Input/output

François Trahay

Contents

In this lecture, we mainly talk about files, as this is the easiest example of I/O to manipulable. However, note that the content of the first 3 sections apply to I/O other than files (eg sockets).

Reminder on files:

- A file is a series of contiguous bytes stored in a medium (for example, a disk) under a name (the "name of the file").
- We distinguish several types of the files:
  - text: containing bytes that can be displayed on the screen. This type of files consists of lines identified by the character end of line (on Unix, ASCII code character `10` while on Windows, ASCII code character `10` followed by a character of ASCII code `13`);

  - binary: containing bytes that cannot be displayed on the screen.

On Unix, the commands `hexdump -C filename`, `bless filename` or `xxd filename` show the exact content of a file. Use them to

1. compare the contents of `helloWorldUnix.c` and `helloWorldWindows.c`
2. see that the file `default_names_fichierIssuDuTP10DuModuleCSC4103.txt` is not quite a text file (and, see also how are the accented characters stored in a file)

- When you "open" a file, the operating system provides a notion of current position (sometimes called *offset* in the rest of this course) for reading or writing.
  - This current position determines which byte in the file will be read-/written during the next I/O operation.
  - This *offset* advances each time a read or write operation is performed.
  - The operating system provides the user with primitives to explicitly change this position (without reading or writing bytes).
- The "end of a file" corresponds to the location behind the last byte of the file. When a program reaches the end of file, it cannot read bytes anymore. On the other hand, the program can write bytes (depending on the mode in which the file was opened).
- There are 3 ways to access a file:
  - Sequential: the bytes are read one after the others from the beginning of the file.
  - Direct: you can set the *offset* without reading bytes before *offset*.
  - Indexed sequential: the file contains records, each record being identified by a key (unique or no). Using the key, you can position the offset at the start of a recording. You can also read the recordings in the order defined by their key.

The Linux system and the C library provide sequential and direct access modes. For an indexed sequential access mode, other libraries are required (Unix NDBM, GDBM, Oracle Berkeley DB, ...).

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*

→ a buffered I/O ≠ 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`

† To be exact, an "unbuffered" I/O generates a system call. The OS can then decide to cache the data or no.

I/O primitives

File open / close

- `int open(const char *path, int flags, mode_t mode)` : returns 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)`

About the `O_SYNC` option in `open`:

- To improve performance, by default, during a write operation, the operating system does not physically write the bytes on disk (they are stored in a kernel cache, waiting to be writen to disk)
- Therefore, in the event of a sudden stop of the machine (example: power outage):
  - data thought to have been written to disk may to be lost because they were in fact in memory;
  - there is also a risk of inconsistency in the data on the disk.

- Solutions to synchronize file data in memory with the disc:
  - implicit synchronization (i.e. on each write): adding the `O_SYNC` option when opening the file;
  - explicit synchronization (i.e. the application decides) via the `int fsync(int fd)` primitive

Note that we can also create a file using the `creat` primitive:

- `int creat(const char *path, mode_t mode)` : return value = f_id
  - which is equivalent to the following call to `open`:
  - `open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)`.

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

In the case where the `read` function is used on a descriptor other than a file (e.g. a pipe, or a socket), the fact that the number of bytes read may not equal `count` may have other meanings:

- for a communication pipe (see the Inter-process communication lecture), the correspondent has closed its end of the pipe.
- for a socket (see course NET4103), the network protocol uses data packets smaller than the size that is requested.

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)

Writing to disk is atomic: if two processes P_1 and P_2 simultaneously write to the same file in the same location, when the two processes have finished their writing, we will find:

- either the data written by P_1,
- either the data written by P_2,
- but never a mix of the data written by P_1 and P_2.

Note that when the file is opened with the option `O_APPEND`, if P_1 and P_2 write simultaneously (at the end of the file, because of `O_APPEND`), when the two processes will have finished their writing, we will find at the end of file:

- either the data written by P_1 followed by the one written by P_2,
- or the data written by P_2 followed by the one written by P_1.

No writing is therefore lost! Attention, this concurrent write at the end of file is not equivalent to two processes simultaneously performing the following operations:

```
lseek(fd,0,SEEK_END); /* move the cursor to the end of file */
write(fd,data,taille);
```

In fact, in the latter case, one of the written data may by overwritten by the other.

The `copy.c` file on the next page illustrates the use of `open`, `read`, `write` and `close`.

```
/************/
/* copy.c */
/************/
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>

#define USAGE "USAGE: copy src dest\n"
#define WRITE_ERROR "write error (no space left on device ?)\n"

int source, dest;
int buf;
int nb_read, nb_written;

int main(int argc, char *argv[]) {
  if (argc != 3) {
```

```
    write(STDERR_FILENO, USAGE, strlen(USAGE));
    return EXIT_FAILURE;
  }
  source = open(argv[1], O_RDONLY);
  if (source < 0) {
    perror(argv[1]);
    return EXIT_FAILURE;
  }
  dest = open(argv[2],
              O_WRONLY|O_CREAT|O_TRUNC,
              S_IRWXU|S_IRWXG|S_IRWXO);
  if (dest < 0) {
    perror(argv[2]);
    return EXIT_FAILURE;
  }
  while ((nb_read = read(source, (void*)&buf, sizeof(buf))) > 0) {
    nb_written = write(dest, (void*)&buf, nb_read);
    if (nb_written <= 0) {
      if (nb_written == 0) {
        write(STDERR_FILENO, WRITE_ERROR, strlen(WRITE_ERROR));
      }
      else {
        perror("write");
      }
      return EXIT_FAILURE;
    }
  }
  if (nb_read < 0) {
    perror("read");
    return EXIT_FAILURE;
  }
  if (close(source) < 0) {
    perror(argv[1]);
    return EXIT_FAILURE;
  }
  if (close(dest) < 0) {
    perror(argv[2]);
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

This operation of copying the contents of one file to another descriptor is an operation frequently performed in web servers. Indeed, these servers must in particular send the content of files to client who have requested them. This is why the linux system offers the `sendfile` primitive (`ssize_t  sendfile (int out_fd, int`

`in_fd, off_t * offset, size_t count`)). It reads `count` bytes of `in_fd` and write them to `out_fd` (which must match an *socket*). `sendfile` is more more efficient than the combination `read` / `write`.

The `fallocate` function is the Linux specific version of the portable function `posix_fallocate`.

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)`

I/O and concurrence

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

The exclusive-lock.c file illustrates exclusive file locking:

```c
/***********/
/* exclusive_lock.c */
/***********/
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main(){
  int fd;
  struct flock lock;

  fd = open("/tmp/ficTest",O_RDWR|O_CREAT, S_IRWXU|S_IRWXG|S_IRWXO);
  if (fd < 0) {
    perror("open");
    exit(EXIT_FAILURE);
  }

  /* Exclusive lock on the 15th byte */
  lock.l_type = F_WRLCK;
  lock.l_whence = SEEK_SET;
  lock.l_start = 15;
  lock.l_len = 1;

  /* Because of the F_SETLKW parameter, we get stuck on the fcntl if */
  /* the lock cannot be acquired                                     */
  printf("attempt to acquire an exclusive lock by process %d...\n",
         getpid());
  if (fcntl(fd, F_SETLKW, &lock) < 0){
    perror("Acquiring lock");
    exit(EXIT_FAILURE);
  }
  printf("... Exclusive lock acquired by process %d\n", getpid());

  /* Here we could do the processing that needed to be protected */
  /* by the lock                                                 */
  sleep(10);

  /* Release the lock */
  printf("Releasing the lock by process %d...\n", getpid());
  lock.l_type = F_UNLCK;
  lock.l_whence = SEEK_SET;
```

```
  lock.l_start = 15;
  lock.l_len = 1;
  if (fcntl(fd, F_SETLK, &lock) < 0){
    perror("Releasing lock");
    exit(EXIT_FAILURE);
  }
  printf("...OK\n");

  return EXIT_SUCCESS;
}
```

The `shared-lock.c` file illustrates the shared locking:

```
/*****************/
/* shared_lock.c */
/*****************/
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main(){
  int fd;
  struct flock lock;

  fd = open("/tmp/ficTest",O_RDWR|O_CREAT, S_IRWXU|S_IRWXG|S_IRWXO);
  if (fd < 0) {
    perror("open");
    exit(EXIT_FAILURE);
  }

  /* Shared lock on the 15th byte */
  lock.l_type = F_RDLCK;
  lock.l_whence = SEEK_SET;
  lock.l_start = 15;
  lock.l_len = 1;

  /* Because of the F_SETLKW parameter, we get stuck on the fcntl if */
  /* the lock cannot be acquired                                     */
  printf("attempt to acquire a shared lock by process %d...\n",
         getpid());
  if (fcntl(fd, F_SETLKW, &lock) < 0){
    perror("Acquiring lock");
    exit(EXIT_FAILURE);
  }
  printf("... shared lock acquired by process %d\n", getpid());
```

```
  /* Here we could do the processing that needed to be protected */
  /* by the lock                                                 */
  sleep(10);

  /* Release the lock */
  printf("Releasing the lock by process %d...\n", getpid());
  lock.l_type = F_UNLCK;
  lock.l_whence = SEEK_SET;
  lock.l_start = 15;
  lock.l_len = 1;
  if (fcntl(fd, F_SETLK, &lock) < 0){
    perror("Releasing lock");
    exit(EXIT_FAILURE);
  }
  printf("...OK\n");

  return EXIT_SUCCESS;
}
```

- If we run `exclusive-lock` first, running `exclusive-lock` or `shared-lock` wait before locking.
- If we run `shared-lock` first, another `shared-lock` can set the (shared) lock. On the other hand, a `exclusive-lock` must wait to be able to lock.
- Note that exclusive_lock may suffer starvation:
  - start a 1st shared_lock.
  - start exclusive_lock: it wait
  - start a 2nd shared_lock. The 1st shared_lock ends. But as the 2nd shared_lock is running, exclusive_lock is still waiting.
  - start a 3rd shared_lock. The 2nd shared_lock ends. But as the 3rd shared_lock is running, exclusive_lock is still waiting.
  - we see that as long as shared_lock starts while the previous shared_lock has not finished running, exclusive_lock must wait: exclusive_lock may face starvation.

  To prevent this starvation, we must add a mutual exclusion.

Offset manipulation

- `off_t lseek(int fd, off_t unOffset, int origine)` : return = new offset
  - allows 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`

Improving the I / O performance

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

Since January 2011, we know that this function is used in Firefox to reduce startup time by 40 % to 50 % by loading more efficiently GUI libraries `xul.dll` and `mozjs.dll` (more information here <https://bugzilla.mozilla.org/show_bug.cgi?id=627591>).

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

For more information on asynchronous I/O, refer to the documentation (`man 7 aio`).

The current implementation of AIO Posix is provided in *user-land* by libc and can cause scalability issues. Another solution is to use the Asynchronous I/O

interface provided by the Linux kernel (see the system calls `io_submit`, `io_setup`, etc.), or the `libaio` library which provides an overlay to Linux system calls.

---

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

To ensure that the memory accesses have been passed on to the disk, you can use the `msync` function.