Lecture 25: Unix I/O

David Hovemeyer

601.229 Computer Systems Fundamentals

Hello, world

```
#include <stdio.h>

// compile program with gcc -o hello hello.c
int main(void) {
  printf("Hello, world\n");
  return 0;
}
```

Where does the output go?

Hello, world

```
#include <stdio.h>

// compile program with gcc -o hello hello.c
int main(void) {
  printf("Hello, world\n");
  return 0;
}
```

Where does the output go?

To the terminal:

```
$ ./hello
Hello, world
```

Hello, world

```
#include <stdio.h>

// compile program with gcc -o hello hello.c
int main(void) {
  printf("Hello, world\n");
  return 0;
}
```

Where does the output go?

To the terminal:

```
$ ./hello
Hello, world
```

To a file:

```
$ ./hello > hello.out
$ cat hello.out
Hello, world
```

File handles

C provides `FILE*` data type to represent a "file handle"

- source of input and/or destination for output
- standard file handles: `stdin`, `stdout`, `stderr`
- can open named files/devices using `fopen`

File handles

C provides `FILE*` data type to represent a "file handle"

- source of input and/or destination for output
- standard file handles: `stdin`, `stdout`, `stderr`
- can open named files/devices using `fopen`

Many C I/O functions take a `FILE*` as a parameter:

```
printf("Hello, world\n");           // print to stdout
fprintf(stdout, "Hello, world\n"); // as above, but explicit
```

File handles

C provides `FILE*` data type to represent a "file handle"

- source of input and/or destination for output
- standard file handles: `stdin`, `stdout`, `stderr`
- can open named files/devices using `fopen`

Many C I/O functions take a `FILE*` as a parameter:

```
printf("Hello, world\n");          // print to stdout
fprintf(stdout, "Hello, world\n"); // as above, but explicit
```

How do file handles work?

File descriptors

Unix-based systems such as Linux and MacOS use *file descriptors* to refer to

- files
- devices (e.g., terminals)
- other kinds of communication channels (e.g., network connections)
- a file descriptor is just an integer
- a C file handle (`FILE*`) is just a "wrapper" for a file descriptor

How can we write programs to work with file descriptors? (And why is it useful to do that?)

Files, Unix filesystem

Files

Basic concept: *file*

Is a sequence of bytes: $b_0, b_1, b_2, \ldots, b_{n-1}$

For all files, bytes can be read and written sequentially

For some files, random access is possible (reading or writing at arbitrary positions in the sequence)

What is a file?

"File" has two related but distinct meanings:

What is a file?

"File" has two related but distinct meanings:

1. A sequence of bytes stored on a medium such as disk or SSD

What is a file?

"File" has two related but distinct meanings:

1. A sequence of bytes stored on a medium such as disk or SSD
2. Any object or device that can be treated as a file (for reading bytes and/or writing bytes)

The first meaning is a special case of the second meaning.

`Lots` of devices and objects in Unix are treated as files (in the second sense):

What is a file?

"File" has two related but distinct meanings:

1. A sequence of bytes stored on a medium such as disk or SSD
2. Any object or device that can be treated as a file (for reading bytes and/or writing bytes)

The first meaning is a special case of the second meaning.

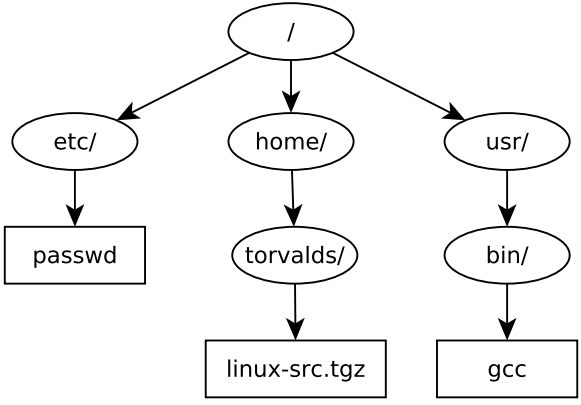
Lots of devices and objects in Unix are treated as files (in the second sense):

- Terminals
- Pipes (e.g., `cat myfile.txt | wc -l`)
- Network connections
- Peripheral devices
- And of course, files (in the first sense)

Filesystem

The Unix *filesystem* is a hierarchical namespace for files.

A *path* names the location of a file or directory in the namespace by describing how to find it, starting from the *root directory* (/), navigating through a sequence of zero or more intermediate directories.

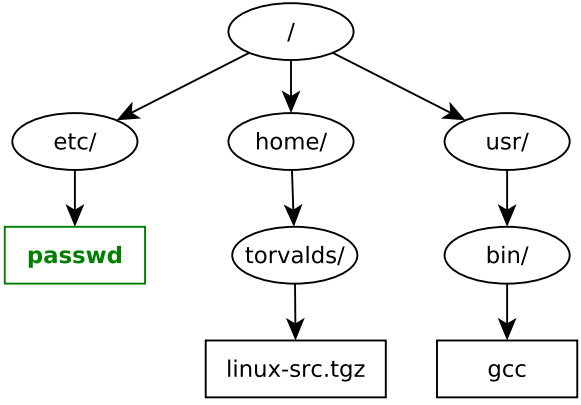


Filesystem

The Unix *filesystem* is a hierarchical namespace for files.

A *path* names the location of a file or directory in the namespace by describing how to find it, starting from the *root directory* (/), navigating through a sequence of zero or more intermediate directories.

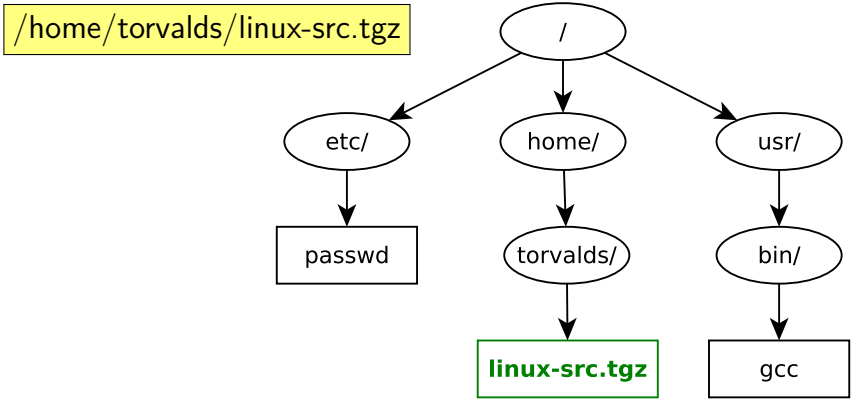
/etc/passwd



Filesystem

The Unix *filesystem* is a hierarchical namespace for files.

A *path* names the location of a file or directory in the namespace by describing how to find it, starting from the *root directory* (/), navigating through a sequence of zero or more intermediate directories.

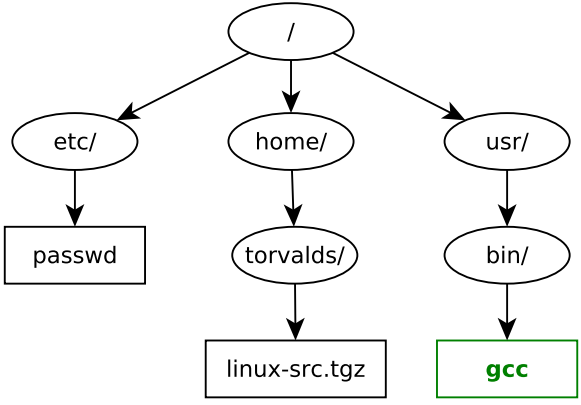


Filesystem

The Unix *filesystem* is a hierarchical namespace for files.

A *path* names the location of a file or directory in the namespace by describing how to find it, starting from the *root directory* (/), navigating through a sequence of zero or more intermediate directories.

/usr/bin/gcc



System calls, Unix I/O

System calls

A *system call* is a mechanism allowing a process (running program) to request a service from the operating system.

To the program, system calls are just function calls. (They are typically implemented using software interrupts.)

System calls are typically very low-level. Most programming languages provide a *run-time library* with higher-level functions.
For example:

- `open`, `read`, `write`: system calls
- `fopen`, `fread`, `fwrite`: run-time library functions

System calls and signals

Signals are a Unix mechanism for asynchronous notification.

They are similar to hardware interrupts, but are delivered to processes (running programs). The progam can register a *signal handler* function to receive them.

Example signals:

- `SIGSEGV`: segmentation violation (ever gotten this? ☺)
- `SIGINT`: interruption (sent when you type control-C in the terminal)
- `SIGALRM`: software timer

Issue: system calls can be interrupted if a signal is received.

Opening files

`open` system call:

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

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

Used for opening a *named* file (one that has a path in the filesystem.) Returns file descriptor (or -1 to indicate error.)

`flags`: one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, bitwise-or'ed with (optionally) `O_CREAT`, `O_TRUNC`, and/or `O_APPEND`

`mode`: access permission bits (only significant when flags contains `O_CREAT`, can omit otherwise)

Closing files

`close` system call:

```
#include <unistd.h>

int close(int fd);
```

Closes file named by specified file descriptor (fd).

Reading data

`read` system call:

```
#include <unistd.h>

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

Read `n` bytes from specified file descriptor, placing data read in `buf`. Returns number of bytes read, or -1 on error.

Short read: fewer than `n` bytes might be read because

- end of file was reached
- some data is not available yet (e.g., reading from a network connection)
- line buffering by terminal

Must check return value!

Dealing with short reads

```
// Try to read n bytes from fd.
// Returns number of bytes read, or -1 on error.
// Returns fewer than n bytes only if EOF is reached.
ssize_t read_fully(int fd, void *buf, size_t n) {
  char *p = buf;
  while (p < (char *) buf + n) {
    size_t remaining = p - (char *) buf;
    ssize_t rc = read(fd, p, remaining);
    if (rc == 0) {
      break;      // reached end of file
    } else if (rc > 0) {
      p += rc;    // read data successfully
    } else if (errno != EINTR) {
      return -1;  // an error occurred
    }
  }
  return (ssize_t) (p - (char *) buf);
}
```

Writing data

`write` system call:

```
#include <unistd.h>

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

Write `n` bytes from `buf` to specified file descriptor. Returns number of bytes written, or -1 on error.

Like `read`, can also return a short write (fewer than `n` bytes written.)

Exercise for reader: implement a `write_fully` function.

Buffered I/O

A program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int fd = open(argv[1], O_RDONLY);
  char *buf = malloc(10000000);
  for (int i = 0; i < 10000000; i++) {
    if (read(fd, buf + i, 1) != 1) {
      fprintf(stderr, "failed read?\n");
      return 1;
    }
  }
  close(fd);
  return 0;
}
```

A program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int fd = open(argv[1], O_RDONLY);
  char *buf = malloc(10000000);
  for (int i = 0; i < 10000000; i++) {
    if (read(fd, buf + i, 1) != 1) {
      fprintf(stderr, "failed read?\n");
      return 1;
    }
  }
  close(fd);
  return 0;
}
```

```
$ gcc -Wall -O2 -o r1 r1.c
$ time ./r1 largefile.mp3

real    0m5.334s
user    0m2.204s
sys     0m3.128s
```

Another program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int fd = open(argv[1], O_RDONLY);
  char *buf = malloc(10000000);
  if (read(fd, buf, 10000000) != 10000000) {
    fprintf(stderr, "failed read?\n");
    return 1;
  }
  close(fd);
  return 0;
}
```

Another program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int fd = open(argv[1], O_RDONLY);
  char *buf = malloc(10000000);
  if (read(fd, buf, 10000000) != 10000000) {
    fprintf(stderr, "failed read?\n");
    return 1;
  }
  close(fd);
  return 0;
}
```

```
$ gcc -Wall -O2 -o r2 r2.c
$ time ./r2 largefile.mp3

real        0m0.010s
user        0m0.000s
sys         0m0.010s
```

System call overhead

System calls (such as `read`) are not like ordinary function calls.

They require a call into operating system kernel (typically via a software interrupt). This requires:

- saving and restoring registers
- switching processor privilege levels
- checking system call arguments
- carrying out the system call (I/O, data transfer to/from program buffer)

This overhead can add up.

Buffering

`read` and `write` system calls can be made less frequent using *buffering*:

- Writing: store data in a buffer in memory, flush (write to actual file) when buffer has become full or before closing file
- Reading: read large chunks of data into buffer infrequently, retrieve data from the buffer when requested

`FILE*` objects (C file handles) have an internal buffer, so, e.g.

- `fgetc` doesn't call `read` each time
- `fputc` doesn't call `write` each time

A third program

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  FILE *in = fopen(argv[1], "rb");
  char *buf = malloc(10000000);
  for (int i = 0; i < 10000000; i++) {
    int c = fgetc(in);
    if (c < 0) {
      fprintf(stderr, "fgetc failed?\n");
      return 1;
    }
    buf[i] = (char) c;
  }
  fclose(in);
  return 0;
}
```

```
$ gcc -Wall -O2 -o r3 r3.c
$ time ./r3 largefile.mp3

real        0m0.042s
user        0m0.038s
sys         0m0.004s
```

Files and processes

What is a file descriptor?

A *file descriptor* is a small integer value identifying an open file.

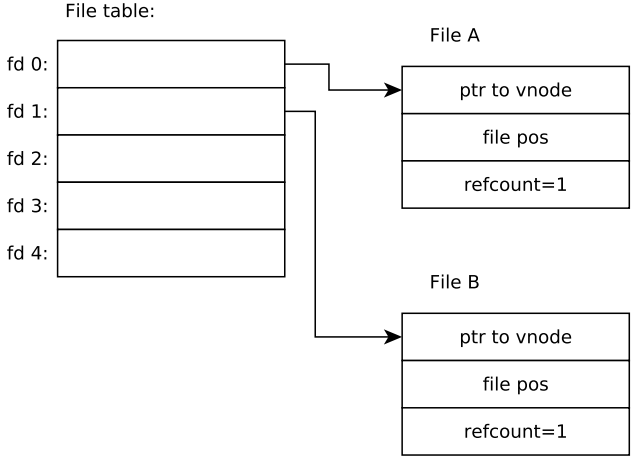
Really, it's an index into a *file table* belonging to the process. The file table is a kernel data structure (not directly accessible to the program.)

Each entry in the table points to a *file object*.

The file object contains:

- Pointer to a *vnode* ("virtual node")
- File position (offset of next byte to be read or written, only relevant for random-access files)
- Reference count

File table example



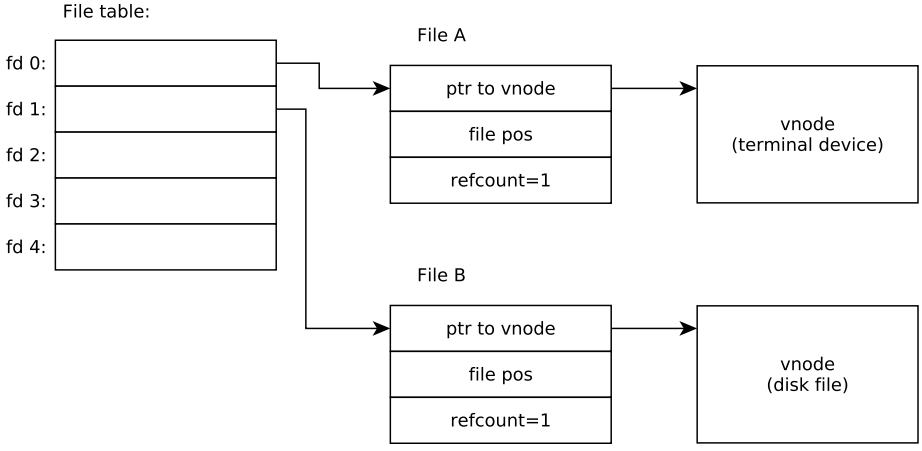
What is a vnode?

A *vnode* is an object representing the actual file data, in whatever its "true" form is

- file on disk or SSD
- terminal device
- pipe
- network connection
- etc.

Example on next slide: fd 0 (standard output) is a terminal, fd 1 (standard input) is a file on disk

vnode example



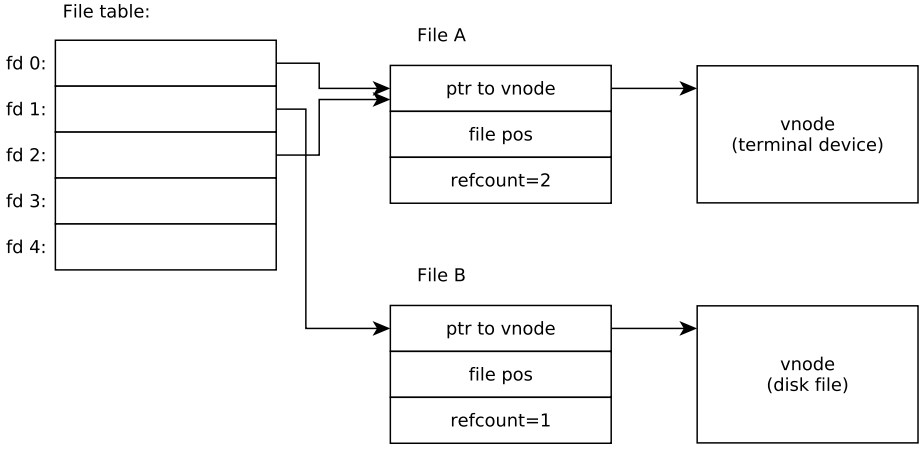
Duplication of file descriptors

More than one file descriptor (file table entry) can refer to the same file object.

The `dup` and `dup2` system calls create a new file descriptor pointing to the same file object as an existing file descriptor.

Example on next slide: fd 2 (standard error) also goes to terminal.

Duplicated file descriptor



Shared file objects

When a process creates a child process using the `fork` system call, the parent process's file table is inherited by the child.

This results in file objects being shared between parent and child.

Example on next slide: fd 0 and fd 1 in parent and child processes are shared following `fork` system call.

Shared file descriptor example