File I/O

CISC 3350 — CUNY Brooklyn College
Lecture Notes

File I/O

In this chapter, we will learn system calls for working with files (and the keyboard/screen, which are special types of files!)

The **Linux kernel** (= the main component of the Linux OS) keeps a list of all the files currently open by every running program. This list is called the **file table**, and its indexes are **file descriptors** (`fd`s), each uniquely representing a file.

- Most file system calls accept an `fd` as an argument (since they need to know which file(s) we want to change.)
- Each file table entry contains a pointer to the file's metadata (inode), including the location of the file's content on disk.
- Legit `fd`s are non-negative integers (`0, 1, 2, ..., n`), so `fd == -1` indicates an error. Each process has an upper limit of how many files can be opened at the same time (by default: `n = 1000`), but we can change this limit if wanted.
- Conventionally, `fd == 0` represents **standard input** (= `stdin` = keyboard), `fd == 1` is **standard output** (= `stdout` = display), and `fd == 2` represents **standard error output** (= `stderr` = also printed to display). All 3 are open by default.
- Instead of writing `fd == 0`, `1`, or `2`, you can use the already-defined constants: `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`. Checking `fd == STDIN_FILENO` is more portable than checking `fd == 0`!
- `fd`s can also represent special files (devices, disks, sockets, etc.) since Linux treats everything as a file.

Opening Files

<table>
<tr><td>Name: open()</td><td colspan="2">System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: Opens an existing file, or creates a new file and opens it.</td></tr>
<tr><td colspan="4">What libraries you must include:

```
#include <sys/stat.h> // Defines modes such as S_IRUSR
#include <fcntl.h> // Defines open() and flags such as O_RDONLY
```

</td></tr>
<tr><td colspan="4">Syntax:

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

</td></tr>
<tr><td colspan="4">Description of arguments:
name: double-quoted name of the file
flags: what to do with the file, e.g.,: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT (O_EXCL), O_TRUNC.
mode: If flags contains O_CREAT, sets default file permissions, e.g.,: S_IRWXU (owner: read, write, and execute), S_IRUSR (owner: read), S_IRGRP (group: read), S_IROTH (others: read), etc.</td></tr>
<tr><td rowspan="3">What type it returns: int</td><td colspan="3">On success: A legit file descriptor (a non-negative integer)</td></tr>
<tr><td colspan="3">On failure: -1</td></tr>
<tr><td colspan="3">If failure, does it set errno? Yes</td></tr>
<tr><td colspan="4">Quick example:

```
int file_fd = open ("text.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IWUSR | S_IRUSR);
```

</td></tr>
<tr><td colspan="4">Other variations: creat()</td></tr>
</table>

Opening Files

<table>
<tr><td>Name: <code>creat()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: Creates a new file and opens it for writing. If a file with the same name already exists, empty (truncate) it before opening it. Same as calling <code>open()</code> with the 3 flags: <code>O_WRONLY | O_CREAT | O_TRUNC</code></td></tr>
<tr><td colspan="3">What libraries you must include:

```
#include <sys/stat.h> // Defines modes such as S_IRUSR
#include <fcntl.h> // Defines creat()
```

</td></tr>
<tr><td colspan="3">Syntax:

```
int creat (const char *name, mode_t mode);
```

</td></tr>
<tr><td colspan="3">Description of arguments:
<code>name</code>: double-quoted name of the file
<code>mode</code>: Sets default file permissions, e.g.,: <code>S_IRWXU</code> (owner: read, write, and execute), <code>S_IRUSR</code> (owner: read), <code>S_IRGRP</code> (group: read), <code>S_IROTH</code> (others: read), etc.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: A legit file descriptor (a non-negative integer)</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:

```
int file_fd = creat ("text.txt", S_IWUSR | S_IRUSR | S_IXUSR);
```

</td></tr>
<tr><td colspan="3">Other variations: <code>open()</code></td></tr>
</table>

More open() and creat() Examples


```c
int fd;
fd = open ("wow.txt",
           O_WRONLY | O_CREAT | O_TRUNC,
           S_IWUSR | S_IRUSR | S_IWGRP
                   | S_IRGRP | S_IROTH);
if (fd == -1) /* To check for errors */
     /* ... */
// Note that the five bitwise ORed modes:
//    "S_..." can be written instead as: 0664.

// In other words, we could instead
//    equivalently write:
int fd = open ("wow.txt", O_WRONLY | O_CREAT |
O_TRUNC, 0664);
```

```c
int fd;
fd = creat ("wow.txt", 0644); // much simpler
if (fd == -1) /* To check for errors */
     /* ... */

// Equivalently:
int fd = open ("wow.txt", O_WRONLY | O_CREAT |
O_TRUNC, 0644);

/* Note: 0664 stands for the file permissions:
        rw-rw-r--
   and 0644 stands for the file permissions:
        rw-r--r--                              */
```

Opening Files

A couple more notes about `open()` and `create()`:

- When you call `open()` without the intention to create a file, you can drop the 3rd argument. For instance, to open `idea.txt` for reading, you would write:

```
int my_file = open ("idea.txt", O_RDONLY);
```

- Because we learned that `create()` is essentially a call to `open()` with the flags `O_WRONLY | O_CREAT | O_TRUNC`, it means `create()` could potentially be implemented, in some Linux architectures, by calling `open()`:

```
int creat (const char *name, mode_t mode)
{
    return open (name, O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

- In the cases of `fd == -1` after calling `open()` or `create()`, a C program should usually either ask the user to enter the name of a different file, or simply print an error message and `exit()` the program.

Reading Files

Once we opened a file, let's learn how to read data from it:

<table>
<tr><td>Name: <code>read()</code></td><td colspan="2">System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: reads data from a file into an array, and returns the # of read bytes.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines read() and the size_t, ssize_t types.</code></td></tr>
<tr><td colspan="4">Syntax: <code>ssize_t read (int fd, void *buf, size_t len); // ssize_t = signed size_t</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>fd</code>: file descriptor of an open file
<code>buf</code>: The array name (or address of a variable, e.g., &<code>var</code>) into which data is read.
<code>len</code>: Maximum number of bytes we can or wish read.</td></tr>
<tr><td rowspan="3">What type it returns: <code>ssize_t</code></td><td colspan="3">On success: A non-negative integer indicating how many bytes we read in. This number will be equal to or smaller than <code>len</code>, and could also be <code>0</code> (end-of-file case.)</td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>ssize_t number_of_chars_read_in = read (fd, arr, 100);</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Reading Files

An example of reading an `unsigned long` integer from a file:

```
unsigned long my_int; // 4-byte type on 32-bit Linux systems, or 8-byte type on 64-bit systems
ssize_t nr = read (fd, &my_int, sizeof(unsigned long));
if (nr == -1) // To check for errors
```

Notes:

- The variable of the array into which we read data must be created (on the stack or the heap) before the call to `read()`.
- The file position is advanced by the number of bytes that are read in. For example, if we read in `"Hello"` at the beginning of the file, the file position will become `5` after we return from `read()`, so that, for future reads, we can read in the content that comes after `"Hello"`.
- If `read()` reads fewer than `len` bytes, we either reached the end of a file or a signal interrupted the read midway.
- When reading from a socket or device file (e.g., the keyboard!), the call will block (wait) if no bytes are available for reading yet and if the open file allows blocking (i.e., `O_NONBLOCK` wasn't used as a flag inside the call to `open()`.)

Reading Files

Notes (Cont'):

- If the returned value is `-1`, either a recoverable or an irrecoverable error occurred. Recoverable errors include:
  - An interruption by a signal before any bytes were read (we will learn more about signals in Topic 10.) If so, `errno` will be set to `EINTR`, so we should call `read()` again.
    - Nonblocking reads return immediately if data isn't available (e.g., the user hasn't typed anything using the keyboard yet.) This approach is useful in that you can move on to read from other files that do have ready data, without the need to wait and waste time.
  - The refusal of `read()` to wait for unavailable data. If the file was opened with the `O_NONBLOCK` flag, and no data is currently available, `errno` is set to `EAGAIN`, so you should call `read()` sometime later in the code, in the hope that data becomes available by then.
- Otherwise, an irrecoverable error occurred. In such a case, `errno` is set to a value other than `EINTR` or `EAGAIN`.
- The code example on the following slide shows how one can read from a file that allows waiting upon I/O (i.e., the file wasn't open using the `O_NONBLOCK` flag, so `errno` won't be set to `EAGAIN`.)
- That example properly checks and handles partial readings (= cases of reading fewer than `len` bytes.)

Reading Files


```c
char buf [1000];
ssize_t ret;
size_t len = 50;
int fd = open ("text.txt", O_RDONLY);
while (len != 0 && (ret = read (fd, buf, len)) != 0) // len bytes filled or EOF
{
    if (ret == -1)
    {
        if (errno == EINTR) // Signal interrupt: continue without updating len/buf.
            continue;
        perror ("read"); // Irrecoverable error: print error message & exit the loop.
        break;
    }
    len -= ret;
    buf += ret;
}
```

Reading Files

And here is an example with a reading that doesn't block (= doesn't wait for data):

```
char buf [1000];
size_t len = 50;
int fd = open ("text.txt", O_RDONLY | O_NONBLOCK); // O_NONBLOCK is included to prevent blocking
ssize_t ret = read (fd, buf, len);
int error_code = errno;
if (ret == -1)
{
    if (error_code == EINTR)
        ssize_t ret = read (fd, buf, len); /* Calling read() again. */
    else if (error_code == EAGAIN)
        /* Call read() later */
    else
        /* irrecoverable error */
}
```

Reading Files

Any questions?

Writing to (= Changing) Files

Let's learn how to write data to files:

<table>
<tr><td>Name: <code>write()</code></td><td colspan="2">System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: writes data from an array into a file, and returns the # of written bytes.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines write() and the size_t, ssize_t types.</code></td></tr>
<tr><td colspan="4">Syntax: <code>ssize_t write (int fd, const void *buf, size_t count);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>fd</code>: file descriptor of a file that was open for writing
<code>buf</code>: The array name (or address of a variable, e.g., <code>&var</code>) from which data is taken.
<code>count</code>: Number of bytes we wish to write into the file.</td></tr>
<tr><td rowspan="3">What type it returns: <code>ssize_t</code></td><td colspan="3">On success: A non-negative integer indicating how many bytes we wrote. This number will be equal to or smaller than <code>count</code>, and could also be <code>0</code> (in cases of some special files.)</td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>ssize_t number_of_chars_written = write (fd, arr, strlen(arr));</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Writing to (= Changing) Files

An example of writing a string to a file:

```
char buf [] = "It's summer soon!";
ssize_t nr = write (fd, buf, strlen (buf));
if (nr == -1) // To check for errors
```

Notes:

- The file position is advanced by the number of bytes that are written into the file each time we call `write()`.
- With `write()` used for regular files, we don't have an "end of file" situation, so usually all the bytes are written at once.
- However, for special files like sockets, which are frequently interrupted, we should use a **`while`** loop to write out all the requested bytes, just like the one we had for `read()` on Slide 10:

```
int fd = open (socket_file, O_WRONLY);
while (len != 0 && (ret = write (fd, buf, len)) != 0)
```

More Information about Writes

- We can append written content to the end of the file (= "Append Mode").
  - We must open such files using the `O_APPEND` flag:

```
int fd = open ("hello.txt", O_WRONLY | O_APPEND);
```

  - It guarantees that writes occur at the end of the file and prevents overwriting of existing content.
- Same as with `read()`, use `O_NONBLOCK` when you write to non-regular files to prevent `write()` from blocking.
  - Instead of `write()` blocking on file unavailability, it returns `-1` immediately and sets `errno` to `EAGAIN`.
- Other possible `errno` values that `write()` could set in case of an error are: `EBADF` (Bad file descriptor), `EFAULT` (bad array), `EFBIG` (File too large), `EINVAL` (Invalid argument), `EIO` (Low-level input/output error), `ENOSPC` (No space left on device), and `EPIPE` (the file is a pipe or socket whose reading end is closed).
- Size limits on `write()`:
  - If the 3rd argument to `write()` is larger than `SSIZE_MAX`, then the results of `write()` will be undefined/chaotic.
  - If this 3rd argument is `0`, the call returns immediately with a return value of `0`.

Writing to Disk

- The Linux OS uses internal memory storage arrays called **buffers** to temporarily keep changes before saving them to disk. `write()` only guarantees copying data from the array (that you pass as the 2nd argument) to this OS buffer, but there is no guarantee that data is written out to disk immediately.
  - In the background, the OS gathers up the buffers containing data newer than what is on disk, sorts them optimally, and writes them out to disk (writeback) at some later point in time.
- **Why so?** Delayed writes allow `write()` to return immediately, without waiting for the data to be written to disk. Also, reading from the buffers is faster than reading from disk.
  - When Linux is more-or-less idle (not very busy), it saves all these buffers to disk in batches (= groups).
- **Problem**: in cases of system crashes or I/O errors, programs might get terminated, so the data on the buffers might never make it to disk!
- **Solutions:**
  - The OS stores an "age" variable for each buffer and, if a buffer passes the maximum age allowed, the OS writes it right away to disk.
  - We can also force writing to disk via synchronization system calls, as we shall see next.

Saving Files' Changes to Disk: Synching

<table>
<tr><td colspan="2">Name: <code>fsync()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: flushes (= forces the saving of) both data and metadata of a file from the Linux OS's I/O buffer to disk and returns only after the drive confirms that data were successfully saved to the disk.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines fsync().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int fsync (int fd);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>fd</code>: file descriptor of a file that was open for writing</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: <code>0</code></td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>int return_value = fsync (my_file_fd);</code></td></tr>
<tr><td colspan="4">Other variations: <code>fdatasync()</code>, <code>sync()</code></td></tr>
</table>

There is a trade-off between performance and the control of when data reaches the disk: in case of large files, the OS might take some time to complete the disk saving of the data, so `fsync()` might take time to return.

Saving Files' Changes to Disk: Synching

<table>
<tr><td>Name: <code>fdatasync()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: flushes (= forces the saving of) only the data (content) and some essential metadata changes of a file that is required to properly access the file in the future (such as the file's size, but not the recent access timestamp or modification timestamp) from the Linux OS's I/O buffer to disk and returns only after the drive confirms that data were successfully saved to the disk.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <unistd.h> // Defines fdatasync().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int fdatasync (int fd);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>fd</code>: file descriptor of a file that was open for writing</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example: <code>int return_value = fdatasync (my_file_fd);</code></td></tr>
<tr><td colspan="3">Other variations: <code>fsync()</code>, <code>sync()</code></td></tr>
</table>

`fdatasync()` skips synchronizing non-essential metadata, which means it is faster, in most cases, than `fsync()`.

Saving Files' Changes to Disk: Synching

<table>
<tr><td colspan="2">Name: <code>sync()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: flushes (= forces the saving of) the data (content) and all metadata changes of all the Linux OS's I/O buffers (for all the open files on the computer) to disk and returns only after the drive confirms that data were successfully saved to the disk.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines sync().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void sync (void);</code></td></tr>
<tr><td colspan="4">Description of arguments: – The function accepts no arguments –</td></tr>
<tr><td rowspan="3">What type it returns: <code>void</code></td><td colspan="3">On success: nothing is returned (this function always succeeds)</td></tr>
<tr><td colspan="3">On failure: nothing is returned (this function never fails)</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>sync();</code></td></tr>
<tr><td colspan="4">Other variations: <code>fsync()</code>, <code>fdatasync()</code></td></tr>
</table>

Since `sync()` syncs the data of all the currently-open files on the OS, it will take much more time than `fsync()` to run. On busy systems, it might even take several minutes or longer to return.

Closing Files

Instead of calling the sync system calls, you may `open()` your file with the `O_SYNC` flag, which will automatically flush changes. This is the same as calling `fsync()` after every call to `write()`!

Name: `close()`	**System call or function:** System call	**Links:** Online Manual | Course Packet
What it does: closes a file that was open with `open()`. It unmaps the open `fd` and disassociates the file from the program. The kernel is free to reuse this `fd` integer for other files that will be opened. Closing a file doesn't affect when the file is flushed to disk, but it may result in an unlinked file being finally physically erased from the disk. The most common error value is EIO, most likely indicating a low-level I/O error, but the file always gets closed.		
What libraries you must include: `#include <unistd.h> // Defines close().`		
Syntax: `int close (int fd);`		
Description of arguments: `fd`: file descriptor of an open file		
What type it returns: `int`	**On success:** `0`	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	
Quick example: `if (close (my_file_fd) == -1) perror("close");`		
Other variations: N/A		

Changing the File Position

<table>
<tr><td colspan="2">Name: <code>lseek()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: updates the file's reading and writing position (it's like moving the cursor elsewhere in a file.)</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines lseek().</code></td></tr>
<tr><td colspan="4">Syntax: <code>off_t lseek (int fd, off_t offset, int whence);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>fd</code>: file descriptor of an open file
<code>offset</code>: a signed long integer indicating by how many places you want to move the position.
<code>whence</code>: one of the following 3 constants:
1. <code>SEEK_CUR</code>: sets file's position to current value + <code>offset</code> (<code>offset</code> could be any integer).
2. <code>SEEK_END</code>: the position is set to the end of file + <code>offset</code>.
3. <code>SEEK_SET</code>: sets the file's position to <code>offset</code>.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>off_t</code></td><td colspan="2">On success: the new file position (a signed long integer)</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>off_t ret = lseek (fd, (off_t) 1825, SEEK_SET);</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Changing the File Position: Examples

To set the file position to byte 734:

```
off_t ret = lseek (fd, (off_t) 734, SEEK_SET);
if (ret == (off_t) 1) /* error */
```

To set the file position to the end of the file:

```
off_t ret = lseek (fd, 0, SEEK_END);
if (ret == (off_t) 1) /* error */
```

To get the current file position:

```
off_t ret = lseek (fd, 0, SEEK_CUR);
if (ret == (off_t) 1) /* error */
```

To prevent errors/warnings, we must implicitly convert some number literals to the `off_t` data type by including: `(off_t)` to the left of the number before comparing the results with ret.

More on `lseek()`

You can move the position past the end of a file. In such a case:

- A subsequent read will return `EOF`.
- A write request will automatically pad zeros (null characters) in the skipped space.
  - Such zero padding (called a **hole**), doesn't occupy any physical disk space, which saves space.
  - A read request to the hole will return the correct number of null characters.

Possible `errno` values in case of errors with `lseek()`:

- `EBADF`: bad `fd`
- `EINVAL`: invalid parameters in `whence`, or the resulting file position would be negative.
- `EOVERFLOW`: offset can't be represented in an `off_t` (C `signed long`). Internally, the kernel stores `offset` in the C `long long` type, causing this error on 32-bit systems.
- `ESPIPE`: the `fd` is associated with an unseekable object (whose position we can't change:) pipe/fifo/socket.

Truncating the File

<table>
<tr><td colspan="2">Name: <code>ftruncate()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: deletes a file's content beyond a specific position. The file must have been opened for writing. It doesn't update the file's position.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines ftruncate().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int ftruncate (int fd, off_t length);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>fd</code>: file descriptor of an open file
<code>length</code>: an integer, indicating the file position beyond which all the content will be deleted.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: the file's new length in bytes.</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>int size = ftruncate (my_fd, 15);</code></td></tr>
<tr><td colspan="4">Other variations: <code>truncate()</code></td></tr>
</table>

While it's less common, the function can use a length that is larger than the original file length; the extended space will be padded with null characters.

Truncating the File

<table>
<tr><td colspan="2">Name: <code>truncate()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: deletes a file's content beyond a specific position. The file must have been opened for writing. It doesn't update the file's position.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines truncate().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int truncate (const char *filename, off_t length);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>filename</code>: a double-quoted name of a file.
<code>length</code>: an integer, indicating the file position beyond which all the content will be deleted.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: the file's new length in bytes.</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>int size = truncate ("text.txt", 15);</code></td></tr>
<tr><td colspan="4">Other variations: <code>ftruncate()</code></td></tr>
</table>

Truncating the File

If the file "hello.txt" contains the following 2 lines of text, what will happen if we run the program below?

```
Edward Teach was a notorious English pirate. He was nicknamed Blackbeard.
```

```c
#include <unistd.h>
#include <stdio.h>
int main()
{
    int ret = truncate ("./hello.txt", 45);
    if (ret == 1)
    {
        perror ("truncate");
        return 1;
    }
    return 0;
}
```

Truncating the File

Any questions?

Reading/Writing without Changing File Position

Suppose we want to read or write into a file without automatically updating the file position. The `read()` and `write()` system calls update the file position, so what system calls could we use instead of `read()` and `write()`?

Answer: We can use the system calls `pread()` and `pwrite()`! The info is on the following slides.

These system calls let us access the file at a position that we indicate (the 4th argument to `pread()` and `pwrite()` is the position at which we want to read or write) instead of reading or writing at the current file position.

As such, if the program uses multiple threads, which are mini processes that access the file at the same time, we can eliminate accidental overwriting of existing content when `pread()` and `pwrite()` are used, because we tell Linux at which exact place we want to read and write. We will cover Threads in Topic 8.

Accept for the file position that remains unchanged, `pread()` and `pwrite()` behave the same way as `read()` and `write()`.

Positional Reads

Name: `pread()`	**System call or function:** System call	**Links:** Online Manual | Course Packet
What it does: reads data from a file into an array *without* changing the file position, and returns the # of read bytes.		
What libraries you must include: `#define _XOPEN_SOURCE 500 // extra function definitions + pread().` `#include <unistd.h> // Defines pread(), size_t, ssize_t, and off_t.`		
Syntax: `ssize_t pread (int fd, void *buf, size_t len, off_t pos);`		
Description of arguments: `fd`: file descriptor of an open file `buf`: The array name (or address of a variable, e.g., `&var`) into which data is read. `len`: Maximum number of bytes we can or wish read. `pos`: The position (byte) in the file from where you want to start reading.		
What type it returns: `ssize_t`	**On success:** A non-negative integer indicating how many bytes we read in. This number will be equal to or smaller than `len`, and could also be `0` (end-of-file case.)	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	
Quick example: `ssize_t number_of_chars_read_in = pread (fd, arr, ARR_SIZE, 50);`		
Other variations: N/A		

Positional Writes

<table>
<tr><td colspan="2">Name: <code>pwrite()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: writes data from an array into a file without changing the file position, and returns the # of written bytes.</td></tr>
<tr><td colspan="4">What libraries you must include:
<code>#define _XOPEN_SOURCE 500 // extra function definitions + pread().</code>
<code>#include <unistd.h> // Defines pwrite(), size_t, ssize_t, and off_t.</code></td></tr>
<tr><td colspan="4">Syntax: <code>ssize_t pwrite (int fd, void *buf, size_t len, off_t pos);</code></td></tr>
<tr><td colspan="4">Description of arguments:
<code>fd</code>: file descriptor of an open file
<code>buf</code>: The array name (or address of a variable, e.g., <code>&var</code>) from which data is taken.
<code>len</code>: Maximum number of bytes we want to write.
<code>pos</code>: The position (byte) in the file from where you want to start writing.</td></tr>
<tr><td rowspan="3">What type it returns: <code>ssize_t</code></td><td colspan="3">On success: A non-negative integer indicating how many bytes we wrote. This number will be equal to or smaller than <code>len</code>, and could also be <code>0</code> (in cases of some special files.)</td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>ssize_t number_of_chars_written = pwrite (fd, arr, strlen(arr), 50);</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

file_io_syscalls.c: a Full C Program


```c
/* A C program that demonstrates uses of the creat(),
 *    open(), read(), write(), pread(), pwrite(),
 *    ftruncate(), fsync(), sync(), lseek(), and
 *    close() file I/O system calls.
 *
 *    Miriam Briskman, 2/27/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#define _XOPEN_SOURCE 500

// Library defining types such as size_t (usually
//    as an unsigned integer:)
```

Multiplexed I/O - Motivation

Scenario: Your C program works with several files and I/O devices at the same time, e.g., with 5 keyboards. The users type info using these keyboards with some delay. If your program starts waiting for data to become available from one of the keyboards, it can't then wait for data from other keyboards (since it is busy waiting for the current one, so the call to `read()` won't return until data is entered.) ***How can we wait for more than just one file/device at the same time?***

Solution 1: We could've used non-blocking I/O, but this, too, will require us to call system calls multiple times with the hope that they become available, which isn't too efficient. What we want instead is that the program waits until one of the devices becomes available, rather than asking each device every time.

Solution 2: We can use special system calls that perform **multiplexed I/O**, which allows the program to concurrently block on multiple files/devices! The program is notified when at least one of them becomes ready to read/write data as below:

1. If no device is ready, sleep.
2. If at least one device becomes ready, wake up.
3. Read or write data into the available devices.
4. Go back to step 1.

Multiplexed I/O: `select()`

<table>
<tr><td>Name: <code>select()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: waits for multiple I/O devices to become available or until the timeout is reached.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <sys/select.h> // Defines select().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);</code></td></tr>
<tr><td colspan="3">Description of arguments:
<code>n</code>: the highest <code>fd</code> value, plus one (<code>fd</code> + 1).
<code>readfds</code>, <code>writefds</code>, <code>exceptfds</code>: sets of <code>fds</code> of files that we want to read from, write into, and that might experience exceptions or receive out-of-band data. We may pass <code>NULL</code> for any ones of these if we are not interested in using it.
<code>timeout</code>: a data structure of seconds + microseconds that we can set to be the timeout.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: The number of files available for I/O, or <code>0</code> if time out.</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example: <code>int files = select (max_fds + 1, &readfds, &writefds, NULL, &timeout);</code></td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

Multiplexed I/O: `select()`

Remember that `readfds`, `writefds`, and `exceptfds` are sets of `fd`s? We call 4 functions to manipulate these sets:

- `FD_CLR (int fd, fd_set *set); // Removes an fd from a set (rarely used)`
- `FD_SET (int fd, fd_set *set); // Add an fd to a set`
- `FD_ZERO (fd_set *set); // Removes all the fds from a set`
- `FD_ISSET (int fd, fd_set *set); // Checks whether an fd is in the set: returns a nonzero if yes`

We usually call `FD_ZERO()` to first clear a set, e.g., `readfds`, from all the previously used `fd`s in it, and then call `FD_SET()` repeatedly on all the files/devices that we want to add to our set.

`select()` will check if any of the files have become available. The available files will remain in their sets, while files that haven't become available will be removed from these sets by `select()` itself.

After `select()` returns, we would call `FD_ISSET()` to check which of the files are still in their original sets (that is, which files are now available for I/O!)

select_syscall_example.c: a Full C Program


```
// The following program exemplifies a call to the
//     select() system call. We wait (block) only
//     on one file: the keyboard (fd = 0 = STDIN_FILENO)
//     for 5 seconds. If the user types some content and
//     presses ENTER before the 5 seconds end, the
//     content will be output to the terminal. Otherwise,
//     the 5 seconds will elapse, so the system call will
//     return without gathering input from the user.
//
// This program is taken from Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     pages 55-56.
```

Multiplexed I/O: `poll()`

<table>
<tr><td>Name: <code>poll()</code></td><td>System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: waits for multiple I/O devices to become available or until the timeout is reached. More efficient than <code>select()</code> in several aspects.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <sys/poll.h> // Defines poll().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int poll (struct pollfd *fds, nfds_t nfds, int timeout);</code></td></tr>
<tr><td colspan="3">Description of arguments:
<code>fds</code>: an array of structures of the type <code>pollfd</code> (see more info on the next slide.)
<code>nfds</code>: the number of structures we passed in <code>fds</code>.
<code>timeout</code>: number of milliseconds that we assign as the timeout for <code>poll()</code> to run. To indicate no timeout (i.e., to set it to "infinity",) set <code>timeout</code> to any negative integer, e.g., <code>-1</code>.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: The number of files available for I/O, or <code>0</code> if time out.</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example: <code>int files = poll (fds, 2, 5000);</code></td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

Multiplexed I/O: `poll()`

Where `select()` uses inefficient 3 bitmask-based sets of `fd`, `poll()` uses a single array of `pollfd` structures:

```
#include <poll.h>
struct pollfd {
        int fd;         /* file descriptor */
        short events;   /* requested events to watch */
        short revents; /* returned events witnessed */
};
```

Each `pollfd` specifies one `fd` to watch, with the `events` field being a bitmask of events to watch for; the user/programmer will set this field.

The `revents` field is a bitmask of events that were witnessed; the OS itself will set this field while `poll()` is running, so don't touch this field before calling `poll()`.

Multiplexed I/O: `poll()`

Here are constants representing some of the `events` that we are looking for with our files and I/O devices. The full list of events constants that you can use is provided on the manual page of `poll()`:

- `POLLIN`: data is available for reading from that file/device.
- `POLLPRI`: some exceptional situation happened with the file descriptor, e.g., there is out-of-band data on a socket device.
- `POLLOUT`: the file/device is ready for writing.
- `POLLWRBAND`: the file/device is ready to accept the writing of priority data.

We can use the above constants both when we set `events` before the call to `poll()`, and after the call to `poll()` when we check out the `revents` field to see which files/devices have become available for reading or writing.

On the next slide is a full C program showing how to use `poll()` to check whether the keyboard is available for reading and whether the screen is available for writing.

poll_syscall_example.c: a Full C Program


```
// The following program exemplifies a call to the
//     poll() system call. We wait (block) on 2 files:
//     the keyboard (fd = 0 = STDIN_FILENO) and the
//     screen (fd = 1 = STDOUT_FILENO) for 5 seconds.
//     We expect the screen to be responsive right
//     away, so we'll see the message: "stdout is
//     writable". Since poll() will return right
//     away, the program will soon thereafter end.
//     As such, to check also for the keyboard,
//     create some file (say, a .txt file) and
//     call the poll_syscall_example program this
//     way: ./poll_syscall_example < text.txt
//     where text.txt is the file you created. Then,
//     the following additional message will be
```

Multiplexed I/O - Comparison

Benefits of `poll()`:

1. No need to pass in the highest-numbered `fd` plus 1: you just pass the overall number of the `fd`s you are watching for as the 2nd argument.
2. `select()`'s `fd` sets are statically sized (either too small or wasting space,) whereas `poll()` has the exact, right size.
3. The `fd` sets for `select()` are reconstructed on return, so a subsequent call must reinitialize them, whereas for `poll()` the `events` and `revents` fields are separate.
4. The `timeout` field in `select()` is modified during `select()`'s execution, requiring re-initialization of the `timeout` for future calls to `select()`.

Benefits of `select()`:

1. More portable - some Unix systems don't support `poll()`.
2. Has better timeout resolution (microseconds) instead of `poll()`'s milliseconds.

Multiplexed I/O

Any questions?

File I/O: Sources

Topic 4: File I/O. Briskman, Miriam. *CISC 3350 OER Course Packet.* 2024, CC BY-NC 4.0.
URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/CISC_3350_OER_Course_Packet.pdf#topic+4:+file+i/o.

Partly based on the teaching of Professor **Yongqing Xiang** of Brooklyn College.