Input/output

François Trahay

CSC4508 – Operating Systems

2022–2023

Outlines

1 Buffered / non-buffered IO 3
2 I/O primitives 4
3 I/O and concurrence 9
4 Improving the I / O performance 12

1 Buffered / non-buffered IO

- Buffered I/O
  - Write operations are grouped in a *buffer* which is written to disc from time to time
  - When reading, a data block is loaded from disk to *buffer*
  - $\rightarrow$ a buffered I/O $\neq$ an operation on the disk
  - eg. `fopen`, `fread`, `fscanf`, `fwrite`, `fprintf`, etc.
  - Data stream identified by an *opaque pointer* `FILE*`
- Unbuffered I/O
  - an unbuffered I/O = an operation on the disk †
  - eg. `open`, `read`, `write`, etc.
  - Open file identified by a *file descriptor* of type `int`

2 I/O primitives

2.1 File open / close 5
2.2 Reading on a file descriptor 6
2.3 Writing on a file descriptor 7
2.4 File descriptor duplication 8

2.1 File open / close

- `int open(const char *path, int flags, mode_t mode)` : retour = f_id

  `flags` can take one of the following values:
  - `O_RDONLY`: read only
  - `O_WRONLY`: write only
  - `O_RDWR`: read and write

  Additional flags:
  - `O_APPEND`: append data (write at the end of the file)
  - `O_TRUNC`: truncate (empty) the file when opening it
  - `O_CREAT`: creation if the file does not exist. The permissions are $(mode \ \& \ \sim umask)$
  - `O_SYNC`: open file in synchronous write mode
  - `O_NONBLOCK` (ot `O_NDELAY`): open and subsequent operations performed on the descriptor will be non-blocking.

- `int close(int desc)`

2.2 Reading on a file descriptor

- `ssize_t read(int fd, void *buf, size_t count)` : return = number of bytes successfully read
  - When `read` returns, the `buf` zone contains the read data;
  - In the case of a file, the number of bytes read may not be be equal to `count`:
    - We reached the end of the file
    - We did a non-blocking read and the data was exclusively locked

2.3 Writing on a file descriptor

- `ssize_t write(int fd, const void *buf, size_t count)` : return value = number of bytes written
  - In the case of a file, the return value (without error) of the write operation means that:
    - Bytes were written to kernel caches unless `O_SYNC` was specify at file open;
    - Bytes have been written to disk if `O_SYNC` was specified.
  - In the case of a file, a number of bytes written that is different from `count` means an error (e.g. No space left on device)

2.4 File descriptor duplication

- Mechanism mainly used to perform redirection of the three standard I/O files.
- `int dup(int old_fd)` : return value = `new_fd`
  associates the smallest available file descriptor of the calling process the same entry in the open files table as the descriptor `old_fd`
- `int dup2(int old_fd, int new_fd)`
  force the file descriptor `new_fd` to become a synonym of the `old_fd` descriptor. If the descriptor `new_fd` is not available, the system first closes `close(new_fd)`

3 I/O and concurrence

3.1 Locking a file 10
3.2 Offset manipulation 11

3.1 Locking a file

- `struct flock { short l_type; short l_whence; off_t l_start; off_t l_len; };`
- `int fcntl(int fd, F_SETLK, struct flock*lock);`
- Locks are attached to an *inode*. So locking a file affects all file descriptors (and therefore all open files) corresponding to this *inode*
- A lock is the property of a process: this process is the only one authorized to modify or remove it
- Locks have a scope of $[integer1 : integer2]$ or $[integer : \infty]$
- Locks have a type:
  - `F_RDLCK`: allows concurrent read access
  - `F_WRLCK`: exclusive access

3.2 Offset manipulation

- `off_t lseek(int fd, off_t unOffset, int origine)` : return = new offset
  allows you to handle the *offset* of the file
- Warning ! Race condition if several threads manipulate the file
- Solutions:
  - Handling of the file in mutual exclusion
  - Using `pread` or `pwrite` instead of `lseek + read` or `lseek + write`

4 Improving the I / O performance

4.1 Giving advices to the kernel 13
4.2 Asynchronous I/O 14
4.3 mmap 15

4.1 Giving advices to the kernel

- `int posix_fadvise(int fd, off_t offset, off_t len, int advice)`
  - examples of advice: `POSIX_FADV_SEQUENTIAL`, `POSIX_FADV_RANDOM`, `POSIX_FADV_WILLNEED`
  - return value = 0 if OK, error number otherwise
  - allows you to tell the kernel how the programm will access a file, which allows the kernel to optimize accordingly

4.2 Asynchronous I/O

- `int aio_read(struct aiocb *aiocbp)`
- `int aio_write(struct aiocb *aiocbp)`
- Starts an asynchronous read / write operation
- Returns immediately
- `int aio_suspend(const struct aiocb * const aiocb_list[], int nitems, const struct timespec *timeout)`
  - Waits for the end of an asynchronous operation
- `int aio_error(const struct aiocb *aiocbp)`
  - Tests the end of an asynchronous operation

4.3 mmap

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
  - "map" a file in memory
  - memory accesses to the buffer are transformed into disk operations
- `int munmap(void *addr, size_t length)`
  - "unmap" a buffer