Advanced Programming in the UNIX Environment

Week 02, Segment 2: open(2) and close(2)

Department of Computer Science
Stevens Institute of Technology

Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

Standard I/O

Basic File I/O: almost all UNIX file I/O can be performed using these five functions:

- `open(2)`
- `close(2)`

- `read(2)`
- `write(2)`
- `lseek(2)`

creat(2)

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

creat(2) returns a file handle in write-only mode. To get a read-write file handle:

```
if ((fd = creat(path, mode) < 0 ) {
        /* error */
}
(void)close(fd);
if ((fd = open(path, O_RDWR) < 0) {
        /* error */
}
/* do stuff with 'fd' ... */
```

creat(2)

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

This interface is made obsolete by open(2).

creat() is the same as:

```
        open(path, O_CREAT | O_TRUNC | O_WRONLY, mode);
```

open(2)

```
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Returns: file descriptor if OK, -1 on error

oflag must be one (and only one) of:

- `O_RDONLY` - open for reading only
- `O_WRONLY` - open for writing only
- `O_RDWR` - open for reading and writing

and may be OR'd with any of these:

- `O_APPEND` – append on each write
- `O_CREAT` – create the file if it doesn't exist; requires *mode* argument
- `O_EXCL` – error if `O_CREAT` and file already exists. (atomic)
- `O_TRUNC` – truncate size to 0
- `O_NONBLOCK` – do not block on open or for data to become available
- `O_SYNC` – wait for physical I/O to complete

`open(2)`

```
#include <fcntl.h>
int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Returns: file descriptor if OK, -1 on error

Additional *oflag*s may be supported on some platforms:

- `O_DIRECTORY` – if path resolves to a non-directory file, fail and set errno to `ENOTDIR`
- `O_DSYNC` – wait for physical I/O for data, except file attributes
- `O_EXEC` – open file for execute only, fail if it is a directory
- `O_NOFOLLOW` - do not follow symlinks
- `O_PATH` – obtain a file descriptor purely for fd-level operations. (Linux >2.6.36 only)
- `O_RSYNC` – block read operations on any pending writes
- `O_SEARCH` – open for search only, fail if it is a regular file
- ...

openat(2)

```
#include <fcntl.h>
int open(const char *pathname, int oflag, ... /* mode_t mode */);
int openat(int dirfd, const char *pathname, int oflag, ... /* mode_t mode);
```

Returns: file descriptor if OK, -1 on error

openat(2) is used to handle relative pathnames from different working directories in an atomic fashion.

Here, *pathname* is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory.

open(2)

```
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Returns: file descriptor if OK, -1 on error

open(2) may fail for a surprising number of reasons. Some of the more common ones include:

- `EEXIST`: `O_CREAT | O_EXCL` was specified, but the file exists
- `EMFILE`: process has already reached max number of open file descriptors
- `ENOENT`: file does not exist
- `EPERM`: lack of permissions
- ...

~~fd = **open**(path, O_RDWR);~~
~~/* do stuff with fd */~~

```
if ((fd = open(path, O_RDWR) < 0) {
        /* error */
}
/* do stuff with fd */
```

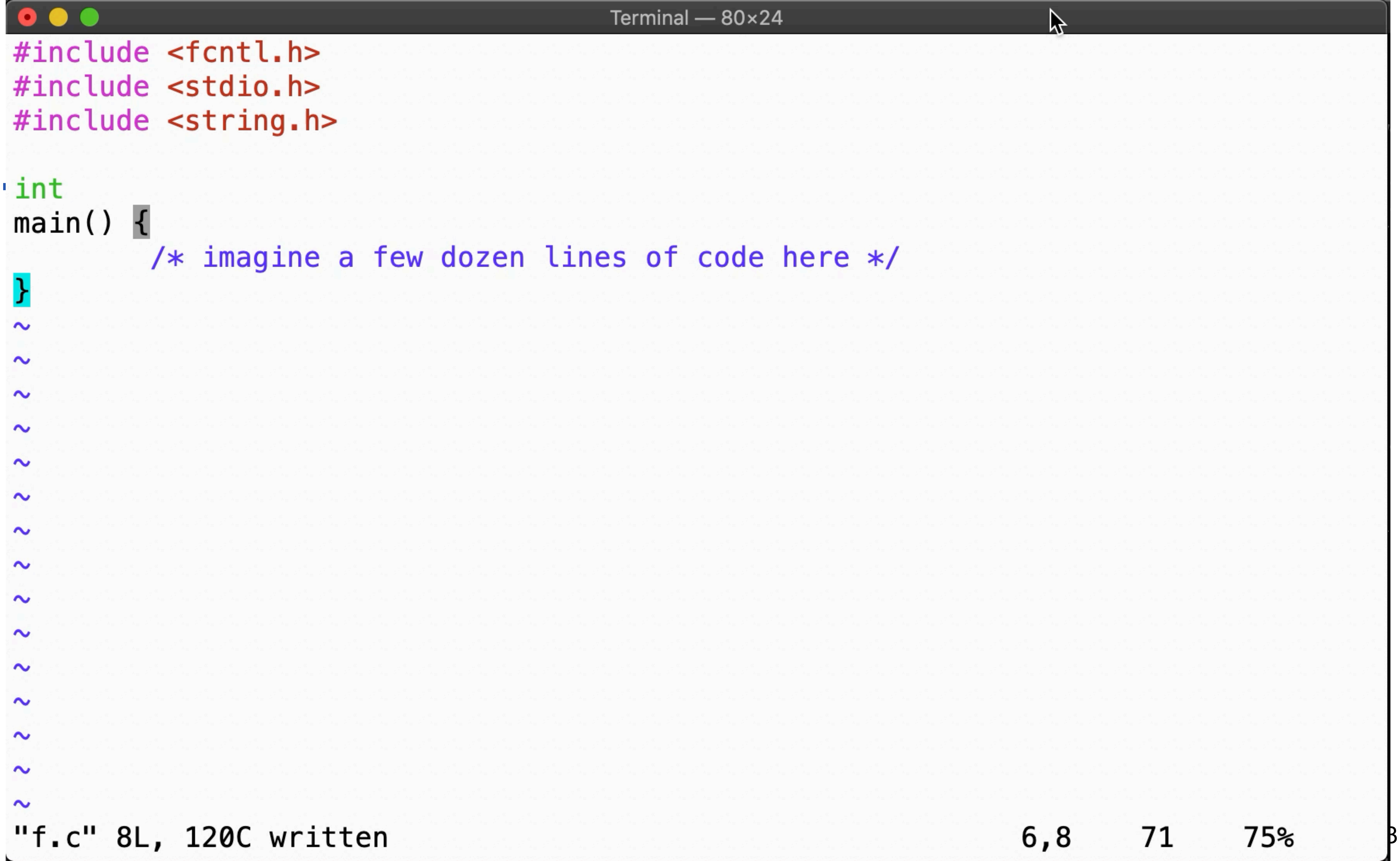
close(2)

```
#include <unistd.h>

int close(int fd);

                                        Returns: 0 if OK, -1 on error
```

- closing a filedescriptor releases any record locks on that file (more on that in future lectures)
- file descriptors not explicitly closed are closed by the kernel when the process terminates.
- to avoid leaking file descriptors, always close(2) them within the same scope


Terminal — 80×24

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

int
main() {
        /* imagine a few dozen lines of code here */
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"f.c" 8L, 120C written                                    6,8    71    75%
```

close(2)

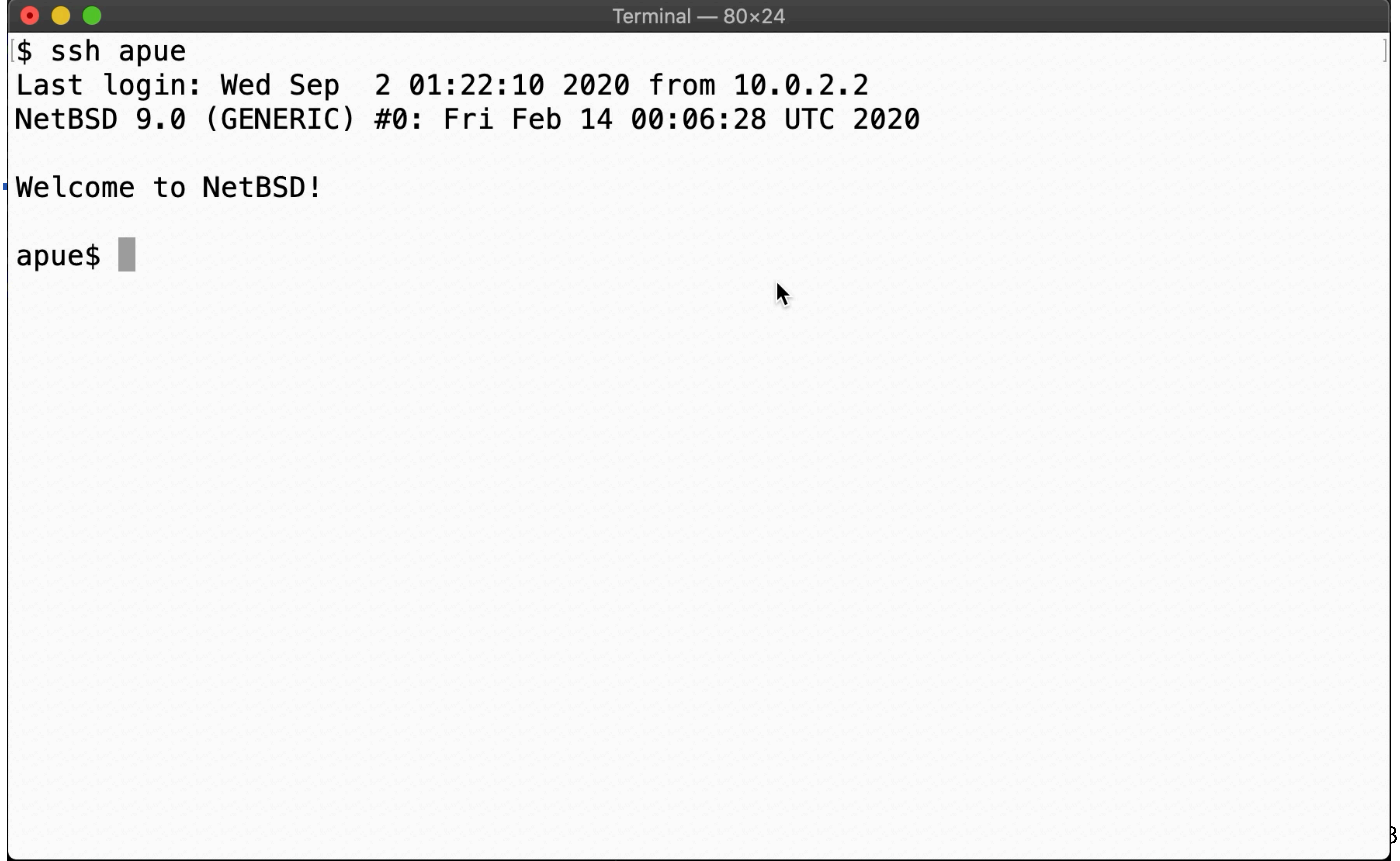
```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, -1 on error

- closing a filedescriptor releases any record locks on that file (more on that in future lectures)
- file descriptors not explicitly closed are closed by the kernel when the process terminates.
- to avoid leaking file descriptors, always close(2) them within the same scope

```
(void)close(fd);
/* you can't do stuff with fd here either way */
```



```
$ ssh apue
Last login: Wed Sep  2 01:22:10 2020 from 10.0.2.2
NetBSD 9.0 (GENERIC) #0: Fri Feb 14 00:06:28 UTC 2020

Welcome to NetBSD!

apue$ 
```

In our next segment...

- `read(2)`
- `write(2)`
- `lseek(2)`

Can you go backwards on a pipe?

What happens when you try to write data way beyond the end of a file?

How efficient is our `simple-cat.c` program?