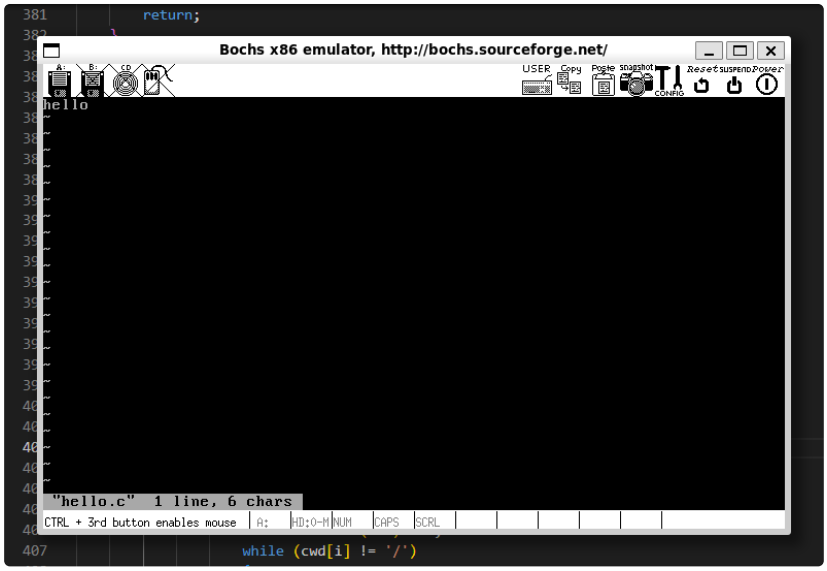
- `exec_ls` 函数的实现：

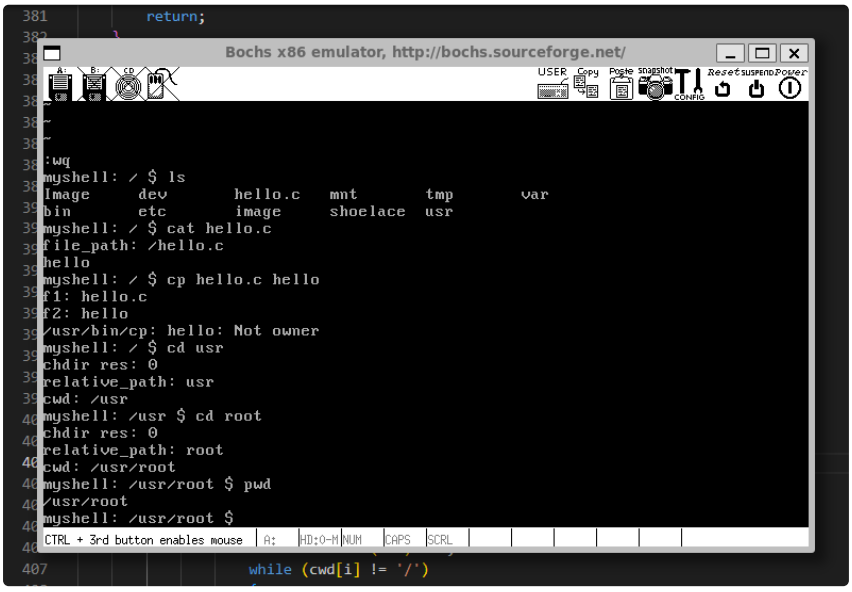
  定义了两个数组变量 `ls_argv` 和 `ls_envp`，分别用于存储 `ls` 命令的参数和环境变量。然后使用 Linux 的 `execve` 系统调用执行 `ls` 命令，将 `ls_argv` 和 `ls_envp` 作为参数传递给 `execve` 函数。

三、实验结果与分析

编译完成后，0.11 内核可正常启动并执行 `myshell` 函数，具体测试情况截图如下：


Bochs x86 emulator, http://bochs.sourceforge.net/
19502/20666 free inodes
myshell: / $ ls
Image dev image shoelace usr
bin etc mnt tmp var
myshell: / $ cd usr
chdir res: 0
relative_path: usr
cwd: /usr
myshell: /usr $ cd ..
chdir res: 0
relative_path: ..
cwd: /
myshell: / $ cat hello.c
file_path: /hello.c
open error
myshell: / $ ls
Image dev image shoelace usr
bin etc mnt tmp var
myshell: / $
CTRL + 3rd button enables mouse
while (cwd[i] != '/')


Bochs x86 emulator, http://bochs.sourceforge.net/
hello
"hello.c" 1 line, 6 chars
CTRL + 3rd button enables mouse
while (cwd[i] != '/')


Bochs x86 emulator, http://bochs.sourceforge.net/
:wq
myshell: / $ ls
Image dev hello.c mnt tmp var
bin etc image shoelace usr
myshell: / $ cat hello.c
file_path: /hello.c
hello
myshell: / $ cp hello.c hello
f1: hello.c
f2: hello
/usr/bin/cp: hello: Not owner
myshell: / $ cd usr
chdir res: 0
relative_path: usr
cwd: /usr
myshell: /usr $ cd root
chdir res: 0
relative_path: root
cwd: /usr/root
myshell: /usr/root $ pwd
/usr/root
myshell: /usr/root $
CTRL + 3rd button enables mouse
while (cwd[i] != '/')

四、问题与建议

问题

无论是在自己编写的 myshell 里还是系统原装的 sh 里，都无法正常执行 cp 命令，如：

```C
myshell: /usr $ cp hello.c hi.c
/usr/bin/cp: hi.c: Not owner
```

源文件 hello.c 有访问权限，hi.c 尚未创建，尚不清楚如何解决。
