Engineering 8894/9875:

[Embedded and] Real-time [Operating] Systems [Design]

Lecture 3:

File abstractions

https://memorialu.gitlab.io/Engineering/ECE/Teaching/operating-systems/website/lecture/3/

Recall

Virtualization

Time-sharing: Each user has the impression of working on a dedicated computer

Virtual memory: Each process has the impression of a dedicated address space

How about files?

Files*

What is a file?

- an array of bytes that *persists*
- can be found via a *path*, e.g.:

```
/home/jon/Teaching/8894/website/config.yaml
```

- **don't** have user-meaningful names

* Reference: OSTEP Ch 39

Comparison with networks: "people get mad"

They *do* have not-terribly meaningful names... we'll explain a bit more about that soon!

Directories

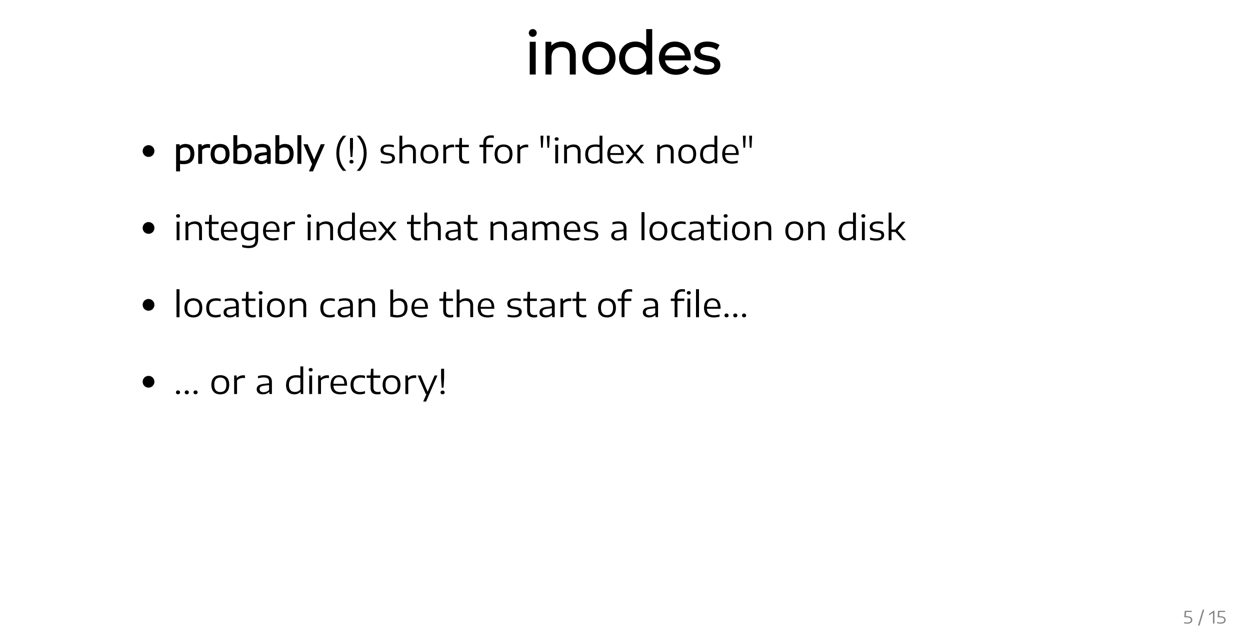
Folders?

Set of entries $(n \to i)$

n: user-meaningful name
i: a file's *inode*

e.g.: $(\texttt{foo.c} \to 925551)$

... um, but what's an inode?



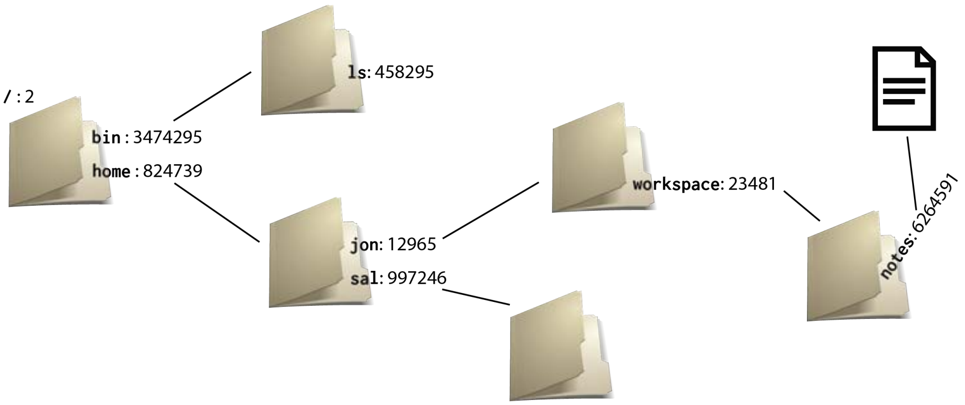
inodes

- **probably** (!) short for "index node"
- integer index that names a location on disk
- location can be the start of a file...
- ... or a directory!

This is the ____________ name that a file itself *really* has

So we can put ________________________________

Another directory hierarchy



... but so could `/home/jon/workspace/notes`!

But `/home/jon/TODO-notes` is *not* the name of the file!

Directory summary

Directories map names to inodes

inodes identify files and directories

Paths can be used to look up files…

… but files and paths change independently

- file contents can change without changing the path
- path can change without changing the file

Process file abstractions

But how do *processes* access files?

```
new PrintWriter(new BufferedWriter(new FileWriter(name))).write("Hello!");
```

```
std::cout << "Hello, world!" << std::endl;
```

```
fwrite(stdout, "Hello, world!\n");
```

```
char message[] = "Hello, world!\n";
int fd = STDOUT_FILENO;
write(fd, message, sizeof(message));
```

Process file abstractions

Each process has a set of integer *file descriptors**

Can use *system calls* to open, close, read, write, etc.

```
int fd = open("/home/jon/hello.txt", O_RDONLY);

/* ... */

write(fd, some_data_bytes, data_length);

/* ... */
```

* We'll come back to what a file descriptor means soon for now, just think of it as a small integer that identifies a particular file.

File I/O system calls

```
/* Open or close a file: */
int      open(const char *, int, ...);
int      close(int);

/* Sequential reading and writing: */
ssize_t  read(int, void *, size_t);
ssize_t  write(int, const void *, size_t);

/* Random reading and writing: */
off_t    lseek(int, off_t, int);
ssize_t  pread(int, void *, size_t, off_t);
ssize_t  pwrite(int, const void *, size_t, off_t);

/* Directories and metadata: */
int      mkdir(const char *, mode_t);
int      rename(const char *, const char *);
int      stat(const char *, struct stat *);
int      unlink(const char *);
```

Run `truss cat <filename>` for effect

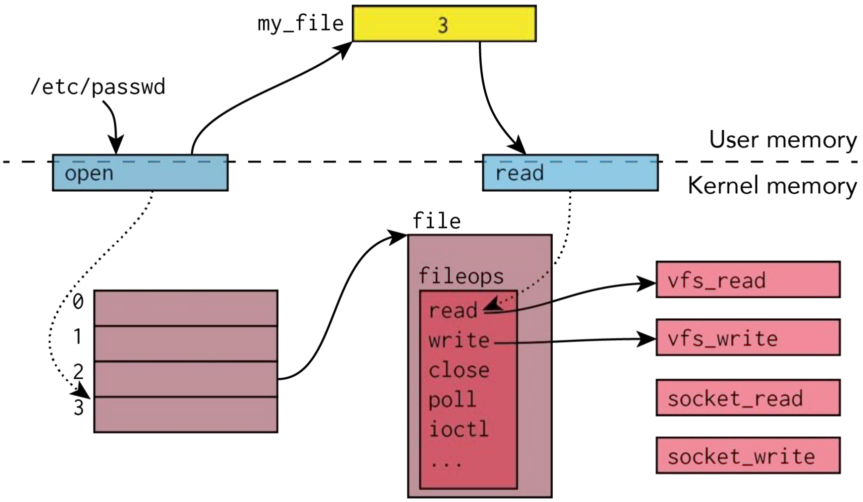
Show how C++ fstream code calls fread/fwrite, which (very indirectly) call read(2)/write(2)

File I/O system calls

From a process' perspective:

- system calls are C functions
- files are named by small integers (e.g., FD 3)
- … but what do these numbers really mean?
- is FD 3 in two processes the same file?

File descriptors



Show some syscall code

What about FD -1?

File descriptors

Small integer numbers

Indicies into a kernel array

- the contents of the array are a bit like objects
- like C "objects" in the tutorial!

Each process has *its own* file descriptor array

- my FD 4 != your FD 4