Implementing Redirection: `dup2()`, `dup()`

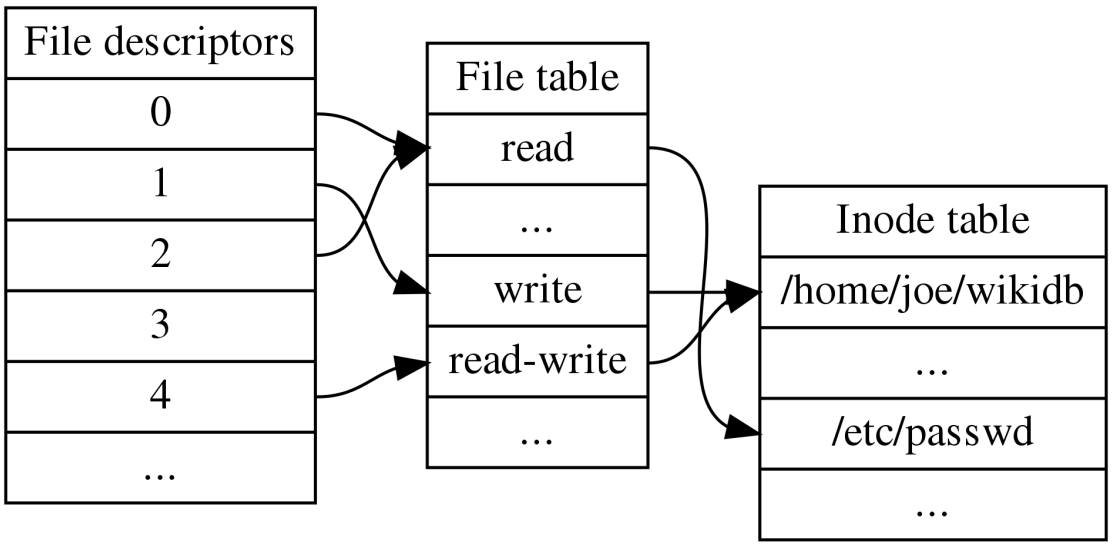
VE482 P1-Pre Group 6

Members: Haorong Lu, Shengyuan Xu, Yuhong Zhan, Kaiyang Chen

File I/O redirection?

- By default, a UNIX command reads from `stdin`, and writes to `stdout`
- Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell.
- By the file I/O redirection, we can change the files the command reads from and writes to
- In Project 1, we need to implement three kinds of file I/O redirection
  - Input redirection `<`, e.g. `cat < 1.txt`, using 1.txt as the input of `cat`
  - Output redirection (overwriting) `>`, e.g. `echo 123 > 1.txt`, writing `123` to file `1.txt`
  - Output redirection (appending) `>`, e.g. `echo 123 >> 1.txt`, appending `123` to the end of file `1.txt`
- They may also appear in one command,
  - e.g. `cat < 1.txt > 2.txt`, copying the content of `1.txt` and writing to `2.txt`

File Descriptor

- In Unix and Unix-like OS, a **file descriptor** is a unique identifier for a file or other input/output resource, such as `int fd[2]; pipe(fd);`.
  - **File descriptors index into a *per-process* file descriptor table, which in turn indexes into a system-wide table of files opened by all processes, called the file table.**
- The **file table** records the mode with which the file has been opened. It also indexes into the **inode table** that describes the actual underlying files (Unix-style file system).
- 

- Each Unix process has three standard POSIX file descriptors,

    - 0 for `stdin`
    - 1 for `stdout`
    - 2 for `stderr`
- `command fd>file` `>file by default is 1>file`
- `command fd<file` `<file by default is 0<file`

`dup()`, `dup2()`

```c
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

// example
dup(in_fd);      // (lowest-unused) 0 -> (in_fd -> file)
dup(out_fd);     // (lowest-unused) 1 -> (out_fd -> file)

dup2(in_fd, 0);  // 0 -> (in_fd -> file)
dup2(out_fd, 1); // 1 -> (out_fd -> file)
```

- The `dup()` system call allocates a new file descriptor that refers to the same open file descriptor as the descriptor `oldfd`. The new file descriptor number is guaranteed to be the **lowest-numbered file descriptor that was unused in the calling process**.
- The `dup2()` system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the **file descriptor number specified in** `newfd`.
    - If the file descriptor `newfd` was previously open, it is closed before being reused.

Implementation!

Other Useful System Call

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags, mode_t mode);

#include <unistd.h>
int close(int fd);

// example
int out_fd = open(out_file, O_WRONLY | O_CREAT | O_APPEND, 0666);
// do something
close(out_fd);
```

- The `open()` system call opens the file specified by **pathname**.
- The return value of `open()` is a file descriptor referring to the open file.
- **Flags** we may use:

    - `O_RDONLY`: read-only
    - `O_WRONLY`: write-only
    - `O_CREAT`: create the file if not exist
    - `O_TRUNC`: the file will be truncated to length 0 if writing is allowed
    - `O_APPEND`: the file is opened in append mode
- **Mode**: 0666, refers to `rw-rw-rw-`, allow all users to read/write

- The `close()` system call closes a file descriptor, so that it no longer refers to any file and may be reused.
- `close()` returns 0 on success. On error, -1 is returned, and `errno` is set appropriately
    - This feature is very useful for error handling

Final Implementation (Assuming no error)

- First, we parse the input line to collect I/O redirection notations `<, >, >>` and corresponding files
- Then we record the redirection information to a per-command data structure `redirect_t`
- Finally, in the child process, we open corresponding files, and use `dup2()` (for simplicity) to duplicate their file descriptor to `stdin/stdout`
- The core redirection code is shown as below,

```
typedef struct {
    int in;
    int out;
    char *in_file;
    char *out_file;
} redirect_t;

void redirect_fd(redirect_t *r) {
    if (r->in == REDIRECT_IN) { /* < input redirection */
        int in_fd = open(r->in_file, O_RDONLY, 0666);
        dup2(in_fd, 0);
        close(in_fd);
    }
    if (r->out == REDIRECT_OUT) { /* > output redirection (overwrite) */
        int out_fd = open(r->out_file, O_WRONLY | O_CREAT | O_TRUNC, 0666);
        dup2(out_fd, 1);
        close(out_fd);
    } else if (r->out == REDIRECT_APPEND) { /* >> output redirection (append) */
        int out_fd = open(r->out_file, O_WRONLY | O_CREAT | O_APPEND, 0666);
        dup2(out_fd, 1);
        close(out_fd);
    }
}
```