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

Syscalls: Files 2: Basic I/O Syscalls

1. Library vs. Syscall I/O
2. **Basic I/O Syscalls** (part a)
   - **open**
   - **close**
3. Basic I/O Syscalls (part b)
4. File Offset
5. Filesystem Syscalls

The Basic I/O System Calls

The set of basic I/O-related syscalls is small:

- **open()** – open a file, possibly creating it
- **close()** – close an open file
- **read()** – read from an open file
- **write()** – write to an open file

While much of the time these four syscalls will be all that are needed, there are a few additional I/O syscalls.

The additional calls can read/write at specific positions in files, deal with particular special files like sockets, etc.

open() System Call

`open()` is the syscall to *open* a file, making it possible to read from and/or write to the file.

It can potentially also cause the file to be *created*.

It has two possible syntax variations:

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

- `pathname` is the file path as a C string
- `flags` specifies the file **access mode** and possibly other options, such as file creation
- `mode` specifies the file *permissions*, and should be provided if the file might be created by the call (`O_CREAT` flag)
- returns a *file descriptor* handle for the open file, else -1 on *error*

open() System Call (contd.)

`flags` *must include one* of the following three **access mode** specs, which determine what operations can be done to the file:

- **O_RDONLY** – allow only reading
- **O_WRONLY** – allow only writing
- **O_RDWR** – allow both reading and writing

Optional flags may be included by *bitwise-or'ing* them with the access mode spec using the C bitwise-or operator ("|"):

```
O_WRONLY | O_CREAT | O_EXCL
```

open() System Call (contd.)

The most useful/common `open()` flags are:

- **O_APPEND** – open file in **append mode**, meaning any writen data is appended to the end of the file automatically
- **O_CREAT** – create the file if it does not already exist
- **O_EXCL** – in conjunction with `O_CREAT`, causes an error if the file already exists
- **O_TRUNC** – if the file exists (and the access mode includes writing), file will be **truncated** (to length zero) when opened
- **O_CLOEXEC** – close the open file if an exec call is made
- **O_NONBLOCK** – open file in **nonblocking mode**, meaning a `read()` will immediately return even if there is no data to be read, etc

open() System Call (contd.)

If the `O_CREAT` flag is supplied so that the file may be *created*, the `mode` argument *should be supplied* (otherwise the file mode will end up being random, which can cause access problems later).

`mode` is often supplied as an *octal* value such as would be used with the `chmod` command:
`open("test",O_WRONLY|O_CREAT,0764)`
[the leading `0` in `0764` tells C to interpret the number as *octal*]

There are also a set of *symbolic constants* that can be bitwise or'd together to specify the desired mode/permissions:
`open("test",O_WRONLY|O_CREAT,S_IRWXU|S_IRGRP|S_IWGRP|S_IROTH)`

Use "`man 2 open`" for a list of the flag/mode symbols.

open() System Call (contd.)

`open()` always returns the *lowest-numbered* file descriptor not currently active for the process.

`open()` returns -1 in case of an error, plus `errno` will be set (> 0).

There are a number of reasons why `open()` can encounter an *error* and fail.

`errno`'s value can be checked to determine the particular error.

Among the most common `open()` errors are:

- **EACCES** – requested access denied due to permissions (user does not have permission for requested access mode)
- **EEXIST** – file exists and `O_CREAT` and `O_EXCL` were used
- **ENOENT** – file does not exist and `O_CREAT` was not used

open() System Call (contd.)

Example of opening an existing file for reading:

```
int infd;
if ((infd = open("test.text",O_RDONLY)) == -1) {
  perror("Opening file test.text failed");
  exit(EXIT_FAILURE); }
...
```

Notes:

- The file descriptor return will virtually always need to be *captured* since this is the handle to access the opened file.
- The return should *always* be checked to see if an error occurred because errors can definitely occur with `open()`.
- The common C idiom is to do both of the above together inside the `if` condition.

open() System Call (contd.)

Example of creating a *new* file and opening it for writing:

```
int outfd;
if ((outfd = open("newfile.text",O_WRONLY|O_CREAT|O_EXCL,0640)) == -1) {
  perror("Creating file newfile.text failed");
  exit(EXIT_FAILURE); }
...
```

Example of opening a file for reading and writing, *creating* the file if it does not exist or *truncating* it if it does:

```
int outfd;
if ((outfd = open("maybenewfile.text",O_RDWR|O_CREAT|O_TRUNC,0640)) == -1) {
  perror("Creating file maybenewfile.text failed");
  exit(EXIT_FAILURE); }
...
```

close() System Call

`close()` is the syscall that closes an open file:

`int close(int fd)`

- `fd` is the open file's file descriptor
- returns 0 (zero) on success, else -1 on error

close() System Call (contd.)

Example of closing an open file (descriptor):

```
int fd;
if ((fd = open(...)) == -1)
...
if (close(fd) == -1) {
  perror("Error closing file");
  exit(EXIT_FAILURE); }
```

Note that it is quite common to ignore the return from `close()` (not do any error checking on it).

Unless the file descriptor argument is wrong, an error is unlikely and nothing can be done about it anyway.

close() System Call (contd.)

When a process terminates, Linux/UNIX systems automatically close all of the process' open files.

This means there is little point in making a bunch of `close()` calls just prior to exiting.

However, in long running processes such as servers, it can be critical to close open files as soon as the program is finished with them:

- the maximum number of per process file descriptors is a *resource* that can be limited by the sysadmin
- open files take up memory in the process' *file descriptor table*
- open files take up memory in the global *open files table*

Syscalls: Files 3: Basic I/O Syscalls

1. Library vs. Syscall I/O
2. Basic I/O Syscalls (part a)
3. **Basic I/O Syscalls (part b)**
   - **read**
   - **write**
   - **blocking vs nonblocking mode**
4. File Offset
5. Filesystem Syscalls

read() System Call

`read()` is the basic *input* syscall:

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

- `fd` is the file descriptor handle
- `buf` is a pointer to the start of the memory block/array where the bytes read in will be stored
- `count` is the number of *bytes* to be read
- returns the number of bytes actually read ($\leq$ `count`), else -1 on error
- a return of 0 (*zero*) indicates an **end-of-file** condition

read() System Call (contd.)

The amount of data to be read is specified in *bytes*.

`read()` does not understand C types; the `sizeof` operator is commonly used to determine the number of bytes to read.

No checking is performed to ensure that the `buf` memory block is large enough to hold `count` bytes—this is left to the programmer!

A `read()` call changes only the first n bytes of `buf` where n is the actual number of bytes that were read (it does not "zero out" the rest of `buf`).

Because the number of bytes read may be less than the number requested, programs will generally have to *capture the return* from `read()` to know how much of `buf` is valid.

read() System Call (contd.)

There are effectively three classes of returns from `read()`:

- > 0: data was read
- 0: an *end-of-file* condition was encountered
- -1: an error occurred

The number of bytes actually read may be *less* than `count` for several reasons:

- less than `count` bytes remain in a regular file
- less than `count` bytes are currently in the kernel buffer for a pipe or socket
- the default behavior for terminals is to return the next *line*
- a `read()` call may be interrupted by a **signal**
- some devices return only the next "record" (e.g., tape drives)

read() System Call (contd.)

Example of reading (up to) 100 characters of ASCII text and storing it as a legal C string:

```
int infd, nread;
char str[101];
...
infd = open(...);
...
if ((nread = read(infd,str,100)) == -1) {
  perror("Error reading from file");
  exit(EXIT_FAILURE); }
//Ensure str is a legal C string:
str[nread] = '\0';
...
```

read() System Call (contd.)

Example of reading a *line* of ASCII text and storing it as a legal C string:

```
int infd, pos, nread;
char next, line[101];
...
infd = open(...);
...
pos = 0;
while ((nread = read(infd,&next,1)) > 0 && next != '\n')
  line[pos++] = next;
line[pos] = '\0';
if (nread == -1)...
...
```

read() System Call (contd.)

Example of reading an `int` (in binary):

```
int infd, nread, fileint = 0;
...
infd = open(...);
...
if (read(infd,&fileint,sizeof(int)) <= 0) {
  fprintf(stderr,"Problem reading int from file");
  exit(EXIT_FAILURE); }
...
```

write() System Call

`write()` is the basic *output* syscall:
`ssize_t write(int fd, const void *buf, size_t count)`

- `fd` is the file descriptor handle
- `buf` is a pointer to the start of the memory that contains the bytes to be written
- `count` is the number of *bytes* to be written
- returns the number of bytes actually written ($\leq$ `count`), else -1 on error

write() System Call (contd.)

Example of writing a C string to a file:

```
char *str;
int outfd;
...
outfd = open(...);
...
if (write(outfd,str,strlen(str)) == -1) {
  perror("Error writing str to file");
  exit(EXIT_FAILURE); }
...
```

Note: a C string's *null sentinel char* is generally *not* written out to text files (files containing null's are considered *binary files*).

write() System Call (contd.)

Example of writing an array of `int`'s (in binary) to a file:

```
int outfd, vector[10];
...
outfd = open(...);
...
for (int i=0; i<10; i++) {
  if (write(outfd,vector+i,sizeof(int)) == -1) {
    perror("Error writing int to file");
    exit(EXIT_FAILURE); }
  }
...
```

Blocking Mode

By default, I/O occurs in **blocking mode**.

In blocking mode, a `read()` or `write()` call can **block**—cause the process/thread to be *suspended*—until data is able to be read or written.

Blocking mode is important to understand, because it can be possible processes/threads to block *indefinitely!*

This is the case for I/O with pipes/FIFOs, sockets, terminals, and non-block devices.

E.g., if you try to `read()` from an empty pipe (that still has its write end open), the process/thread will block until some data becomes available in the pipe.

Blocking Mode (contd.)

Note, however, that I/O involving **regular files** or **block devices** cannot cause indefinite blocking.

This is because data being transferred from/to regular files and block devices gets *automatically* **cached** (**buffered**) by the kernel, to speed up reading/writing of files to secondary storage.

The effect is that `read()`/`write()` cannot block indefinitely:

- `read()` may have to wait on a disk read if the requested data is not in the **page cache**, but it will "block" only briefly
- `write()` usually returns immediately because the data simply gets written to the page cache (written to storage later), or may block briefly if **synchronized I/O** has been set

Nonblocking Mode

The possibility of `read()` or `write()` blocking indefinitely can be a problem for some applications.

Then, a file descriptor can be put into **nonblocking mode**.

When in nonblocking mode, a `read()` or `write()` call to an FD *always returns immediately*:

- if it can "immediately" read/write data, then it does (just as in blocking mode)
- if it *cannot* immediately read/write data (so call would *block*), -1 is returned (error), and `ERRNO` is set to `EAGAIN` or `EWOULDBLOCK`

Nonblocking mode can be appropriate to use when I/O involves pipes/FIFOs, sockets, terminals, and some devices.

Nonblocking Mode (contd.)

A file descriptor can be set to nonblocking mode in various ways:

- `open()` and `O_NONBLOCK` flag
- `fcntl()` and `F_SETFD` command, to set `O_NONBLOCK` flag
- `pipe2()` and `O_NONBLOCK` flag (Linux specific)
- `accept4()` and `SOCK_NONBLOCK` flag (Linux specific)

Syscalls: Files 4: File Offset

1. Library vs. Syscall I/O
2. Basic I/O Syscalls (part a)
3. Basic I/O Syscalls (part b)
4. **File Offset**
   - **file offset**
   - **lseek**
   - **multiple processes and open files**
   - **race conditions and atomicity**
   - **append mode**
   - **pread and pwrite**
5. Filesystem Syscalls

File Offset

The kernel's *open files table* holds each file's current **offset**: the position (in bytes) in the file where the next read/write operation will occur.

The terms **read/write pointer** or **read/write position** are also used for offset.

When a file is first opened, its offset is by default 0 (zero), which means the *start* of the file.

If a file contains n bytes, the offset would be n when the entire files has been read, and further calls to `read()` will cause 0 (zero) to be returned, meaning *end of file*.

Both `read()` and `write()` *automatically advance a file's offset* by the number of bytes that they actually read/write.

File Offset (contd.)

Most of the time code will simply use `read()`/`write()` and move sequentially through the file automatically.

Sometimes, though, it will be necessary to move to specific locations in a file, such as to read/write a particular **record** or to add data to the end of the file.

Some file devices are **seekable**: the file offset can be set to any desired value, and the next I/O operation will take place at that position in the file.

Seekable devices include regular disk files and tape drives.

Non-seekable devices include pipes, sockets, and terminals.

File Offset (contd.)

One interesting element of Linux/UNIX is that it is possible to set the current offset beyond the current end of a file.

While a call to `read()` would result in a zero/end-of-file return, a call to `write()` effectively causes the space between the old file end and the position where new data is to be written to be filled with zero bytes.

Such sequence of zero bytes are known as **file holes**, and the files that contain them are called **sparse files**.

Many Linux/UNIX filesystems can efficiently represent sparse files (i.e., they don't literally store the zero-byte sequences).

lseek() System Call

`lseek()` is the syscall for manually setting file offset:
`off_t lseek(int fd, off_t offset, int whence)`

- `fd` is the file descriptor handle;
- `offset` is an integer representing the new offset as a number of bytes relative to the file position specified by `whence` (a positive value means bytes past the position, a negative value means bytes before it)
- `whence` is one of:
  - `SEEK_SET` – file start (zero offset)
  - `SEEK_CURR` – current file position/offset
  - `SEEK_END` – file end
- returns the new offset (in bytes), else -1 on error

lseek() System Call (contd.)

If one attempts to call `lseek()` on a non-seekable file device, -1 is returned and `errno` is set to `ESPIPE`.

`lseek()` merely sets the offset value in the kernel's *open files table*, so it will not cause any movement of the devices associated with the open file.

lseek() System Call (contd.)

Set to start of file (`read()` first byte next):
`lseek(fd,0,SEEK_SET)`

Set to `read()` the 10th byte in the file next:
`lseek(fd,9,SEEK_SET)`

Set to end of file (`read()` would return end-of-file):
`lseek(fd,0,SEEK_END)`

Set to `read()` the last byte of file next:
`lseek(fd,-1,SEEK_END)`

Find out (return) the current file offset/position:
`lseek(fd,0,SEEK_CURR)`

Open Files and Multiple Processes

An important aspect of Linux/UNIX I/O is that *multiple processes* (running programs) can have the *same file open simultaneously*, and all be reading and/or writing to it.

If the processes are **unrelated**, each process' file descriptor will be associated with a *separate entry* in the kernel's *open files table*.

I.e., each process will have a separate offset for the file.

Thus, I/O operations in one process will not change where another process will read/write in the file.

Of course if one process `write()`'s to such a shared file, this may obviously change what any other process would subsequently `read()` from the file.

Open Files and Multiple Processes (contd.)

Processes are "**related**" when some were created from others by `fork()` syscalls.

With related processes, operations on open files can be even more interrelated.

`fork()` creates a *copy* of a process, so the **child** gets a copy of the **parent**'s *file descriptor table*.

This results in the child's FD entries pointing to the the same entries in the kernel's *open files table* as the parent's FD entries point to.

This means that parent and child will *share the same offset* for each file that was open prior to the `fork()`.

Open Files and Multiple Processes (contd.)

A `read()`/`write()` in one of the related processes will silently change the offset for the other process(es), moving where these others would next read/write in the file.

The result is that when the required `read()`'s/`write()`'s in the processes get executed in different orders, different results may be obtained.

Because the order processes are run is determined by the kernel **scheduler**, this is another example of a **race condition**.

Race Conditions

Concurrent programs are programs that create multiple processes or threads that run "in parallel."

One of the potential complications of *concurrent programs* is that results can depend on the exact order that operations in the processes/threads get run.

This characteristic is referred to as a **race condition**.

A race condition is a type of **logic error** in a concurrent program.

Atomicity of Operations

Sometimes preventing a race condition will require the ability to ensure that an operation is performed **atomically**.

An program operation is said to be **atomic** if it is performed as a *single uninterruptible step* (no other process can change relevant system state during the operation).

Generally, individual *system calls are atomic*, but library functions are *not*.

This is one reason why it may sometimes be necessary to use system call I/O rather than library I/O.

Append Mode

Avoiding the possibility of race conditions is the reason why it may be necessary to open a file in **append mode**.

Consider a situation in which you have several running programs that all write their logging messages to a common **log file**.

Each program should always add its log messages at the *end of the log file*.

You might imagine that the following code could accomplish this:

```
  ...
  //Set offset to current end of file:
  lseek(logfd,0,SEEK_END);
  //Write log message (at end of file):
  write(logfd,logstr,strlen(logstr));
  ...
```

Append Mode (contd.)

Consider this situation:

- two running programs, $P1$ and $P2$
- every log message is 20 bytes
- six messages have already been written to the log file (so the file is 120 bytes long)

Now suppose the processes get scheduled as follows:

1. $P1$ runs through the `lseek()` in the above code, but is then suspended (e.g., because it has exhausted its time slice)
2. $P2$ is run and executes through both the `lseek()` and `write()` calls before being suspended
3. $P1$ resumes running, executing the logging `write()` call

Append Mode (contd.)

Here is what happens given the above scheduling:

1. $P1$ sets its offset for the log file at the current file end, 120 bytes, before being suspended
2. $P2$ also sets its offset at 120, then `write()`'s a new 20-byte log message, making the log file now 140 bytes in size and automatically updating $P2$'s offset for the log file to 140
3. $P1$ resumes execution at the `write()`, with its offset for the log file still 120, and `write()`'s its 20-byte messaage as bytes 121 to 140 in the file, leaving the log file with a size of 140 bytes and automatically updating its offset to 140

Thus, $P1$ writes its log message right over the message just written by $P2$, wiping out $P2$'s latest log message.

Append Mode (contd.)

The source of the race condition is that the `lseek()`, `write()` *sequence of calls* are not an *atomic operation*, so other processes can change system state between the two calls.

There are two possible solutions:

- Prevent more than one process from running the `lseek()`, `write()` sequence of calls on the same file "at the same time" (e.g., with locking or semaphores).
- Provide the ability to seek to the end of the file and write data as a *single atomic operation*.

The second approach is exactly what is accomplished by opening a file in **append mode**: `write()` calls will automatically write data at whatever the current end of the file is.

pread() and pwrite() System Calls

For some situations, special system calls have been added that accomplish the same functionality of a sequence of other syscalls, but turn the operations into a single *atomic* operation.

The I/O syscalls `pread()` and `pwrite()` are examples of such calls:

- `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 offset)`

These calls are similar to `read()` and `write()`, but take an `offset` argument and `read()`/`write()` at the specified position in the file.

In addition, they do *not* automatically change the current *offset* for the open file (stored in the kernel's *open files table*).

pread() and pwrite() System Calls (contd.)

So to read a particular **record** in a file we could do:

```
pread(fd,record,rec_size,rec_size * rec_num-1)
```

This single call is functionally equivalent to:

```
//Save current offset:
ssize_t offset_orig = lseek(fd,0,SEE_CURR);
//Move to where to write record:
lseek(fd,rec_size * rec_num-1,SEEK_SET);
//Read record:
read(fd,record,rec_size);
//Restore offset to original value:
lseek(fd,offset_orig,SEEK_SET);
```

pread() and pwrite() System Calls (contd.)

Obviously it is much easier to use a single syscall rather than four, and wanting to read/write random (fixed-size) records in files is quite common.

The other critical advantage that the single syscalls have over their equivalent four-call sequence is that they are executed as *single, atomic operations*.

Without the atomicity provided by these special syscalls, race conditions could arise when multiple programs are trying to `read()` and `write()` at specific positions in a single file.

pread() and pwrite() System Calls (contd.)

In addition to the atomic operation aspect of `pread()` and `pwrite()`, we already noted that they do not automatically adjust file *offsets*.

It turns out that this is extremely useful to address the potential race condition noted earlier with *shared offsets* in related processes.

The same situation can arise in **multithreaded** programs, since multiple threads will share file descriptors and so offsets.

Syscalls: Files 5: Filesystem Syscalls

1. Library vs. Syscall I/O
2. Basic I/O Syscalls (part a)
3. Basic I/O Syscalls (part b)
4. File Offset
5. **Filesystem Syscalls**

Filesystem System Calls

Create and open a file:

- `int creat(const char *pathname, mode_t mode)`

Delete file (remove name or hard link):

- `int unlink(const char *pathname)`
- `int remove(const char *pathname)`

Rename and maybe move a file:

- `int rename(const char *oldpath, const char *newpath)`

Truncate (zero out) a file:

- `int truncate(const char *path, off_t length)`
- `int ftruncate(int fd, off_t length)`

Filesystem System Calls (contd.)

Create a new hard or soft link:

- `int link(const char *oldpath, const char *newpath)`
- `int symlink(const char *oldpath, const char *newpath)`

Read a symblic link:

- `ssize_t readlink(const char *path, char *buf, size_t bufsiz)`

Get detailed file metadata:

- `int stat(const char *path, struct stat *buf)`
- `int fstat(int fd, struct stat *buf)`
- `int lstat(const char *path, struct stat *buf)`

Filesystem System Calls (contd.)

Change file mode/permissions:

- `int chmod(const char *path, mode_t mode)`
- `int fchmod(int fd, mode_t mode)`

Change file owner and group:

- `int chown(const char *path, uid_t owner, gid_t group)`
- `int fchown(int fd, uid_t owner, gid_t group)`
- `int lchown(const char *path, uid_t owner, gid_t group)`

Check whether process can access file as specified:

- `int access(const char *pathname, int mode)`

Filesystem System Calls (contd.)

Change file access and modification times (atime and mtime):

- `int utime(const char *filename, const struct utimbuf *times)`
- `int utimes(const char *filename, const struct timeval times[2])`

Misc. File System Calls

Two special/unique file syscalls are:

- `int fcntl(int fd, int cmd, ... /* arg */ )`
- `int ioctl(int fd, int request, void *argptr)`

`fcntl()` can be used for various operations on open files, including changing *access mode* or other `open()` *flags*, *record locking*, managing *signals*.

`ioctl()` can be used for manipulating parameters of *devices* (open "*special files*").