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.