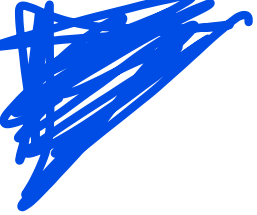

Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Files and file descriptors

Two different interfaces to files

1. *streams* of type

```
FILE * my_f;
```

(`FILE` is a `struct` defined in `stdio.h`) and

- input/output is *buffered* to improve performance:
  - incovenient to write on a disk byte by byte
  - data to be written to disk is stored to a memory area (*buffer*) only
  - the buffer is written to the disk (*flushed*) depending on the buffering policy

2. *file descriptors* of type

```
int my_fd;
```

a file descriptor is just an integer (which is the index in a table managed by the operating system)

- lower level interface
- not buffered
- file descriptors are used for more general purpose than writing on a disk file (interprocess communication, communicate via TCP/IP, etc.)

Outline

Streams: opening/closing

- Before being used streams must be opened by

```
FILE * fopen(const char *path, const char *mode);
```

  - `path` is the path of the file to be open
  - `mode` is a **string** (not a character) specifying the read/write opening mode. Example: `"rw"` Check: `man fopen` for full description
  - a pointer `(FILE *)` is returned

  Example: opening `"my_file.txt"` in read mode

```
FILE * my_f;

my_f = fopen("my_file.txt", "r");
```

- After its usage, a stream must be closed by

```
int fclose(FILE * stream);
```

  if streams are not closed, the OS may be unable to open new files

Streams: reading

- For each open file, the OS keeps and updates a *file position indicator*
- Every reading happens at the current position, which is incremented by the number of bytes read

```
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int fscanf(FILE *stream,const char *format,...);
```

- they all read from the current position
- `fgetc` reads and returns the byte (`char`) read into a (`int`). If end-of-file is reached, then the `int` non `char`-representable `EOF` macro is returned (tipycally of value `-1`)
  - the byte `0xFF` can be distinguished by `EOF`
- `fgets` reads an array of up to `size` bytes. Returned `NULL` if end-of-file is reached.
- `fscanf` read by `scanf/printf` format

Streams: writing

- Every writing happens at the current position, which is incremented by the number of written bytes

```
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int fprintf(FILE *stream,const char *format,...);
```

- `fputc` writes the byte `c`, casted to `char`, to the file
- `fputs` writes the zero-terminated string `s` **without** the terminating 0 byte
- `fprintf` writes to file by the `printf` format

Streams: controlling the position over a file

- The position over a file can be controlled by `fseek()`

```
int fseek(FILE *stream, long offset, int whence);
```

  sets the file pointer of `stream` as follows:
  - if `(whence == SEEK_SET)`, position is set equal to `offset`
  - if `(whence == SEEK_CUR)`, position is moved by `offset`
  - if `(whence == SEEK_END)`, position is moved by `offset` from the end

  notice that `offset` may be negative (to move the position backward)
- To know the current position over a file

```
long ftell(FILE *stream);
```

  - The first byte of a file is at position `0`
  - The last byte of a file is at position `<size>-1`
  - When the position is equal to `<size>`, then we reached the end-of-file
- *test-file.c*

Standard streams: `stdin`, `stdout`, `stderr`

- `stdin`, `stdout`, and `stderr` are all streams (of type `FILE *`) defined by the operating system with a special usage
- `stdin` is "standard input" and it is the stream of characters entered by the keyboard
- `stdout` is "standard output" and it is the stream of characters printed on the terminal
- `stderr` is the "standard error" stream. It is used to print error messages and it is printed on the terminal as well

Streams: buffering

- The interaction between the (fast) processor and the (slow) devices may degrade the performance
  - it is not convenient to write a single byte to the disk every time `fputc()` is invoked
- I/O may be buffered: "buffered" read/write are delayed until the "buffering" condition is true. Three types of buffering
  1. unbuffered: all I/O operations happen immediately
  2. block buffered: I/O operations are executed when the buffer is full
  3. line buffered: I/O operations are executed when newline `'\n'` read
- `stdout` is line-buffered
- `stderr` is not-buffered (normally, we want to see the error messages as soon as they happen): during debugging use `stderr`
- other files are block buffered, unless specified differently

Streams: controlling the buffering

- To force the buffer to be written to the device

```
int fflush(FILE *stream);
```

  by `fflush(NULL)`, all open output streams are flushed.

- the function `setvbuf` changes the buffering policy of `stream`

```
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

  if `mode` is:

  1. `_IONBF`, stream is unbuffered (every single byte is written/read immediately)
  2. `_IOLBF`, the buffer is written as soon as newline is found
  3. `_IOFBF`, write to disk only whe buffer is full

  `man setvbuf` for more information

- Examples

```
/* set no buffering to stream */
setvbuf(stream, NULL, _IONBF, 0);
```

Outline

File descriptors and files

- **Streams** (not files) are of type `(FILE *)` (the name "`FILE`" is only for historical reasons)
- **File descriptors** are of type `int` (sometime called "I/O streams")
- A file descriptor (fd) identifies a source/destination of a sequence of bytes
- File descriptors are a lower level interface than streams
- File descriptors are more general than streams
  - all streams have a file descriptor
  - there may be file descriptors which are not streams
- File descriptors are opened by different functions depending on their usage:
  - `int open(...)` (not `fopen(...)`) binds a file in the file system to the returned descriptor
  - `int socket(...)` binds the data coming-from/going-to a UDP/TCP (and others) connection to the returned descriptor
  - `pipe(...)` creates a "pipe": two descriptors attached to each other (more details later in the course)

File descriptors linked to standard streams

- `stdin`, `stdout`, and `stderr` are standard streams opened by the OS and allowing the program to:
  - read from keyboard (from `stdin`)
  - write normal output to terminal (to `stdout`)
  - write error messages to terminal (to `stderr`)
- Standard file descriptors are associated to these streams:
  - the integer 0 is the file descriptor of `stdin`
  - the integer 1 is the file descriptor of `stdout`
  - the integer 2 is the file descriptor of `stderr`

Redirecting stdout and/or stderr

- To redirect stdout to a file, truncate if existing
  ```
  COMMAND 1> filename
  ```
- To redirect stdout to a file, append if existing
  ```
  COMMAND 1>> filename
  ```
- To redirect stderr to a file, truncate if existing
  ```
  COMMAND 2> filename
  ```
- To redirect stderr to a file, append if existing
  ```
  COMMAND 2>> filename
  ```
- To redirect stdout and stderr to a file, truncate if existing
  ```
  COMMAND &> filename
  ```
- To redirect stdout and stderr to a file, append if existing
  ```
  COMMAND &>> filename
  ```

Opening/closing a file descriptor of a file

```
int open(const char *pathname, int flags);
```

- `open(...)` opens a file and returns a fd (`man 2 open` for details)
  - `pathname`, a string with the pathname of the file
  - `flags`, specifies how to open (about 20 flags).
    Flags are set by making the bitwise OR "|" among the selected macros
  - Each macro has one "1" bit only
    - ★ **must include** one among `O_RDONLY`, `O_WRONLY`, `O_RDWR`
    - ★ `O_APPEND`, file opened in append mode
    - ★ `O_CREAT`, create the file if doesn'e exist
    - ★ `O_TRUNC`, if file exists, it is truncated
- After being used, file descriptors must be closed

```
int close(int fd);
```

otherwise we may run out of available file descriptors

Reading from a file descriptor

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

- reads from the file descriptor `fd` up to `size` bytes and store them to `buf`
- it returns the number of bytes actually read (it may be less than `size`)
- if it returns zero, then end-of-file is reached
- if it returns -1 then an error has occured

Writing to a file descriptor

```
ssize_t write(int fd, const void *buf, size_t size);
```

- write `size` bytes from the buffer `buf` to the file descriptor `fd`
- it writes immediately the data, not buffered as `fprintf`
- formatted output over a file descriptor `fd` by

  ```
  int dprintf(int fd, const char *format, ...);
  ```

- WARNING: by mixing `fprintf` and `write` to the same fd/stream you must be careful
  - `fprintf` uses a buffer to write, while `write` doesn't
  - the output written by `fprintf` may be delayed w.r.t. the output made via `write`
- *test-buf.c*

Positioning over a file descriptor

- This position over a file descriptor is controlled by `lseek()`

```
off_t lseek(int fd, off_t offset, int whence);
```

- set the file pointer of `fd` as follows:
  - if (`whence == SEEK_SET`), position is set equal to `offset`
  - if (`whence == SEEK_CUR`), position is moved by `offset`
  - if (`whence == SEEK_END`), position is moved by `offset` from the end
- notice that `offset` may be negative (to move the position backward)
- File descriptors of different types (not associated to files) do not allow positioning by `lseek(...)`