# Buffered File I/O

## CISC 3350 — CUNY Brooklyn College
## Lecture Notes

# Buffered File I/O - Motivation



The file I/O system calls we learned about work relatively fast, but there is a drawback to using them. Let's understand it.

The smallest unit of storage for a filesystem is called a **block**, which is a chunk of bytes, usually 512 bytes. All I/O operations work with integer (>= 1) multiples of the block size, and not with single bytes/characters.

In most cases a computer program wishes to work with a few bytes (e.g., a string), a line of text, or other data (e.g., integers), which isn't exactly 512 bytes long. This means we are forced to read multiples of 512 bytes from the disk and use only a small part of it, which is inefficient in terms of time!

***Fun class activity***: Run each of the following commands in Linux. Which one runs for a longer time?

- `$ time dd bs=1 count=2097152 if=/dev/zero of=myfile.txt`
- `$ time dd bs=4096 count=512 if=/dev/zero of=myfile2.txt`

# Buffered File I/O - Motivation



The Linux command `dd` is used to copy bytes from a file/device into another file. We tell `dd` what the source file ("`if`") and destination file ("`of`") are, how many blocks of data we want to copy ("`count`"), and how many bytes should be in each such block ("`bs`" = **b**lock **s**ize).

In both commands of the activity on the previous slide, we copied the same overall number of bytes from a Linux device called `/dev/zero`, which provides a stream of null characters: this number is 2097152 bytes.

***However, the difference was*** in the division of these bytes into blocks: in the first call, we attempted to copy 2097152 blocks of 1 byte each, and, in the 2nd call, we copied 512 blocks containing 4096 bytes each. The rule is that if the block size is a multiple of the OS's block size and is close in magnitude to the block size, the copying would go much faster!

Specifically, if you read data of the sizes of 512, 1024, 2048, etc., bytes, the OS will return you the data much faster than if you were to ask for it character-by-character (since, anyway, the OS will repetitively read in blocks and simply discard the majority of them, so it'll just waste time.) The most common block sizes are 4K = 4096 and 8K = 8192, and reading this number of bytes at a time is usually pretty efficient.

To find the block size supported by a certain device (e.g., disk), you can use the `stat` Linux command:

```
$ stat fileOrDiskName
```

---

**To solve this time-efficiency issue**, you could save the entire block of characters that you read from the device into a buffer (array), and then access this array anytime you need data near the bytes you extracted earlier. This way, you won't need to repetitively read whole blocks of bytes from the device.

You can implement this yourself, of course. However, using the functions of the standard I/O library: `<stdio.h>`, which implements this bufferization *for you*, is much better!

`<stdio.h>` is a platform-independent (= portable), user-buffering solution used since 1989. In other words, the functions of this library can be used not only on Linux systems, but also on Windows and other OS platforms.

# Standard I/O (`stdio.h`) Library



Instead of operating on `fd`s, the `<stdio.h>` library allows you to create **file pointers**, which are pointer (address) variables to a data type called `FILE`.

The `FILE` data type is defined in `<stdio.h>` as a structure, whose implementation is hidden. That is, aside from creating `FILE` pointers, you aren't required to know how they are implemented.

What we do know is that, internally, the `FILE` structure is linked (mapped) to the `fd` that represents the file.

We call a file that was opened via the functions of `<stdio.h>` a **stream**.

Streams may be opened for reading (input streams), writing (output streams), or both (input/output streams.)

Well, it's time now to learn how to open files via the `fopen()` function!

# <stdio.h>: Opening Streams



<table>
<tr><td colspan="2">Name: fopen()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: opens a file stream and returns a pointer to it.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines fopen().</code></td></tr>
<tr><td colspan="4">Syntax: <code>FILE *fopen (const char *path, const char *mode);</code></td></tr>
<tr><td colspan="4">Description of arguments: path: a double-quoted name of the file or path to the file<br>mode: a double-quoted opening mode: r, r+, w, w+, a, or a+. See next slide for description.</td></tr>
<tr><td rowspan="3" colspan="2">What type it returns: FILE*</td><td colspan="2">On success: A non-NULL file pointer to the opened file stream</td></tr>
<tr><td colspan="2">On failure: NULL</td></tr>
<tr><td colspan="2">If failure, does it set errno? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>FILE *stream = fopen ("myfile.txt", "r"); // Open "myfile.txt" for reading</code></td></tr>
<tr><td colspan="4">Other variations: fdopen()</td></tr>
</table>

Since the fopen() may fail to open the file, we must check whether the returned file pointer isn't NULL:

```
FILE  *stream = fopen ("myfile.txt", "r");
if (stream == NULL) // Error occurred!
```

# `<stdio.h>`: Opening Streams



Here are descriptions of what the `mode` types mean:

| | |
|---|---|
| `"r"` | Reading from the file. The stream is positioned at the file's start in all modes except for `"a"`. |
| `"r+"` | Reading and writing (but not truncating or creating the file.) |
| `"w"` | Writing to the file. If the file exists, it truncates the file to zero length; if not, it creates it. |
| `"w+"` | Reading and writing, together with truncating and creating as in mode `"w"`. |
| `"a"` | Writing in append mode, and creating the file if it doesn't exist. Only with `"a"`, the stream is positioned at `EOF` (i.e., all writes will append to the file, and no truncation of the original content occurs.) |
| `"a+"` | Reading and writing in append mode; like mode `"a"`, but also allows reading. |

Sometimes, you already opened a file using the `open()` system call, and got its `fd`. You can convert the `fd` into a file pointer, and continue accessing the file with the functions of `<stdio.h>` instead of system calls.

This can be achieved with the `fdopen()` function, as we shall see next.

# <stdio.h>: Opening Streams



<table>
<tr><td colspan="2">Name: fdopen()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: opens a file stream from an existing fd and returns a pointer to it.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines fdopen().</code></td></tr>
<tr><td colspan="4">Syntax: <code>FILE *fdopen (int fd, const char *mode);</code></td></tr>
<tr><td colspan="4">Description of arguments:<br>fd: file descriptor of an open file<br>mode: a double-quoted opening mode: r, r+, w, w+, a, or a+. See slide 7 for description.</td></tr>
<tr><td rowspan="3" colspan="2">What type it returns: FILE*</td><td colspan="2">On success: A non-NULL file pointer to the opened file stream</td></tr>
<tr><td colspan="2">On failure: NULL</td></tr>
<tr><td colspan="2">If failure, does it set errno? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>FILE *stream = fdopen (my_fd, "w+");</code></td></tr>
<tr><td colspan="4">Other variations: fopen()</td></tr>
</table>

You can use the same modes as for fopen(), but no truncation occurs for "w" or "w+", and the file position isn't changed.

- Modes must be compatible with those you used in the system call open(): reading for reading, writing for writing, etc.

# `<stdio.h>`: Opening Streams



After calling `fdopen()`, from then on, I/O should be performed via the file stream and not via the `fd`.

Closing the stream will close the `fd`, too, so don't call the `close()` system call after calling `fdopen()`.

Example of calling `fdopen()`:

```
FILE *stream;
int fd = open ("info.txt", O_RDONLY);
if (fd == -1) /* checks for errors */
   /*     */
// ...
stream = fdopen (fd, "r");
if (!stream) /* checks for errors: same as if (stream == NULL) */
   /*     */
```

# <stdio.h>: Closing Streams



| **Name:** `fclose()` | **System call or function:** Function | **Links:** Online Manual \| Course Packet |
|---|---|---|
| **What it does:** closes a file that was opened with `fopen()` or with `fdopen()`. | | |
| **What libraries you must include:** `#include <stdio.h> // Defines fclose().` | | |
| **Syntax:** `int fclose (FILE *stream);` | | |
| **Description of arguments:** `stream`: the open file stream that was returned by `fopen()` or `fdopen()` | | |
| **What type it returns:** `int` | **On success:** `0` | |
| | **On failure:** The constant `EOF` (usually equal to `-1`, but still use `EOF` for portability.) | |
| | **If failure, does it set `errno`?** Yes | |
| **Quick example:** `if (fclose(stream) == EOF) perror ("fclose");` | | |
| **Other variations:** `fcloseall()` | | |

Any buffered data is first flushed (= saved) to disk (which is not guaranteed to happen when using the `close()` system call.) This is why it is very important to close files after using them!

The other variant, `fcloseall()`, closes all the open streams at once.

# <stdio.h>: Closing Streams



| **Name:** `fcloseall()` | **System call or function:** Function | **Links:** Online Manual \| Course Packet |
|---|---|---|
| **What it does:** closes all the files streams that were opened with `fopen()` or with `fdopen()` by the program. | | |
| **What libraries you must include:**<br>`#define _GNU_SOURCE /* non-standard GNU/Linux-specific extension functions. */`<br>`#include <stdio.h> // Defines fclose().` | | |
| **Syntax:** `int fcloseall (void);` | | |
| **Description of arguments:** – Function doesn't accept any arguments – | | |
| **What type it returns:** `int` | **On success:** `0` | |
| | **On failure:** The constant `EOF` (usually equal to `-1`, but still use `EOF` for portability.) | |
| | **If failure, does it set `errno`?** Yes | |
| **Quick example:** `if (fcloseall() == EOF) perror ("fcloseall");` | | |
| **Other variations:** `fclose()` | | |

All the buffered data for all the files is first flushed (= saved) to disk.

# `<stdio.h>`: Reading from Streams, Character-by-Character



Opened streams for reading (in any mode except for `"w"` or `"a"`) can be read in 3 ways: (1) one character at a time, (2) an entire line at a time, or (3) as binary data.

<table>
<tr><td>Name: <code>fgetc()</code></td><td colspan="2">System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: reads a single byte (character) from a stream that was opened for reading. The returned <code>unsigned char</code> type is cast to an <code>int</code> to be able to detect <code>EOF</code> (= <code>-1</code>).</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines fclose().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int fgetc (FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>stream</code>: the open file stream that was returned by <code>fopen()</code> or <code>fdopen()</code></td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: A non-negative integer ASCII value of the character that was read in.</td></tr>
<tr><td colspan="3">On failure: The constant <code>EOF</code> (usually equal to <code>-1</code>, but still use <code>EOF</code> for portability.)</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:<br><code>if ((c = fgetc (stream)) == EOF)  // Checking for errors. c is an int.</code><br><code>else printf ("c = %c\n", (char)c); // (char)c casts int to char.</code></td></tr>
<tr><td colspan="4">Other variations: <code>ungetc()</code></td></tr>
</table>

# <stdio.h>: Reading from Streams, Character-by-Character



| **Name:** ungetc() | **System call or function:** Function | **Links:** Online Manual \| Course Packet |
|---|---|---|
| **What it does:** pushes a character back into a stream that is open for reading: the file itself (on disk) isn't changed. Calling fgetc() afterwards will return this character. However, calling a seeking function will discard all pushed-back characters. ungetc(), therefore, lets us 'peek' at the stream and put the character back to the stream if we don't want to use it. | | |
| **What libraries you must include:** `#include <stdio.h> // Defines ungetc().` | | |
| **Syntax:** `int ungetc (int c, FILE *stream);` | | |
| **Description of arguments:** c: the character we want to push back into the I/O buffer<br>stream: a file stream that was opened for reading | | |
| **What type it returns:** int | **On success:** A non-negative integer ASCII value of the character that was pushed back. | |
| | **On failure:** EOF (= -1). | |
| | **If failure, does it set errno?** Yes | |

**Quick example:**

```
if ((c = ungetc ('A', stream)) == EOF) // Checking for errors. c is an int.
else printf ("c = %c\n", (char)c);     // (char)c casts int to char.
int result_char = fgetc (stream);      // fgetc() should return 'A'.
```

**Other variations:** fgetc()

# `<stdio.h>`: Reading from Streams, Character-by-Character



Any questions?

# <stdio.h>: Reading from Streams, Line by Line



| **Name:** fgets() | **System call or function:** Function | **Links:** Online Manual \| Course Packet |
|---|---|---|
| **What it does:** It reads up to (size - 1) bytes from stream, storing them in str. The reason for reading (size - 1) bytes is that the last byte should be kept for the null '\0' character. fgets() also stops at file's end and at '\n'. | | |
| **What libraries you must include:** `#include <stdio.h> // Defines fgets().` | | |
| **Syntax:** `char * fgets (char *str, int size, FILE *stream);` | | |
| **Description of arguments:** str: a char array where the line will be copied to<br>size: the number of characters we want to read up to<br>stream: a file stream that was opened for reading | | |
| **What type it returns:** char* | **On success:** A non-NULL pointer to the start of the char array containing the copied line. | |
| | **On failure:** NULL | |
| | **If failure, does it set errno?** Yes | |
| **Quick example:** | | |

```
char buf[1000]; // buffer on the stack
if (!fgets (buf, 1000, stream)) // Same as checking for (fgets(...) == NULL)
   /* Handle the errors */
```

**Other variations:** N/A

# `<stdio.h>`: Reading from Streams, Line by Line



Note that `fgets()` automatically adds the null `'\0'` character into the char array at the end of the line.

Want a function that stops at a **delimiter** (= stopping character) other than `'\n'`? Here is a possible implementation:

```
void * read_line (char * str, int str_size, int delimiter) {
    int c = 0;
    while (--str_size > 0 && (c = fgetc (stream)) != EOF && (*str++ = c) != delimiter);
    if (c == delimiter) // checking if it's the delimiter.
        *--str = '\0'; // if so, replacing the delimiter character with '\0'.
    else
        *str = '\0'; // all (str_size - 1) chars or EOF were read; add '\0' to end of str.
}
```

This function behaves the same as `fgets()` except that we stop at `delimiter`, not at `'\n'`.

# `<stdio.h>`: Reading from Streams, as Binary Data



<table>
<tr><td colspan="2">Name: <code>fread()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: reads up to <code>nr</code> elements of data, each of <code>size</code> bytes, from <code>stream</code> into the array <code>buf</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines fread().</code></td></tr>
<tr><td colspan="4">Syntax: <code>size_t fread (void *buf, size_t size, size_t nr, FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>buf</code>: an array (or variable's address) into which the binary data is read<br><code>size</code>: the size in bytes of each data element that we want to read (e.g., <code>sizeof(double)</code>)<br><code>nr</code>: the amount of such data elements to read<br><code>stream</code>: a file stream that was opened for reading</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>size_t</code></td><td colspan="2">On success: The amount of elements that were read in (not the number of total bytes)</td></tr>
<tr><td colspan="2">On failure: A number smaller than <code>nr</code>. This can also happen when the end-of-file was reached: we'll see how to distinguish these cases apart on the next slide.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:<br><code>int arr [3];<br>size_t number_of_elements_read = fread (arr, sizeof(int), 3, stream);</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

# `<stdio.h>`: Reading from Streams, as Binary Data



The file position is advanced by the number of bytes that are read-in by `fread()`.

As we explained, a return value smaller than the amount of elements we wanted to read could mean either that an error happened or that the end of the file was reached. To distinguish between the two cases, we should call two functions after calling `fread()`: `ferror()` and `feof()` as follows:

```
int arr [3];
size_t number_of_elements_read = fread (arr, sizeof(int), 3, stream);
if (number_of_elements_read < 3 && ferror(stream))
    /* Handle the error */
if (number_of_elements_read < 3 && feof(stream))
    /* The end of the line was reached */
clearerr(stream); // Function that clears the EOF and error indicators
```

The visiting cards of `ferror()`, `feof()`, and `clearerr()` are on the following slides.

# `<stdio.h>`: Reading from Streams, as Binary Data



<table>
<tr><td colspan="2">Name: <code>ferror()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: checks if an error happened during the execution of <code>fread()</code>. If yes, it returns a non-zero integer.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines ferror().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int ferror (FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>stream</code>: a file stream that was opened for reading</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: A non-zero integer if an error indeed happened, or <code>0</code> if an error hadn't happened.</td></tr>
<tr><td colspan="3">On failure: – This function never fails –</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No; it never fails</td></tr>
<tr><td colspan="4">Quick example:<br><code>if (ferror (stream)) // Same as: if (ferror (stream) != 0)<br>exit (EXIT_FAILURE);</code></td></tr>
<tr><td colspan="4">Other variations: <code>feof()</code>, <code>clearerr()</code></td></tr>
</table>

# `<stdio.h>`: Reading from Streams, as Binary Data



<table>
<tr><td>Name: <code>feof()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: checks if the end of file was reached during the execution of <code>fread()</code>. If yes, it returns a non-zero integer.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <stdio.h> // Defines feof().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int feof (FILE *stream);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>stream</code>: a file stream that was opened for reading</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: A non-zero integer if the end of file was reached during the call to <code>fread()</code>, or <code>0</code> if the end-of-file wasn't reached.</td></tr>
<tr><td colspan="2">On failure: – This function never fails –</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No; it never fails</td></tr>
<tr><td colspan="3">Quick example:

```
if (feof (stream)) // Same as: if (feof (stream) != 0)
    if (num_of_elements_read > 0) // At least some elements were read in!
        /* Do something with the elements that were */
        /*    read in before the end of file was reached. */
```

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

# `<stdio.h>`: Reading from Streams, as Binary Data



| **Name:** `clearerr()` | **System call or function:** Function | **Links:** Online Manual \| Course Packet |
|---|---|---|
| **What it does:** clears (nullifies) the flags in `<stdio.h>` that indicate an error or end of file cases, so that we can freshly call `ferror()` and `feof()` again in the future. Call this function only after checking for errors/end-of-file. | | |
| **What libraries you must include:** `#include <stdio.h> // Defines clearerr().` | | |
| **Syntax:** `void clearerr (FILE *stream);` | | |
| **Description of arguments:** `stream`: a file stream that was opened for reading | | |
| **What type it returns:** `void` | **On success:** – it returns nothing and always succeeds – | |
| | **On failure:** – This function never fails – | |
| | **If failure, does it set `errno`?** No; it never fails | |

**Quick example:**

```
if (ferror (stream))
    /* ... */
if (feof (stream))
    /* ... */
clearerr (stream);
```

**Other variations:** `ferror()`, `foef()`

# `<stdio.h>`: Reading from Streams, as Binary Data



Any questions?

# `<stdio.h>`: Writing to Streams, Character-by-Character



We can write to streams (those that were open in any mode except for `"r"`) in 3 ways, too: (1) one character at a time, (2) an entire line at a time, or (3) as binary data.

<table>
<tr><td>Name: <code>fputc()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: writes a single character <code>c</code> to a stream <code>stream</code> that was opened for writing.<br>Question: What special file does the function <code>int putchar (int c);</code> write a character to? Some of you used it in HW 2!</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <stdio.h> // Defines fputc().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int fputc (int c, FILE *stream);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>c</code>: the character that we want to write to the stream<br><code>stream</code>: a file stream that was open for writing</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: A non-negative integer ASCII value of the character that was written.</td></tr>
<tr><td colspan="2">On failure: The constant <code>EOF</code> (= <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>if (fputc ('A', stream) == EOF) /* Handle the errors */</code></td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

# `<stdio.h>`: Writing to Streams, Line by Line



<table>
<tr><td colspan="2">Name: <code>fputs()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: writes a null-terminated array of characters <code>str</code> to the stream <code>stream</code> (note that <code>'\0'</code> isn't written out to the stream.)</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines fputs().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int fputs (const char *str, FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>str</code>: a char array ending with <code>'\0'</code><br><code>stream</code>: a file stream that was open for writing</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: Some non-negative integer (could also be <code>0</code>.)</td></tr>
<tr><td colspan="2">On failure: <code>EOF</code> (= <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:<br><code>if (fputs ("Hello, World!\n", stream) == EOF)<br>    /* Handle the error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

# <stdio.h>: Writing to Streams, as Binary Data



<table>
<tr><td colspan="2">Name: fwrite()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: writes up to nr elements of data, each of size bytes, to stream from the array buf.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines fwrite().</code></td></tr>
<tr><td colspan="4">Syntax: <code>size_t fwrite (void *buf, size_t size, size_t nr, FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: buf: an array (or variable's address) from which the binary data is taken<br>size: the size in bytes of each data element that we want to write (e.g., sizeof(double))<br>nr: the amount of such data elements to write<br>stream: a file stream that was opened for writing</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns:<br>size_t</td><td colspan="2">On success: The amount of elements that were read in (not the number of total bytes)</td></tr>
<tr><td colspan="2">On failure: A number smaller than nr. There is no end-of-file case when we write to files, so a number smaller than nr indicates only an error.</td></tr>
<tr><td colspan="2">If failure, does it set errno? Yes</td></tr>
<tr><td colspan="4">Quick example:<br><code>int arr [3] = {-4, 0, 10598};<br>size_t num_of_elements_written = fwrite (arr, sizeof(int), 3, stream);</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

# buffered_io.c: Working with Binary Data



```
/* The following program exemplifies calls to
 *    fopen() fwrite(), fread(), and fclose(). We
 *    write the binary data of a single structure
 *    into a file. This same file is then opened
 *    for reading, and we read this data back in
 *    again as binary data into another structure
 *    instance, s. We then print the data fields
 *    of s to the terminal.
 *
 *    Miriam Briskman, 3/8/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */
```

# files.c: Reading from a File



```c
/* C program that reads from and writes to files.
 *
 *    Miriam Briskman, 2/8/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <stdio.h>
#include <stdlib.h>

// Constant storing array size:
const int ARR_SIZE = 400;

int main ()
```

# Buffered File I/O

Any questions?

# <stdio.h>: Changing File Position

| | | |
|---|---|---|
| **Name:** `fseek()` | **System call or function:** Function | **Links:** Online Manual \| Course Packet |
| **What it does:** changes the file position of the file `stream` to `offset` based on the starting point at `whence`. | | |
| **What libraries you must include:** `#include <stdio.h> // Defines fseek().` | | |
| **Syntax:** `int fseek (FILE *stream, long offset, int whence);` | | |
| **Description of arguments:** `stream`: any open file stream<br>`offset`: a signed long integer indicating by how many places you want to move the position.<br>`whence`: one of the following 3 constants:<br>1. `SEEK_CUR`: sets file's position to current value + `offset` (`offset` could be any integer).<br>2. `SEEK_END`: the position is set to the end of file + `offset`.<br>3. `SEEK_SET`: sets the file's position to `offset`. | | |
| **What type it returns:** `int` | **On success:** `0` | |
| | **On failure:** `-1` | |
| | **If failure, does it set `errno`?** Yes | |
| **Quick example:** `if (fseek (stream, 1825, SEEK_CUR) == -1) /* Handle the error */` | | |
| **Other variations:** `ftell()`, `fsetpos()`, `rewind()` | | |

# <stdio.h>: Changing File Position



<table>
<tr><td colspan="2">Name: ftell()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: returns the current position (integer) of the file.<br>Question: How do you call the lseek() system call to return you the current file position? Fill in the blanks:<br><code>off_t pos = lseek (fd, _____, _____);</code><br>[ Hint: Replace the 1st blank with some integer and the 2nd blank with one of the SEEK_* constants. ]</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines ftell().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int ftell (FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: stream: any open file stream</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: int</td><td colspan="2">On success: A non-negative integer representing the current file position.</td></tr>
<tr><td colspan="2">On failure: -1</td></tr>
<tr><td colspan="2">If failure, does it set errno? Yes</td></tr>
<tr><td colspan="4">Quick example:
<pre>int pos = ftell (stream);
if (pos == -1)
    /* Handle the error */
printf ("file position = %d", pos);</pre></td></tr>
<tr><td colspan="4">Other variations: fseek(), fsetpos(), rewind()</td></tr>
</table>

# <stdio.h>: Changing File Position



| **Name:** fsetpos() | **System call or function:** Function | **Links:** Online Manual \| Course Packet |
|---|---|---|

**What it does:** sets the file position of `stream` to `pos` .
***Question***: How do you call the `fseek()` function to set the position exactly to `pos`? Fill in the blank:

```
int result = fseek (stream, pos, _____);
```

[ ***Hint***: Replace blank with one of the `SEEK_*` constants. ]

**What libraries you must include:** `#include <stdio.h> // Defines fsetpos().`

**Syntax:** `int fsetpos (FILE *stream, fpos_t *pos);`

**Description of arguments:** `stream`: any open file stream
`pos` a pointer to an `fpos_t` variable (non-negative integer) that contains the desired position

| **What type it returns:** `int` | **On success:** `0` |
|---|---|
| | **On failure:** `-1` |
| | **If failure, does it set `errno`?** Yes |

**Quick example:**

```
fpos_t pos = 5;
if (fsetpos (stream, &pos) == -1)
    /* Handle the error */
```

**Other variations:** `fseek()`, `ftell()`, `rewind()`

# <stdio.h>: Changing File Position



**Name:** rewind()

**System call or function:** Function

**Links:** Online Manual | Course Packet

**What it does:** sets the file position of stream to 0 (the start of the file.)

***Question***: How do you call the fseek() function to set the position exactly to the start of the file? Fill in the blanks:

```
int result = fseek (stream, _____, _____);
```

[ ***Hint***: Replace the 1st blank with an integer and the 2nd blank with one of the SEEK_* constants. ]

**What libraries you must include:** `#include <stdio.h> // Defines rewind().`

**Syntax:** `void rewind (FILE *stream);`

**Description of arguments:** stream: any open file stream

**What type it returns:** void

**On success:** – This function doesn't return anything –

**On failure:** – This function doesn't return anything –

**If failure, does it set errno?** Yes

**Quick example:**

```
errno = 0;
rewind (stream);
if (errno != 0)
    /* Handle the error */
```

**Other variations:** fseek(), ftell(), fsetpos()

# `<stdio.h>`: I/O Buffers



***So, what's the advantage of using the functions of `<stdio.h>`?***

**Answer**: The `<stdio.h>` library conveniently stores and manages a buffer (array) called the **user buffer** or **`stdio` buffer** in which it keeps a copy of a file's content. This buffer stores multiple blocks of bytes, so there is no need in calling `read()` too frequently, so it avoids repetitive copies of huge chunks of bytes from disk.

***And disadvantage?***

**Answer**: Remember that the OS itself manages a buffer called the **kernel buffer** in which, for example, changes that we wrote to files are temporarily stored before being saved to the disk.

This creates a slight disadvantage for `<stdio.h>`: when we need to read a file's content from disk, it is first copied to the kernel buffer, then from the kernel buffer to the user buffer, and then to the char array that we supplied to `fgets()`, `fgetc()`, etc. The writing case is similar: when we write to a file, the string from our char array is copied to the user buffer, then copied to the kernel buffer, and then eventually to disk. As such, since the same data to several vessels, the execution of the `<stdio.h>` functions may be slower than this of the system calls in certain cases.

# <stdio.h>: Flushing Data to Kernel Buffer



<table>
<tr><td colspan="2">Name: <code>fflush()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: for writing file: copies the content of the user buffer of <code>stream</code> to the kernel (OS) buffer. For reading files: discard any changes made to the buffer (e.g., pushing a character via <code>ungetc()</code>.)</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines fflush().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int fflush (FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>stream</code>: any open file stream. Pass <code>NULL</code> to flush the buffers of all the open streams.</td></tr>
<tr><td rowspan="3" colspan="2">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>EOF</code> (= <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 examples:
<pre>
if (fflush (stream) == EOF) // Flush one file's user buffer
    /* Handle the error */
if (fflush (NULL) == EOF) // Flush all the open files' user buffers
    /* Handle the error */
</pre></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Note that `fflush()` doesn't force the copy of the kernel buffer to disk as the system calls `fsync()` or `sync()`.

# <stdio.h>: Getting a File's fd



<table>
<tr><td colspan="2">Name: fileno()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: returns the file stream's fd, so that you could issue system calls on the file, too.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines fileno().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int fileno (FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: stream: any open file stream.</td></tr>
<tr><td rowspan="3">What type it returns: int</td><td colspan="3">On success: A non-negative integer representing the fd of the file.</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 examples:<br><code>int fd = fileno (stream)<br>if (fd == -1)<br>/* Handle the error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Notice that mixing <stdio.h> function calls and system calls is not advised normally: syscalls may conflict with the user buffering. A good practice is to flush the stream to the kernel buffer using fflush() before issuing system calls.

# `<stdio.h>`: Thread Safety



Multi-threaded processes (= running programs) are like several processes sharing the same address space.

On a multi-processor system, multiple threads could run concurrently.

Threads may overwrite shared data unless the measures below are taken:

- Synchronize access to data (= **locking**), or
- Make the data thread-local (= **thread confinement**).

An OS provides locking mechanisms to allow concurrent `<stdio.h>` calls, but you may want to have more refined control.

- While individual `<stdio.h>` functions are thread-safe, you may want to lock a whole code section so that only 1 thread accesses it at a time, to prevent race conditions.

# <stdio.h>: Manual File Locking



<table>
<tr><td colspan="2">Name: flockfile()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: waits (blocks) until stream is no longer locked, increases the lock count, and then acquires the lock, thus becoming the thread that owns of stream.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines flockfile().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void flockfile (FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: stream: any open file stream.</td></tr>
<tr><td rowspan="3" colspan="2">What type it returns: void</td><td colspan="2">On success: – This function doesn't return anything –</td></tr>
<tr><td colspan="2">On failure: – This function doesn't return anything –</td></tr>
<tr><td colspan="2">If failure, does it set errno? No</td></tr>
<tr><td colspan="4">Quick examples: flockfile (stream);</td></tr>
<tr><td colspan="4">Other variations: funlockfile(), ftrylockfile()</td></tr>
</table>

# <stdio.h>: Manual File Locking



<table>
<tr><td colspan="2">Name: <code>funlockfile()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: decreases the lock count, and if the lock count reaches zero, the current thread relinquishes ownership of the <code>stream</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdio.h> // Defines funlockfile().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void funlockfile (FILE *stream);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>stream</code>: any open file stream that was locked with <code>flockfile()</code>.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>void</code></td><td colspan="2">On success: – This function doesn't return anything –</td></tr>
<tr><td colspan="2">On failure: – This function doesn't return anything –</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick examples: <code>funlockfile (stream);</code></td></tr>
<tr><td colspan="4">Other variations: <code>flockfile()</code>, <code>ftrylockfile()</code></td></tr>
</table>

# <stdio.h>: Manual File Locking



| **Name:** `ftrylockfile()` | **System call or function:** Function | **Links:** Online Manual \| Course Packet |
|---|---|---|
| **What it does:** it will return a nonzero integer immediately if the `stream` is currently locked. Otherwise, it returns `0` and works similarly to `flockfile()`. | | |
| **What libraries you must include:** `#include <stdio.h> // Defines ftrylockfile().` | | |
| **Syntax:** `int ftrylockfile (FILE *stream);` | | |
| **Description of arguments:** `stream`: any open file stream | | |
| **What type it returns:** `int` | **On success:** A nonzero integer if stream is locked, or `0` otherwise. | |
| | **On failure:** – This function doesn't fail – | |
| | **If failure, does it set `errno`?** No | |
| **Quick examples:** `int result = ftrylockfile (stream);` | | |
| **Other variations:** `flockfile()`, `funlockfile()` | | |

# `<stdio.h>`: Unlocked Stream Operations



Individual `<stdio.h>` functions are internally locked, which incurs overhead.

Linux provides a family of unlocked counterparts of `<stdio.h>` unctions for boosting performance (this is Linux-specific, not POSIX.)

- Such as `fgets_unlocked()`, `fread_unlocked()`, etc.

It is the programmer's responsibility to manually acquire and release locks for them if desired.

How to gain in performance without using locks:

- Delegating all I/O to a single thread, or
- Delegating all I/Os to a pool of threads, where each file stream is mapped to exactly one thread in the pool.

# `<stdio.h>`: Conclusions



Standard I/O is a powerful and popular user-buffering library that is a part of the standard C library.

Best used for:

- Minimizing overhead (incurred penalty) by internally combining many system calls.
- Performance gain by ensuring all I/O occurs in block-sized chunks.
- When your access patterns are character or line-based (`fgets()` is easier to use.)
- When you prefer a higher-level interface to the low-level Linux syscalls.

A great website for reading about `<stdio.h>` functions: https://cplusplus.com/reference/cstdio/

# `<stdio.h>`: Conclusions



Any questions?

# Buffered File I/O: Sources

Topic 5: Buffered 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+5:+buffered+file+i/o.

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