Filesystem on Unix / POSIX

adrien.poteaux@lifl.fr

LIFL, Université Lille 1

Year 2014-2015

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

http://creativecommons.org/licenses/by-nc-sa/3.0/
Normalisation of the interface

- **Unix:**
  - Operating system,
  - Ken Thompson and Denis Ritchie, Bell Labs, 1969,
  - Source code distributed,
  - Several versions... 

- **POSIX:**
  - Portable Open System Interface eXchange,
  - Portable Open System Interface X for Unix,
  - Standard IEEE, 1985,
  - Standardized interface of what the system has to offer.

- One POSIX function = one Unix system call.
A good programming language:
- good semantic,
- efficient,
- access to all the computer structures (registers, bits...),
- explicit allocation memory.

Natural interface with the system:
- Libraries are written in C,
- One can use them from C

Other approaches are possible!
Libraries and system call

- System call similar at using a library’s function.
- Standard C library: section 3 of `man`
- POSIX functions: section 2 of `man`

Other differences:
- No link edition,
- System code execution,
- Standard libraries are a higher level abstraction.

**Important**: system calls are costly → minimize them!
Files: persistent memory

- used to store data ("for good"),
- managed by the system: structure, naming, access, protection...
- filesystem is part of the OS.
- Dissociation OS / Filesystem (e.g., Linux can mount NTFS)
- Unix approach: “everything” is a file
Filesystem: a graph organisation with nodes
Operating on a file

- Informations:
  - peripheric number, inode number
  - file’s type, size...
  - dates...
  - owner, group, rights

- Scanning the hierarchy: listing, moving in the hierarchy

- Changing the hierarchy:
  - Creation or destruction of nodes,
  - Physical and symbolic links

- Reading and writing in ordinary files.
Informations on a file

- **Structure struc stat ; access via:**
  ```c
  # include <sys/types.h>
  # include <sys/stat.h>
  int stat(const char *path, struct stat *sb);
  int lstat(const char *path, struct stat *sb);
  ```

- **Identification of a node:**
  ```c
  struct stat {
    dev_t st_dev;
    ino_t st_ino;
    ...
  }
  ```

- A lot more informations: cf lecture notes.
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

struct stat sb;
int status;
status = stat(pathname, &sb);
if (status) {
    perror("stat call");
    return;
}
if (S_ISREG(sb.st_mode)) {
    printf("Ordinary file");
    if (sb.st_mode & S_IXUSR)
        printf("\, executable by its owner");
}

(cf lecture notes to understand the different functions and macros)
Depending on the type of the file (first thing to check):

- Ordinary file: access to the data.
- Directory: access to the list of related nodes ("children").
- Symbolic link: access to the name of the pointed file (than same as above).
- Special file: access to some peripheric data, possible limitations, specific operations might be possible.
Scanning directories

- Content of a directory = linked list → loop on the list.

- Opening and closing a directory:
  
  ```
  #include <dirent.h>
  DIR *opendir(const char *dirname);
  int closedir(DIR *dirp);
  ```

- Loop on the input:
  
  ```
  #include <dirent.h>
  struct dirent {
    ino_t d_ino;
    char d_name[];
  }

  struct dirent *readdir(DIR *dirp);
  ```
One example

```c
static int
lookup(const char *name)
{
    DIR *dirp;
    struct dirent *dp;
    if ((dirp = opendir(".")) == NULL) {
        perror("couldn’t open ’.’");
        return 0;
    }
    while ((dp = readdir(dirp))) { /* uses linked list */
        if (! strcmp(dp->d_name, name)) {
            printf("found %s\n", name);
            closedir(dirp);
            return 1;
        }
    }
    if (errno != 0) /* cf man 3 errno for details */
        perror("error reading directory");
    else
        printf("failed to find %s\n", name);
    closedir(dirp);
    return 0;
}
```
File pointed by a symbolic link

- Path of the pointed file

```c
#include <unistd.h>
ssize_t readlink(const char *path, char *buf, size_t bufsize);

// (returns the number of characters of the path ; −1 if error)
```

- Typical use:

```c
char buf[PATH_MAX+1];
ssize_t len;
if ((len = readlink(path, buf, PATH_MAX)) != -1)
    buf[len] = '\0'; /* do not forget ! */
else
    perror("error reading symlink");
```
One can also...

- create nodes (recursively if the directory does not exist),
- create links,
- destroy nodes,
- destroy directories,
- ...

(not detailed in this lecture)
Reading and writing in a file...

- **C language:** the `<stdio.h>` library
  - FILE structure,
  - FILE *
    ```c
    fopen(const char *path, const char *mode);
    fclose(FILE *fp);
    fprintf, fscanf,...
    ```
  - Fortran language:
    - `open(unit=FD,file='filename'), close(FD)`
    - `write(FD,*), read(FD,*)`

- Behind both: system functions
  - `<sys/types.h>, <sys/stat.h>, <fcntl.h>, <unistd.h>`
  - `int open(const char *pathname, int flags, mode_t mode);
    close(int fd);`
  - `ssize_t read(int fd, void *buf, size_t count);
    write(int fd, const void *buf, size_t count);`
A process need to use a file → designed by an integer

File descriptor: index of a table containing informations related to the file.

The first three elements are:
- the standard input stdin (index 0) ; by default the keyboard,
- the standard output stdout (index 1) ; by default the screen,
- the error output stderr (index 2) ; also the screen by default.

<stdio.h> uses the same numbers (but one uses macros !)

Fortran uses 5 (stdin) and 6 (stdout)
One example using directly system calls

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#define BUFSIZE 4096
static void
copy_file(const char *src, const char *dst)
{
    int fdsrc, fddst;
    char buffer[BUFSIZE];
    int nchar;
    fdsrc = open(src, O_RDONLY);
    fddst = open(dst, O_WRONLY | O_CREAT | O_TRUNC,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    while ((nchar = read(fdsrc, buffer, BUFSIZE))) {
        write(fddst, buffer, nchar);
    }
    close(fdsrc);
    close(fddst);
}
```

*(missing: error considerations)*
The `strace` command

- prints set of system calls made by a process,
- functions usually detailed in `man` (section 2),
- Default output is on `stderr`! To separate the printing of the program and the one of `strace`:

```
$ strace ./a.out 2>trace.txt
```

You can run `strace` on the executable produced by the following code to see how `<stdio.h>` is using system calls

```c
#include <stdio.h>
int main()
{
    int i;
    printf("Hello");
    putchar(\n');
    printf("Hello again ");
    printf("world\n");
    putchar('H');
    putchar('e');
    putchar('l');
    putchar('l');
    putchar('o');
    for (i=0;i<500;i++)
        putchar('3');
    putchar(\n');
    for (i=0;i<50;i++)
    {
        putchar('2');
        putchar(\n');
    }
    for (i=0;i<3000;i++)
        putchar('p');
    putchar(\n');
    return 0;
}
```

adrien.poteaux@lifl.fr

Filesystem
Why libraries then?

- Higher level abstraction,
  - Formatted input / output
    - `fprintf()`, `fscanf()`...
  - More efficient:
    - input and output use buffers,
    - reduces costly system calls,
    - one system call $\approx 1000$ instructions,
    - call read / write only when the buffer is full / empty.
A better example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
static void
copy_file(const char *src, const char *dst)
{
    struct stat stsrc, stdst;
    FILE *fsrc, *fdst;
    int c;
    lstat(src, &stsrc);
    lstat(dst, &stdst);
    if (stsrc.st_ino == stdst.st_ino && stsrc.st_dev == stdst.st_dev) {
        fprintf(stderr, "%s and %s are the same file\n", src, dst);
        return;
    }
    fsrc = fopen(src, "r");
    fdst = fopen(dst, "w");
    while ((c = fgetc(fsrc)) != EOF)
        fputc(c, fdst);
    fclose(fsrc);
    fclose(fdst);
}
```

**Warning:** avoid mixing system calls and library calls (that uses system calls !)
Process and program

- What is a program?
  - static code description,
  - sequence of instructions.

- What is a process?
  - dynamic activity,
  - it has a life: creation, execution, end.

- One process = one execution of a program; one can have:
  - several executions of programs,
  - several executions of the same program,
  - execution “at the same time” of different programs,
  - execution “at the same time” of the same program.
Processor

- Physical entity,
- Affectation to a process:
  - time allocation,
  - enables the process to progress.
- Affectation choice: ordering of the process... dealt by the OS,
- We consider only a single processor, i.e. sequential operations:

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

→ time
## Process characteristics

<table>
<thead>
<tr>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>process ID</td>
</tr>
<tr>
<td>father’s process ID</td>
</tr>
<tr>
<td>Owner</td>
</tr>
<tr>
<td>True owner</td>
</tr>
<tr>
<td>user that started the process</td>
</tr>
</tbody>
</table>

Inside a shell:

- several possible states: in background, foreground, stopped...
- job identifier (%1, %2, ...)
Process hierarchy

1 (init)

18696 (xterm)
18698 (bash)
17325 (emacs)

18706 (xterm)
18708 (bash)

15242 (firefox)

15213 (xterm)
15215 (bash)
22904 (ps)
Starting a program

Two steps:

1. Creation of a new process by cloning the father: copy everything except its identity (pid),
2. Mutation to execute the new program,

No other way

Cloning: use the `fork()` system call,

Mutation: use the system call `execve()` ; there is a family of similar functions `exec*()`
Cloning a process: the `fork()` system call

```c
#include <unistd.h>

pid_t fork(void);
```

- duplicates the current process,
- returns:
  - the pid of the child in the father’s process,
  - 0 in the created process (child),
  - −1 in case of error.
One example

```c
int main()
{
    pid_t status;
    printf("[%d] I will generate a process\n", getpid());
    status = fork();
    switch (status) {
    case -1 :
        perror("Process creation");
        exit(EXIT_FAILURE);
    case 0 :
        printf("[%d] I’m just born !\n", getpid());
        printf("[%d] My father is %d\n", getpid(), getppid());
        break;
    default:
        printf("[%d] I gave birth !\n", getpid());
        printf("[%d] My child is %d\n", getpid(), status);
    }
    printf("[%d] I’m now ending\n", getpid());
    exit(EXIT_SUCCESS);
}
```
Using it:

$ ./a.out
[18727] I will generate a process
[18728] I’m just born !
[18728] My father is 18727
[18728] I’m now ending
[18727] I gave birth !
[18727] My child is 18728
[18727] I’m now ending
Using it:

$ ./a.out
[18727] I will generate a process
[18728] I’m just born !
[18728] My father is 18727
[18728] I’m now ending
[18727] I gave birth !
[18727] My child is 18728
[18727] I’m now ending

father
process 18727

int status

fork()
Using it:

$ ./a.out
[18727] I will generate a process
[18728] I’m just born!
[18728] My father is 18727
[18728] I’m now ending
[18727] I gave birth!
[18727] My child is 18728
[18727] I’m now ending
Using it:

$ ./a.out

[18727] I will generate a process
[18728] I’m just born !
[18728] My father is 18727
[18728] I’m now ending
[18727] I gave birth !
[18727] My child is 18728
[18727] I’m now ending

father
process 18727

int status
18728

→

child
process 18728

int status 0

→

fork()
Using it:

$ ./a.out
[18727] I will generate a process
[18728] I’m just born !
[18728] My father is 18727
[18728] I’m now ending
[18727] I gave birth !
[18727] My child is 18728
[18727] I’m now ending

This could be:

$ ./a.out
[18727] I will generate a process
[18727] I gave birth !
[18727] My child is 18728
[18727] I’m now ending
[18728] I’m just born !
[18728] My father is 1
[18728] I’m now ending

• “race” between the process: the printing order is random,
• if the father ends before the child, it is adopted by init (1).
• Remark: the sleep() function can be useful to “force” the order.
Inheritance

- Memory of the child → copy of the father’s memory,
- Variables are not shared,
- Buffers of `<stdio.h>` are also copied! Empty them before using fork!
- Open descriptors are shared! (more details later on)
int main()
{
    pid_t status;
    printf("beginning ");
    status = fork();
    switch (status) {
    case -1 :
        perror("Process creation");
        exit(EXIT_FAILURE);
    case 0 :
        printf("[%d] child\n", getpid());
        break;
    default:
        printf("[%d] father\n", getpid());
    }
    printf("[%d] end\n", getpid());
    exit(EXIT_SUCCESS);
}
Ending children

- Use exit(),
- The father can know the returned value (success or not...)
- Waiting a child:

```c
#include <sys/wait.h>
pid_t wait(int *pstatus);
```

- returns the pid of the child,
- −1 if error (no child...),
- suspends the process if no child ends,
- *pstatus provides informations on the way the child ended.
Informations on the child ending

- stored in the integer status pointed by par pstatus:
  
<table>
<thead>
<tr>
<th>macro</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIFEXITED(status)</td>
<td>process used exit()</td>
</tr>
<tr>
<td>WIFSIGNALED(status)</td>
<td>process received a signal</td>
</tr>
<tr>
<td>WIFSTOPPED(status)</td>
<td>process has been stopped</td>
</tr>
</tbody>
</table>

- returned value: if WIFEXITED(status) only!
  
  - on 8 bits,
  
  - access with WEXITSTATUS(status)

- Signal that made the ending / stop of the process:
  
  - if WIFSIGNALED(status) or WIFSTOPPED(status),
  
  - access with WTERMSIG(status) or WSTOPSIG(status)
One example

int main(int argc, char *argv[])
{
    int status;
    pid_t pid, pidz;
    switch (pid = fork()) {
        case -1:
            perror("Process creation");
            exit(EXIT_FAILURE);
        case 0:
            printf("[%d] Child dying\n", getpid());
            exit(2);
        default:
            printf("[%d] Father created %d\n", getpid(), pid);
            pidz = wait(&status);
            if (WIFEXITED(status))
                printf("[%d] My child %d ended normally,\n" "[%d] returned value: %d\n",
                        getpid(), pidz,
                        getpid(), WEXITSTATUS(status));
            else
                printf("[%d] My child ended in a bad way\n", getpid());
            break;
    }
    exit(EXIT_SUCCESS);
}
Waiting a specific child

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *pstatus, int options);
```

- `pid` is the child we want to wait,
- `pid = -1` for any child,
- `options`: binary combination of `WNOHANG` (non blocking call) and `WUNTRACED` (stopped process)
Zombie process

- Process ended but not yet waited by its father,

- Avoid them: system keeps its formations for the father (memory usage),

- One solution: double fork (not seen here).
# Mutation of a process

```c
#include <unistd.h>
int execve(const char *filename,
            char *const argv[],
            char *const envp[]);
```

- Execute the program `filename`,
  - The process ends at the end of the program execution,
  - If no error, no return to the current program!

- Parameters `argv[]`
- Environment variables `envp[]`
- (similar to the `main()` function in C)
- Other functions (that use `execve` - make the call easier):
  - `execl`, `execv`, `execlp`, `execvp`. 

---

```
adrien.poteaux@lifl.fr
Process
38 / 55
```
How the shell starts a new process

$ command

```
$ commmand
```

![Diagram showing the process of how the shell starts a new process]

- `fork()`: Creates a new process
- `wait()`: Waits for a child process to exit
- `exec(command)`: Executes the command
- `exit()`: Exits the process

(email: adrien.poteaux@lifl.fr)
Executing a command in background

$ command &

sh (fpid) → fork() ← sh (fpid)

sh (cpid) → exec(command) ← command (cpid) → exit()

The shell does not wait the process anymore!
Conditional execution

$ \text{cmd1} \&\& \text{cmd2}$
Redirect of input/output

```c
int dup2(int oldfd, int newfd);

- makes newfd be the copy of oldfd,
- closes newfd first if necessary.

One example:
```
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

static void print_inode(const char *str, int fd)
{
    struct stat st;
    fstat(fd, &st);
    fprintf(stderr, "\t%s : inode %d\n", str, st.st_ino);
}
```
int main(int argc, char *argv[]) {
    int fdin, fdout;
    fdin = open(argv[1], O_RDONLY);
    fdout = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC,
                  S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    fprintf(stderr, "Before dup2() :
");
    print_inode("fdin", fdin);
    print_inode("fdout", fdout);
    print_inode("stdin", STDIN_FILENO);
    print_inode("stdout", STDOUT_FILENO);
    dup2(fdin, STDIN_FILENO);
    dup2(fdout, STDOUT_FILENO);
    fprintf(stderr, "After dup2() :
");
    print_inode("fdin", fdin);
    print_inode("fdout", fdout);
    print_inode("stdin", STDIN_FILENO);
    print_inode("stdout", STDOUT_FILENO);
    exit(EXIT_SUCCESS);
}

$ ./a.out
Before dup2() :
    fdin : inode -1567691942
    fdout : inode 30539837
    stdin : inode 8
    stdout : inode 8
After dup2() :
    fdin : inode -1567691942
    fdout : inode 30539837
    stdin : inode 8
    stdout : inode 30539837
How the shell works:

$ command > file
Signals: interruptions

- Sent to a process... by another process or the system.

- Dealing with signals: handlers (function called when receiving the signal)

- One example: shell managing jobs (bg, fg...).
Some signals

<table>
<thead>
<tr>
<th>name</th>
<th>event</th>
<th>behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>interruption &lt;intr&gt;, C-c</td>
<td>terminates</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>interruption &lt;quit&gt;, C-\</td>
<td>terminates + core</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>immediate ending</td>
<td>terminates (unchangeable)</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>ending</td>
<td>terminates</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>suspension &lt;susp&gt;, C-z</td>
<td>suspended</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>suspension signal</td>
<td>suspended (unchangeable)</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>a stopped process continues</td>
<td>restarts (if stopped !)</td>
</tr>
</tbody>
</table>
A process sends a signal to another:

```c
#include <signal.h>
int kill(pid_t pid, int sig);
```

- Rights needed,
- returns −1 in case of error,
- `sig= 0` means no signal (testing the pid).

Waiting a signal

```c
#include <unistd.h>
int pause(void);
```

- blocks the process until it receives a signal,
- then acts according to the sent signal.
Handling signals

- Handler executed by the signal target,
- function's prototype:
  ```c
  void handler(int signum);
  ```
- function's type:
  ```c
  void (*phandler)(int);
  ```
- Structure `struct sigaction`
  ```c
  struct sigaction {
      void (*sa_handler)();
      sigset_t sa_mask;
      int sa_flags;
  };
  ```
- Primitive `sigaction()`
  ```c
  # include <signal.h>
  int sigaction (int sig, const struct sigaction *new, struct sigaction *old);
  ```
One example

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#define NSIGMAX 5

static void set_default()
{
    struct sigaction sa;
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa,
              (struct sigaction *) 0);
}

static void int_handler(int sig)
{
    static int nsig = 0;
    if (nsig++ < NSIGMAX)
        printf(" C-c won’t kill me\n");
    else {
        printf(" unless you insist...\n");
        set_default();
    }
}

int main()
{
    struct sigaction sa;
    sa.sa_handler = int_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa,
              (struct sigaction *) 0);
    for(;;)
        pause();
    fprintf(stderr, "bye\n");
    exit(EXIT_SUCCESS);
}
```

$ ./a.out
^C C-c won’t kill me
^C C-c won’t kill me
^C C-c won’t kill me
^C C-c won’t kill me
^C C-c won’t kill me
^C unless you insist...
C $
Child ending

- Father receives the SIGCHLD signal when a child ends,
- By default: ignored
- Specific treatment:
  - eliminating a zombi process,
  - wait() or waitpid() call.
- Father also warned when a child is suspendend (also ignored by default).
Job manager

1. Command
2. Foreground work
3. Background work
4. Stopped work
5. Command &
6. Stop %job
7. SIGSTOP
8. Control-z
9. SIGSTOP
10. SIGINT
11. Control-c
12. SIGINT
13. Kill %job
14. Kill job
15. FG %job
16. Background work
17. End
18. Foreground work
19. End
20. Command
Communication between process: pipe

- Needed communication between process:
  - Two process cooperating,
  - Combining the results of two or more process.

- Several possible communications:
  - `exit()` & `wait()`: returned value only.
  - `kill()`: communicating a signal.
  - `write()` & `read()`: with an intermediate file.

- Another (better) way: pipes
  - Communication between reader(s) and writer(s),
  - No intermediate file necessary!

  ```
  cat file | more
  ```
Anonymous pipe

- One file without any name
  - not in the filesystem data,
  - cannot use open()

- Creating a pipe:
  ```
  # include <unistd.h>
  int pipe(int *fd)
  ```

- Creates two descriptors:
  - `fd[0]` for reading,
  - returns 0 (success) or -1 (error),
  - closing a descriptor ? no more access,
  - inheritance of descriptors after a fork()
Using pipes

- Typical use:
  - one process writes,
  - another reads.

- How to do that?
  - A pipe is created in a parent process,
  - Father forks to create a child,
  - Each process have access to the descriptors
  - Each uses one end of the pipe (thus closes the useless one)

```c
fork() != 0
fd[1]
/* close(fd[0]) */
```

```c
fork() == 0
fd[0]
/* close(fd[1]) */
```
One example: the shell

$ cmd1 | cmd2