

Issues involved with Programming a Shell

Program Execution
Unix I/O
File Redirection
Pipes

Basic Code Outline

```
while (1) {  
    get a line from the user  
    if line is a builtin command  
        process builtin  
    else  
        process external command  
}
```

Shell Built-in (internal) commands

Bash:

- set, echo, cd, true, false, type, ...
- if – then, while -do, for, test, ...
- history, fg, bg, kill, ...
- lots more: try “help”
- Homework #3:
 - set and print (like HW1)
 - where (find out what file corresponds to a command).
 - cd and pwd

External Commands (executable files)

- Shell needs to know where to look for executable files:
 - it would take too long to look in all directories
 - it is possible for there to be multiple executable files with the same name.
- Real shells use PATH environment variable:

```
PATH = /usr/bin:/usr/local/bin:/opt/foo:.
```
- HW3: use HWPATH environment variable.

Exec functions

- There are a bunch of them.
 - different ways to specify the command line.
 - different ways the executable is specified.
 - some support new environments
- For HW3 – you need to use `execl` or `execv`, not `execlp` or `execvp`.
 - the 'p' versions use `PATH`, you need to search yourself and use `HWPATH`.

I/O Redirection

- Your shell must support I/O redirection for standard input and standard output:

```
ls > savedls
```

```
sort < savedls
```

```
cat < savedls
```

```
cat savedls > anothercopy
```

etc.

Implementing I/O Redirection

- The general idea is to set up standard input (or output) before calling `exec()`.
 - open a file and assign it as standard input, or standard output.
 - the new program inherits `stdin` and `stdout`
 - The program was written to read from `stdin` and write to `stdout`, it doesn't have to know what these are actually connected to.

Unix File I/O

- C and C++ include standard libraries
 - `cin`, `cout`, `iostream`, `fstream`, ...
 - `fopen`, `fclose`, `fread`, `fwrite`, ...
- We will deal with Unix directly with system calls (not via libraries).
 - `open`, `close`, `read`, `write`, ...
 - system calls use “file descriptors” to refer to open files.

open ()

```
int open(const char *path, int flags, ...);
```

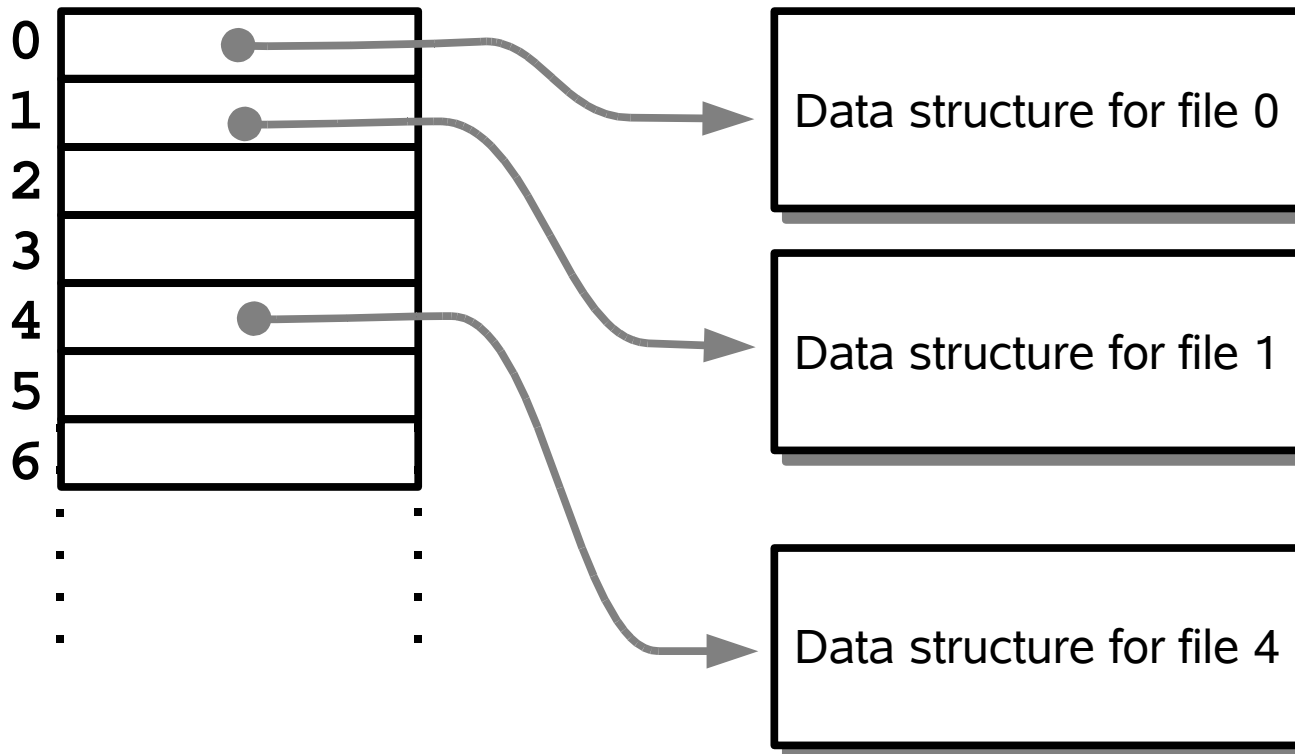
- path is C string containing a pathname.
- flags indicate what kind of access:

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NONBLOCK	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
... (there are more)	

open () return value

- Anything less than 0 means ERROR
 - check `errno` for specific error.
- `open()` returns a *file descriptor* (a small integer).
- The O.S. manages a *file descriptor table* for each process, a *file descriptor* is an index into this table.

File Descriptor Table



Using File Descriptors

- When calling Unix I/O system calls, you give them an open file descriptor:

```
write(fd, "hello" , 5 ) ;
```

```
read(fd, buff , 22 ) ;
```

```
close(fd) ;
```

- File descriptors can refer to open files, pipes, fifos, sockets, etc.

read()

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

- `fd` is an open file descriptor.
- `buf` is the address where you want read to store what it reads.
- `nbytes` is the maximum number of bytes
 - often the size of the buffer.

`read ()` continued.

- The return value is the number of bytes read, 0 means EOF. Anything less than 0 means ERROR (check `errno`).
- By default, `read` will block until there is some data available (it could return less than `nbytes`).
- For files, `read` will always read `nbytes` unless it hits the end of the file first.
 - reading from pipes and sockets is a little different.

write ()

```
ssize_t write(int fd, void *buf, size_t nbytes);
```

- `fd` is an open file descriptor.
- `buf` is the address of the bytes you want to write.
- `nbytes` is the number of bytes to be written.
- Write does not know anything about strings!
 - doesn't know about null termination, etc.

`write ()` continued.

- Return value is the number of bytes written. Negative number means an error occurred.
- Write can block!
 - max capacity of a pipe has been reached.
 - network buffer full.
- Typically doesn't block when writing to a file...

dup () and dup2 ()

- open creates a new file descriptor
 - typically the lowest file descriptor that is not currently used.
- dup (fd) will duplicate a file descriptor, creates a new file descriptor that refers to the same file as fd.
- dup2 (fd , newfd) will duplicate a file descriptor, the new descriptor will be newfd.

Using dup2

- dup2 can be used to assign an open file to stdin:

```
dup2(fd, 0);
```

```
/* or dup2(fd, STDIN_FILENO); */
```



Before \longrightarrow `dup2(fd, 0);` \longrightarrow After

Example: `sortfile`

- We want to make a command that will sort a file for us (never mind that `sort` will do this without any help...).

```
sortfile somefile
```

- `sortfile` will
 - open `somefile`
 - `dup2` the new file descriptor to `STDIN_FILENO`
 - `exec` the `sort` command.

sortfile.c

- Complete source is on the web.
- Abbreviated source here is missing lots of stuff
 - includes
 - error checking
 - comments...

sortfile.c (abbreviated)

```
int main(int argc, char **argv) {
    int fd;

    fd = open(argv[1], O_RDONLY);    /* errors! */
    dup2(fd, 0);                    /* errors! */
    execlp("sort", "sort", NULL);  /* errors! */

    /* should never get here */
    return(1);
}
```

Re-directing STDOUT

- Same idea, but now replace file descriptor 1
 - `STDOUT_FILENO` is a better name than 1
- Sample code: `logusers.c`
 - uses the `who` command to write list of logged-in users to a log file
 - appends to the log file each time run.
 - also writes timestamp to the file

logusers.c (abbreviated)

```
int main(int argc, char **argv) {
    int fd;
    char *curtime;
    time_t t;
    fd = open(argv[1], O_WRONLY | O_APPEND | O_CREAT, 0755);
    time(&t); curtime = ctime(&t);
    write(fd, curtime, strlen(curtime));
    dup2(fd, STDOUT_FILENO);
    execlp("who", "who", NULL);
}
```

Pipes

- Your shell must support command lines that include pipes:

```
ls | grep fred | sort
```

- The `ls` command should run, its output is sent to the `grep` command, which acts as a filter (printing only those lines that contain the string “fred”). The `grep` command's output is feed to the `sort` command, the result is a sorted list of all files whose named contain “fred”.

Unix Pipes



- A *pipe* is basically a buffer held by the kernel, and two file descriptors
 - one for reading, one for writing.
 - The pipe maintains the order of the bytes, so that the first byte written to the pipe is the first byte read from the pipe.
- The `pipe ()` system call creates a pipe and generates two file descriptors.

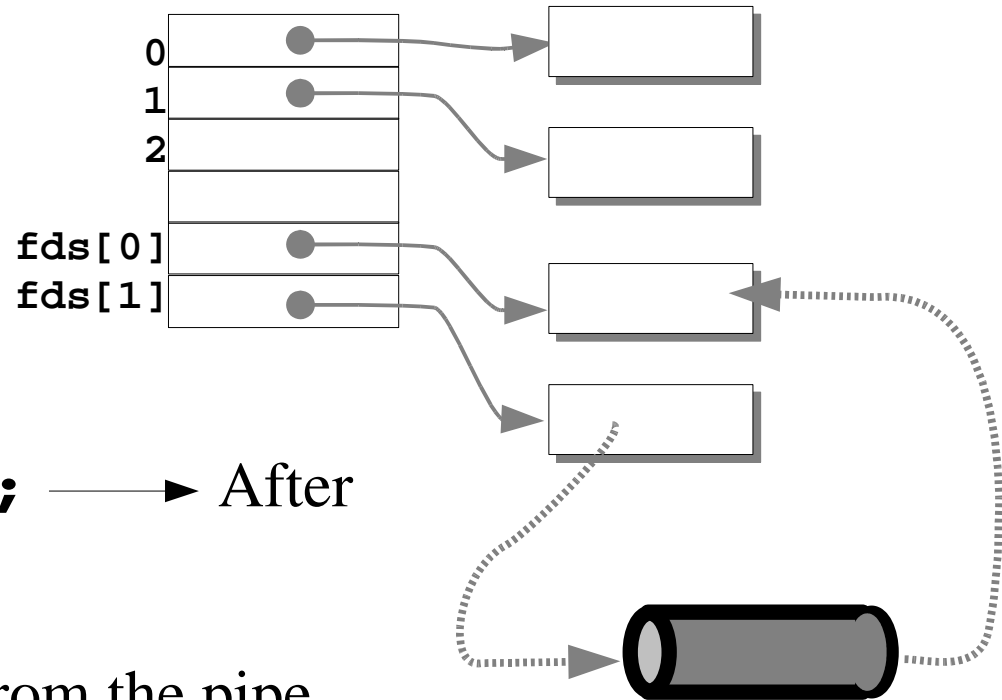
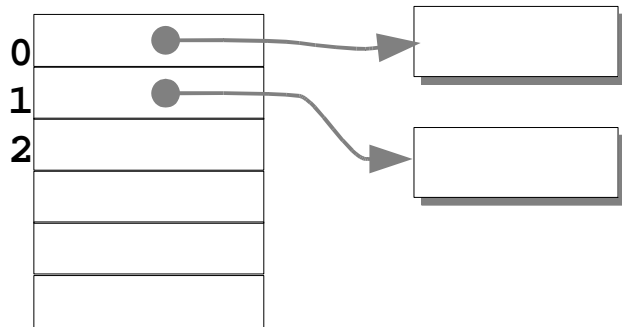
pipe ()

```
int pipe(int *fildes)
```

- `fildes` is an array of (at least) two `ints`. This array must exist before calling `pipe`!
- `pipe` returns 0 if successful, -1 on error.
- If successful, `fildes[0]` is a file descriptor for reading from the pipe, and `fildes[1]` is for writing to the pipe.

Calling `pipe()`

```
int fds[2];  
pipe(fds);
```



Before → `pipe(fds);` → After

`fds[0]` can be used to read from the pipe.
`fds[1]` can be used to write to the pipe.

Example: peterpiper.c (abbreviated)

```
int main(int argc, char **argv) {
    int n, fds[2];
    char buff[10];

    pipe(fds);
    write(fds[1], data, sizeof(data));
    close(fds[1]);
    while ( (n=read(fds[0], buff, 10)) > 0)
        write(STDOUT_FILENO, buff, n);
}
```

Pipe Capacity

- The O.S. has some buffer space that is used by the pipe – there is a limit.
 - if you hit the limit, write will block!

```
while (write(fds[1],data,BUFSIZE)>0) {  
    tot += BUFSIZE;  
    printf("pipe can hold %u\n",tot);  
}
```

Run this and see what the largest number printed is. At this point the process is blocked (it won't quit – just waits!). Hit ^C to kill it...

sortedls.c

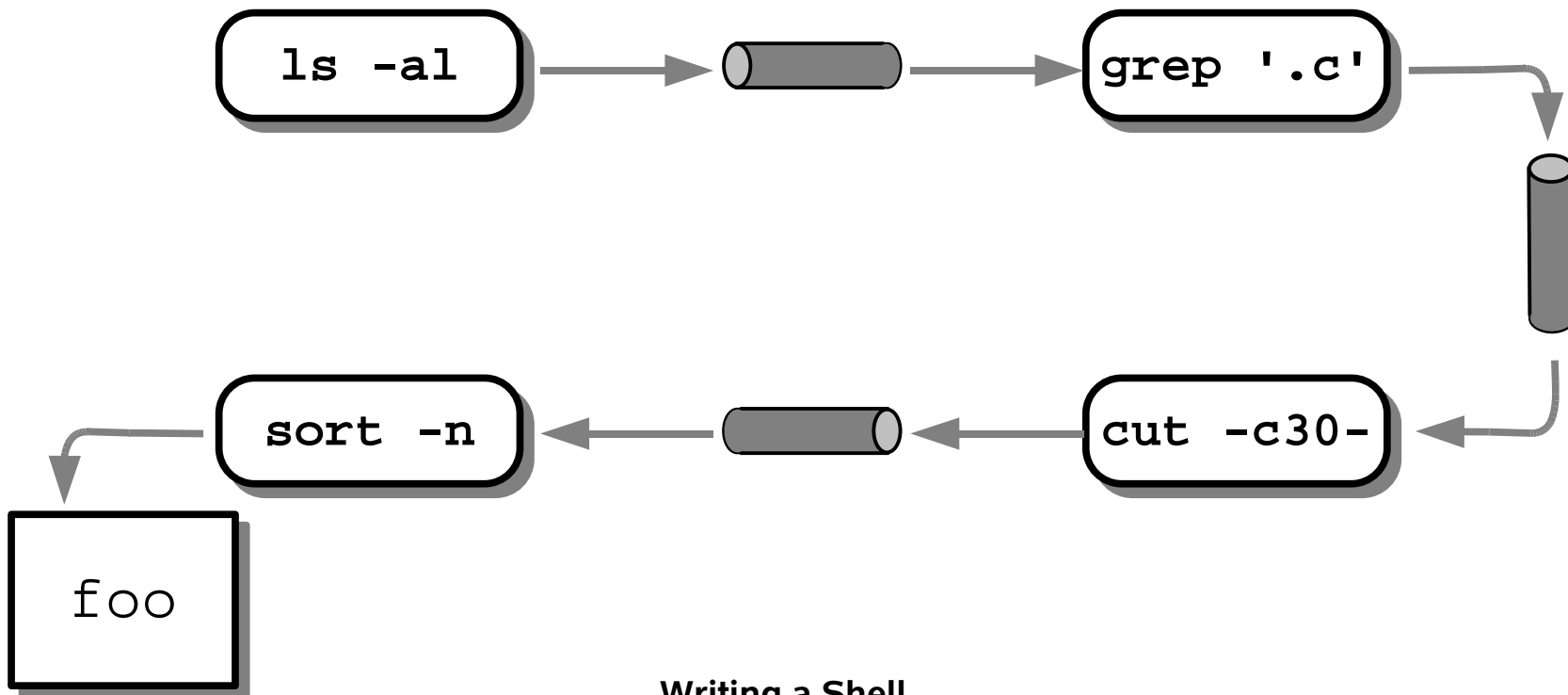
- Sample program:
 - creates a pipe.
 - forks
 - parent process attaches the writing end of the pipe to stdout.
 - child process attaches the reading end of the pipe to stdin.
 - parent exec's `ls`
 - child exec's `sort`.
- Result is basically this: `ls | sort`

sortedls.c (abbreviated)

```
int main() {
    int fds[2];
    pipe(fds);
    if (fork()) {
        dup2(fds[1],STDOUT_FILENO);
        execlp("ls","ls",NULL);
    } else {
        dup2(fds[0],STDIN_FILENO);
        execlp("sort","sort",NULL);
    }
}
```

Putting is all together

```
ls -al | grep '.c' | cut -c30- | sort -n > foo
```



Pipe Issues

- If you want two processes to share a pipe, you create the pipe first, then `fork()`.
 - The parent and child will have copies of the pipe file descriptors.
- Close any unused descriptors before calling `exec`
 - If any process has an open file descriptor that is the writing end of a pipe, EOF will never be seen!

Parsing Idea

```
ls -al | grep '.c' | cut -c30- | sort -n > foo
```

is

```
ls -al | something
```

- Parse the first command (everything before the first '|').
 - fork – child will become ls.
 - Have child create pipe, and fork.
 - child execs ls.
 - grandchild processes *something* (in the same way).