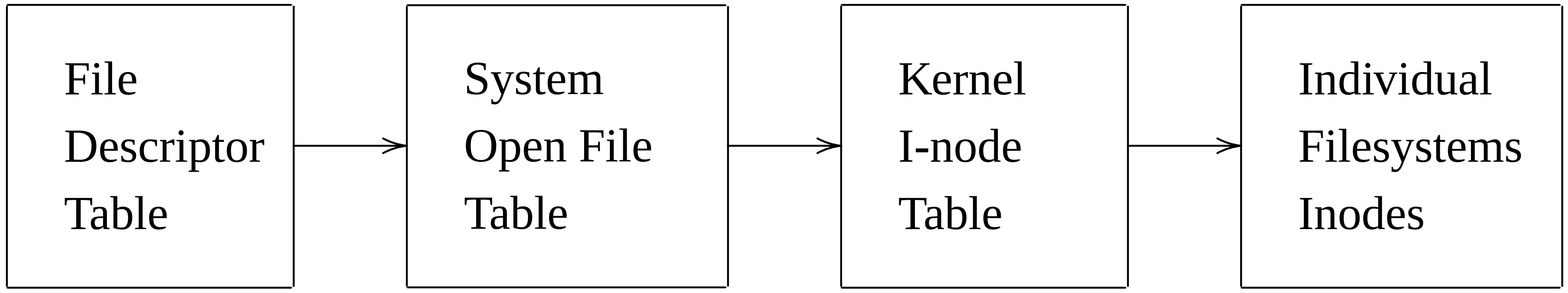
File Sharing

Data Structures for open files

The kernel maintains three distinct structures for open files:



Each table entry contains a pointer to an entry in the next table.

File Descriptor Table

Each process has its own File Descriptor Table.

- The File Descriptor Table is an array of open file descriptors for a process. A file descriptor is an index to this array.
- Each table entry consists of
  - A file descriptor flag: there is only one `FD_CLOEXEC`, which closes the file descriptor when an `exec` function is called. This is set using `fcntl(fd, F_SETFD, FD_CLOEXEC)`.
  - A pointer to an entry in the System Open File Table.
- The File Descriptor Table is copied on `fork` for the new process.
- The maximum number of open file descriptors a process can have is given by `OPEN_MAX` given in <`limits.h`>.

System Open File Table

The kernel maintains a system-wide Open File Table.

- Different processes can share the same Open File Table entry, but one process must be a descendant of the other (created by a chain of `fork`s).
- Each table entry consists of
  - The file status flags for the file (read, write, append, nonblocking, etc.) passed by call to `open`.
  - The current file offset
  - A counter of the number of file descriptors which currently point to this entry.
  - A pointer to the inode entry of the file in the Kernel I-node Table.
- A new open file entry is created on each call to `open`, so the same file can have several table entries.

Kernel I-node Table

The kernel maintains a table of recently accessed files.

- There is one entry in the Kernel I-node Table for each file.
- Serveral Open File Table entries may point to the same file entry in the Kernel I-node Table.
- The Kernel I-node Table is the kernel's abstraction of a file maintained by the Virtual File System. (Stevens calls these entries *v-nodes* to contrast them with the *i-nodes* which are actually implemented in each individual file system.)
- Each entry contains at least the following information:
  - File information as found on `struct stat`: type of file, size, ownership, etc. But *not* the name of the file!!
  - Filesystem device number and i-node in this filesystem, which together *uniquely* identifies the file.
  - Pointers to functions which operate on files in the filesystem (read, write, get i-node, update i-node, etc.)

Individual Filesystems

The Virtual Filesystem is an abstraction by the Kernel to hide many individual filesystems.

- Several operating systems may access the same filesystem. (Think when you log into two or more CS computers.)
- Filesystems have traditionally maintained file information (user, size, etc.) by means of *i-nodes*; but, this is not necessary for the Virtual Filesystem. POSIX now calls i-nodes *file serial numbers* and leaves their meaning and implementation to each filesystem.
- Traditionally, the name of the file is *not* kept with the i-node, but only stored in directories.
- Traditionally, each directory stores the name of the file and the inode of files within the directory.

Putting it together: I/O operations

- When a file is `open`ed the *lowest* file descriptor available is used on the process File Descriptor Table. An entry in the Open File Table is created, the `flag` parameter on `open` is copied to the entry, and the offset is set to 0, unless `O_APPEND` is specified, in which case the offset is set to the file size. The descriptor count on this table is set to 1. If the file is not located on the Kernel I-node Table, the kernel must create an entry here as well.
- After each `lseek` the Open File Table offset is adjusted, but no other action is taken.
- After a `close` the file descriptor entry is closed on the File Descriptor Table and the count for the relevant entry in the Open File Table is decremented. This entry is removed if the count reaches 0.

Putting it together: I/O operations

- After each `write` is complete, the current file offset in the Open File Table entry is incremented by the number of bytes written. If this causes the offset to exceed the current file size, the file size in the i-node table entry of the Kernel I-node Table is adjusted (and eventually, the i-node in the filesystem will be changed as well.)
- If the `O_APPEND` flag is set in the Open File Table, on each `write` the current file offset is set to the file size (the end of the file) obtained from the Kernel I-node Table.
- After a `fork` the File Descriptor Table is copied to the child and all counts for relevant entries in the Open File Table are incremented.

First Lesson in Synchronization: Atomic Operations

An *atomic operation* is an operation composed of multiple steps, but where all steps are performed together or none are.

- What is wrong with the following code:

```
lseek(fd, 0L, SEEK_END);  /* position to EOF */
write(fd, buff, 100);  /* append to EOF */
```

  (Multiple processes have access to the same file. It is possible that between the system calls `lseek` and `write` another process writes to the file.)

- Appending to a file is made atomic by using the flag `O_APPEND` when opening.

Second Lesson in Synchronization: File Sharing

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
int main(void) {
   FILE      *fs;
   fs = fopen("myfile", "w");
   fork();
   fprintf(fs, "Process %ld writing\n", (long)getpid());
   sleep(1);  /* Process doing other things */
   fprintf(fs, "Process %ld writing more\n", (long)getpid());
}
```

What could happen if parent and child were writing large blocks of data?

dup,dup2: duplicating file descriptors

dup,dup2: duplicating file descriptors

SYNOPSIS

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Return:

new file descriptor if OK -1 on error

Usage of dup,dup2

- Both `dup` and `dup2` create a copy of the file descriptor `oldfd`:
  - `dup` uses the lowest-number unused descriptor available for the new descriptor
  - `dup2` copies `oldfd` to `newfd` (closing `newfd` if necessary)
- Both `oldfd` and the new descriptor point to the *same* Open File Table entry, so share the same current file offset.
- The file descriptor flag (close-on-`exec`) is *not* copied on duplication–it is cleared on duplication.
- The most common use of duplicating file descriptors is for redirecting standard input and standard output.

Duplicating standard input

The following steps redirect standard input to *file*:

1. Open: `fd = open(`*file*`,...);`
2. Duplicate: `dup2(fd, 0);`
3. Close: `close(fd);`
4. Read, Write: `read(0,...);`, `write(0,...);`

Since`fd` was no longer needed it was closed. In general, it is a good practice to close file descriptors that will no longer be used, to conserve a limited system resource.

Filters

A filter is a program which reads from standard input, writes to standard output and reports errors to standard error.

- Examples of filters: `head`, `grep`, `sort`, `diff`, `cat`
- The shell handles redirection of standard input and output by duplicating file descriptors. Example:

  ```
  cat < my_input > my_output
  ```

  - `fork` for a new process
  - Redirect standard input and output in child
  - `exec` the `cat` program.
- The filter reads from standard input, writes to standard output and reports errors to standard error.

Why dup2: Atomic Operations

Consider the followining alternative to re-directing standard input

```
close(0);
open("my_input",O_RDONLY);
```

Usually, this is an acceptable alternative to `dup2`, but it duplicates in *two* operations. `dup2` is atomic and does it in *one* operation. If you have a signal handler and recieve a signal between closing standard input and opening `my_input` then your program will be executing input between these operations which may be a problem. Alternative lines of execution, such as signal handlers called to handle signals, can cause nasty difficulties in ensuring correct execution. Having an atomic duplication operation eliminates this problem.

Possible errors on dup, dup2

`errno`	
`EBADF`	`oldfd` is not an open file descriptor, or `newfd` is out of the allowed range for file descriptors
`EMFILE`	The process already has the maximum number of open file descriptors.