Université Nice SOPHIA ANTIPOLIS

Programmation-système
La norme Posix.1

Jean-Paul Rigault

Polytech'Nice Sophia Antipolis

jpr@polytech.unice.fr

Adaptation au cours de 2009 / 2010
Erick Gallesio <eg@unice.fr>

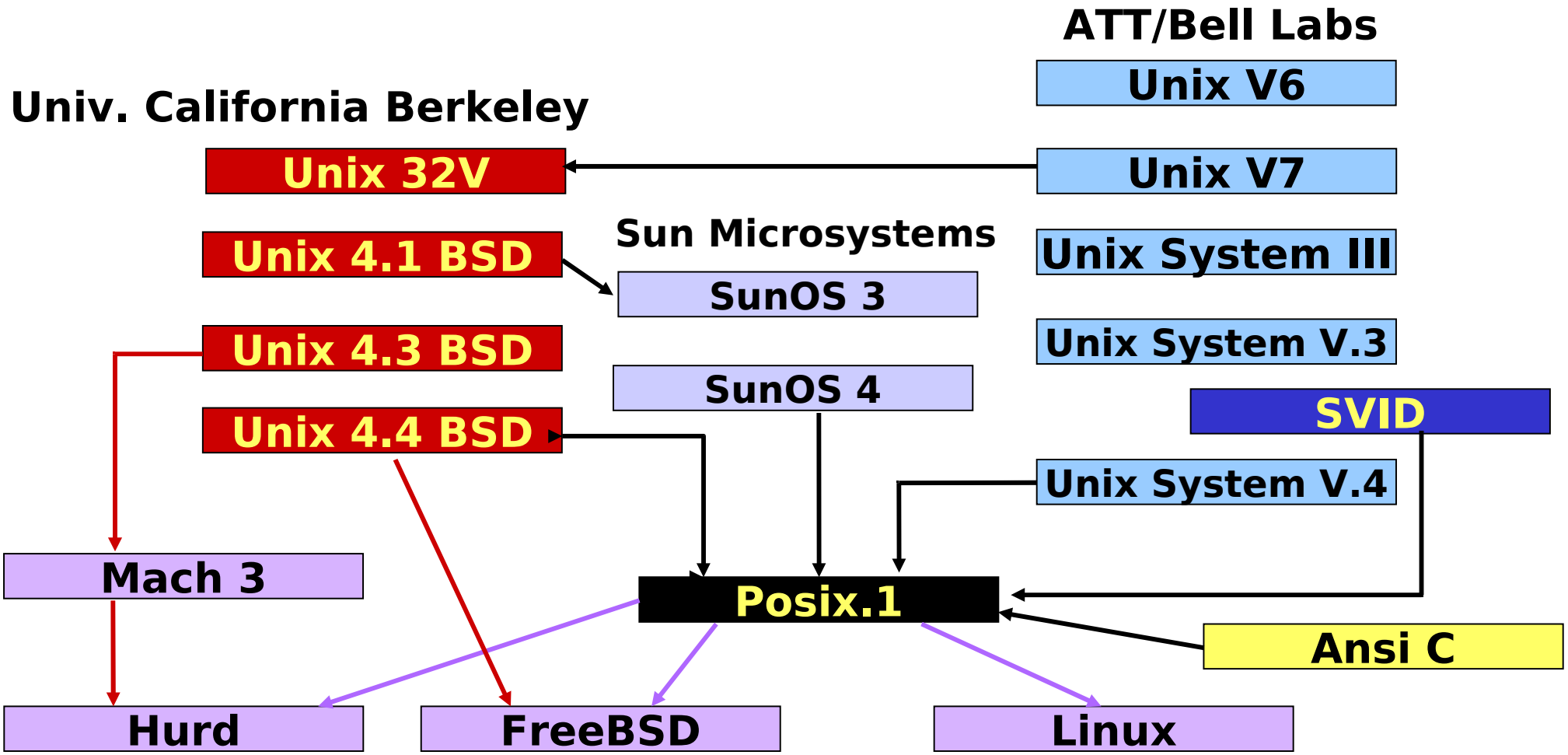
Plan

- Introduction
- Gestion d’entrées-sorties
- Gestion de processus
- Signaux
- Information à l’exécution
- Gestion du terminal
- Gestion du temps partagé :
  - `/etc/init`
  - shell
- Internationalisation et localisation

La norme Posix.1

Introduction

D'Unix à Posix
Petite histoire d'Unix



D’Unix à Posix
Genèse de Posix

- **Motivations**
  - API : Interface portable de programmation d’applications
  - Portabilité au niveau source
  - Compatibilité avec Ansi C
  - Unification des versions d’Unix et ouverture à d’autres systèmes d’exploitation

- **Déroulement des travaux**
  - Groupes de travail : /usr/group, IEEE 1003, Ansi
  - Normes : ISO 9945-1 (1990) ; révisions en 1996, 2001

D'Unix à Posix
Les normes Posix

- **Interface portable de programmation**
  - Posix.1, 1a/1b (extensions temps-réel), 1c (threads)
  - Shell et commandes de base (Posix.2)

- **Supports divers**
  - Réseaux
  - Langages (Fortran, Ada…)
  - Systèmes (bases de données, transactionnels…)
  - etc.

Posix.1-2001
Compatibilité Posix (1)

- **Posix et Unix**
  - Posix est fondé sur Unix BSD et Unix SVR4
  - La plupart des systèmes Unix actuels proposent la compatibilité Posix

- **Posix et les systèmes non-Unix**
  - Bibliothèques d'émulation
  - E.g., Cygwin

Posix.1-2001
Compatibilité Posix (2)

- **Posix et Ansi C**
  - Description première de l'API en Ansi C
  - Support d'autres langages prévu
  - L'API Posix est un **sur-ensemble** de la bibliothèque standard d'Ansi C

Posix.1-2001
L'API *(1)*

- **Ensemble de fonctions**
  - appels-système *(2)*
  - Bibliothèque *(3)*
  - fichiers d'entête

```
<assert.h> <dirent.h> <ctype.h> …
<sys/times.h> <sys/types.h> …
```

- **Compatibilité stricte**

```
gcc -ansi -pedantic -D_POSIX_SOURCE=1 …
```

Posix.1-2001
L'API *(2)*

- **Valeur de retour en cas d'erreur**
  - si **`int`** : -1
  - si pointeur : 0 (**`NULL`**)

- **Code de l'erreur (`EACCESS`, `E2BIG`...)**

```
extern int errno;
```

- **Messages d'erreur**

```
#include <stdio.h>
void perror(const char *my_message);

#include <string.h>
char *strerror(int errnum);
```

Posix.1-2001
L'API (3)

```
#include <...>
.....
int main()
{
  char *p, path[2];  /* un peut court probablement */

  p = getcwd(path, 2);
  if (!p) {
    fprintf(stderr, "errno=%d  Message='%s'\n", errno, strerror(errno));
    /* pas de sortie ici */
  }

  if (open("foo", O_RDONLY) < 0) {
    fprintf(stderr, "Valeur de errno: %d\n", errno);
    perror("just-a-test");
    exit(EXIT_FAILURE);
  }

  fprintf(stderr, "Fin de just-a-test\n");
  return EXIT_SUCCESS;
}
```

A l'exécution:

```
$ just-a-test
errno=34  Message='Numerical result out of range'
Valeur de errno: 2
just-a-test: No such file or directory
```

La norme Posix.1

Gestion d’entrées-sorties

Fichiers et répertoires
Modèle

- **Modèle hiérarchique**

- **Types de fichiers**
  - ordinaire
    - données textuelles, binaires…
  - répertoire
  - spéciaux (périphériques)
    - modes bloc et caractères (voir dans **`/dev`**)
  - fichier FIFO (« pipe nommé »)
  - communication (sockets…)
  - etc.

Fichiers et répertoires
Répertoire courant

```
#include <unistd.h>
```

- **Changement de répertoire**

```
int chdir(const char *path);
```

- **Consultation du répertoire courant**

```
char *getcwd(char *dirname,
             size_t size);
```

  - **`size`** est la taille du tableau pointé par **`dirname`** (et qui a dû être alloué par l'appelant)

Fichiers et répertoires
Consultation des attributs d'un fichier (1)

- **La fonction `stat`**

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
```

- **Structure `stat`**

`st_dev, st_ino`	**périphérique et numéro du fichier**
`st_mode`	**type et droits d'accès**
`st_nlink`	**nombre de liens**
`st_uid, st_gid`	**propriétaire et son groupe**
`st_size`	**taille du fichier (nombre de caractères)**
`st_atime`	**date du dernier accès,**
`st_mtime`	**date de la dernière modification,**
`st_ctime`	**date du dernier changement d'attributs**

Fichiers et répertoires
Consultation des attributs d'un fichier *(2)*

- **Macros de décodage de `st_mode`**
  - Sélection des champs

    `S_IRWX[UGO]` `S_IS[UG]ID`
  - Décodage des droits d'accès

    `S_I[RWX](USR|GRP|OTH)`
  - Type du fichier (argument `m` : `st_mode`)

    `S_ISCHR(m)` `S_ISBLK(m)`

    `S_ISDIR(m)` `S_ISREG(m)`

    `S_ISFIFO(m)`

Fichiers et répertoires
Consultation des attributs d'un fichier *(3)*

- **Exemple**

```
struct stat sbuf;
char *path = "foo/bar";

if (stat(path, &sbuf) >= 0) {
   int m = sbuf.st_mode;
   if (S_ISREG(m)) {
      /* le fichier est un fichier ordinaire */ …
      if (m & (S_IWUSR | S_IWGRP)) {
         /* le fichier est inscriptible par le
                  propriétaire ou son groupe */ …
      }
   }
}
```

Fichiers et répertoires
Lecture des répertoires (1)

- **Fonctions de lecture**

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *dirpath);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
int rewinddir(DIR *dirp);
```

- **Champs de `struct dirent`**
  - un seul champ portable, **`d_name`**, tableau de caractères de dimension **`NAME_MAX`**

Fichiers et répertoires
Lecture des répertoires (2)

- **Exemple : une version rustique de ls**

```
struct dirent *dentry;

DIR *dirp = opendir("foo");
if (dirp == NULL) {
  fprintf(stderr, "répertoire inaccessible\n");
  exit(EXIT_FAILURE);
}
while((dentry = readdir(dirp)) != 0) {
  printf("%s\n", dentry->d_name);
}
closedir(dirp);
```

Fichiers et répertoires
Manipulations de répertoires *(1)*

- **Création/destruction de répertoires**

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
int rmdir(const char *path)
```

  - **mode** ne doit contenir que des droits d'accès
  - on ne peut détruire qu'un répertoire vide

- **Exemple**

```
mkdir("foo", S_IRUSR|S_IWUSR|S_IRGRP);
```

Fichiers et répertoires
Manipulations de répertoires *(2)*

- **Création et destruction d'entrées dans un répertoire**

```
#include <unistd.h>

int link(const char *oldpath,const char *newpath);

int unlink(const char *path);

int rename(const char *oldpath, const char *newpath);
```

Entrées-sorties standards
Bibliothèque Ansi C

- **Posix.1 contient l'ensemble de la bibliothèque d'entrée-sortie standard de C (`<stdio.h>`)**

- **Les E/S on lieu avec « bufferisation »**
  - sauf si le support est un terminal
  - la fonction `fflush(FILE *)` vide le buffer

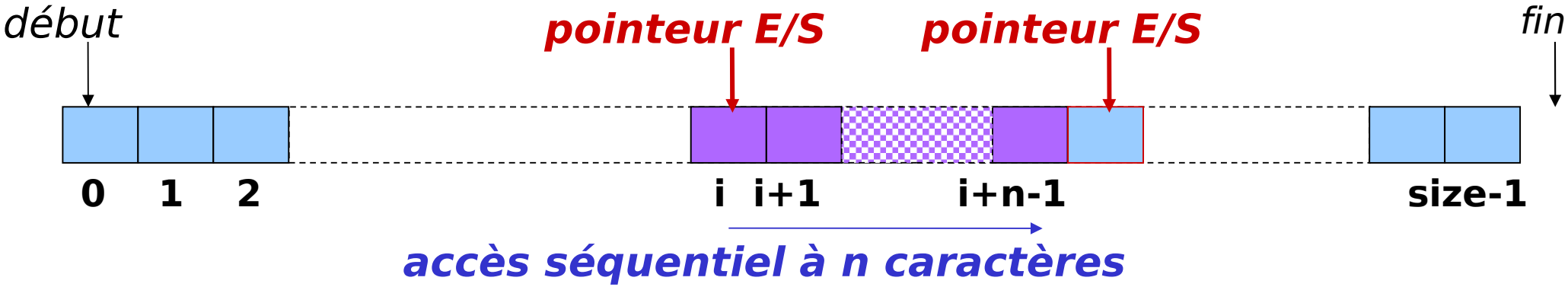
- **En fait ces fonctions d'E/S sont réalisées avec des fonctions primitives qui constituent l'API d'E/S de base de Posix**

Entrées-sorties de base
Descripteurs de fichiers

- **Un « file descriptor », `fd`, est un numéro d'unité logique qui permet de référencer un fichier**
- **`fd` est un entier positif ou nul**
  - 0, 1 et 2 correspondent respectivement à l'entrée, la sortie et l'erreur standard
- **Les « file descriptors » sont alloués par processus**
- **Les fonctions qui retournent un descripteur de fichier (e.g., `open`) retournent en général la plus petite valeur disponible dans le processus courant**

Entrées-sorties de base
Modèle de base des fichiers

- **Tableau de caractères**
  - indexé à partir de 0
- **Pointeur d'E/S**
  - index courant
  - manipulable directement (accès direct)
- **Lecture/écriture séquentielle**
  - à partir de l'index courant



Entrées-sorties de base
Ouverture et création de fichiers *(1)*

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

int open(const char *path, int oflag [, mode_t mode]);
```

- Flag **oflag**

O_RDONLY, O_WRONLY, O_RDWR	**mode d'ouverture**
O_APPEND	**place le pointeur d'E/S en fin de fichier**
O_CREAT	**crée le fichier s'il n'existe pas**
O_EXCL	**si O_CREAT et si le fichier existe, erreur**
O_TRUNC	**tronquer le fichier s'il existe**

etc.

Entrées-sorties de base
Ouverture et création de fichiers *(2)*

- **Troisième argument de `open` (`mode`)**
  - utilisé seulement si `O_CREAT`
  - positionne les droits d'accès au nouveau fichier
    - filtrage par le **`UMASK`**

- **Exemple**

```
if (open("foo", O_WRONLY|O_CREAT|O_TRUNC,
          S_IRUSR|S_IWUSR|S_IRGRP) < 0) {  /* rw- r-- --- */
  perror("open");
}
```

Entrées-sorties de base
Masque de création de fichiers (`cmask`)

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mode);
```

- **`mode`** ne doit contenir que des droits d'accès
- la fonction retourne la valeur précédente du masque
- mode contient les droits que l'on veut ***dénier***
  après `fd = open("foo",…|O_CREAT, m);`
  le mode est : `m & ~cmask`
- le masque est un attribut du processus

- **Exemple**

```
oldmask = umask(S_IWGRP|S_IWOTHR);
```

Entrées-sorties de base
Lecture/écriture séquentielles *(1)*

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t n);
ssize_t write(int fd, void *buf, size_t n);
```

- **E/S séquentielles**
  - Le pointeur d ’E/S est avancé du nombre de caractères effectivement transférés
  - Par défaut, **`read()`** est bloquant ; **`write()`** peut l’être
- **Valeur de retour**
  - Nombre de caractères effectivement transférés
  - Pour **`read()`**, valeur de retour 0 ⇒ fin de fichier

Entrées-sorties de base
Lecture/écriture séquentielles *(2)*

- **Exemple : une version rustique et partielle de copie de fichiers**

```
   char buffer[MAX];
int n;
int fdin  = open("foo", O_RDONLY|O_EXCL);
int fdout = open("bar", O_WRONLY|O_TRUNC|O_CREAT, …);

while((n = read(fdin, buffer, MAX)) > 0)
     write(fdout, buffer, n);

if (n == 0) /* eof */
else        /* n == -1 => erreur */
```

Entrées-sorties de base
Accès direct *(1)*

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- **Manipulation directe du pointeur d'E/S**
  - Nouvelle position : *Orig*(**`whence`**) + **`offset`**
    - *Orig* **`(SEEK_SET)`** début du fichier (0)
    - *Orig* **`(SEEK_CUR)`** position courante
    - *Orig* **`(SEEK_EOF)`** fin de fichier
  - Valeur de retour : nouvelle position absolue (i.e., depuis début du fichier)

Entrées-sorties de base
Accès direct *(2)*

- **Exemples**

```
off_t off = lseek(fd, 0, SEEK_CUR);
```

  - Retourne la position absolue courante, sans la modifier (**`ltell(fd)`**)

```
lseek(fd, 0, SEEK_SET);
```

  - Retourne au début du fichier (**`rewind(fd)`**)

```
off_t off = lseek(fd, 100, SEEK_EOF);
```

  - Ajoute 100 caractères (nuls) après la fin de fichier
  - Le fichier doit avoir été ouvert en écriture

Entrées-sorties de base
Accès direct *(3)*

- **Exemple : accès direct à une structure de données**

```
struct Data { … } buf;
int i = … ;

/* accès au ième élément et lecture */
lseek(fd, i*sizeof(struct Data), SEEK_SET);
read(fd, &buf, sizeof(struct Data));
```

Entrées-sorties de base
Mélange avec la bibliothèque Ansi C

- **Descripteur de fichier → `FILE *`**

```
#include <stdio.h>
int fd;
FILE *fp = fdopen(fd, type);
```

  - voir **`fopen()`** pour la signification de **`type`**

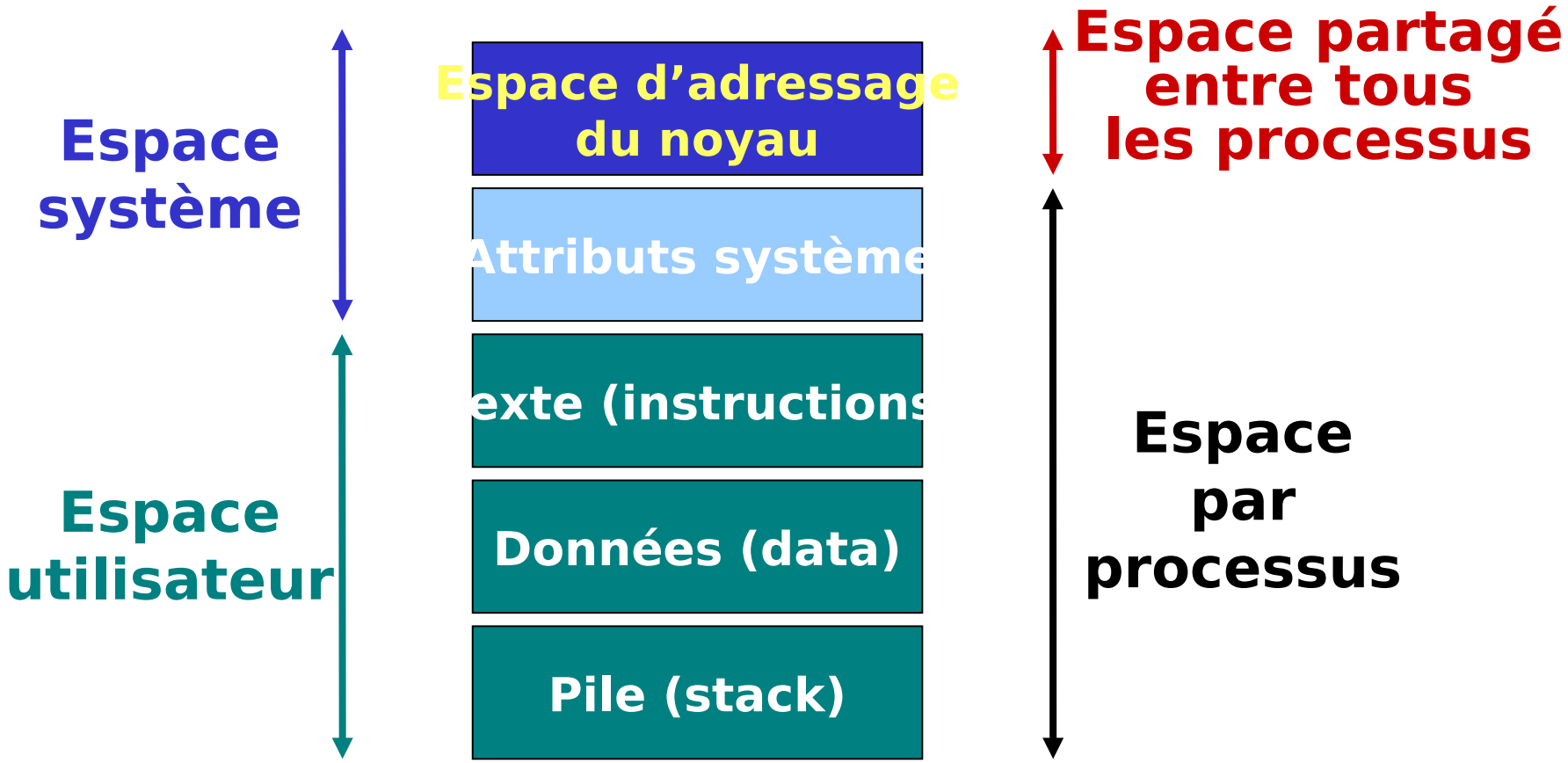
- **`FILE *` → descripteur de fichier**

```
#include <stdio.h>
FILE *fp;
int fd = fileno(fp);
```

La norme Posix.1

Gestion de processus

Espace d’adressage d’un processus (1)

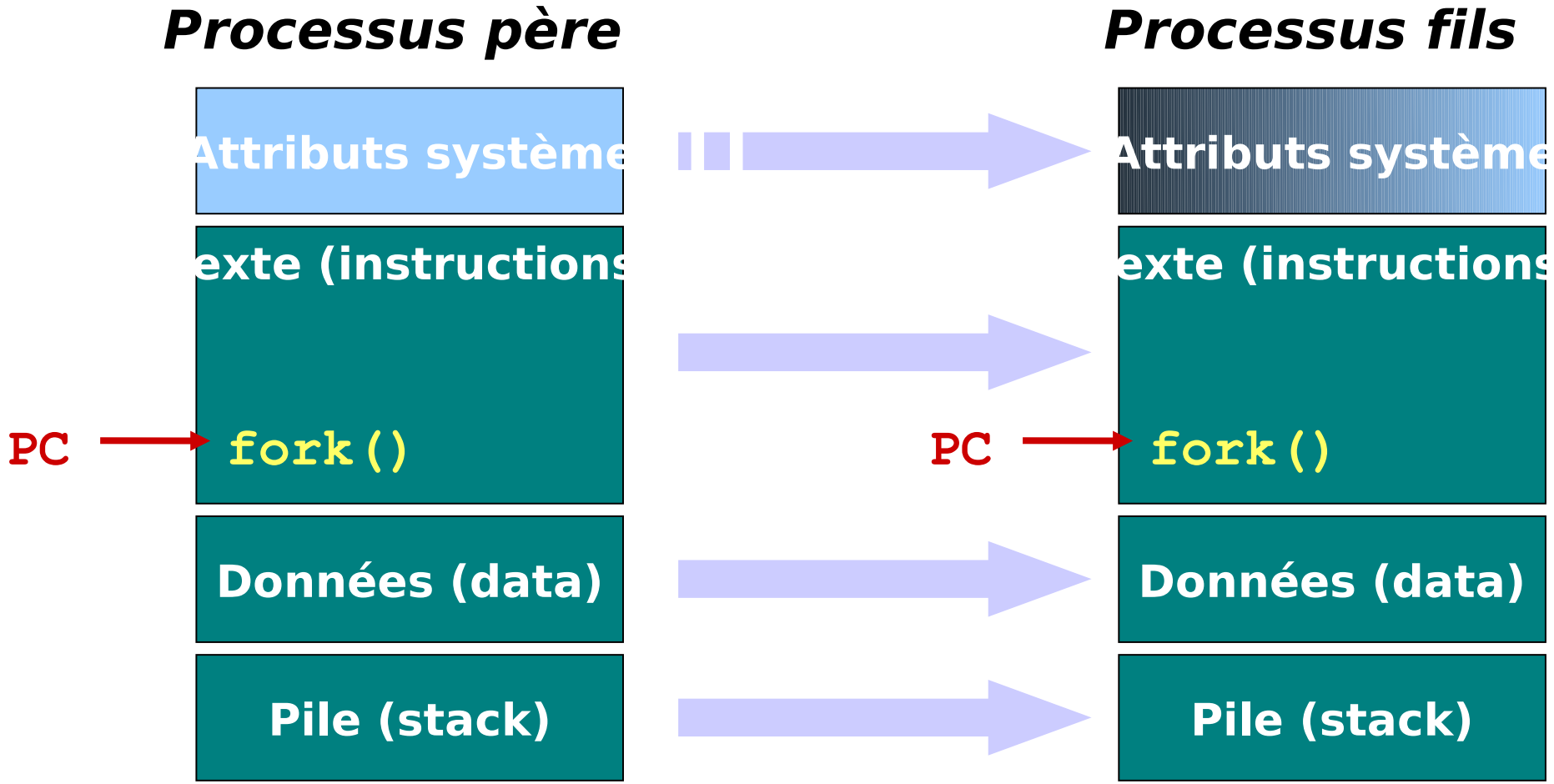


Espace d'adressage d'un processus (2)

- **Attributs système d'un processus**
  - identification (**`pid`**) : unique à un instant donné
  - **`uid`**, **`gid`** effectifs et réels
  - descripteurs de fichiers ouverts
  - racine et répertoire courants
  - états des signaux
  - masque de création des fichiers (**`cmask`**)
  - *adresses (mémoire, disque), information de gestion de mémoire virtuelle, priorité, etc.*

Création d'un processus

fork() (1)



Création d'un processus
fork() (2)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

- **Création de processus par « clonage »**
  - Duplication des segments de texte, de données, de piles *et de la plupart* des attributs système

- **Après `fork()`, les deux processus exécutent le même programme, mais *indépendamment***

Création d'un processus
fork() (3)

- **`fork()` est donc appelé une fois mais a deux retours**
  - un dans le fils, avec la valeur 0
  - un dans le père avec comme valeur le pid du fils

- **Héritage des attributs système**
  - (descripteurs de) fichiers ouverts
    - **le pointeur d'E/S est partagé entre le père et le fils**
  - **`uid`**, **`gid`**, répertoire courant, terminal de contrôle, masque de création, état des signaux, etc.

Création d'un processus
fork() (4)

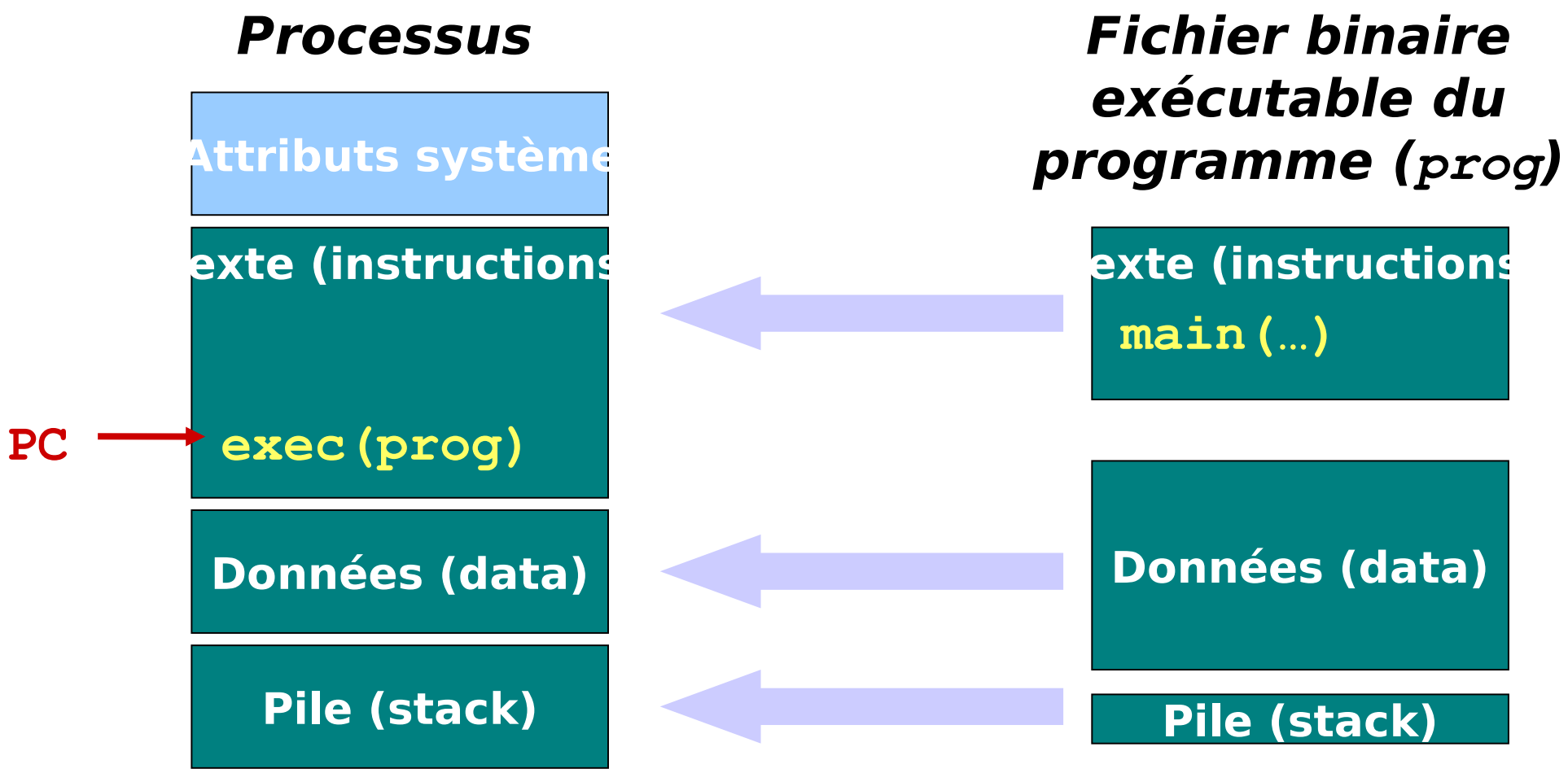
```
switch (fork()) {
case -1 :
      perror("fork");
      exit(1);
case 0 :
      printf("le fils\n");
      break;
default :
      printf("le père\n");
      break;
}
printf("père+fils\n");
```

```
% test-fork
père
fils
père+fils
père+fils
%
```

- **L'ordre d'exécution entre le père et le fils peut être quelconque**

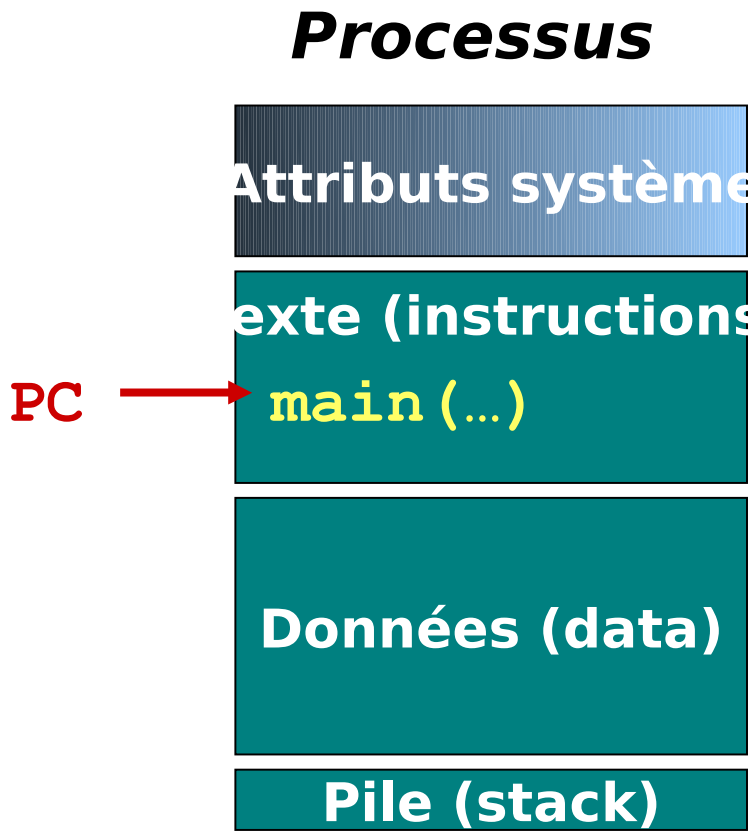
Association programme/processus

exec() *(1)*



Association programme/processus

exec() (2)



- **Le pid du processus n'a pas changé**
  - c'est le même processus
- **Le code a changé**
  - il exécute un autre programme
  - ce programme démarre au début (`main()`)
- **L'état de l'ancien programme est oublié**
  - on ne peut revenir d'un `exec` réussi !

Association programme/processus
exec() (3)

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., NULL);
int execv(const char *path, char *const argv[]);

int execle(const char *path, const char *arg0,..., NULL,
           char **envp);
int execve(const char *path, char *const argv[], char **envp);

int execlp(const char *path, const char *arg0,... , NULL);
int execvp(const char *path, char *const argv[]);
```

Association programme/processus
exec() *(4)*

- **Remplacement des segments d'un processus par ceux d'un programme pris dans leur état initial**

- **Argument**
  - Le fichier à exécuter (**`path`**)
  - les arguments du **`main`**
    - **`argv[0]`** est le nom (de base) du fichier à exécuter
    - La liste (ou le tableau **`argv[]`**) se termine avec un pointeur **`NULL`**

Association programme/processus
exec() (5)

- **exec[lv]p utilisent la variable PATH**
- **exec[lv]e passent l'environnement en dernier paramètre**
- **Conservation de la plupart des attributs système**
  - (descripteurs) de fichiers ouverts
    - avec la même valeur du pointeur d'E/S qu'avant **exec()**
  - **uid**, **gid**, répertoire courant, terminal de contrôle, masque de création, certains états des signaux, etc.

Association programme/processus

exec() *(6)*

```
printf("début\n");
execlp("ls", "ls", "-l", "-R","/usr", NULL);
/* On ne passe ici qu'en cas d'erreur de exec */
perror("exec");
```

Shell-like example

```
execlp("foo.sh", "./foo.sh", NULL);
/* Si l'éxécution échoue, on essaie avec /bin/sh */
execlp("/bin/sh", "sh", "-c", "./foo.sh", NULL);
```

Terminaison volontaire d'un processus
exit() (1)

```
#include <stdlib.h>
void exit(int status);
void abort();

#include <unistd.h>
void _exit(int status);
```

- Toutes ces fonctions terminent le processus courant
- **_exit()** et **exit()** transmettent le code de retour **status** au processus père
- **abort()** produit un fichier **core** (signal **SIGABRT**)

Terminaison volontaire d'un processus
exit() (2)

- **Terminaison normale : `exit()`**
  - appelle les fonctions enregistrées par **`atexit()`**
  - « flushe » tous les fichiers de **stdio**
  - détruit les fichiers temporaires (**`tempfile()`**)
  - appelle **`_exit()`**

- **Terminaison forcée : `_exit()`**
  - ferme tous les fichiers et répertoires
  - *réveille le processus père (si nécessaire)*
  - *provoque éventuellement l'adoption du processus, etc.*

Attente d'un processus fils
wait() (1)

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

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

- **Attente de la terminaison d'un fils**
  - **`wait()`** est réveillé par la fin d'un fils quelconque
  - **`waitpid()`** est réveillé par la fin du fils indiqué
- **Retour immédiat si un/le fils déjà terminé**

Attente d'un processus fils
wait() (2)

- **Exemple**

```
while (....) {
   ....
   .... fork() ....
   ....
}

/* Attendre la fin de tous les fils */
while (wait(NULL) != -1) /* ne rien faire */;

printf("Tous les fils sont terminés\n");
```

Version simplifiée de `system()` (1)

- **Exemple d'utilisation**

```
int status;

printf("avant system\n");
status = system("ls -la -R /usr > foo");
printf("après system %d\n", status);
```

Version simplifiée de `system()` (2)

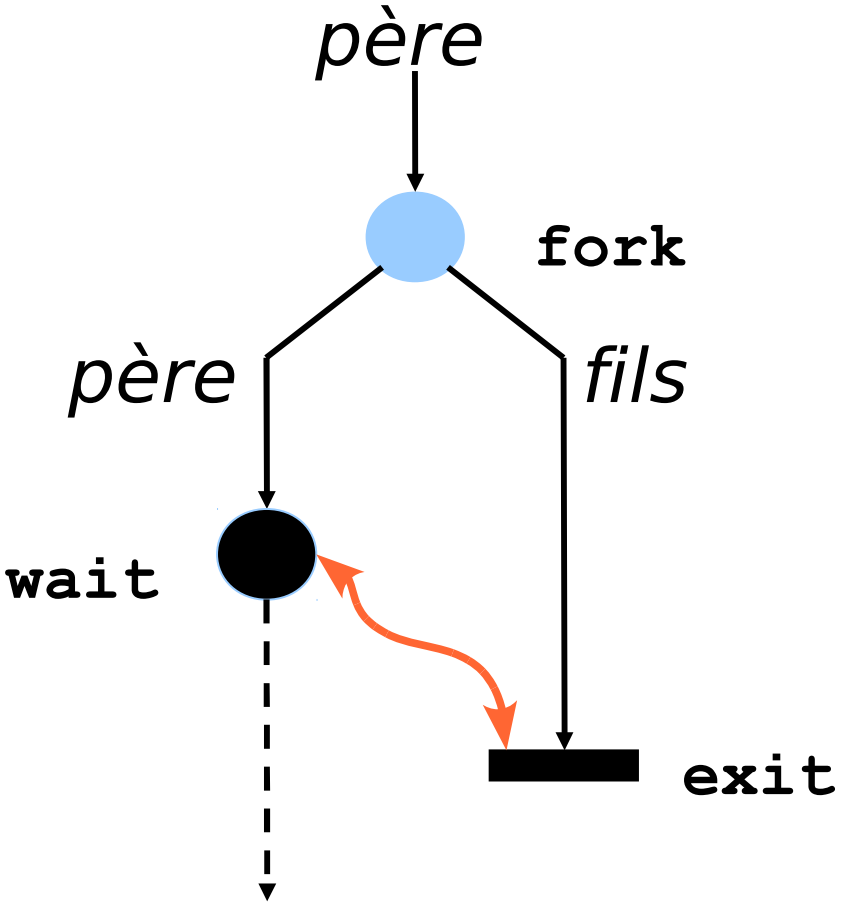
```
#define BAD 1

int system(const char *cmd) {
        int status;

        if (fork()) {
            wait(&status);
            return status;
        } else {
            execl("/bin/sh", "sh", "-c", cmd, NULL);
            perror("exec");
            return BAD;    /* PROBLEME */
        }
}
```

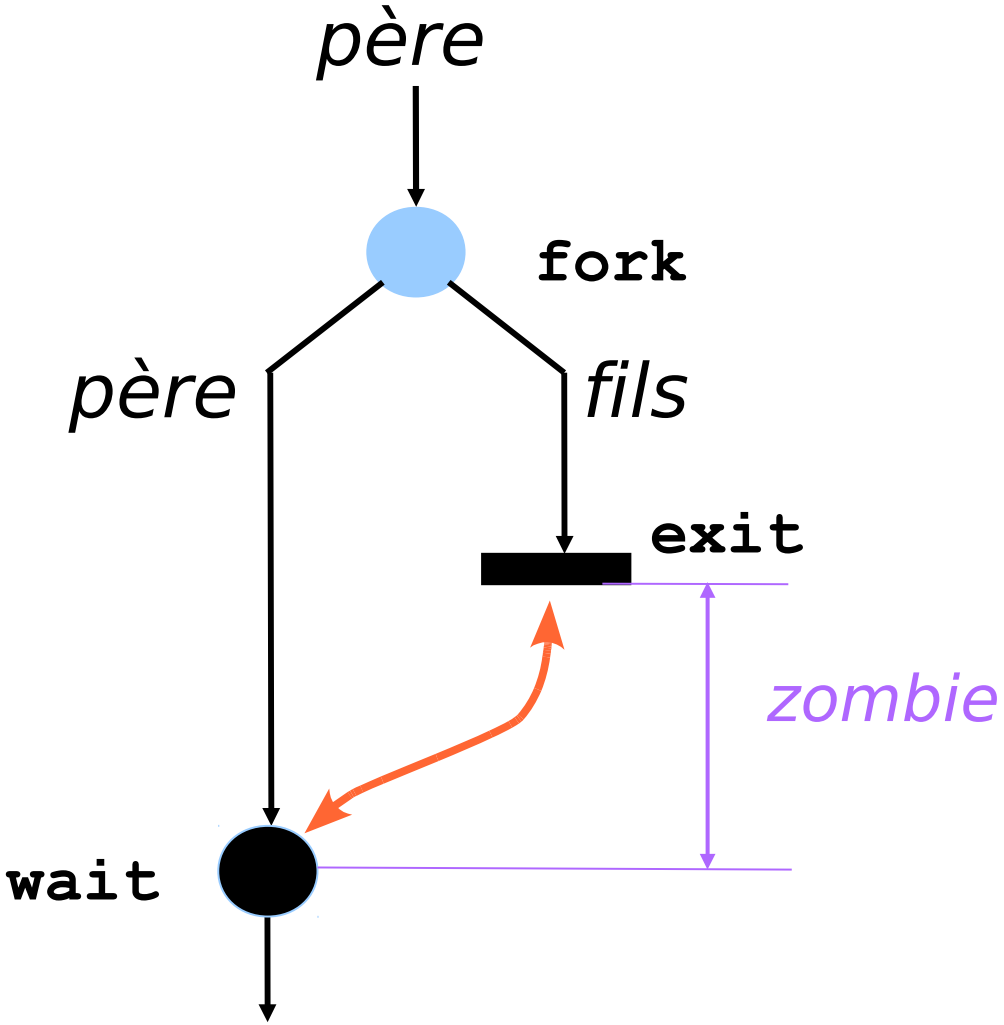
Version simplifiée de `system()` (3)

```
#define BAD 1

int system(const char *cmd) {
      int status;

      if (fork()) {
          wait(&status);
          return status;
      } else {
          execl("/bin/sh", "sh", "-c", cmd, NULL);
          perror("exec");
          exit(BAD);
      }
}
```

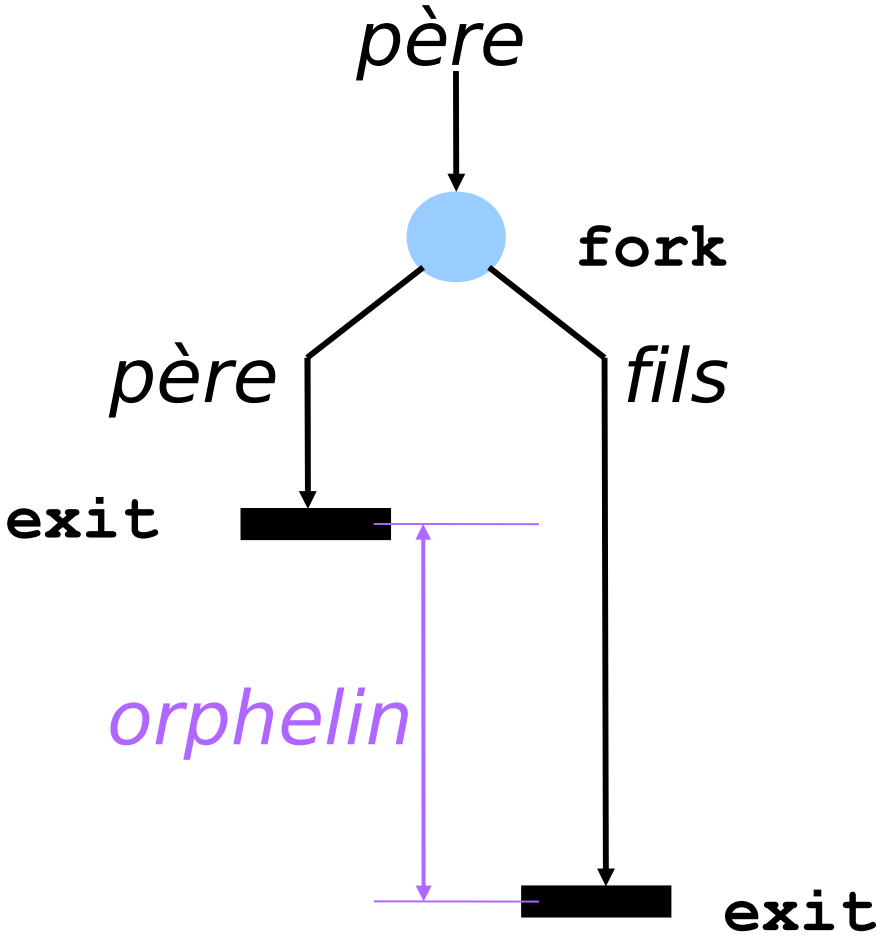
Processus père et fils : « zombie » et orphelin

(1)





Processus père et fils : « zombie » et orphelin (2)



- **Les orphelins sont adoptés par le processus de pid 1**
- **Ce processus 1 est associé au programme `/etc/init`**
- **Ce processus est aussi le gestionnaire du temps partagé**

Redirection des flux d'E/S
Contrôle des opérations sur fichier

```
#include <unistd.h>
#include <fcntl>
int fcntl(int fd, int cmd, ...);
```

- Le nombre de paramètres dépend de **cmd**

- Commandes (**cmd**)
  - **F_DUPFD** duplique le descripteur (cf diapo suivante)
  - **F_GETFL** consulte indicateurs de **open()**
  - **F_GETLK** consulte état verrouillage
  - **F_SETFL** modifie indicateurs de **open()**
  - **F_SETLK** ou **F_SETLWK** modifie état verrouillage
  - etc.

Redirection des flux d'E/S
Duplication de descripteurs (1)

- **Duplication de descripteur**

```
int newfd = fcntl(fd, F_DUPFD, minfd);
```

  - **fd** et **newfd** sont *synonymes* ; ils désignent le même fichier, avec le même pointeur d'E/S

- **Fonctions spéciales de duplication**

```
int newfd = dup(fd);
```

  - équivalent à `newfd = fcntl(fd, F_DUPFD, 0);`

```
int newfd = dup2(fd, desiredfd);
```

  - équivalent à

```
close(desiredfd);
newfd = fcntl(fd, F_DUPFD, desiredfd);
```

Redirection des flux d'E/S
Duplication de descripteurs (2)

- **Exemple de duplication de descripteur : redirections du shell**

```
% ls > foo
```

```
int fd = open("foo",
              O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
dup2(fd, 1);
close(fd);
execlp("ls", "ls", NULL);
```

Communication de données entre processus
Tube : `pipe()` *(1)*

```
#include <unistd.h>
int pipe(int fds[2]);
```

- Flot ***non structuré*** de caractères
- Gestion FIFO
- Synchronisation producteur/consommateur
  - **`write()`** peut être bloquant (**`read()`** aussi, bien sûr !)

Communication de données entre processus
Tube : `pipe()` (2)

- **Plusieurs processus peuvent se partager les deux extrémités du pipe**
  - Atomicité des lectures et écritures de moins de **`PIPE_BUF`** caractères
  - Aucune structure des E/S n'est conservée dans le pipe

- **Conditions aux limites**
  - Si **aucun processus** n'a le pipe ouvert en écriture, une tentative de lecture reçoit une fin de fichier (**`EOF`**)
  - *Si **aucun processus** n'a le pipe ouvert en lecture, un processus tentant d'écrire recevra le signal **`SIGPIPE`** (et, par défaut, se terminera)*

Communication de données entre processus
Tube : pipe() (3)

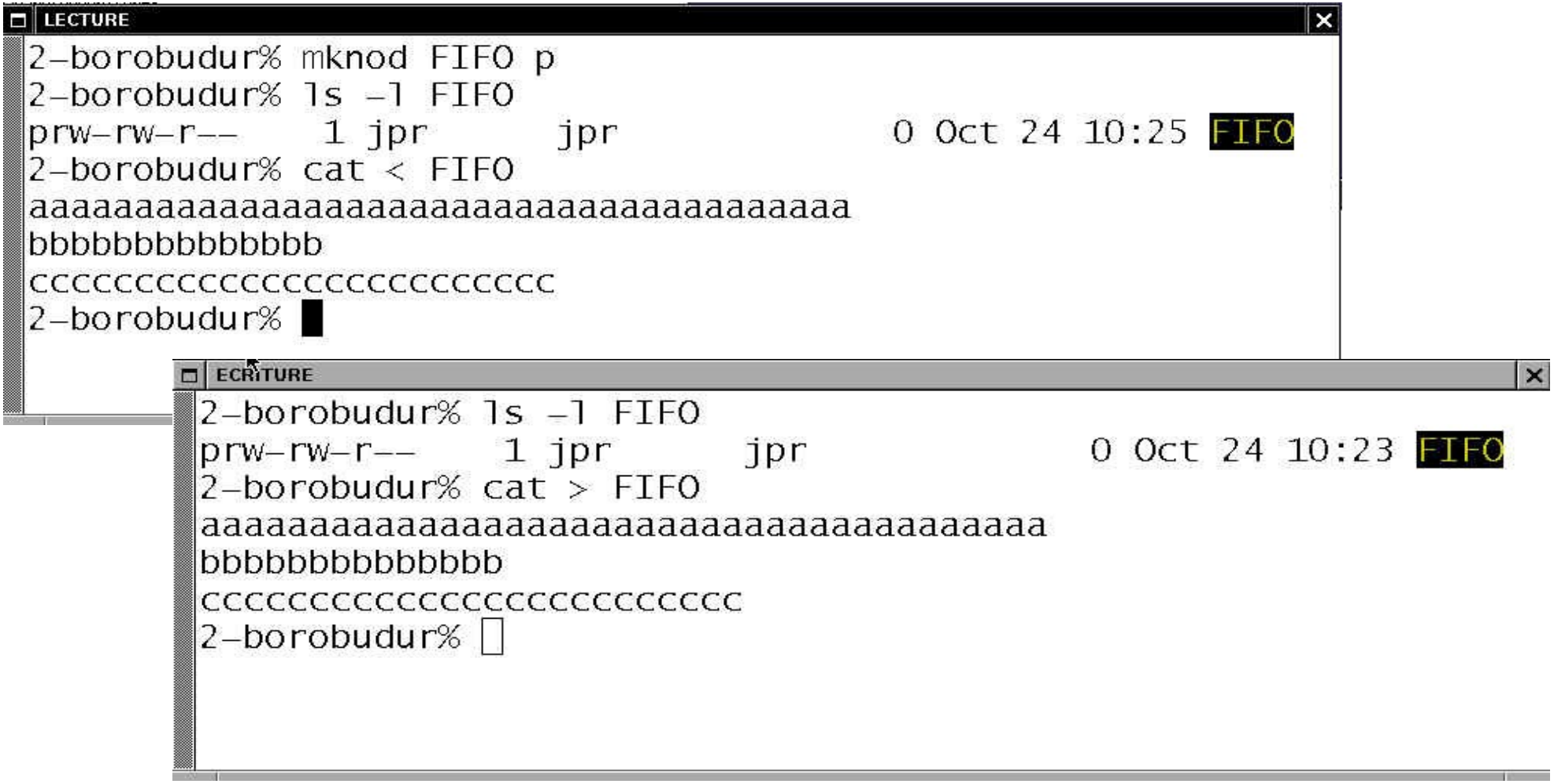
```
#define MAXL 100
char Msg[MAXL];
. . .
int fds[2];
pipe(fds);
if (fork()) {
    close(fds[0]);
    write(fds[1], "Salut !", sizeof("Salut !");
    . . .
} else {
    close(fds[1]);
    read(fds[0], Msg, MAXL);
    printf("%s\n", Msg);
    . . .
}
```

Communication de données entre processus
Pipe nommé (fichier fifo) *(1)*

- **Inconvénient des pipes**
  - Le pipe doit être créé par un ancêtre commun aux processus communiquants
    - Les descripteurs correspondants sont hérités lors des **`fork()`**
- **Pipe nommé (fichier fifo)**
  - Structure et synchronisation semblable au pipe
  - Désigné par un nom de fichier

```
#include <sys/types.h>
#include <sys/stat.h.h>
int mkfifo(const char *path, mode_t mode);
```

Communication de données entre processus
Pipe nommé (fichier fifo) *(2)*



La norme Posix.1

Informations à l’exécution

Identification des processus

```
#include <sys/types.h>
#include <unistd.h>
```

- **Processus courant**

```
pid_t getpid();
```

- **Processus père**

```
pid_t getppid();
```

- **Groupe de processus**

```
int setpgrp();
pid_t getgrp();
```

Information sur les utilisateurs (1)

- **Noms de login**

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

  - Champs de struct passwd (voir **/etc/passwd**)

```
pw_name pw_uid pw_gid pw_dir pw_shell
```

- **Groupes d'utilisateurs**

```
#include <gid.h>
struct group *getgrgid(gid_t gid);
struct passwd *getgrnam(const char *name);
```

  - Champs de **struct group** (voir **/etc/group**)

```
gr_name gr_gid gr_mem
```

Information sur les utilisateurs (2)

- **Liste des groupes d'un utilisateur**

```
#include <sys/types.h>
#include <unistd.h>
int getgroups(int size, gid_t grlist[]);
```

Date et heure (1)

```
#include <time.h>
```

- **Mesure du temps**
  - `time_t` : secondes depuis le 01/01/1970, 00:00 UTC
  - `struct tm` : date éclatée en jour, mois, année, heure, etc.
- **Temps « absolu »**

```
time_t time(time_t *tp);
time_t mktime(struct tm *timep);
```

- **Transformation en temps local ou GMT**

```
struct tm localtime(time_t *tp);
struct tm gmtime(time_t *tp);
```

Date et heure (2)

- **Formattage des dates**

```
size_t strftime(char *buffer, size_t sz,
                const char *format,
                const struct tm *timep);
char *asctime(const struct tm *timep);
char *ctime(const time_t *tp);
```

- **Différence de dates**

```
double difftime(time_t t1, time_t t2);
```

Date et heure (3)

- **Temps d'exécution des processus**
  - Exprimé en « clock ticks » : **`clock_t`**
    - À diviser par **`CLOCKS_PER_SEC`** pour obtenir le nombre de secondes
  - Temps CPU total utilisé

```
clock_t clock();
```

  - Temps CPU utilisateur et système, pour le processus lui-même et pour ses enfants

```
clock_t times(struct tms *buf);
```

  - Voir aussi la commande **time** du shell

Divers

- **Identification du système**

```
#include <sys/utsname.h>
int uname(struct utsname *name);
```

- **Valeurs locales des limites d'implémentation (<limits.h>…)**

```
#include <unistd.h>
long sysconf(int pname);
long pathconf(const char *path, int pname);
long fpathconf(int fd, int pname);
```

La norme Posix.1

Gestion de signaux

Signaux
Définition, état, action associée *(1)*

- **Notion de signal**
  - Evénement asynchrone
    - interruption terminal (`^c`, `^z`, `^\`)
    - terminaison d'un processus fils...
  - Evénement synchrone (exception)
    - erreur arithmétique (division par 0)
    - violation de protection mémoire...

Signaux
Définition, état, action associée *(2)*

- **État d'un signal ; action associée**
  - Ignoré (et donc perdu !)
  - Associé à son action par défaut
    - dépend du signal (rien, suspension, reprise, terminaison avec ou sans core…)
  - Associé à une action définie par l'utilisateur (piégé, capturé, « trappé »)
    - « handler » de signal (fonction utilisateur)
- **Signal différé (masqué, bloqué)**
  - Le signal est mémorisé
  - L'action sera effectuée lors du déblocage (démasquage)

Signaux
Liste des signaux de Posix.1 *(1)*

Signal	Signification	core	Action déf.	Remarque
SIGABRT	abort	✓	fin	
SIGALRM	alarme (time-out)		---	
SIGFPE	exception calcul flottant	✓	fin	exception
SIGHUP	coupure ligne terminal		fin	
SIGILL	instruction illégale	✓	fin	exception
SIGINT	interruption		fin	^c au terminal
SIGKILL	terminaison forcée		fin	ni ignorable, ni piégeable, ni blocable
SIGPIPE	écriture sur pipe sans lecteur		fin	exception
SIGQUIT	interruption	✓	fin	^\ au terminal
SIGSEGV	violation mémoire	✓	fin	exception

Signaux
Liste des signaux de Posix.1 *(2)*

Signal	Signification	core	Action déf.	Remarque
`SIGTERM`	terminaison normale		**fin**	
`SIGUSR1`	signal utilisateur		**---**	
`SIGUSR2`	signal utilisateur		**---**	
`SIGCHLD`	changement état d'un fils		**---**	
`SIGSTOP`	suspension forcée		**suspension**	ni ignorable, ni piégeable, ni blocable
`SIGCONT`	reprise		**reprise**	(**fg** et **bg**)
`SIGTSTP`	suspension douce		**suspension**	`^z` au terminal
`SIGTTIN`	lecture terminal depuis un processus background		**suspension**	exception
`SIGTTOUT`	écriture terminal depuis un processus background		**suspension**	exception

Signaux
Liste des signaux de Posix.1 *(3)*

- **Aucune priorité entre les différents signaux**
- **L'ordre de délivrance de plusieurs signaux « simultanés » n'est pas garantie**

Délivrance d'un signal à un processus (1)

- **Caractères spéciaux au terminal**
  - **^C, ^Z, ^\, ...**
- **Fonctions spéciales du shell**
  - **fg**, **bg**, **kill**...
- **Primitive Posix : fonction `kill()`**

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig)
```

Délivrance d'un signal à un processus (2)

- **La valeur du pid dans kill permet de s'adresser à différents processus**

pid	qui reçoit le signal?
> 0	Le processus de PID = pid
= 0	Tous les processus du groupe de processus de pid
=-1	Tous les processus à qui le processus appelant peut envoyer des signaux
< -1	Tous les processus dont le groupe de processus est égal à -pid

Etat d'un signal en Ansi C
Fonction `signal()` *(1)*

```
#include <signal.h>
void (*signal(int sig, void (*ph)(int))(int);
```

ou, si l'on préfère

```
typedef void (*Ptr2Handler)(int);
Ptr2Handler signal(int sig, Ptr2Handler ph);
```

- Positionne l'action associée à la réception du signal **`sig`**
- L'action associée est **`ph`** (« pointer to handler »)
  - **`SIG_IGN`** : signal ignoré
  - **`SIG_DFL`** : action par défaut
  - une fonction utilisateur (paramètre **`int`**, retour **`void`**) : piégé
- Retourne l'ancienne action associée

Etat d'un signal en Ansi C
Fonction signal() (2)

```
#include <signal.h>

void on_signal(int sig) {
 printf("*** signal %d\n", sig);
}

main() {
 void (*ph)(int);

 signal(SIGQUIT, SIG_IGN);
 ph = signal(SIGINT, SIG_IGN);
 printf("INT et QUIT ignorés\n");
 sleep(5);
 signal(SIGQUIT, on_signal);
 signal(SIGINT, on_signal);
 printf("INT et QUIT trappés\n");
 sleep(5);

 signal(SIGQUIT, SIG_DFL);
 signal(SIGINT, ph);
 printf("INT restauré "
        "QUIT défaut\n");
 sleep(5);
}
```

Etat d'un signal en Ansi C
Fonction signal() *(3)*

```
% test-signal
INT et QUIT ignorés
^\^CINT et QUIT trappés
^\*** signal 3
^\*** signal 3
^C*** signal 2
INT restauré QUIT défaut
^C
%
```

Etat d'un signal en Posix

- **Inconvénients des signaux d'Ansi C**
  - Impossibilité de consulter l'action/état courant(e)
  - Impossibilité de bloquer (masquer) d'autres signaux pendant l'exécution du handler
  - Pas de possibilité d'extension
  - Gestion de la « réentrance » peut être compliquée
  - Durée de vie du handler dépend de l'implémentation

- **Posix introduit de nouveaux mécanismes**
  - Blocage (masquage) de signaux (emprunté à BSD)
  - Fonction **`sigaction()`** comme remplacement de **`signal()`**

Etat d'un signal en Posix
Action associée

	Bloqué	**Débloqué**
Ignoré	rien	rien
Associé à l'action par défaut	action différée au déblocage	action immédiate
Piégé	exécution du *handler* différée au déblocage	exécution du *handler* immédiate

Etat d'un signal en Posix
Masque des signaux *(1)*

```
#include <signal.h>
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *old_set);
```

- **set** contient l'ensemble des signaux à masquer ou démasquer
- how détermine la fonction à effectuer
  - **SIG_BLOCK** : bloque les signaux de **set**
  - **SIG_UNBLOCK** : débloque les signaux de **set**
  - **SIG_SETMASK** : positionne le masque du processus à **set**
- **old_set** contient l'ancien masque

```
int sigpending(sigset_t *set);
```

- **sigpending** retourne les signaux bloqués en attente

Etat d'un signal en Posix
Masque des signaux *(2)*

- **Ensemble de signaux**
  - Ensemble de bits, 1 bit par signal

- **Fonctions de manipulation**

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
int sigismember(const sigset_t *set, int sig);
```

État d'un signal en Posix
Fonction `sigaction()` *(1)*

```
#include <signal.h>
int sigaction(int sig,
              const struct sigaction *actp,
              struct sigaction *old_actp);
```

- **Champs de `struct sigaction`**
  - `void (*sa_handler)(int)` fonction de capture (identique à `signal()`)
  - `sigset_t sa_mask` masque des signaux à bloquer lors de l'exécution du handler
  - `int sa_flags` utile seulement pour `SIGCHLD`

État d'un signal en Posix
Fonction sigaction() *(2)*

```
#include <signal.h>

void on_signal(int sig) {
 printf("*** signal %d\n", sig);
 sleep(5);
 printf("*** fin handler\n");
}

main() {
 struct sigaction sigact;
 sigset_t msk_int, msk_quit;

 sigemptyset(&msk_int);
 sigaddset(&msk_int, SIGINT);
 sigemptyset(&msk_quit);
 sigaddset(&msk_quit, SIGQUIT);
 sigact.sa_handler = on_signal;
 sigact.sa_mask = msk_quit;
 sigaction(SIGINT, &sigact,
           NULL);
 sigact.sa_mask = msk_int;
 sigaction(SIGQUIT, &sigact,
           NULL);

 printf("INT et QUIT trappés\n");
 sleep(10);
}
```

État d'un signal en Posix
Fonction `sigaction()` *(3)*

```
% test-sigaction
     INT et QUIT trappés
     ^C*** signal 2
     ^\^\*** fin handler
     *** signal 3
     *** fin handler
%
```

État d'un signal en Posix
Durée de vie du handler (1)

- **Lorsque la fonction `sigaction()` est utilisée pour trapper un signal, le handler est valide jusqu'à ce qu'un prochain `sigaction()` l'invalide**

- **En revanche la durée de vie du handler établi par `signal()` est dépendante de l'implémentation**
  - après réception du signal, l'action par défaut est rétablie
  - on est donc souvent conduit à réarmer le *handler* dans le *handler* lui-même (cas d'Unix SVR4, de Solaris...)

État d'un signal en Posix
Durée de vie du handler (2)

Code « Classique » en ANSI/C

```
#include <signal.h>

void on_signal(int sig) {
 static int in_handler = 0;

 signal(sig, on_signal);
 if (in_handler) { .... }
 else { in_handler = 1; .... }
 }

int main() {
  signal(SIGINT, on_signal);
  .....
  .....
}
```

Interaction entre les signaux et les primitives `fork()` et `exec()`

- **Attributs de processus**
  - État des signaux (ignoré, action par défaut, trappé)
  - Masque des signaux bloqués
- **Héritage de l'état et du masque lors d'un `fork()`**
- **Transmission de l'état et du masque à travers un `exec()`**
  - sauf pour les signaux *piégés* qui sont rétablis à l'*action par défaut*

Autres fonctions liées aux signaux

```
#include <unistd.h>
int sleep(unsigned int seconds);
```

- Suspend le processus pendant le nombre de secondes indiqués, ou jusqu'à ce qu'un signal (non ignoré) arrive

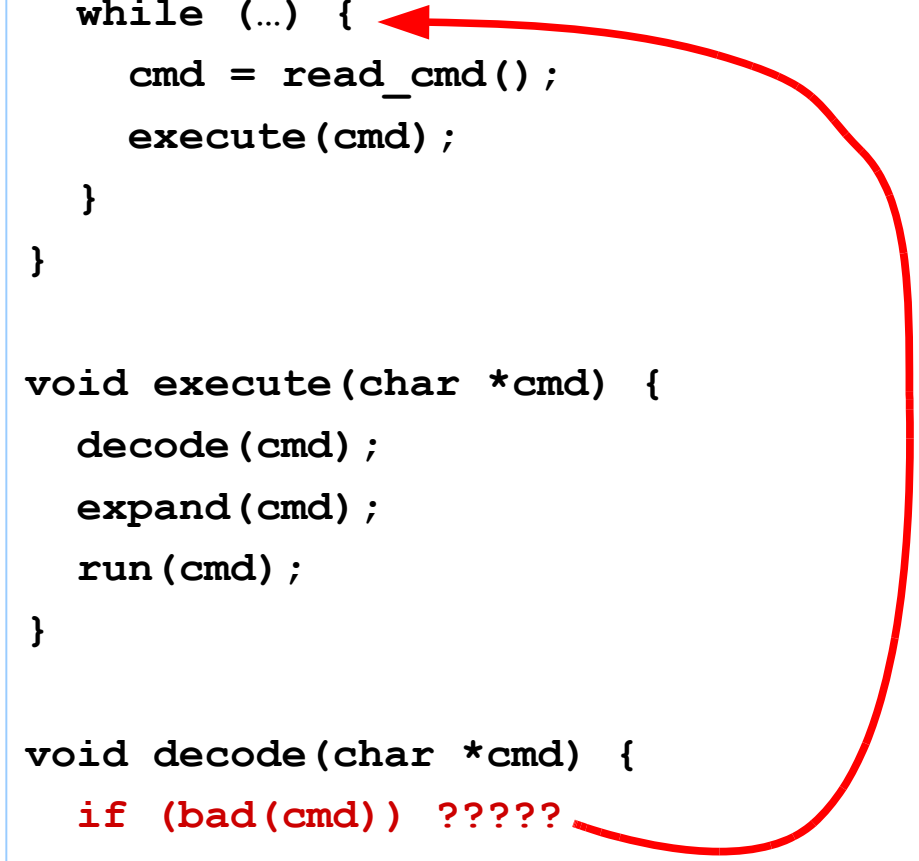
```
#include <unistd.h>
int pause();
```

- Suspend le processus jusqu'à ce qu'un signal (non ignoré) arrive
- Cette fonction retourne après que le *handler* (éventuel) ait été exécuté

Points de reprise

- **Motivation**
  - Permettre d'abandonner une fonction pour reprendre le traitement à un niveau plus élevé dans la chaîne (dynamique) d'appel
  - Mécanisme primitif d'exception
  - Goto non local

```
main() {
  while (…) {
    cmd = read_cmd();
    execute(cmd);
  }
}

void execute(char *cmd) {
  decode(cmd);
  expand(cmd);
  run(cmd);
}

void decode(char *cmd) {
  if (bad(cmd)) ?????
}
```



Points de reprise en Ansi C
setjmp() et longjmp() (1)

```
#include <setjmp.h>
int setjmp(jmp_buf env);
int longjmp(jmp_buf env, int v);
```

- **setjmp()** positionne un point de reprise
  - les informations sont mémorisées dans **env**
  - **env** doit être une variable gobale
  - lors du premier passage, **setjmp()** retourne 0

- **longjmp()** se branche au point de reprise **env**, cad au **setjmp()** correspondant
  - la valeur **v** est alors retournée par **setjmp()** (ou 1 si **v** = 0)
  - l'environnement **env** doit être *actif*

Points de reprise en Ansi C
setjmp() et longjmp() (2)

```
jmp_buf env;




main() {

  while (…) {
    if (setjmp(env) > 0)
      puts("Recommencez !");
    cmd = read_cmd();
    execute(cmd);
  }
}
```

```
void execute(char *cmd) {
  decode(cmd);
  expand(cmd);
  run(cmd);
}

void decode(char *cmd) {
  if (bad(cmd)) longjmp(env, 1);
}
```

Points de reprise en Ansi C
setjmp() et longjmp() (3)

```
jmp_buf env;

void on_int(int sig) {
  long_jmp(env, 2);
}

main() {
  signal(SIGINT, on_int);
  while (…) {
    if (setjmp(env) > 0)
      puts("Recommencez !");
    cmd = read_cmd();
    execute(cmd);
  }
}
```

```
void execute(char *cmd) {
  decode(cmd);
  expand(cmd);
  run(cmd);
}

void decode(char *cmd) {
  if (bad(cmd)) longjmp(env, 1);
}
```

Points de reprise Posix
sigsetjmp() et siglongjmp()

- **Les fonctions `setjmp()` et `longjmp()` ne sauvegardent pas (et donc ne restaurent pas) le masque des signaux bloqués**

- **Posix a donc introduit `sigsetjmp()` et `siglongjmp()`**
  - utilisation identique (**`sigsetjmp`** a 2 arguments)
  - à préférer aux versions Ansi C

- ***Dans les deux cas , les variables locales sont dans un état* indéfini *après un `longjmp()`***
  - *à moins d'avoir été déclarées **`volatile`***

La norme Posix.1

Gestion des terminaux

Introduction

- **Avant Posix**
  - Primitive `ioctl()`
    - Introduite par Unix V7
    - Présente sur pratiquement tous les Unix
    - Primitive à tout faire, pour tous types de périphériques
  - Structure `termio`
    - Introduite par Unix System V
    - Spécifique à la gestion des terminaux
- **Solutions trop spécifiques à Unix**
- **Nécessité d'un mécanisme portable**

Gestion des terminaux en entrée

- **Posix.1 définit deux modes d'entrée**

- **Mode canonique**
  - Mode ligne par ligne
  - Les caractères ne sont transmis au programme que sur une fin de ligne

- **Mode non canonique**
  - Plus de notion de ligne
  - Les caractères peuvent être transmis au programme n'importe quand

Gestion des terminaux en entrée
Mode canonique

- **Les caractères ne sont disponibles pour le programme que sur une *fin de ligne***

	`read(0, buf, 5);` ➔ **bloqué**
`123456789\n23\n`	`read(0, buf, 5);` ➔ **"12345"**
`6789\n23\n`	`read(0, buf, 50);` ➔ **"6789\n"**
`23\n`	`read(0, buf, 10);` ➔ **"23\n"**

Gestion des terminaux en entrée
Mode non canonique

- **Plus de délimiteur de ligne**
- **Deux paramètres**
  - **Min** : nombre minimum de caractères pour débloquer **`read()`**
  - **Time** : délai (en 1/10 s) pour débloquer **`read()`**

Min > 0 **Time > 0**	**`read`** attend **Minimum**(Min, Time) (le premier caractère déclanche le timer)
Min > 0 **Time = 0**	**`read`** termine quand Min caractères sont disponibles
Min = 0 **Time > 0**	**`read`** termine au bout de Min 1/10 s au plus (le timer se déclanche tout de suite)
Min = 0 **Time = 0**	**`read`** termine tout de suite (le nombre de caractères lus peut être nul)

Gestion des terminaux en entrée
Autres fonctions

- **Gestion des caractères spéciaux**
  - Caractères liés aux signaux
    - **`^C`** (**`SIGINT`**) , **`^\`** (**`SIGQUIT`**), **`^Z`** (**`SIGTSTP`**)…
  - Caractères de gestion du flux
    - **`^J`**, **`^M`** (fin de ligne)
    - **`^Q`**, **`^S`** (xon/xoff : contrôle de flux)
    - **`^D`** (fin de fichier, flush)…
  - Caractères d'édition
    - **`^H`** (backspace)…

Gestion des terminaux en entrée
Les trois modes des « vieux » Unix

- **Mode « cooked »**
  - Mode canonique (ligne)
  - Tous caractères spéciaux traités
- **Mode « cbreak »**
  - Mode non canonique
  - Seuls les caractères spéciaux liés aux signaux sont traités
- **Mode « raw »**
  - Mode non canonique
  - Aucun caractère spécial traité

Gestion des terminaux Posix (1)

```
#include <termios.h>
#include <unistd.h>
struct termios {
        tcflags_t c_iflag;      /* input modes */
        tcflags_t c_oflag;      /* output modes */
        tcflags_t c_cflag;      /* control modes */
        tcflags_t c_lflag;      /* local modes */
        cc_t c_cc[NCCS];        /* control chars */
};

int tcgetattr(int fd, struct termios *tiop);
int tcsetattr(int fd, int actions, struct
                termios *tiop);
```

- Voir aussi la commande **stty** (et **stty -a**)

Gestion des terminaux Posix (2)

- **`c_iflag`**
  - parité, traduction cr/lf, contrôle de flux…
- **`c_oflag`**
  - traduction cr/lf, sortie tout en majuscules…
- **`c_cflag`**
  - aspects hardware du terminal
- **`c_lflag` (le plus utile)**
  - mode canonique ou non
  - echo ou non
  - traitement des signaux
- **`c_cc`**
  - définition des associations de caractères spéciaux
  - définition de Min et Time (**`VMIN`** et **`VTIME`**)

Gestion des terminaux Posix
Exemples

- **Passage en mode non canonique sans écho**

```
struct termios tios;
tcgetattr(0, &tios);
tios.c_lflags &= ~(ICANON | ECHO);
tios.c_cc[VMIN] = 1;
tios.c_cc[VTIME] = 0;
tcsetattr(0, TCSANOW, &tios);
```

- **Retour au mode canonique**

```
tios.c_lflags |= ICANON;
tcsetattr(0, TCSANOW, &tios);
```

Terminal de contrôle

- **Identification du terminal de contrôle**

```
#include <stdio.h>
char *ctermid(char s[L_ctermid]);
```

  - Voir aussi la commande shell **tty et `/dev/tty`**

- **Le terminal de contrôle est un attribut du processus**
  - donc hérité par **`fork()`** et transmis lors d'un **`exec()`**

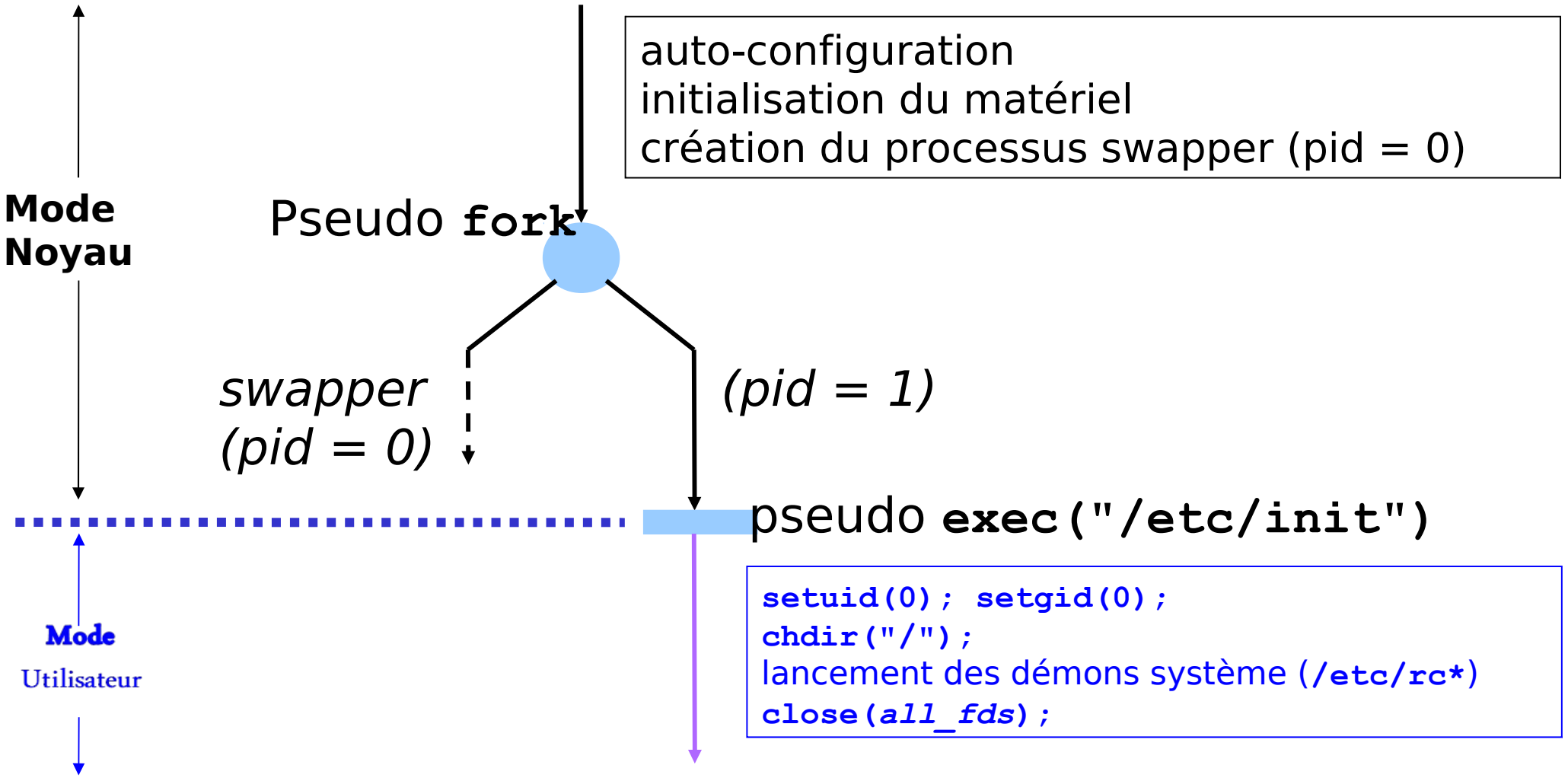
- **Certains processus se détachent de leur terminal de contrôle**
  - en particulier les « démons » système

La norme Posix.1

Gestion du temps partagé :

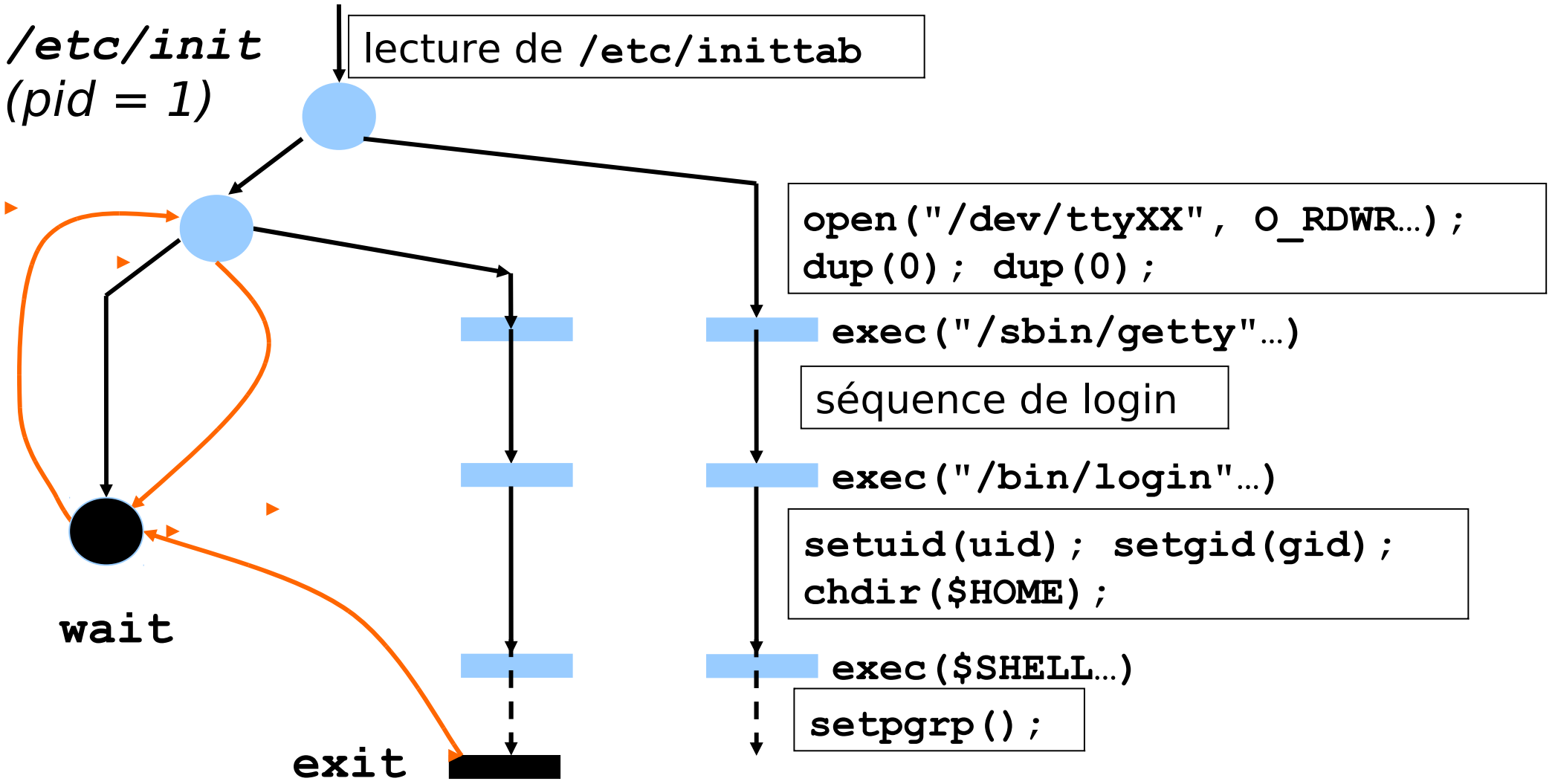
`/etc/init` et **shell**

Gestion du temps partagé
Initialisation (boot)



Gestion du temps partagé

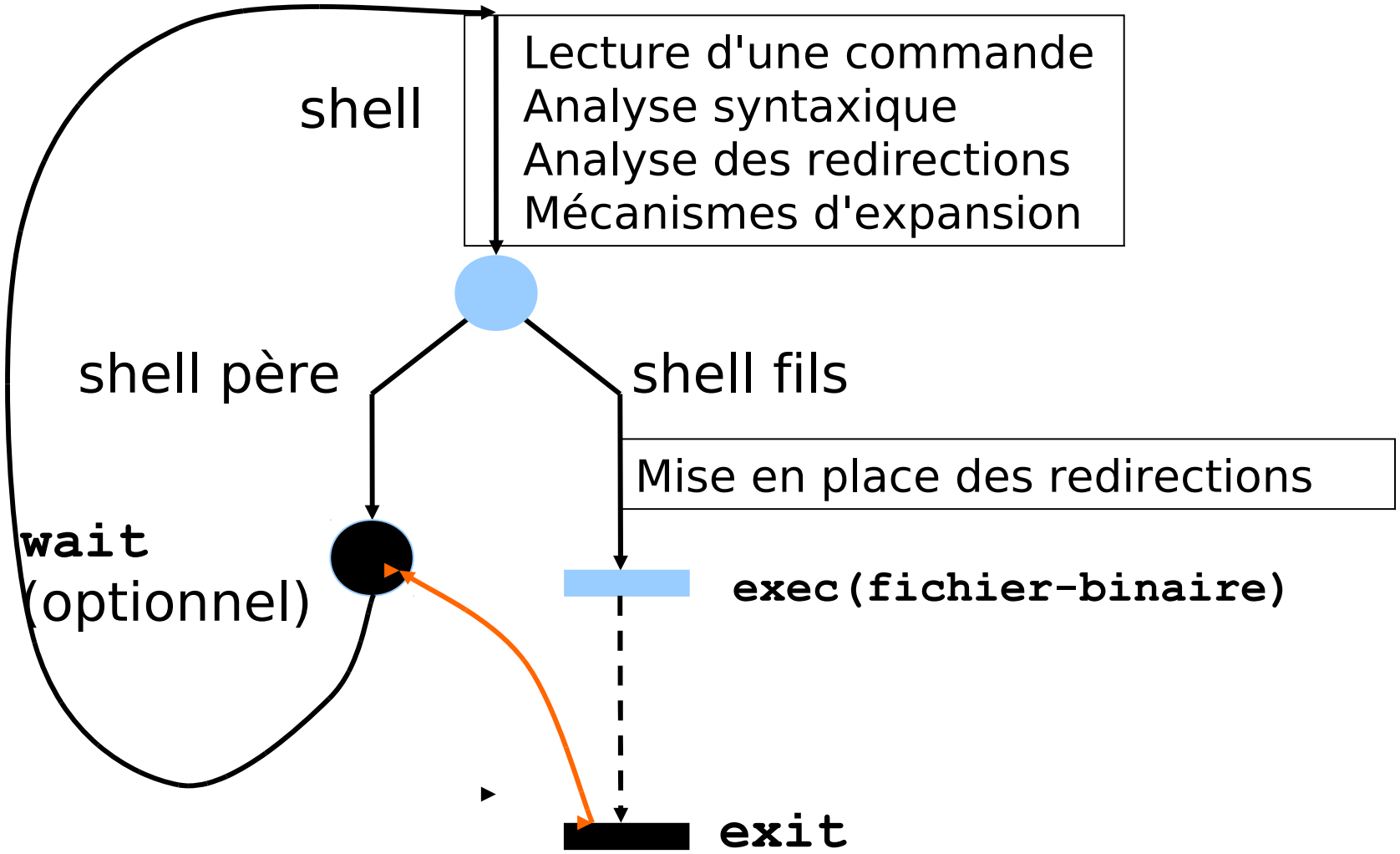
/etc/init (1)



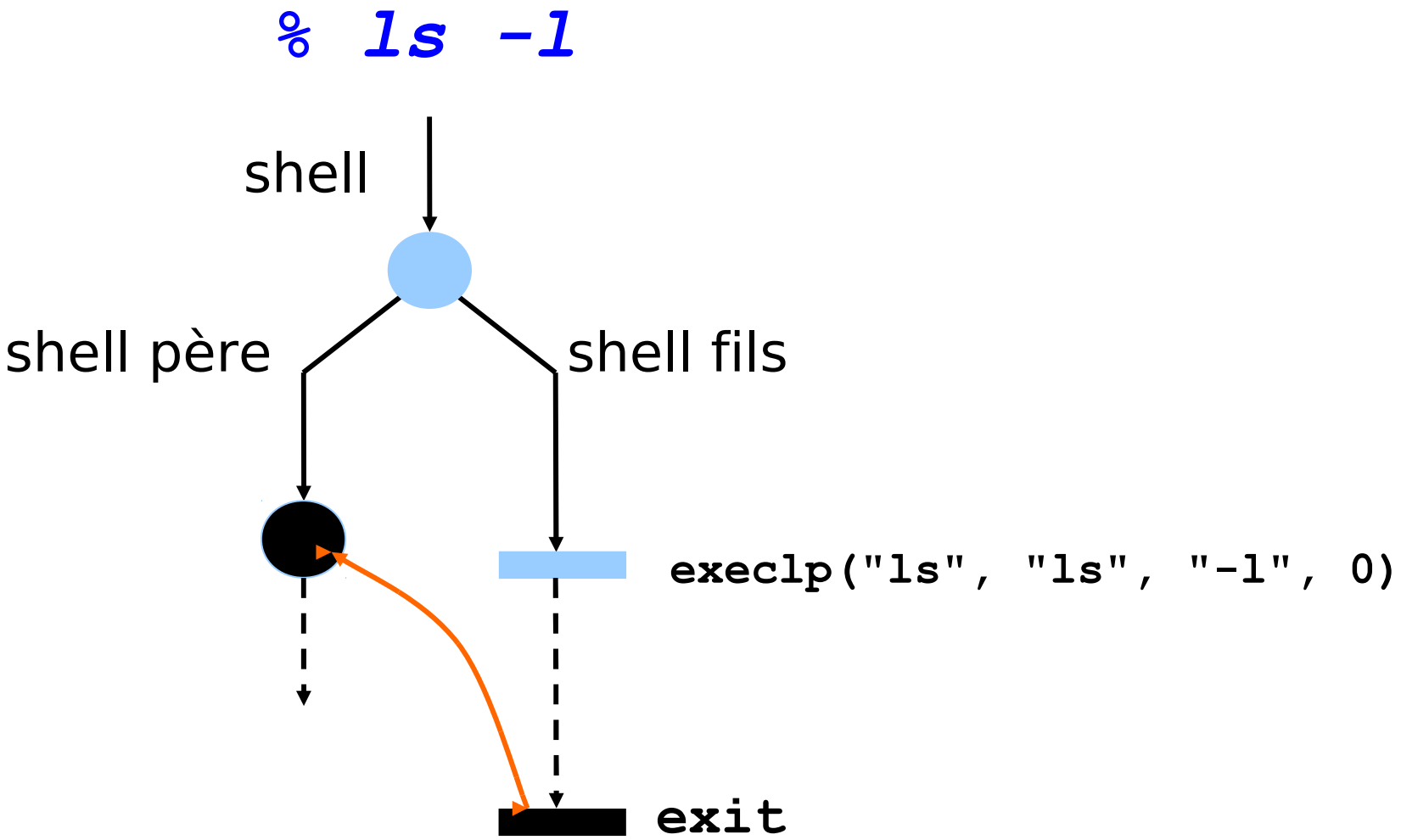
Gestion du temps partagé
`/etc/init` *(2)*

- **Comportement sur les signaux**
  - **`/etc/init`** trappe **`SIGHUP`** et **`SIGTERM`**
  - **`SIGHUP`**
    - Force la relecture du fichier **`/etc/inittab`** et la création éventuelle de nouvelles séquences de login
  - **`SIGTERM`**
    - Terminaison normale du système (shutdown)
    - Termine tous ses fils (y compris les démons système)

Exécution du shell
Principe général

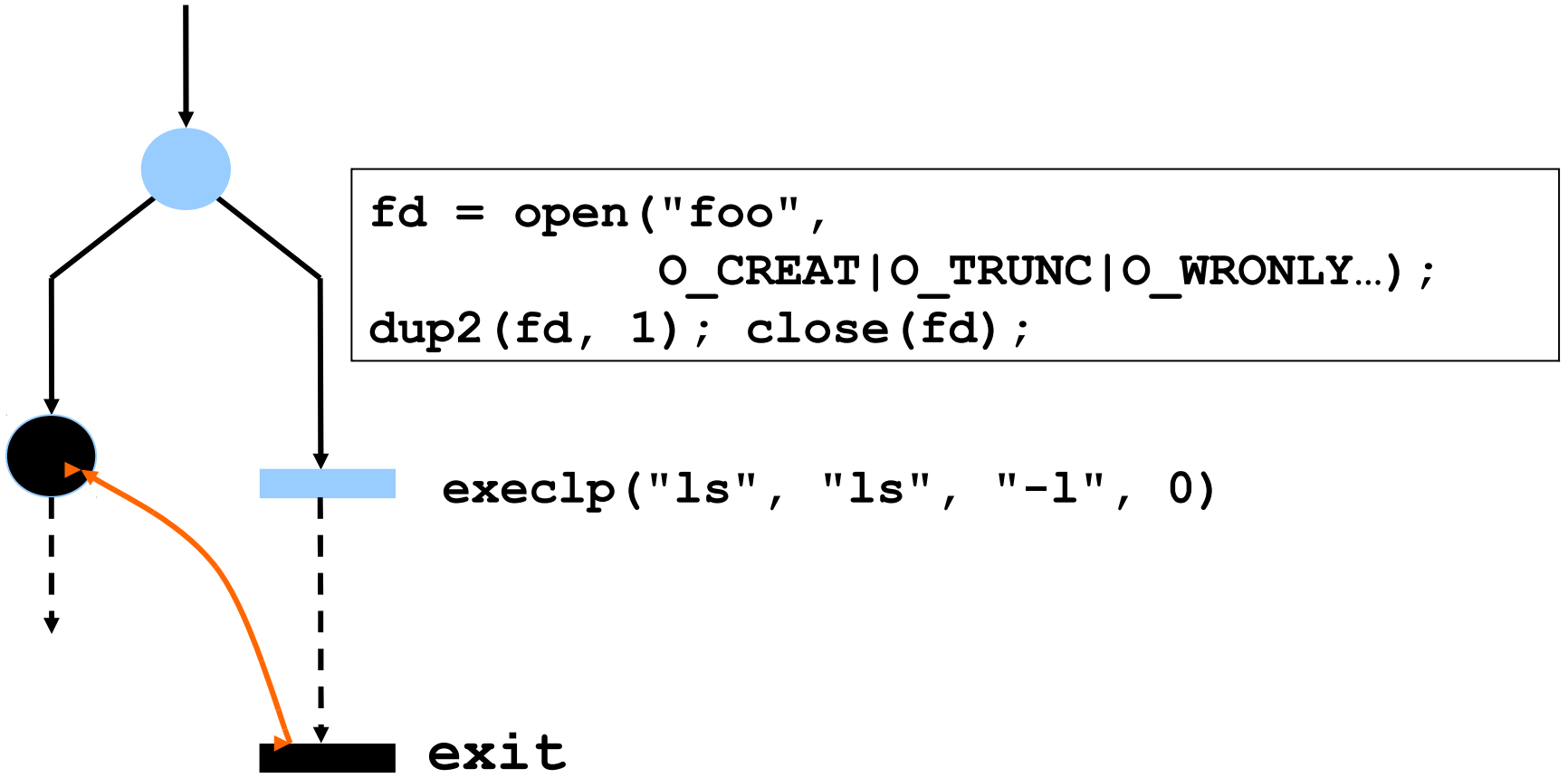


Exécution du shell
Commande simple



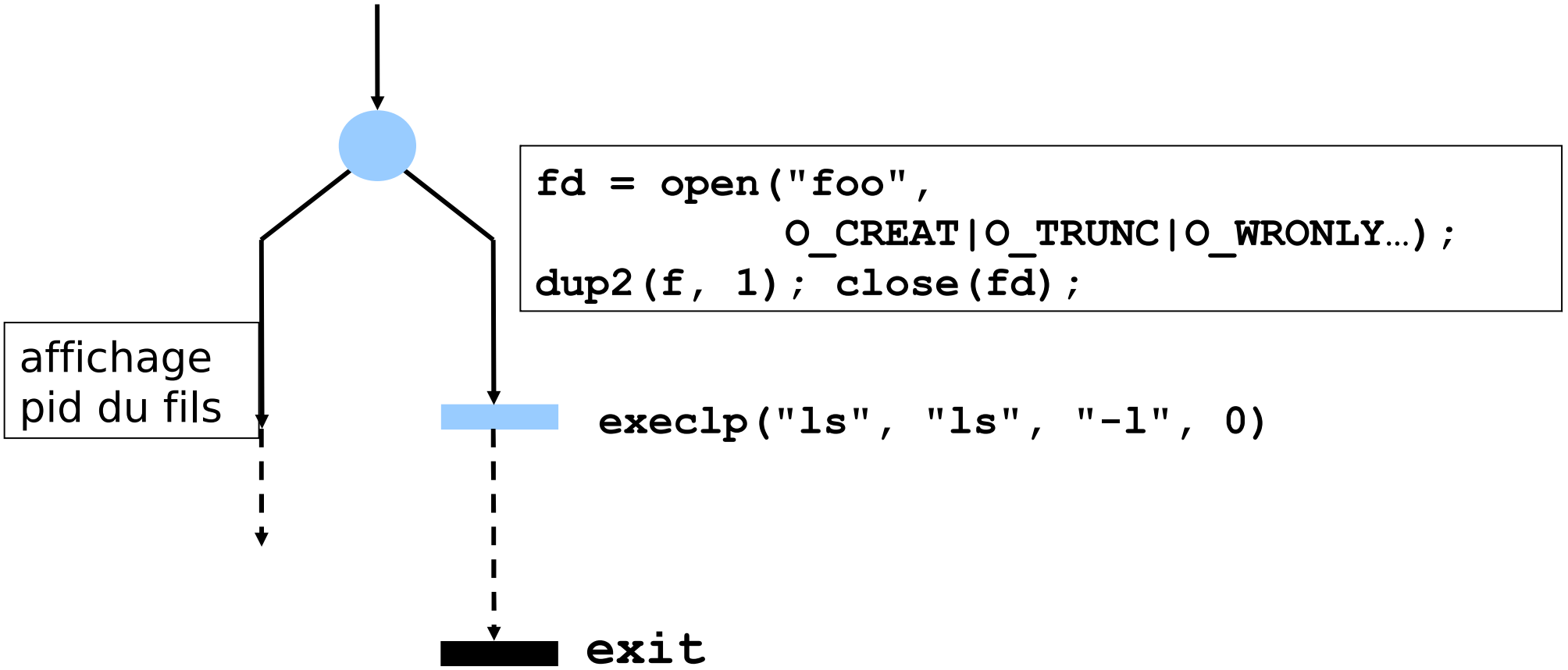
Exécution du shell
Commande avec redirection

```
% ls -l > foo
```

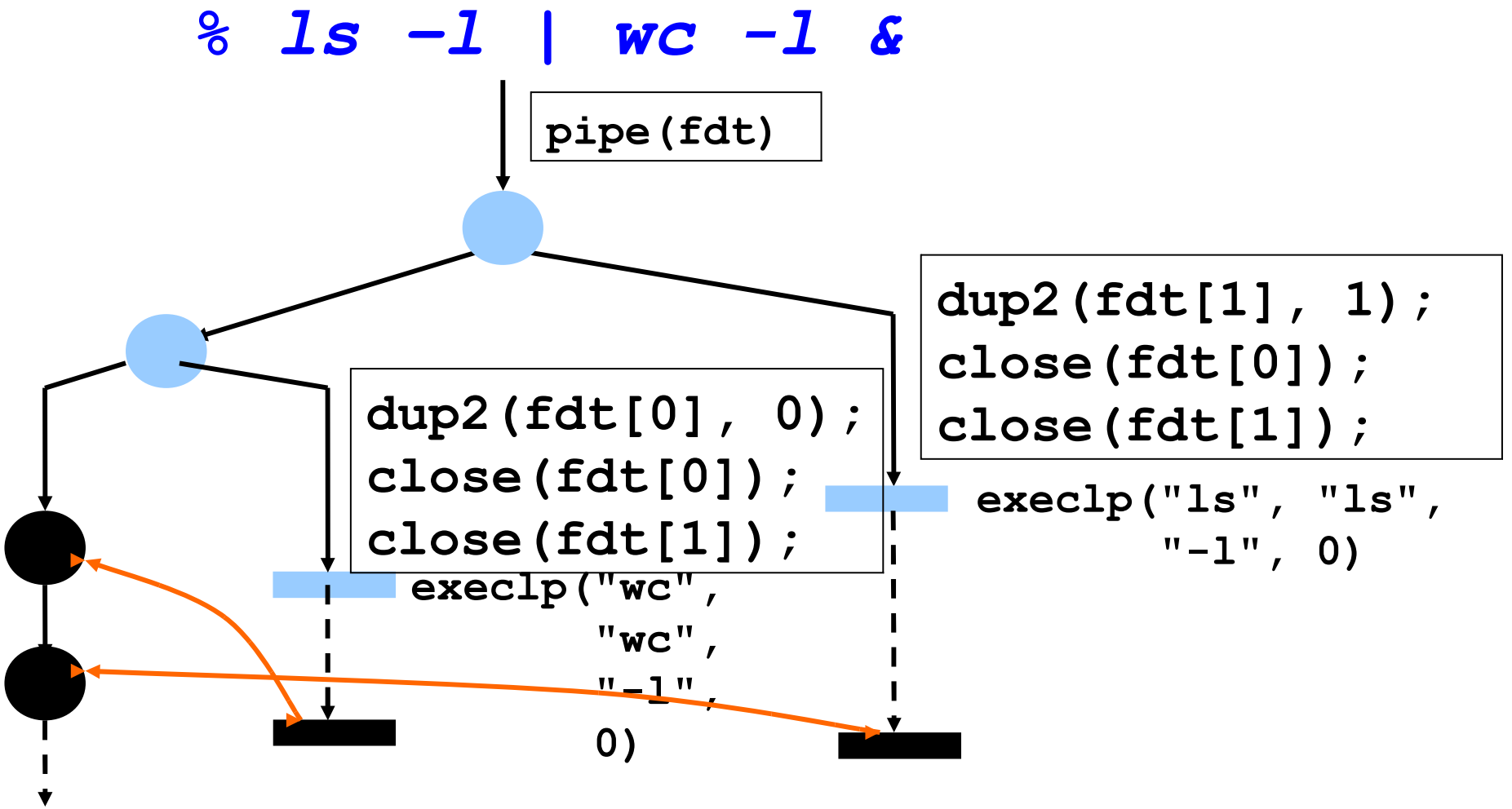


Exécution du shell
Commande en arrière plan

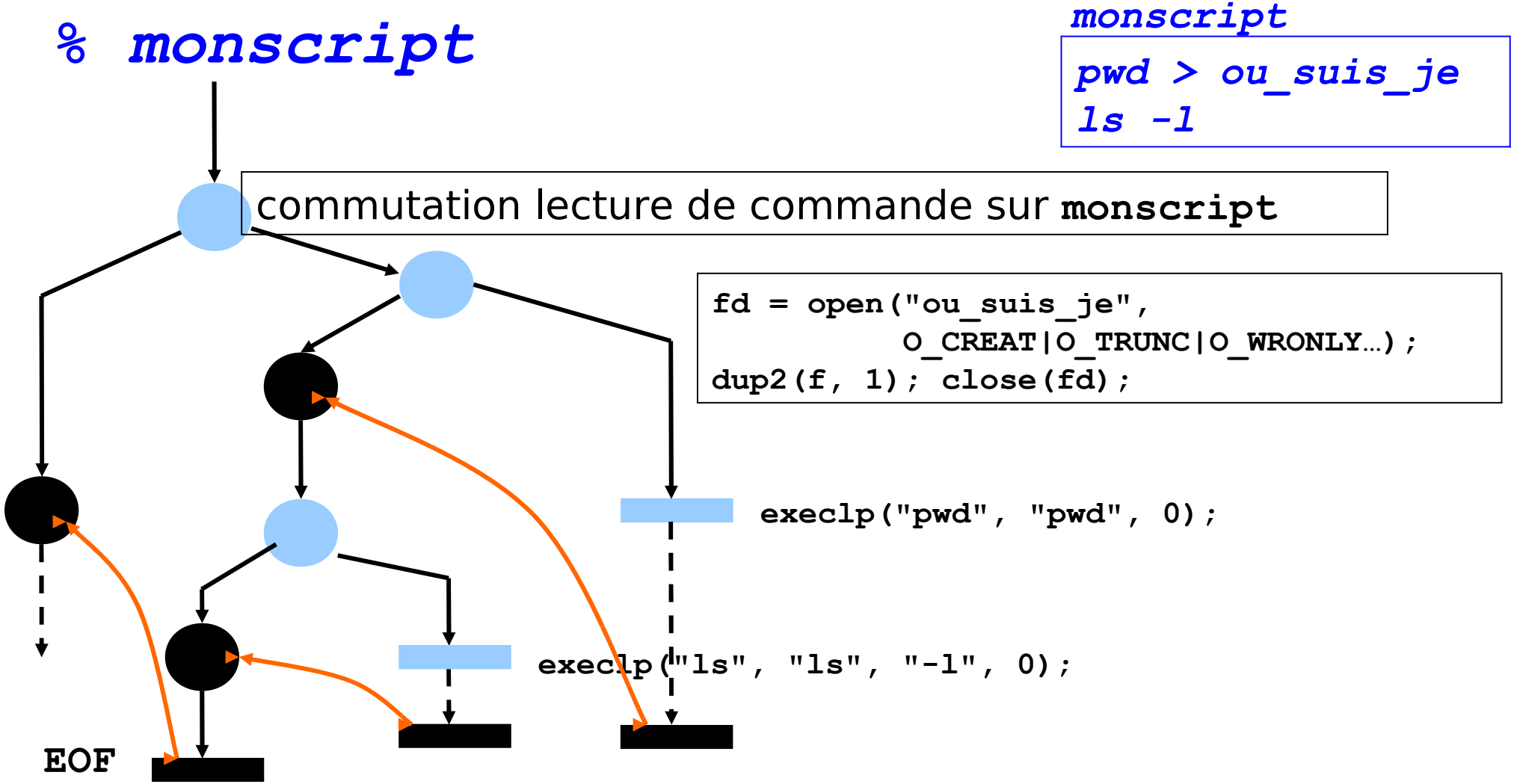
```
% ls -l > foo &
```



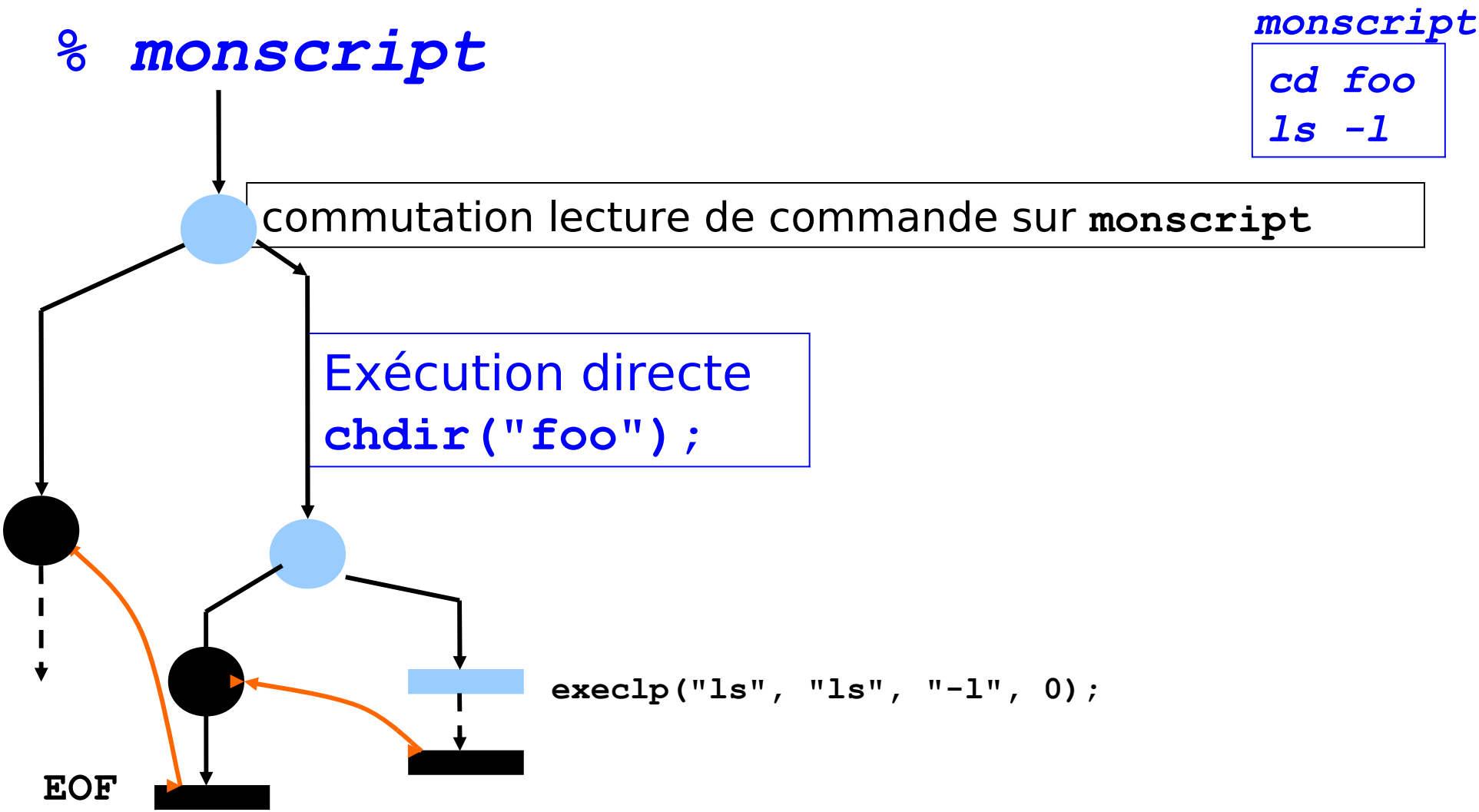
Exécution du shell
Commande avec pipe



Exécution du shell
Script (fichier de commandes)



Exécution du shell
Commande interne (builtin)



La norme Posix.1

Internationalisation et localisation

Définitions

- **Internationalisation**
  - Conception de programmes *neutres* par rapports aux différents pays
- **Localisation**
  - Adaptation d'un programme aux habitudes et conventions d'un pays particulier
  - Ne devrait pas changer la logique du programme mais seulement la présentation des données et des résultats

Conventions locales

```
#include <locale.h>
char *setlocale(int category,
                const char *locale);
```

- **Catégories**

`LC_COLLATE`	comportement de `strcoll()`
`LC_CTYPE`	comportement de `<ctype.h>`
`LC_MONETARY`	affichage des nombres
`LC_TIME`	comportement de `strftime()`
`LC_ALL`	tout

- **Locales**

`NULL`

`C` POSIX

langage[_*territoire*][.*codeset*]@*modifieur*

`fr_BE.UTF-8@euro`

Localisation des messages d'erreur

- **Posix.1 a abdiqué !**
- **X/Open (un consortium industriel) a défini la notion de catalogue de messages d'erreur**
  - Chaque message est repéré par son indice dans le catalogue
  - Voir les fonctions

`catopen()`

`catclose()`

`catgets()`