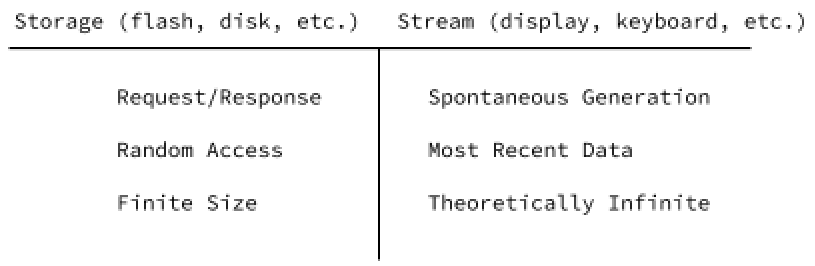
6. Files

Properties:
- I/O is very slow compared to CPU operations, so adding CPU overhead is fine
- robustness and security are the focuses
- our API must work for many types of devices

We are temped to make a complex API to handle all this, but that would make usage difficult!

WE WANT: a simple, portable API that can work with the two major I/O device types:
1. storage
2. stream



Storage (flash, disk, etc.)	Stream (display, keyboard, etc.)
Request/Response	Spontaneous Generation
Random Access	Most Recent Data
Finite Size	Theoretically Infinite

So can we write an API that can handle both?
~ of course! Linux did it! How?
~ by treating everything as a file!

BUT we have an issue: <u>random access</u>
SO in addition to the standard OPEN/READ/WRITE/CLOSE, we use

```
off_t lseek(int fd, off_t offset, int whence);
// where whence = SEEK_START || SEEK_END || SEEK_CURR}
// returns an error if used on a stream
```

The introduces the idea of API <u>orthogonality</u>:
we want features to be “at right angles” to each other
~ the function should be independent of the function call—read() has it, but not lseek()!

Here is a non-orthogonal API for reference:

```
int creat(const char *path, mode_t mode);
// using an open functionality requires a different function c
all to create a file!
```

Solution: introduce an O_CREAT flag to open()...and we dont need creat() any more!

Similar issues arise with concurrent lseek() & read/write, so we now have two special commands

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t of
fset);
```

BUT ORTHOGONALITY SOMETIMES FIGHTS CORRECTNESS

```
int unlink(const char *pathname);
int rename(const char *old, const char *new);
// these two function should be independent, but can we collid
e the axes?

// Process 1:
open("f", O_RDONLY);
// ...
read("f"...);

// Process 2:
// ...
unlink("f");

//somehow, the read does not fail
// Linux addresses this by waiting to delete the file until no
one has it open
```

Here is an example of when a lack of modularity opens security flaws:

```
# our commands:
$ touch secret
$ ls -l secret
-rw-r--r-- 0 ... secret
$ chmod 600 secret
$ ls -l secret
-rw------- 0 ... secret
$ echo $password > secret
# this is secure, right?...of course not!
# bad process:
$(sleep 100; cat) < secret
# a bad process could open it before you restrict the permissi
on and wait!
```

As a general rule, concurrent access to a file must be managed very carefully

```
$ (cat a & cat b) > file
# file becomes an aggregation of a and b
$ (cat > a & cat > b) < file
# file is split amongst a and b
```

The gap between file descriptors and files can create many such errors, including

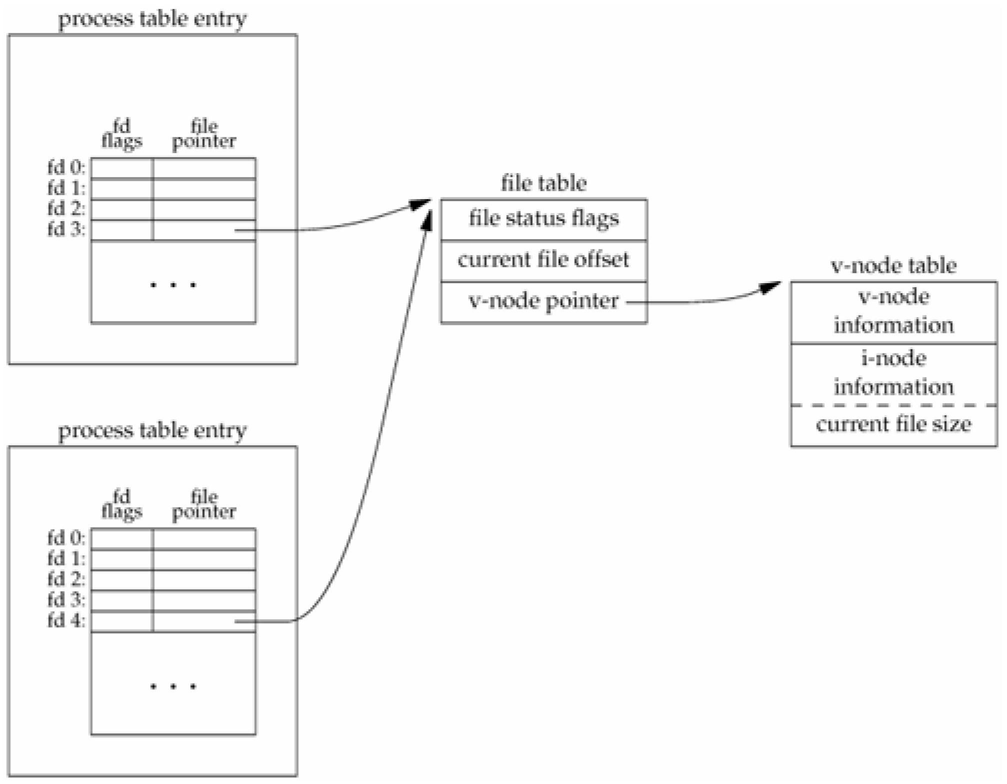
- races
- open but unnamed files (EMFILE = too many files)
- file descriptor leaks
- file descriptor not open (EBADF)
- I/O error (EIO—this one sucks to debug because it is so vague)
- EOF (return 0)
- device no longer there
- empty stream

In the case of the empty stream, we have two choices:

1. hang and wait for data
2. throw EAGAIN

But people will not be happy with either

File Descriptors



- File descriptors are indices in a table of pointers
  - vnode information includes “pipe or non-pipe” value
- On fork(), a copy of the parent’s fd table is made
- On dup(), the child’s pointer is removed, but the file remains for the parent
- Any number of the fd table entries may be present

How do these handle a complex case like $ cat file > output?

```
read(fd...);
write(1...);


// the shell must manipulate file descriptors to get the outpu
t! (like so:)
```

```
pid_t p = fork();
if (p==0) {
  int fd = open("output", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  dup2(fd, 1);
  close(fd); // so we do not write to the wrong file
  execlp("/usr/bin/cat", (char* []){"cat", "file", 0});
}
```

But this type of error can happen on accident as well

```
int fd = open("/tmp/foo", O_RDWR | O_CREAT, 0600);
if (fd < 0)
  if (unlink("/tmp/foo") != 0) abort();
// this thread does not let two concurrent threads run it!
```

Linux complicates the API to provide a solution to the earlier naming problem:

- O_EXCL ~ if the file already exists, an error is thrown

But we don't want failure...we don't care what the name is!

SOLUTION:

```
do {
  char buf[100];
  sprintf(buf, "tmp/foo%ld", random());
  fd = open(buf, O_RDWR | O_CREAT, 0600);
} while (fd < 0 && errno == EEXIST);
// but the temp directory could fill, and we would lose our (u
nlinked) file!
```

Here are a few more weird cases:

```
$ cat f > f
# f is left empty!
$ cat <f >f
# cat is emptied
```

```
$ cat <f >>f
# infinite loop! (so long as f wasn't empty)
```

To handle these cases, we need to introduce another structure called a “Pipe”