PennOS Companion Document

GROUP 7

Catherine Hu, Patrick Wingo, Jason Kahn, Sung Hwang
catheriz, pwingo, kajason, sunghw

November 2012

Contents

1 Introduction **3**

2 Process Related System Calls **4**
2.1 p_spawn(2) . . . 5
2.2 Signals . . . 6
2.3 p_kill(2) . . . 7
2.4 p_wait(2) . . . 9
2.5 Wait macros . . . 11
2.6 p_exit(2) . . . 12

3 Scheduling Related System Calls **13**
3.1 p_nice(3) . . . 14
3.2 p_info(3) . . . 15
3.3 p_sleep(3) . . . 16

4 File System Related System Calls **17**
4.1 f_open(4) . . . 18
4.2 f_read(4) . . . 20
4.3 f_write(4) . . . 21
4.4 f_close(4) . . . 22
4.5 f_unlink(4) . . . 23
4.6 f_lseek(4) . . . 24

5 Shell Commands **26**
5.1 ls . . . 27
5.2 cat . . . 27
5.3 nice . . . 27
5.4 nice_pid . . . 27
5.5 sleep . . . 27
5.6 busy . . . 27
5.7 touch . . . 27
5.8 rm . . . 27
5.9 ps . . . 27
5.10 man . . . 27
5.11 bg . . . 28
5.12 fg . . . 28
5.13 jobs . . . 28
5.14 logout . . . 28

5.15 whoami . . . 28
5.16 robertmead . . . 28
5.17 custom . . . 28
5.18 mv . . . 28
5.19 cd . . . 28
5.20 chmod . . . 28
5.21 cp . . . 29
5.22 grep . . . 29
5.23 quit . . . 29
5.24 pwd . . . 29
5.25 sudo . . . 29
5.26 uptime . . . 29
5.27 echo . . . 29
5.28 wc . . . 29
5.29 tail . . . 29
5.30 head . . . 29
5.31 find . . . 30
5.32 which . . . 30

6 Citations 31

Chapter 1

Introduction

Hey, I just met you
And this is crazy
But here's PennOS
So run it, maybe

These systems calls do
Really sweet things
But don't believe us
Just call them, maybe

Chapter 2

Process Related System Calls

The system calls in this chapter allow the user to manipulate processes. The user can create a new process, send predefined signals to specific processes, exit a process unconditionally, and make a process wait for child processes. Several wait macros are also defined to allow the user to obtain information about processes that have been waited on.

2.1 p_spawn(2)

Name

p_spawn – Creates a new process

Synopsis

```
#include <ucontext.h>
#include ‘‘schedule.h’’
#include ‘‘fatsyscalls.h’’

void p_spawn (void* func);
```

Description

`p_spawn()` creates a new process which executes the function referenced by *func*. The process inherits the process group id of the process that created it (its parent process), begins with a state value of RUNNING, and defaults to priority level MID_PRIORITY (0).

Some exceptions apply; for example, the shell process defaults to priority HIGH_PRIORITY (-1).

Errors

`p_spawn()` will execute successfully unless function pointer *func* is invalid (NULL or points to a non-existent function), in which case no new process is created and an error message is printed to the user.

Example Code

```
Sample code here
```

2.2 Signals

Name

Signals - overview of signals

Signal Dispositions

Every defined signal has a *disposition* which describes the default behavior of a process when it is delivered the signal.

Stop — by default, a process receiving this signal is stopped.

Cont — by default, a process receiving this signal continues if it is currently stopped.

Term — by default, a process receiving this signal terminates.

Ign — by default, a process receiving this signal ignores it.

Signal Masking & Pending Signals

Because we emulate asynchronous signal handling, there are times during which it is useful to block certain signals. . .

Standard Signals

PennOS supports the following standard signals:

Signal	**Val.**	**Disposition**	**Comment**
S_SIGSTOP	0	Term	Stop signal; stops a process if it's not already stopped
S_SIGCONT	1	Cont	Continuation signal; process is resumed if currently stopped
S_SIGTERM	2	Term	Termination signal; process is unconditionally terminated
S_SIGCHLD	3	Ign	Indicates that a child process was stopped or terminated
S_SIGALRM	4	Term	Timer signal; sent from system call `p_alarm(3)`

2.3 p_kill(2)

Name

p_kill – Sends a signal to a process

Synopsis

```
#include <ucontext.h>
#include ``schedule.h''
#include ``fatsyscalls.h''

void p_kill (int pid, int signal);
```

Description

`p_kill()` acts on the process associated with *pid* based on the signal specified by *signal*. Valid signals are defined on the Signals page of this chapter. The behavior of `p_kill()` with respect to each valid signal is as follows:

S_SIGSTOP

If the process control block of the process associated with *pid* has a *state* value of RUNNING or BLOCKED, *state* is changed to STOPPED for this process.

S_SIGCONT

If the process control block of the process associated with *pid* has a *state* value of STOPPED, *state* is changed to RUNNING for this process.

S_SIGTERM

The process associated with *pid* is terminated and its process control block value *normal_exit* is set to 0, or false, to indicate that the process was terminated by a signal. Any non-terminated child processes of this process become orphan processes (process control block *state* = ORPHANED).

S_SIGCHLD

If the process control block of the process associated with *pid* has an *is_waiting* value of 1, or true, and a *state* value of BLOCKED, then this process's *state* is changed to RUNNING and *is_waiting* is set to 0, or false. Otherwise, this signal is ignored.

S_SIGALRM

FILL THIS OUT

If there is no non-terminated or non-orphaned process with process id *pid*, an error message is printed. If *signal* does not match up with a defined signal, PennOS informs the user that the specified signal was unrecognized. In both cases, `p_kill()` does nothing.

Errors

`p_kill()` will execute successfully unless:

- Input *signal* is not a valid/recognized signal
- No process exists whose pid matches the input *pid*

- The process whose pid matches the input *pid* is a zombie or an orphan

If any of the above occurs, an error message is printed to the user and `p_kill()` does nothing.

Example Code

```
Sample code here
```

2.4 p_wait(2)

Name

p_wait – Sets process to blocked until child changes state

Synopsis

```
#include <ucontext.h>
#include ‘‘schedule.h’’
#include ‘‘fatsyscalls.h’’

wait_t* p_wait (int mode);

typedef struct wait_struct{
        int pid;
        int status;
} wait_t;
```

Description

`p_wait()` provides the calling process with the ability to “wait” on its child processes. If the calling process has no non-terminated children and its zombie queue is empty, `p_wait()` returns NULL. If the zombie queue is not empty, `p_wait()` populates the *last_wait* wait_t struct of the calling process with the process id *pid* and the termination status *state* of the first element in the zombie queue. This termination status is either TERMINATED, indicating a normal exit (e.g. from a call to `p_exit()`), or SIG_TERMINATED, indicating termination due to a signal. The calling process's zombie queue pointer is updated to the second element in the zombie queue, and the first element (process) is fully freed from memory.

If the zombie queue of the calling process is empty, but non-terminated child processes exist, the process control block value *is_waiting* of the calling process is set to 1 and its *state* is set to BLOCKED. The calling process, which has now been set up to wait for state changes of any kind from a child process (not only termination), then context switches into the next process selected by the scheduler.

A calling process which sets itself to waiting and blocked will not return from `p_wait()` until one of its child processes changes state and populates *last_wait* (a wait_t struct in its process control block) with the appropriate information (*pid*, *state*). Once this happens, the calling process is unblocked and returns *last_wait*. In this case, possible *state* values are TERMINATED, SIG_TERMINATED, STOPPED, and RUNNING.

The user may pass in W_WNOHANG as the input *mode*. If this is the case, behavior remains unchanged if there are no children and the zombie queue is empty or if there are already terminated child processes in the calling process's zombie queue. However, if non-terminated children exist and the zombie queue is empty, the specification of W_WNOHANG causes `p_wait()` to return immediately and print a message to the user stating that there are no immediately waitable child processes.

If *mode* is anyting other than W_WNOHANG, it is ignored.

Return Values

On success, `p_wait()` returns a wait_t struct containing the process id *pid* and state *status* of the last child process waited on by the calling process. On error, `p_wait()` returns NULL.

Errors

`p_wait()` will execute successfully unless:

- The calling process has no child processes, terminated or non-terminated
- W_WNOHANG was input as *mode* but although child processes exist, none of them are currently available to be waited on

If any of the above occurs, `p_wait()` returns NULL.

Example Code

```
Sample code here
```

2.5 Wait macros

Name

Wait macros - overview of wait macros

Description

These macros are provided to the user to aid in retrieving information about the last child waited on by the calling process—note that information can only be retrieved about the last child process waited on by the calling process, and not any other child processes.

W_WIFEXITED (int status)

Synopsis — `int W_WIFEXITED (int` *`status`*`);`

Return Values — Returns 1 (true) if exited normally, 0 (false) otherwise

W_WIFSTOPPED (int status)

Synopsis — `int W_WIFSTOPPED (int` *`status`*`);`

Return Values — Returns 1 (true) if was stopped by a signal, 0 (false) otherwise

W_WIFCONTINUED (int status)

Synopsis — `int W_WIFCONTINUED (int` *`status`*`);`

Return Values — Returns 1 (true) if was continued by a signal, 0 (false) otherwise

W_WIFSIGNALED (int status)

Synopsis — `int W_WIFSIGNALED (int` *`status`*`);`

Return Values — Returns 1 (true) if was terminated by a signal, 0 (false) otherwise

2.6 p_exit(2)

Name

p_exit – Exits current process unconditionally

Synopsis

```
#include <ucontext.h>
#include ``schedule.h''
#include ``fatsyscalls.h''

void p_exit ();
```

Description

`p_exit()` exits the calling process unconditionally.

Errors

The calling process is terminated unconditionally, but if non-terminated child processes of the calling process exist, they become orphan processes (process control block *state* = ORPHANED). These orphaned processes are inherited by the INIT process, which kills them off (i.e. frees them from memory).

Example Code

```
Sample code here
```

Chapter 3

Scheduling Related System Calls

The system calls in this chapter provide the user with a means to interact with the process scheduler. The user is able to change the priority level of a process, retrieve information about any existing process, and "sleep" processes (block them for a specified period of time).

3.1 p_nice(3)

Name

p_nice – Sets the priority of a process

Synopsis

```
#include <ucontext.h>
#include ``schedule.h''
#include ``fatsyscalls.h''

void p_nice (int pid, int prio);
```

Description

`p_nice()` sets the priority level (process control block value *priority*) of the process with process id *pid* to the input *prio*, which can be one of three possible values: HIGH_PRIORITY (-1), MID_PRIORITY (0), and LOW_PRIORITY (0).

Some exceptions exist; for example, the shell process (which initializes in priority level HIGH_PRIORITY (-1)) cannot be assigned to a different priority level.

Errors

`p_nice()` will execute successfully unless:

- No process exists with process id *pid*
- Priority level input *prio* does not correspond to a valid priority level value (-1, 0, or 1)

If any of the above occurs, `p_nice()` will print an error message to the user and no process will be set to a new priority level.

Example Code

```
Sample code here
```

3.2 p_info(3)

Name

p_info – Returns information about a process

Synopsis

```
#include <ucontext.h>
#include ‘‘schedule.h’’

info_t* p_info (int pid);

typedef struct info_struct{
        int pid;
        int status;
        char* command;
        int priority;
} info_t;
```

Description

`p_info()` returns an info_t struct containing information about the process with process id *pid*.

Return Values

On success, `p_info()` returns an info_t struct containing up-to-date information about the process associated with process id *pid*. If an error occurs, `p_info()` returns NULL.

Errors

`p_info()` will execute successfully unless no process exists with process id *pid*; in this case, an error is printed to the user and `p_info()` returns NULL.

Example Code

```
Sample code here
```

3.3 p_sleep(3)

Name

p_sleep – Sets a process to blocked for a period of time

Synopsis

```
#include <ucontext.h>
#include ‘‘schedule.h’’
#include ‘‘fatsyscalls.h’’

void p_sleep (int ticks);
```

Description

`p_sleep()` sets the calling process to BLOCKED, performs a context switch, and does not allow the process to resume until # *ticks* of the clock occur. Once complete, the process is reset to RUNNING.

`p_sleep()` does not return until the process resumes from being blocked, but it is possible for the process to be terminated by an S_SIGTERM signal while asleep.

Errors

`p_sleep()` will execute successfully unless the process is terminated by a S_SIGTERM signal.

Example Code

```
Sample code here
```

Chapter 4

File System Related System Calls

The system calls in this chapter allow the user to interact with the file system. With these calls, the user can create files, manipulate/edit existing files, and delete files.

4.1 f_open(4)

Name

f_open – open a file or possibly create it if it does not exist in the specified mode

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include ``fatsyscalls.h''

int f_open (const char* fname, int mode, char* user);
```

Description

Given an *fname* for an existent or non-existent file, `f_open()` returns the given file descriptor for the file once it has been opened. A file descriptor is a non-negative integer greater than 1 which can be used in the following file system calls: `f_read()`, `f_write()`, `f_close()`, and `f_lseek()`—please refer to the companion document pages for these specific system calls for more details.

If the file named *fname* has already been opened and has been assigned a file descriptor, `f_open()` will open a new file descriptor for the same file. However, reading from and writing to the same file with multiple file descriptors results in undefined behavior, especially when closing and deleting the file.

If successful, `f_open()` returns a new file descriptor whose value is the lowest value not already in use by a currently open file. The file descriptor must be a value greater than or equal to 2 because the values 0 and 1 are assigned to STDIN and STDOUT, respectively. This new file descriptor, the mode in which the file is opened, and the file offset value are stored in a table containing all the descriptors for each open file.

The second input parameter *mode* specifies the access mode in which the user is attempting to open the file *fname*. *mode* can be one of three defined modes: F_WRITE, F_READ, and F_APPEND. These modes correspond respectively to opening the file in read/write mode, read-only mode, and read/append mode:

F_WRITE (0)

When accessing a file in F_WRITE mode, `f_open()` truncates a file named *fname* if it already exists in the current directory or creates a file named *fname* in the current directory if it does not already exist. This mode allows for the user to read and write to the file as they please. The file offset is set to the beginning of the file – 0 – as the file has no content (size = 0).

F_READ (1)

When accessing a file in F_READ mode, the user may only read from the file—writing or appending is forbidden. If the specified file *fname* does not exist in the current directory, `f_open()` will return an error. The file offset is automatically set to the beginning of the file.

F_APPEND (2)

When accessing a file in F_APPEND mode, the user is able to both read and write to the file. If the file *fname* already exists in the current directory, it will not be truncated as it is in F_WRITE mode. However, if *fname* does not exist, `f_open()` will create a file with name *fname* in the current directory. The file offset is automatically set to the end of the file.

Before performing any of the aforementioned operations, `f_open()` checks the *user* input parameter to ensure that the specified *user* has the correct permissions to access, read from, and write to the file *fname*. If the file *fname* does not exist, it is created with default permissions and the owner of the file is set to the current user (*user*). If the file already exists, `f_open()` first checks to see if *user* is the owner/creator of the file. If so, `f_open()` checks the user-permissions of the file *fname* against the *mode* in which the *user* is trying to access the file. If *user* is not the owner/creator of the file *fname*, then `f_open()` checks the group of the file *name* against the *mode* in which the *user* is trying to access the file. If `f_open()` determines that the user does not have the correct permissions, an error is returned.

Return Values

On success, `f_open()` returns the new file descriptor. If an error occurs, `f_open()` returns -1.

Errors

`f_open()` will execute successfully unless:

- An attempt was made to open a non-existent file *fname* in F_READ mode
- The specified *mode* was invalid (i.e. not F_WRITE, F_READ, or F_APPEND)
- The specified *user* did not have the necessary permissions
- *fname* specified a directory instead of a file

If any of the above occurs, `f_open()` returns -1.

Example Code

```
#include <sys/types.h>
#include <sys/stat.h>
#include ``fatsyscalls.h''

int main(){
    int fd = f_open(``test.txt'', F_WRITE, ``user1''); // open file ``test.txt'' in write mode

    if (fd < 0){ // if f_open returns an error (-1)
       printf(``f_open error'');
       return 1;
    }

    if (f_write(fd, ``Hello world!'', 12) != 12){
       printf(``f_write error'');
       return 1;
    }

    return 0;
}
```

4.2 f_read(4)

Name

f_read – read a specified number of bytes into a char buffer

Synopsis

```
#include <unistd.h>
#include ``fatsyscalls.h''

int f_read (int fd, char* buf, int num_bytes);
```

Description

`f_read()` attempts to read *num_bytes* from the file associated with file descriptor *fd* into the char buffer *buf*. If *num_bytes* is 0, then `f_read()` returns 0 and *buf* is unaltered. The file pointer and specified offset associated with *fd* determine where in the file to begin reading from. This value can be changed by the mode the file is opened in or by the `f_lseek()` function—companion document pages f_open(2) and f_lseek(2) describe this in more detail.

If file descriptor input *fd* is 0 (STDIN_FILENO) or 1 (STDOUT_FILENO), then `f_read()` reads *num_bytes* from stdin or stdout. If *num_bytes* is greater than SSIZE_MAX, the behavior of `f_read()` is unspecified.

Return Values

On success, `f_read()` returns the number of bytes read into the char buffer *buf*. If *num_bytes* is not 0 but the file pointer is at the end of the file, `f_read()` returns 0. If the function is interrupted by some signal, `f_read()` returns 0. If there were fewer bytes available to read from the file (prior to EOF) than were requested, `f_read()` returns some number less than *num_bytes* representing the actual number of bytes read into character buffer *buf*. If an error occurred while reading, `f_read()` returns -1.

Errors

`f_read()` will execute successfully unless:

- The specified file descriptor *fd* is invalid (either does not exist or is not linked to a valid file)
- The number of bytes requested (*num_bytes*) is invalid

If any of the above occurs, `f_read()` returns -1.

Example Code

```
#include <unistd.h>
#include ``fatsyscalls.h''

int main() { //copy the input to the output
    char buf[MAX_SIZE];
    int n;

    while ((n = f_read(0, buf, MAX_SIZE)) > 0){ // fd = 0 because reading stdin
          f_write(1, buf, n);
    }

    return 0;
}
```

4.3 f_write(4)

Name

f_write – write a specified number of bytes to a file descriptor

Synopsis

```
#include <unistd.h>
#include ``fatsyscalls.h''

int f_write (int fd, const char* str, int num_bytes);
```

Description

`f_write()` writes *num_bytes* characters from character string *str* into the file referenced by file descriptor *fd*. The file pointer and specified offset associated with *fd* determine where to begin writing from. When complete, the file pointer is incremented by the number of bytes written to the file. If a user attempts to write to a file opened in F_READ (read-only) mode, `f_write()` returns -1. If the value of *fd* is 0 (STDIN_FILENO) or 1 (STDOUT_FILENO), *num_bytes* of *str* are written to stdin or stdout, respectively; otherwise, `f_write()` writes to the file associated with *fd*. The actual number of bytes written may be fewer than *num_bytes* if there is insufficient space in the file system's memory.

Return Values

On success, `f_write()` returns the number of bytes written to the file associated with file descriptor *fd*. If an error occurrs, `f_write()` returns -1.

Errors

`f_write()` will execute successfully unless:

- File descriptor *fd* is invalid or associated with a file currently open in F_READ mode
- The value of *num_bytes* is invalid

If any of the above occurs, `f_write()` returns -1.

Example Code

```
#include <unistd.h>
#include ``fatsyscalls.h''

int main(){
    int fd = f_open(``testfile.txt'', F_WRITE, ``user1'');

    if (fd < 0){
       printf(``f_open error'');
       return 1;
    }

    if (f_write(fd, ``Hello world!'', 12) != 12){ // write to testfile.txt
       printf(``f_write error'');
       return 1;
    }

    return 0;
}
```

4.4 f_close(4)

Name

f_close – close a file descriptor

Synopsis

```
#include <unistd.h>
#include ``fatsyscalls.h''

int f_close (int fd);
```

Description

`f_close()` closes the file associated with file descriptor *fd*. Upon successful execution, *fd* no longer references any open file and is available for reuse. As part of the execution of `f_close()`, the stored resources associated with the file descriptor *fd*—the file pointer value/offset value and the mode in which the file was opened—are freed from memory. If the file associated with *fd* has been opened more than once, it is associated with more than one file descriptor, behavior is undefined and `f_close()` uses the most recently created file descriptor associated with the file from a call to `f_open()`.

Return Values

On success, `f_close()` returns 0. If an error occurs, `f_close()` returns -1.

Errors

`f_close()` will execute successfully unless:

- File descriptor *fd* is invalid, or exists but is not currently being used by an open file
- File descriptor *fd* is 0 (STDIN_FILENO) or 1 (STDOUT_FILENO)

If any of the above occurs, `f_close()` returns -1.

Example Code

```
#include <unistd.h>
#include ``fatsyscalls.h''

int main(){
    int fd = f_open(``testfile.txt'', F_WRITE, ``user1''); // open a file descriptor fd

    if (fd < 0) {// file did not open successfully
       printf(``f_open error\n'');
       return 1;
    }

    /* PERFORM READ/WRITE OPERATIONS ON testfile.txt */

    if (close(fd) < 0){ // close the file descriptor fd
       printf(``f_close error\n'');
       return 1;
    }

    return 0;
}
```

4.5 f_unlink(4)

Name

f_unlink – remove a file

Synopsis

```
#include <unistd.h>
#include ``fatsyscalls.h''

int f_unlink (const char* fname);
```

Description

`f_unlink()` deletes the file specified by *fname* from the file system and removes the symbolic link from the linked list of files. However, users are restricted from deleting the "." and ".." files from the file system. If the file specified by *fname* is currently open in the file system, `f_unlink()` first calls `f_close()` to delete the associated file descriptor from the table and then proceeds to remove the file as described above. It is important to remember in this case that `f_close()` exhibits undefined behavior if the file it acts on has been opened with `f_open()` more than once.

Return Values

On success, `f_unlink()` returns 0. If an error occurs, `f_unlink()` returns -1.

Errors

`f_unlink()` will execute successfully unless:

- The file system and/or current directory does not contain a file named *fname*
- *fname* specifies a "." or ".." file

If any of the above occurs, `f_unlink()` returns -1.

Example Code

```
#include <unistd.h>
#include ``fatsyscalls.h''

int main() {
    int fd = f_open(``testfile.txt'', F_WRITE, ``user1''); // open a file descriptor fd

    if (fd < 0){
       printf(``f_open error'');
       return 1;
    }

    /* PERFORM SOME READ/WRITE ACTIONS ON testfile.txt */

    if (f_unlink(``testfile.txt'') != 0) { // delete the file ``testfile.txt''
       printf(``f_unlink error'');
       return 1;
    }

    return 0;
}
```

4.6 f_lseek(4)

Name

f_lseek – repositions a file descriptor's file pointer to the offset relative to whence

Synopsis

```
#include <unistd.h>
#include ''fatsyscalls.h''

int f_lseek (int fd, int offset, int whence);
```

Description

`f_lseek()` changes the file pointer value of an open file associated with file descriptor *fd*. The new file pointer value is calculated using the inputs *whence* and *offset* and then saved to the table of file descriptors. All values in `f_lseek()` are measured in bytes. The user must choose one of the following values for *whence*:

SEEK_SET (0)
The file pointer is set to the beginning of the file (0) plus *offset* bytes.

SEEK_CUR (1)
The file pointer is set to the file pointer's current position plus *offset* bytes.

SEEK_END (2)
The file pointer is set to the size of the file plus *offset* bytes.

The user should note that `f_lseek()` can set the file pointer beyond the total size of the file associated with *fd*. However, the size of the file does not change if this is the case—so if data is later written to the file when the file pointer is still set to this value, null bytes ('\0') are written into the gap between the original end of the file and the position of the file pointer.

Return Values

On success, `f_lseek()` returns the new file pointer value, measured as the total number of bytes from the beginning of the file. If an error occurs, -1 is returned.

Errors

`f_lseek()` will execute successfully unless:

- File descriptor *fd* is invalid
- *whence* is invalid (not SEEK_SET, SEEK_CUR, or SEEK_END)
- File descriptor *fd* is 0 (STDIN_FILENO) or 1 (STDOUT_FILENO), because PennOS does not allow users to call `f_lseek()` on stdin or stdout

If any of the above occurs, `f_lseek()` returns -1.

Example Code

```
#include <unistd.h>
#include ‘‘fatsyscalls.h’’

int main() {
    int fd = f_open(‘‘testfile.txt’’, F_WRITE, ‘‘user1’’); // open a file descriptor fd

    if (fd < 0){
       printf(‘‘f_open error’’);
       return 1;
    }

    char buffer[12];

    if (read(file,buffer,6) != 6){
       return 1;
    }

    printf(‘‘Print 1:  %s\n’’,buffer);

    if (lseek(file,6,SEEK_SET) < 0){
       return 1;
    }

    if (read(file,buffer,12) != 12){
       return 1;
    }

    printf(‘‘Print 2:  %s\n’’,buffer);

    return 0;
}
```

Example Output

```
shell> cat testfile.txt
Hello World!
shell> ./testing
Print 1: Hello
Print 2: World!
```

Chapter 5

Shell Commands

Our shell recognizes the commands outlined in this chapter.

5.1 ls

5.2 cat

5.3 nice

5.4 nice_pid

5.5 sleep

5.6 busy

5.7 touch

5.8 rm

5.9 ps

5.10 man

5.11 bg

5.12 fg

5.13 jobs

5.14 logout

5.15 whoami

5.16 robertmead

5.17 custom

5.18 mv

5.19 cd

5.20 chmod

5.21 cp

5.22 grep

5.23 quit

5.24 pwd

5.25 sudo

5.26 uptime

5.27 echo

5.28 wc

5.29 tail

5.30 head

5.31 find

5.32 kill

5.33 passwd

5.34 quota

5.35 date

5.36 cal

Chapter 6

Citations

These are our sources: