# Syscalls: Files 1: Library vs. Syscall I/O

1. **Library vs. Syscall I/O**
   - **file/filesystem syscalls**
   - **"files" in Linux/UNIX**
   - **library vs. syscall I/O**
   - **file descriptors**
   - **mixing library/syscall I/O**
2. Basic I/O Syscalls (part a)
3. Basic I/O Syscalls (part b)
4. File Offset
5. Filesystem Syscalls

# Syscalls: Files and Filesystems

There are a large number of *system calls* that deal with *files* and *filesystems*.

Some of these syscalls are involved with **input/output** (**I/O**) operations.

Other file-related syscalls involve operations on **filesystems**: creating or deleting files, changing file permissions, etc.

Together, these syscalls represent the Linux/UNIX OS' **API** for files and filesystems.

# Syscalls: Files and Filesystems (contd.)

Recall that it is the OS that is responsible for providing filesystems and for controlling file access.

Thus, all programs that perform I/O or make filesystem changes must ultimately make file syscalls.

This includes *library functions* such as `fopen()` and `fprintf()`.

It also includes *shell commands* such as `cat`, `rm`, and `touch`.

# "Files" in Linux/UNIX

A key aspect of the Linux/UNIX model is that a variety of entities can be treated as if they were normal *disk files* ("**regular files**").

These other entities—termed **special files**—include devices, directories, pipes, and sockets.

When we say special files can be treated like regular files, we mean that the same system calls we would use to interact with regular files can be used to interact with special files.

While this is generally true under Linux/UNIX, not every syscall can be used with all types of special files nor can every one be used in exactly the same way.

When we use the term **file** in the following slides, we will be referring to both regular and special files unless we specifically indicate otherwise.

# C Library vs. Syscall I/O

There are two frameworks for *input/output* (I/O) operations:

- **library (stream)** – C library provided functions
- **system call (file descriptor)** – OS provided functions

A key difference between these approaches are their **file handles**:

- library I/O uses **FILE*** type handles
- syscall I/O uses **integers** called **file descriptors**

Other important differences:

- library routines understand C data types, while syscalls understand only **bytes**
- library routines generally automatically **buffer** data, while syscalls do not (must be done manually if desired)

# C Library vs. Syscall I/O (contd.)

The I/O syscalls represent the API for I/O that is provided by Linux/UNIX OS's.

Programs can invoke the syscalls *directly* to have complete control over file I/O operations.

The syscalls are invoked *indirectly* when a program calls library I/O functions.

For example, `fopen()` sets up data structures required for C library I/O and then calls the syscall `open()` to actually open the file.

The `ltrace` and `strace` utilities can be used to gain a better understanding of the relationship between library functions and system calls.

# C Library vs. Syscall I/O (contd.)

The library I/O functions are easier to use than the syscalls, and will provide better efficiency by default.

In general one will want to use library I/O if it is appropriate.

So why study system call I/O:

- since the library I/O functions are ultimately implemented using syscalls, it helps one understand how library I/O works
- syscall I/O allows a programmer more control, so code can be tailored to achieve optimal efficiency for each application
- syscall I/O can be required for some special files and certain I/O functionality

# C Library vs. Syscall I/O (contd.)

Summary of key differences:

| | syscall I/O | C library I/O |
|---|---|---|
| open file handle | file descriptor (`int`) | stream (`FILE*`) |
| automatic buffering | no | yes |
| understands C types | no | yes |
| works for all file types | yes | no |

# File Descriptors

File-related system calls reference a file in one of two ways:

- via a filesystem **pathname**
- via a **file descriptor**

In order to be able to read from or write to a file, the file must have been **opened**.

Each **open file** is associated with a **file descriptor**—i.e., the **handles** for I/O syscalls are *file descriptors*.

File descriptors are *non-negative integers*: 0,1,2,....

# File Descriptors (contd.)

Each process maintains a table of its active file descriptors and their associated files in the process' **file descriptor table**.

The main element of each *entry* in a process' *file descriptor table* is a *pointer* to an **open file description**, which is an entry in the kernel's global **open files table**.

Each **open file description** in the *open files table* contains the following information about the associated open file:

- its access mode and status flags (read, write, append, etc.)
- the current **offset** (position for next I/O operation in file)
- a pointer to the actual file in the filesystem

# File Descriptors (contd.)

It is sometimes important to understand that the *flags* and *offset* are not stored in the per process *file descriptor table* but in the global *open files table*.

When a **child process** is created (via `fork()`), the child process gets a *separate file descriptor table*.

However, the entries in this table point to the *same entries* in the *global open files table* as those in the **parent process**.

This means that the parent and child process end up having (separate) file descriptors that point to the *same open file description*.

This can result in I/O operations in one process affecting those in other the process, most commonly by changing the file offset.

# Default File Descriptors

By default, *every process* automatically starts with the following three open "files" associated with standard file descriptors:

- FD **0**: **standard input**
- FD **1**: **standard output**
- FD **2**: **standard error**

While one can use the explicit FDs 0,1,2 in programs, `unistd.h` defines the following symbols that are better to use:

- `STDIN_FILENO`
- `STDOUT_FILENO`
- `STDERR_FILENO`

# Default File Descriptors (contd.)

These file descriptors (and their symbolic names) correspond to the C library I/O streams `stdin`, `stdout`, `stderr`.

Note that while the files associated with the three default file descriptors are typically going to be the *terminal*, this can be changed when a program is run.

**Piping** and **redirection** both cause changes in the files associated with the standard FDs.

While there are calls that allow a program to determine the device that is associated with an FD, this is rarely necessary or desirable.

In other words, it is considered a "feature" that a program thinks it is reading from the terminal when it is actually reading from a file.

# Default File Descriptors (contd.)

Consider the following pipeline command:

```
ls -l | grep "Jun" > jun-files
```

The pipeline and redirection affect the default FDs as follows:

- FD 1 (standard output) in `ls` is associated with the *write end of the pipe*
- FD 0 (standard input) in `grep` is associated with the *read end of the pipe*
- FD 1 (standard output) in `grep` is associated with the open file `jun-files`

# Mixing Library and Syscall I/O

It is usually *not* a good idea to *mix* the use of library and syscall I/O functions with the same open file.

The reason for this is that library I/O is normally buffered while syscall I/O is not, which can result in reading/writing being done in the wrong place in the file.

However, there are occasions where one wants to be able to use both types of I/O calls, or one has one type of handle but needs to use the other type of calls.

The following library functions useful for such situations:

- `fdopen()` – creates and returns a stream (`FILE*`) for an existing file descriptor
- `fileno()` – returns the file descriptor for a stream