System and Device Programming

The UNIX I/O

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

POSIX Standard Library

- I/O in UNIX can be entirely performed with only **5** functions
  - open, read, write, lseek, close
- This type of access
  - Is part of POSIX and of the Single UNIX Specification, but not of ISO C
  - It is normally defined with the term "unbuffered I/O", in the sense that each read or write operation corresponds to a system call

System call open()

- In the UNIX kernel a "file descriptor" is a non-negative integer
- Conventionally (also for shells)
  - Standard input
    - 0 = STDIN_FILENO
  - Standard output
    - 1 = STDOUT_FILENO
  - Standard error
    - 2 = STDERR_FILENO

These descriptors are defined in the headers file **unistd.h**

System call open()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *path, int flags);

int open (const char *path, int flags,
  mode_t mode);
```

- It opens a file defining the permissions
- Return value
  - The descriptor of the file, on success
  - The value -1, on error

System call open()

- It can have 2 or 3 parameters
  - The **mode** parameter is optional
- **Path** indicates the file to open
- **Flags** has multiple options
  - Can be obtained with the OR bit-by-bit of constants defined in the header file **fcntl.h**
  - One of the following three constants is mandatory
    - O_RDONLY open for read-only access
    - O_WRONLY open for write-only access
    - O_RDWR open for read-write access

```
int open (const char *path, int flags, mode_t mode);
```

System call open

- Optional constants
  - O_CREAT creates the files if not exist
  - O_EXCL error if O_CREAT is set and the file exists
  - O_TRUNC remove the content of the file
  - O_APPEND append to the file
  - O_SYNC each write waits that the physical write operation is finished before continuing

...

  - O_NONBLOCK nonblocking I/O

Analyzed in the next section

```
int open (const char *path, int flags, mode_t mode);
```

System call open()

- **Mode** specifies permission access
  - S_I[RWX]USR rwx --- ---
  - S_I[RWX]GRP --- rwx ---
  - S_I[RWX]OTH --- --- rwx

```
int open (
   const char *path,
   int flags,
   mode_t mode
);
```

When a file is created, actual permissions are obtained from the **umask** of the user owner of the **process**

System call read()

```
#include <unistd.h>

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

- Read from file **fd** a number of bytes equal to **nbytes**, storing them in **buf**
- Returned values
  - The number of read bytes, on success
  - The value -1, on error
  - The value 0, in the case of EOF

System call write()

```
#include <unistd.h>

int write (int fd, void *buf, size_t nbytes);
```

- Write **nbytes** bytes from **buf** in the file identified by descriptor **fd**
- Returned values
  - The number of written bytes (i.e., normally **nbytes),** in the case of success
  - The value -1, on error

System call write()

- Remarks
  - Function **write** writes on the system buffer, not on the disk
    - fd = open (file, O_WRONLY | O_SYNC);
  - O_SYNC forces the sync of the buffers, but only for ext2 file systems

```
int write (int fd, void *buf, size_t nbytes);
```

Examples

```
float data[10];

if ( write(fd, data, 10*sizeof(float))==(-1) ) {
  fprintf (stderr, "Error: Write %d).\n", n);
}
```

Write a data array (of float values)

```
struct {
  char name[L];
  int n;
  float avg;
} item;

if ( write(fd,&item,sizeof(item)))==(-1) ) {
  fprintf (stderr, "Error: Write %d).\n", n);
}
```

Write a structured item (with 3 fields) in binary form

File Pointers

- In UNIX
  - It is possible to explicitly modify file pointers to perform **random walks** on the file
  - Random walks can be implemented using the function lseek

System call lseek()

```
#include <unistd.h>

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

- The current position of the file offset is associated to each file
  - This position indicates the one from which the next read/write operation starts
  - The system call lseek assigns the value **offset** to the file offset

System call lseek()

- Whence specifies the interpretation of offset
  - If whence==SEEK_SET
    - The offset is evaluated from the beginning of the file
  - If whence==SEEK_CUR
    - The offset is evaluated from the current position
  - If whence==SEEK_END
    - The offset is evaluated from the end of the file

The value of **offset** can be positive or negative

It is possible to leave "holes" in a file (filled with zeros)

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

System call lseek()

- Return value
  - New offset, on success
  - -1, on error

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

System call close()

```
#include <unistd.h>

int close (int fd);
```

- It closes the file of descriptor **fd**
  - Notice that, all the open files are closed automatically when the process terminates
- Return value
  - The value 0, on success
  - The value -1, on error

Example

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define BUFFSIZE 4096

int main(void) {
  int nR, nW, fdR, fdW;
  char buf[BUFFSIZE];
  fdR = open (argv[1], O_RDONLY);
  fdW = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC,
                       S_IRUSR | S_IWUSR);
  if ( fdR==(-1) || fdW==(-1) ) {
    fprintf (stdout, "Error Opening a File.\n");
    exit (1);
  }
```

Example

```
while ( (nR = read (fdR, buf, BUFFSIZE)) > 0 ) {
    nW = write (fdW, buf, nR);
    if ( nR!=nW )
      fprintf (stderr,
        "Error: Read %d, Write %d).\n", nR, nW);
  }

  if ( nR < 0 )
    fprintf (stderr, "Write Error.\n");

  close (fdR);
  close (fdW);

  exit(0);
}
```

Error check on the last reading operation

This program works indifferently on text and binary files