escola superior de tecnologia e gestão
instituto politécnico de leiria

Standard I/O

High-level

Patrício R. Domingues

Departamento de Eng Informática

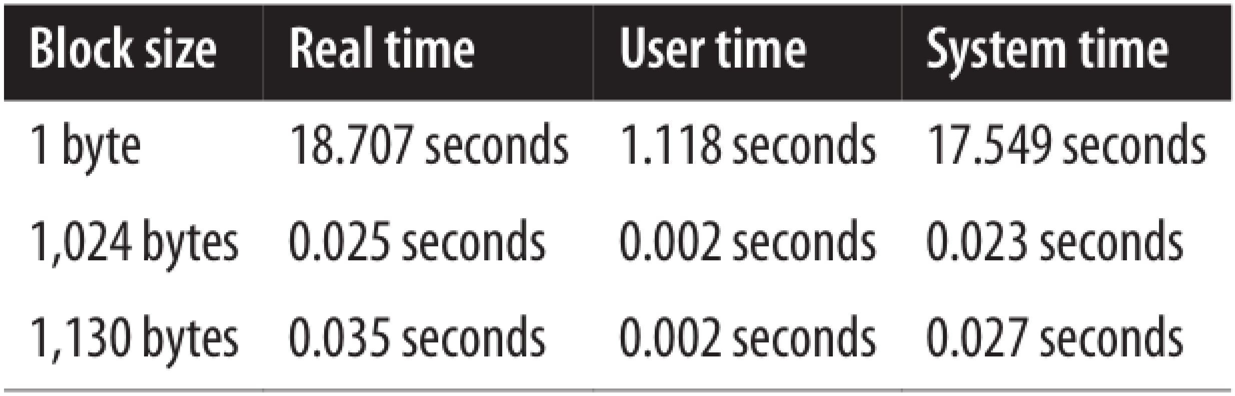
ESTG/IPLeiria

Buffered I/O (1)

- Compared to CPU and RAM, persistent storage is slow
  - Example: access disk is around 10 ms for HDD and .1 ms for SDD (approximate values)
    - Much slower than RAM (~80 ns)
- Persistent storage is organized in blocks
  - A disk (SSD, HDD,etc.) is a set of blocks
  - We can only access disk to read/write **blocks**
  - Blocks are typically 512 bytes , 1024, 2048, 4096 and 8192 bytes
  - A disk does not read/write a byte. It reads/writes a block.

Performance *byte* vs. *block*

- Effect of block size on performance
  - Copy of 2 MiB (HDD)
    - 1 byte at the time
    - 1024 bytes at the time
      - 1024 bytes at the time is much faster than 1 byte at the time



Block size	Real time	User time	System time
1 byte	18.707 seconds	1.118 seconds	17.549 seconds
1,024 bytes	0.025 seconds	0.002 seconds	0.023 seconds
1,130 bytes	0.035 seconds	0.002 seconds	0.027 seconds

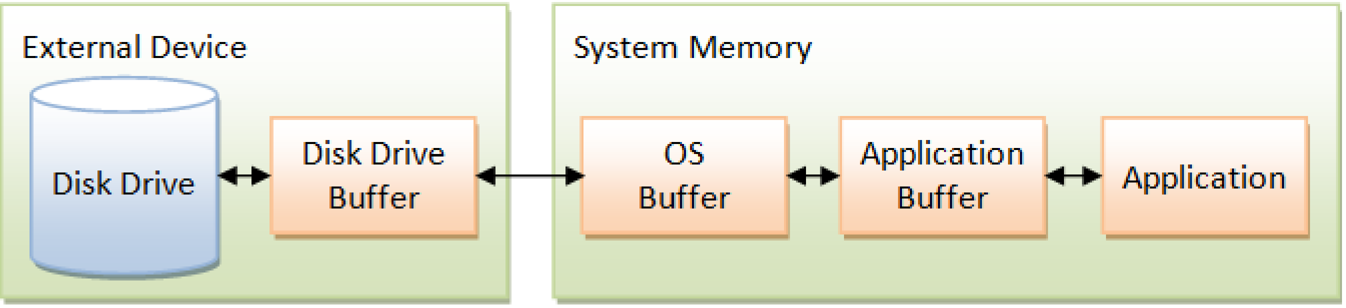
- Source: *"Buffered I/O", Chapter 3 - Linux System Programming*, Robert Love, 2nd Edition, O'Reilly, 2013

Buffered I/O (2)

- At the storage level, read and write operations are done in blocks
  - The operating system and the device driver already **buffer** writes and reads
    - Kernel buffers, etc.
  - But programmers work with fields, lines and characters
    - Programmers work at the semantic level
    - It would be improductive to force programmers to work with block...
- Solution
  - User buffered I/O
    - Standard I/O in C
    - #include <stdio.h>

Buffered I/O (3)

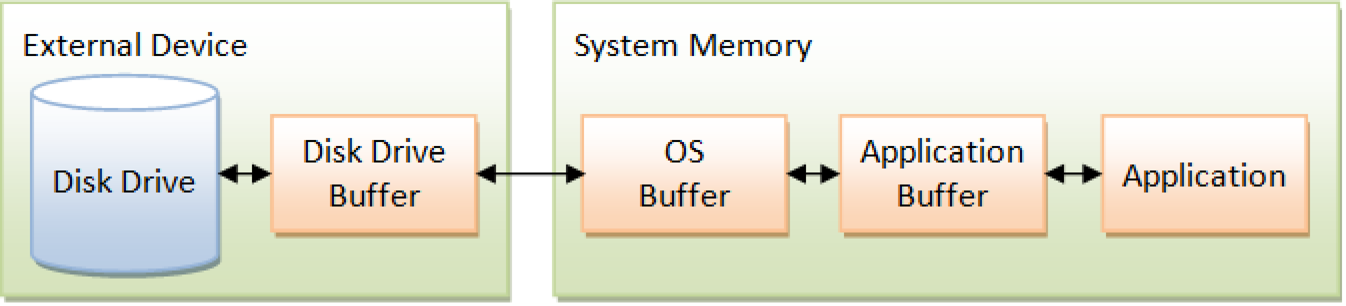
- In C, user buffered I/O is available through STDIO
  - STDIO ➔ standard Input/Ouput
  - FILE *, fopen, etc.
- Basics of user buffered I/O
  - The write and read operations use buffers that are set in <u>**user space**</u>
    - Buffer: zone of memory (e.g., chunk of 1024 bytes)



Source: http://bit.ly/1PjBkxJ

Buffered I/O (4)

- The functions of user buffered I/O (`fread`, `fwrite`, `fprintf`, etc.) transparently deal with writes and reads to/from persistent storage
- They user *write* and *read* buffers in user-space memory
- Whenever a buffer is *full* (write) or *empty* (read), appropriate actions are taken
  - Flush to disk (write); reload of buffer (read)



FILE *

- Standard I/O routines operate on file through so-called *file pointers*
- File pointer type: FILE* (defined in <stdio.h>)
  - Example: FILE *f;
- Documentation often refers to an open FILE* as a *stream*
  - *Input stream*
  - *Output stream*
  - *Input/Output stream*

The *fopen* function (1)

- `FILE *fopen (const char *`**`path`**`, const char *`**`mode`**`);`
  - *path*: path for the file to be opened
  - *mode*: how the file will be used (**string**)
    - r: reading
    - r+: reading and writing (file position at the start of the file)
    - w: writing. If file exists, it is truncated to zero
    - w+: writing and reading. If file exists, it is truncated to zero. File position at the start of the file.
    - a: writing in append mode. File created if it does not exist. File position at the end of the file.
    - a+: writing and reading. All writes will append to the file.

The *fopen* function (2)

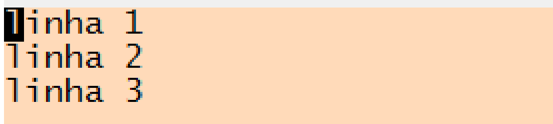
- *mode*: how the file will be used
  - Some OSes (e.g., Microsoft Windows) distinguish between text and binary files
    - Next slides
  - For this OS, the mode parameter needs to be complemented with **b** for binary files.
    - e.g. **rb**, **rb+**, etc.

text vs. *binary* mode (windows)

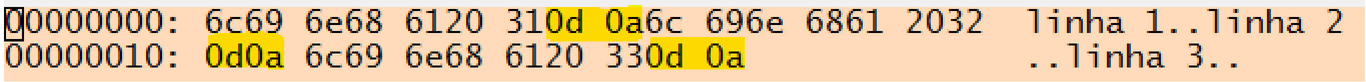
- Differences **text vs. binary** files (e.g. Windows)
  - Newlines in **text mode**
    - \n → \r\n when file is written to disk;
    - \r\n → \n when file is read from disk in TEXT mode.
    - **No** conversion in BINARY mode.
  - \r is carriage return (hex code: 0x0D)
  - \n is line feed (hex code: 0x0A)
- Representation of numbers
  - Text mode: string (e.g., "64502" → five characters)
  - Binary: integer (e.g., 64502 → four bytes: 0xFBF6)

DOS/Windows vs. Unix

- Example
  - DOS/Windows format
    - `0x0d0A: \r\n`

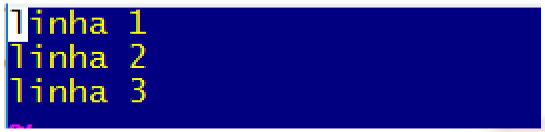


```
linha 1
linha 2
linha 3
```

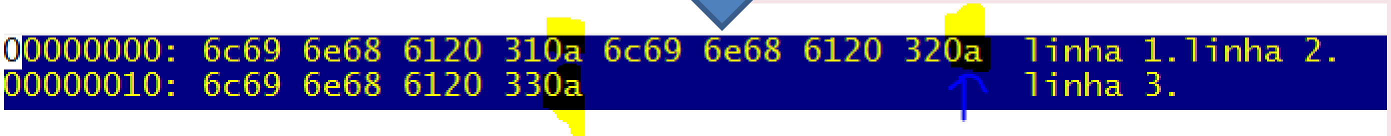


```
00000000: 6c69 6e68 6120 310d 0a6c 696e 6861 2032  linha 1..linha 2
00000010: 0d0a 6c69 6e68 6120 330d 0a              ..linha 3..
```

  - Unix format
    - `0x0A: \n`



```
linha 1
linha 2
linha 3
```



```
00000000: 6c69 6e68 6120 310a 6c69 6e68 6120 320a  linha 1.linha 2.
00000010: 6c69 6e68 6120 330a                      linha 3.
```

The *fopen* function (3)

- `FILE *fopen (const char *path, const char *mode);`
  - *Returns NULL on error*
  - *Returns a pointer to the file if everything went ok*
    - *errno keeps the error*

Closing streams

- `int fclose (FILE *stream);`
  - Any buffered and not-yet-written data are first flushed. On success, `fclose()` returns `0`.
  - On failure, it returns **EOF** and sets **errno** appropriately.
- `int fcloseall(void);`
  - Flush all streams of process and then close them all streams
    - *All streams* means that `stdin`, `stdout` and `stderr` are also closed
  - **Warning:** Linux-specific

Reading from a FILE*

- There are several functions in STDIO for reading from a FILE*
  - Reading one character at a time: **fgetc** and **ungetc**
  - Reading an entire line: **fgets**
  - Reading binary data: **fread**
  - Reading formated data: **fscanf**

fgetc **&** ungetc >>

fgetc & ungetc

- Reading a character
  - `int fgetc (FILE *stream);`
- Returns
  - EOF at end of file, otherwise it returns the read character
- *Pushing back* a character to the bufer
  - int ungetc(int c, FILE *stream);
  - Puts back character **c** to stream

fgets >>

Reading a line - *fgets*

```
char *fgets (char *str, int size, FILE *stream);
```

- This function reads up to **one less than** *size* **bytes** from stream and stores the results in *str* .
- A null character ('\0 ') is stored in the buffer after the last byte read in.
- Reading stops after an EOF or a newline character is reached.
- If a newline is read, the \n is stored in *str* .
- On success, *str* is returned;
- on failure, NULL is returned.

getline function >>

getline function

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

- POSIX function
- It reads a line (up to "\n") from a stream and...
  - Copies to **lineptr* if **lineptr* is not NULL (and **n* is set to the size of the buffer pointed by **lineptr*). If the buffer pointed by *lineptr is not large enough it gets reallocated
    - It needs to be an allocated buffer
  - Allocates a buffer and copies the line to this allocated buffer if **lineptr* is NULL. The address of the buffer is returned in **lineptr*
- **n* is returned with the number of bytes written to **lineptr*
- *getline* returns the number of chars read or -1 on ERROR or EOF

getdelim function

```
ssize_t getdelim(char **lineptr, size_t *n,
                                               int delim, FILE *stream);
```

- Similar to *getline*
- Only difference is parameter “delim”
  - It identifies the end-of-line terminator
    - *getline* always uses '\n' as end-of-line terminator
    - *getdelim* allows to specify a different one

Binary data - reading

- Reading chunks of binary data
  - Example: set of C structures

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);

  - Read up to `nr` elements, each of `size` bytes from `stream` into the buffer pointed by `buf`
  - It returns the number of elements read (not the number of bytes!)
    - if the number of elements read is less than *nr*, then…
      - The function might have reached EOF . Test it with `feof()`.
      - There might be an error. Test it with `ferror()`.
- Example

```
char buf[128];
size_t nr;
nr = fread (buf, sizeof(buf), 1, stream);
if (nr == 0){
    /* error */
```

Reading formated data

- `fscanf` for reading formatted data
- `int fscanf (FILE *stream,const char *format,...);`
  - *Returns*
    - *the number of matched an assigned inputs*
    - *EOF on error*
- *There is a family of scanf functions*
  - `scanf`*: reads from stdin*
  - `sscanf`*: reads from a string*

Writing to a stream

- Writing to a stream
  - Three main operations
    - A character
    - A string
    - A chunk of binary data

Writing a character to a stream >>

Writing a char to a stream

- `int fputc (int c, FILE *stream);`
  - `fputc()` writes the byte specified by `c` (cast to an unsigned char) to the stream pointed by *`stream`*.
    - `stream` must be open for writing
  - On success, it returns `c`
  - On error, it returns EOF , and `errno` is set appropriately.
- Example

```
if (fputc ('p', stream) == EOF){
/* error */
…}
```

Writing a string to a stream (1)

- `int fputs (const char *str, FILE *stream);`
  - `fputs()` writes the '\0' terminated string *`str`* to *`stream`*.
    - `stream` must be open for writing
  - On error, it returns EOF , and `errno` is set appropriately.
- Example

```
if (fputs ("texto", stream) == EOF){
/* error */
…}
```

Writing a string to a stream (2)

- Another function to write a string to a stream is…

```
int fprintf(FILE *stream, const char *format, ...);
```

- Example
  - Remember, `stderr` is a stream!

```
char *str = "this is a test!";
fprintf(stderr,"%s",str);
```

Writing binary data

- `size_t fwrite (void *buf, size_t size, size_t nr, FILE *stream);`
  - write to `stream` up to `nr` elements from buffer `buf`
  - each element has `size` bytes
  - `fwrite` returns the number of elements successfully written to `stream`
    - A return value less than `nr` denotes error.

Working with binary data

- Binary data are dependent on many issues
  - Differences in variable sizes, alignment, padding, and byte order
  - Binary data written with one application may not be readable by a different application
- Solution
  - Standardized formats
    - Examples
      - GZIP file format: RFC 1952 (https://tools.ietf.org/html/rfc1952)
      - PGM file format: http://netpbm.sourceforge.net/doc/pgm.html
      - PNG specification - ISO/IEC 15948:2003 (E): https://www.w3.org/TR/PNG/
      - Etc.

Set the position of a stream

- Standard I/O functions to deal with the position of a stream
  - `int fseek (FILE *stream, long offset, int whence);`
    - `whence`:SEEK_SET, SEEK_CUR, SEEK_END
  - `void rewind (FILE *stream);`
    - Reset the file position
    - same as `fseek(stream, 0,SEEK_SET)`

Get the position of a stream

- What’s the current position of a stream?
- `long ftell (FILE *stream);`
  - Returns the current stream position
  - On error, it returns -1 and errno is set appropriately

Flushing a stream (1)

- Flushing a stream
  - Have the buffer written through the `write()` system call
- Function `int fflush (FILE *stream);`
  - Any unwritten data in `stream` is flushed to the OS
  - If stream is NULL , <u>all open input streams</u> in the process are flushed
- Returns:
  - 0 on success
  - EOF on error and sets `errno` appropriately

Flushing a stream (2)

- Flushing a stream
  - The standard I/O buffer is written through the `write()` system call to the OS buffer
  - It does not mean that the content is actually written to file
    - Standard I/O buffers are in user-space
    - OS buffers are in kernel space
- Using standard I/O functions, we minimize the number of (costly) system calls
  - A system call (e.g., *write*) is issued only when the disk or some other storage has to be accessed.

Flushing a stream (3)

- To provide for effective write to persistent storage, the programmer needs to...
  - *fflush* the stream
  - *fsync* the associated file descriptor
    - The file descriptor of a stream can be obtained with:
      - `int fileno (FILE *stream);`

Errors and End-of-File

- Function `ferror` tests if the error indicator is set on *stream*
  - `int ferror (FILE *stream);`
  - Returns non-zero on error, 0 otherwise
- Function `feof` tests if file position of stream is at end-of-file
  - `int feof (FILE *stream);`
- Function `clearerr` clears error status of `stream`
  - `void clearerr (FILE *stream);`

Controlling buffering (1)

- Types of buffering
  - *Unbuffered*
    - No user buffer is performed. Data are sent directly to the kernel. STDERR is, by default, unbuffered. Otherwise, this mode is rarely used.
  - *Line-buffered*
    - With each newline (\n), the buffer is submitted to the kernel. This is the default for stdout.
  - *Block-buffered* or *full-buffering*
    - A block is a fixed number of bytes.

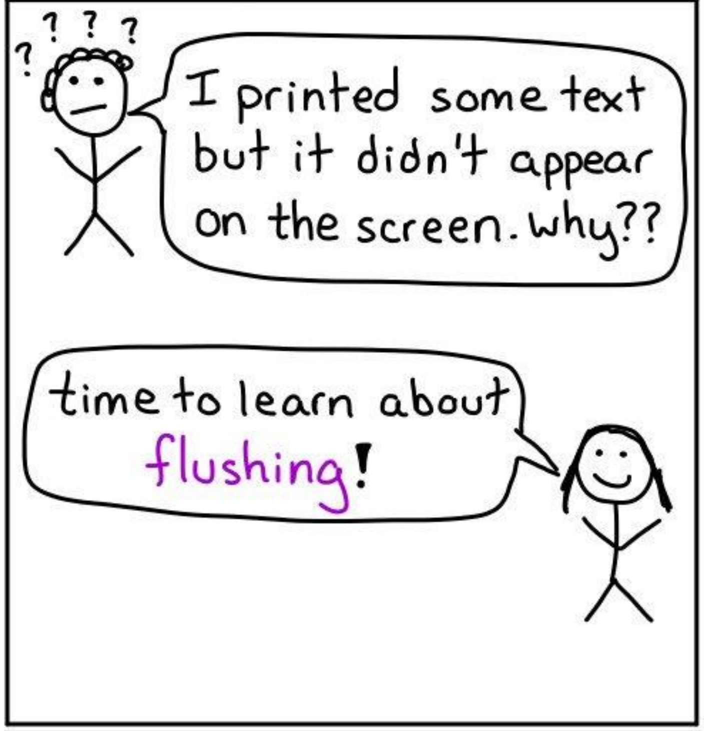
Controlling buffering (2)

- Function `setvbuf`
  - To be called right after opening `stream`
  - `int setvbuf (FILE *stream, char *buf, int mode, size_t size);`
- Mode can be
  - _IONBF: unbuffered
  - _IOLBF: line-buffered
  - _IOFBF: Block-buffered (*full-buffered)*
- `buf` may point to a buffer of `size` bytes that standard I/O will use as the buffer for `stream`.
- If `buf` is NULL , a buffer of `size` bytes is allocated automatically

drawings.jvns.ca

file buffering

JULIA EVANS
@b0rk


? ? ? ?
I printed some text but it didn't appear on the screen. why??
time to learn about flushing!

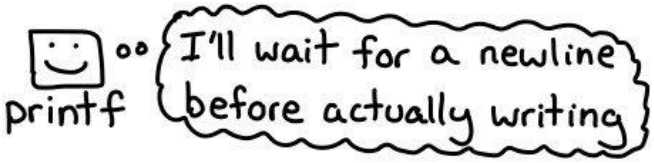
On Linux you write to files & terminals with a system call called

♥ write ♥

please write "I ♥ cats" to file #1 (stdout)
okay!
Linux

I/O libraries don't always call write when you print

```
printf("I ♥ cats");
```



This is called buffering and it helps save on syscalls

3 kinds of buffering

(defaults vary by library)

1. None. This is the default for stderr
2. Line buffering. (write after newline). The default for terminals.
3. "full" buffering. (write in big chunks). The default for files and pipes.

flushing

To force your IO library to write everything it has in its buffer right now, call flush!

stdio
I'll call write right away!!

when it's useful to flush

- when writing an interactive prompt!

  Python example:

  ```
  print("password: ", flush=True)
  ```

- when you're writing to a pipe / socket

program
no seriously, actually write to that pipe please

remove function

```
int remove(const char *pathname);
```

- Remove is a function available in stdio
- It removes an entry in the filesystem
  - file
  - directory
- In practice, it calls the appropriate system call
  - **`unlink`** for a file
  - **`rmdir`** for a directory

References

- *“Buffered I/O”, Chapter 3 - Linux System Programming*, Robert Love, 2nd Edition, O’Reilly, 2013
- Man pages for standard I/O functions
  - man 3 printf
  - man 3 fopen
  - …