操作系统原理实验报告

实验一 小型 shell 模拟

3022206045-陆子毅

实验内容：

实现运行原生的 linux 程序，实现输入输出重定向，管道符

实验过程：

一、执行简单命令

根据手册指引，找到了 runcmd 所在位置，并用 man 3 exec 查看了 exec 函数原型。在 runcmd 函数中，编写了代码来处理简单命令。使用 execv 函数来执行用户输入的命令，并在执行失败时打印错误消息。

查阅资料得知，

execv 和 execvp 都是用于在一个进程中执行另一个程序的函数，它们属于类 Unix 操作系统的系统调用。它们之间的主要区别在于参数的传递方式和搜索可执行文件的方式。

execv 函数：

原型：int execv(const char *path, char *const argv[]);

接受两个参数，第一个参数是要执行的程序的路径名，第二个参数是一个字符串数组，其中包含了传递给新程序的命令行参数。

需要明确指定可执行文件的路径，即必须提供完整的路径，否则需要确保当前工作目录包含可执行文件。不会搜索 PATH 环境变量中指定的目录，因此需要提供完整路径或者在当前目录中执行可执行文件。

execvp 函数：

原型：int execvp(const char *file, char *const argv[]);

接受两个参数，第一个参数是要执行的程序的名称，第二个参数是一个字符串数组，其中包含了传递给新程序的命令行参数。

会搜索 PATH 环境变量中指定的目录，以查找可执行文件。因此，你可以只提供可执行文件的名称而无需提供完整路径。

如果可执行文件的路径可以在 PATH 中找到，execvp 会自动查找并执行它。

总之，主要区别在于 execv 需要提供完整路径，而 execvp 可以根据 PATH 环境变量中的目录来查找可执行文件。通常情况下，execvp 更为常用，因为它更方便，不需要显式指定完整路径，但如果你需要精确控制可执行文件的路径，可以使用 execv。

```
SYNOPSIS
    #include <unistd.h>

    extern char **environ;

    int execl(const char *pathname, const char *arg, ...
              /* (char  *) NULL */);
    int execlp(const char *file, const char *arg, ...
              /* (char  *) NULL */);
    int execle(const char *pathname, const char *arg, ...
              /*, (char *) NULL, char *const envp[] */);
    int execv(const char *pathname, char *const argv[]);
    int execvp(const char *file, char *const argv[]);
    int execvpe(const char *file, char *const argv[],
              char *const envp[]);

  Feature Test Macro Requirements for glibc (see feature_test_macros(7)):


  v - execv(), execvp(), execvpe()
    The char *const argv[] argument is an array of pointers to null-terminated strings
that
    represent  the argument list available to the new program.  The first argument, by
con-
    vention, should point to the filename associated with the file being executed.  The
ar-
    ray of pointers must be terminated by a null pointer.
```

```c
case ' ':
  ecmd = (struct execcmd*)cmd;
  if(ecmd->argv[0] == 0)
    exit(0);
  execvp(ecmd->argv[0], ecmd->argv);
  perror("execvp");
  //fprintf(stderr, "exec not implemented\n");
  // Your code here ...
  break;
```

测试成功，并且在运行命令时不用写/bin

二、输入输出重定向

解析器已经能够识别>和<符号，并构建了 redircmd。只需要填写 runcmd 函数中>和<的代码部分，根据实验提示，应该使用 open 和 close 等系统调用。我们确保在系统调用失败时打印错误消息。

使用命令查看 open，close 的函数原型使用说明。

```
SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

       int openat(int dirfd, const char *pathname, int flags);
       int openat(int dirfd, const char *pathname, int flags, mode_t mode);

       /* Documented separately, in openat2(2): */
       int openat2(int dirfd, const char *pathname,
                   const struct open_how *how, size_t size);


       SYNOPSIS
            #include <unistd.h>

            int close(int fd);
```

open 和 close 是 Unix-like 操作系统中的两个系统调用，它们用于文件操作。它们的作用如下：

open 函数：

open 用于打开文件或创建新文件，并返回一个文件描述符，该文件描述符是一个非负整数，代表打开的文件。

open 函数的原型为：int open(const char *path, int flags, mode_t mode);

参数说明：

path：指定要打开的文件的路径或名称。

flags：指定文件打开的方式和选项，例如只读、只写、追加等。常见的标志包括 O_RDONLY（只读）、O_WRONLY（只写）、O_RDWR（读写）、O_CREAT（创建文件）、O_APPEND（在文件末尾追加）等。

mode：通常用于指定新文件的访问权限，例如 0666 表示文件的读写权限。

open 函数返回一个文件描述符，可以在后续的文件操作中使用这个描述符来标识和访问文件。

close 函数：

close 用于关闭一个已打开的文件，释放相关资源，并使文件描述符不再有效。

close 函数的原型为：int close(int fd);

参数 fd 是要关闭的文件描述符。

当你完成了对文件的操作后，应该调用 close 函数来释放文件描述符，以避免资源泄漏和确保文件被正确关闭。

dup2 是一个 Unix-like 操作系统中的系统调用，用于复制文件描述符（file

descriptor)。它的作用是创建一个新的文件描述符，使其与已有的文件描述符指向相同的文件或流。dup2 允许你将一个文件描述符的副本与另一个文件描述符相关联，以便在后续的文件操作中同时使用这两个描述符。

dup2 函数的原型如下：

int dup2(int oldfd, int newfd);

oldfd 是要复制的旧文件描述符，它必须是有效的文件描述符。

newfd 是要创建的新文件描述符，它通常是一个未使用的文件描述符。如果 newfd 已经被占用，dup2 会首先关闭它，然后将其重定向到 oldfd 指向的文件或流。

在 dup2(file_fd, rcmd->fd) 中，dup2 函数用于复制文件描述符 file_fd 到 rcmd->fd。这通常是用于文件描述符的重定向操作，将一个文件描述符与另一个文件描述符关联，以实现输入/输出重定向或者管道操作。

具体来说，这行代码的作用是将 file_fd 指向的文件描述符的内容复制到 rcmd->fd 指向的文件描述符中。这可能会涉及以下操作：

如果 rcmd->fd 已经打开并与某个文件或流相关联，那么 dup2 会首先关闭 rcmd->fd，然后将其重定向到 file_fd 指向的文件或流，使得 rcmd->fd 成为 file_fd 的一个副本。

如果 rcmd->fd 未打开，dup2 会简单地将其指向 file_fd 指向的文件或流，使得 rcmd->fd 现在与 file_fd 具有相同的文件内容。

这种操作通常用于重定向进程的标准输入、标准输出或标准错误，或者在创建管道时将一个进程的输出与另一个进程的输入相连。

需要注意的是，在执行 dup2 后，rcmd->fd 将继续指向与 file_fd 相同的文件或流，因此所有的读写操作将影响到这个文件或流。

```
case '>':
case '<':
  rcmd = (struct redircmd*)cmd;
  int file_fd = open(rcmd->file, rcmd->mode, 0666);
  if (file_fd < 0) {
      perror("open");
      exit(-1);
  }
  if (dup2(file_fd, rcmd->fd) < 0) {
      perror("dup2");
      exit(-1);
  }
  close(file_fd);
  // Your code here ...
  runcmd(rcmd->cmd);
  //fprintf(stderr, "redir not implemented\n");
  break;
```

测试运行，成功

```
2440078$ ./a.out < sh
open: No such file or directory
2440078$ ./a.out < t.sh
    6    10    58
    6    10    58
rm: cannot remove 'y1': No such file or directory
cat: y1: No such file or directory
rm: cannot remove 'y1': No such file or directory
    5     9    56
rm: cannot remove 'yrm': No such file or directory
rm: cannot remove 'y': No such file or directory
2440078$
```

三、管道操作

管道操作需要用到使用了 pipe、fork、close 和 dup 等系统调用。使用 man 命令查询具体函数。

pipe、fork、close 和 dup 是在 Unix-like 操作系统中常用于创建管道和进程间通信的系统调用，它们通常一起使用。

pipe 函数：

pipe 用于创建一个管道，它可以在两个进程之间传递数据。管道有两个文件描述符，一个用于读取数据，另一个用于写入数据。

pipe 函数的原型为：int pipe(int pipefd[2]);

pipefd 是一个长度为 2 的整数数组，用于存储管道的两个文件描述符，pipefd[0] 用于读取数据，pipefd[1] 用于写入数据。

fork 函数：

fork 用于创建一个新的进程，新进程是调用进程的副本。这两个进程几乎完全相同，但有不同的进程 ID（PID）。

子进程通常用于执行不同的任务，例如在管道通信中，一个进程用于写入数据，另一个进程用于读取数据。

父进程和子进程都会继续执行后续代码，但可以通过返回值来区分哪个进程是父进程，哪个是子进程。

close 函数：

close 用于关闭一个文件描述符，释放相关资源。在管道通信中，通常在父子进程中使用 close 来关闭不需要的文件描述符，以防止资源泄漏。

dup 函数：

dup 用于复制文件描述符，创建一个副本，该副本与原始文件描述符指向相同的文件或流。这可以用于在不同的文件描述符上进行读写操作，以实现进程间通信。

例如，在子进程中，可以使用 dup 复制标准输出文件描述符，然后使用 close 关闭原始标准输出文件描述符，以将子进程的标准输出重定向到管道。

综合使用这些系统调用，可以在父子进程之间创建管道，将数据从一个进程传递到另一个进程，通过 fork 创建子进程，使用 close 关闭不需要的文件描述符，以及使用 dup 进行输入/输出重定向。这是实现进程间通信的常见方法之一。

```c
case '|':
  pcmd = (struct pipecmd*)cmd;
  int pipe_fd[2];


  if (pipe(pipe_fd) < 0) {
      perror("pipe");
      exit(-1);
  }


  int left_pid = fork1();
  if (left_pid == 0) {
      // 子进程：左侧命令
      close(pipe_fd[0]); // 关闭管道的读端
      dup2(pipe_fd[1], 1); // 将标准输出重定向到管道写端
      close(pipe_fd[1]); // 关闭管道的写端
      runcmd(pcmd->left);
  }


  int right_pid = fork1();
  if (right_pid == 0) {
      // 子进程：右侧命令
      close(pipe_fd[1]); // 关闭管道的写端
      dup2(pipe_fd[0], 0); // 将标准输入重定向到管道读端
      close(pipe_fd[0]); // 关闭管道的读端
      runcmd(pcmd->right);
  }


  // 关闭父进程中的管道文件描述符
  close(pipe_fd[0]);
  close(pipe_fd[1]);


  // 等待左右两个子进程结束
  int status;
  waitpid(left_pid, &status, 0);
  waitpid(right_pid, &status, 0);
  // Your code here ...
  break;
  //fprintf(stderr, "pipe not implemented\n");
}
```

实验总结：

实验不是很难，但是需要我们去认证阅读 linux 编程手册，在这期间，准确快速地阅读英文手册是一个难点，希望能够通过不断的锻炼渐渐摆脱翻译软件。

通过本次实验，我们成功实现了一个基本的 Unix Shell，具备了运行原生 Linux 程序、输入输出重定向和管道操作的功能。我们熟悉了系统调用接口和 Shell 的工作原理，提高了操作系统原理的实际编程能力。本次实验使我们更深入地理解了操作系统原理，特别是 Shell 的实现。我们成功地实现了所需的功能，同时也学到了如何查看系统调用文档和调试 Shell 程序。在接下来的实验中，我们将继续学习和扩展 Shell 的功能，以更好地理解操作系统的工作原理。