R3.05 Programmation Système

BUT Informatique

Michaël Hauspie

11 octobre 2023

Plan du cours

Organisation du cours

- 6 semaines
- 1h d'amphi
- 3h de TP

Équipe pédagogique

- Julien Baste
- Bruno Beaufils
- Jean Carle
- *Michaël Hauspie*

Contenu

- Programmation système en C
  - Utilisation des primitives de la spécification POSIX
- Gestion mémoire
- Système de fichiers
- Gestion de processus
- Communication inter-processus

Évaluation

- Chaque TP sera vérifié
  - TP validé = 1 point
  - 6 TPs = 6 points
- Contrôle TP à la fin du module
  - Noté sur 14
- Note finale du module = Note CTP + Notes TPs

Plan du cours

Rôles d'un système

- Fournir un environnement d'exécution aux applications
- Abstraire le matériel
- Partager les ressources
  - Assurer une utilisation équitable
  - Protéger les ressources

Système Multi-*

- Les systèmes POSIX sont
  - Multi-utilisateurs : plusieurs entités peuvent avoir des droits d'accès différents
  - Multi-tâches : plusieurs tâches s'exécutent en *même temps*, en apparence

Fournir un environnement d'exécution

Un application a besoin

- De mémoire
- De ressources
  - temps de calcul
  - fichier, réseau...

Pour exécuter un programme, le système crée un processus qui va exécuter ce programme.

Accèder aux ressources : abstraction

- Fournir une vue indépendante du matériel
  - portabilité
- Deux *côtés*
  1. Proposer une interface pour accéder au matériel pour les processus
     - ★ Sous Unix, tout est fichier
  2. Permettre au noyau de gérer les matériels du même type
     - ★ notion de pilotes (*drivers*)

Partage d'accès au ressources

- Partage et protection de la mémoire
  - Un processus ne doit pas pouvoir accéder à la mémoire d'un autre processus
    - ★ gestion des protections
  - Plusieurs processus doivent pouvoir avoir de la mémoire
    - ★ gestion des allocations
- Partage et protection des autres ressources
  - Limiter les accès aux fichiers, aux matériels

Partage du temps : ordonnancement

- Un cœur ne peut exécuter qu'une seule instruction à la fois
  - il faut partager le temps d'utilisation à tous les processus du système
- Le noyau du système choisi quel processus s'exécute à quel moment et pendant combien de temps
  - c'est l'ordonnancement

Exemples d'ordonnancement

- Ordonnancement par ordre de soumission (premier arrivé, premier servi)
  - on choisi un processus dans l'ordre d'arrivé, et on l'exécute tant qu'il peut s'exécuter
- Ordonnancement par tourniquet
  - on choisi les processus chacun leur tour, et on les exécute pendant un temps fixé avant de passer au suivant
- Ordonnacement par priorité préemptive
  - Même chose que pour le tourniquet, mais on donne priorité à certains processus.
    - ★ C'est un ordonnancement de cet famille qui est utilisé sous linux

Plan du cours

Organisation mémoire d'un processus

La mémoire d'un processus est organisée en plusieurs *parties*

1. Les données *statiques*
2. le code, *i.e* les instructions machines à exécuter
3. la pile
4. le tas (ou /heap)

Les parties 1, 2 et 3 sont de taille fixe

Les données statiques

Ce sont les données globales du processus, initialisées à partir du fichier exécutable. On y trouve :

- les variables globales
  - initialisées à partir de la valeur données dans le programme
  - mise à 0 sinon
- les constantes globales

Ces données statiques sont *valides* du début à la fin de l'exécution du processus

Les données statiques – Exemple

```c
#include <stdio.h>

int a = 1; int b;
const int c = 3;

int main(void) {
    printf("&a: %p\n", &a);
    printf("&b: %p\n", &b);
    printf("&c: %p\n", &c);
    printf("chaîne: %p\n", "Une constante");
    return 0;
}
```

```
&a: 0x100db4000
&b: 0x100db4004
&c: 0x100daff80
chaîne: 0x100daffa9
```

Le code

Les instructions machines sont chargées en mémoire au moment de l'exécution du programme par le processus.

```c
#include <stdio.h>

int main(void) {
    printf("main: %p\n", main);
    return 0;
}
```

```
main: 0x1001cbf5c
```

La pile

Contient les contextes d'exécutions du processus. A chaque appel de fonction, un contexte de fonction (*stack frame*) est empilé. Il contient principalement :

- l'adresse de retour (où revenir quand la fonction est terminée)
- les paramètres de la fonction
- les variables locales

La pile – Avertissements

Les variables locales ne sont pas initialisées. Leur valeur dépend de l'évolution passée de la pile

Quand la fonction est quittée, la stack frame n'est plus valide. Aucun pointeur vers cette frame ne doit être utilisé en dehors de la fonction

La pile – Exemple

```c
#include <stdio.h>

void b(void) {
    int j = 42;
    printf("(%p) j = %d\n", &j, j);
}
void a(void) {
    int i;
    printf("(%p) i = %d\n", &i, i);
}

int main(void) {
    a();
    b();
    a();
    return 0;
}
```

```
(0x16dcf329c) i = 1
(0x16dcf329c) j = 42
(0x16dcf329c) i = 42
```

Le tas (*Heap*)

Toutes les zones de mémoires précédentes ont une taille fixe

- pile : paramètre du noyau
- données statique et code : dépendent du fichier exécutable

Et une durée de vie non *contrôlable*

- pile : durée de vie de la *stack frame*
- données statique et code : toute la durée de vie du programme

Un processus doit pouvoir obtenir de la mémoire en fonction d'une taille non connue à la compilation et décider de sa durée de vie

- traitement de volume de données importants
- données qui *survivent* à une fonction

Le tas (*Heap*)

Le tas est une zone mémoire dont la taille varie en fonction des besoins du processus.
C'est le processus qui décide d'agrandir ou réduire le tas avec `brk`, `sbrk`, `mmap`, `munmap`.
Les développeurs utilisent plutôt `malloc`, `realloc`, `calloc` et `free` : interface simplifiée pour l'allocation de mémoire contiguë.

Le tas (*Heap*) – Exemple

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    void *p = malloc(10);
    printf("p: %p\n", p);
    free(p);
    return 0;
}
```

```
p: 0x6000036a0050
```

Le tas *(Heap)* – Exemple

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *nouvelle_chaine_vide(size_t taille_max) {
    // +1 pour le '\0'
    char *p = malloc(taille_max + 1);
    if (p == NULL)
        return NULL;
    p[0] = '\0';
    return p;
}

int main(void) {
    char *s = nouvelle_chaine_vide(20);
    strncpy(s, "Hello world", 20);
    printf("> %s\n", s);
    free(s);
    return 0;
}
```

```
> Hello world
```

Le tas *(Heap)* – Mauvais Exemple

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *nouvelle_chaine_vide(size_t taille_max) {
    // tableau de taille variable autorisés depuis C99
    // MAIS: si la taille est trop grande, le comportement est indéfini, à éviter absolument !
    char p[taille_max + 1];
    // On retourne l'adressse d'une variable située dans la stack frame
    // cette adresse ne sera plus valide en sortie de fonction
    return p;
}

int main(void) {
    char *s = nouvelle_chaine_vide(20);
    strncpy(s, "Hello world", 20);
    printf("> %s\n", s);
    return 0;
}
```

```
file.c:14:12: error: address of stack memory associated with local variable 'p'
      returned [-Werror,-Wreturn-stack-address]

    return p;
           ^
1 error generated.
```

Plan du cours

Principes

Au niveau noyau, un fichier est une suite non-structurée d'octets

Le système permet d'accéder à cette suite :

- sans formatage
- sans conversion
  Seuls des octets sont lus et écrits
- Sans tampon d'entrées/sorties
- Appel direct au système, sans intermédiaire

Processus et fichiers

Chaque processus gère une table de fichiers ouverts

- chaque index est appelé *descripteur de fichiers*

Accès au fichiers par les appels systèmes :

1. `open` : ouverture
2. `read` ou `write` : lecture ou écriture
3. `close` : fermeture
4. `lseek` : déplace du *curseur*

Ouverture (1/2)

```
int open(const char *path, int oflag);
```

- `path` : chemin vers le fichier à ouvrir (absolu ou relatif)
- `oflag` : option d'ouverture, combinable avec l'opérateur |
  - `O_RDONLY`
  - `O_WRONLY`
  - `O_RDWR`
  - ...

Retourne le descripteur du fichier ouvert, ou -1 en cas d'erreur

Ouverture (2/2)

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

Si l'option `O_CREAT` est utilisée (pour créer un fichier qui n'existe pas), les permissions demandées par l'application doivent être précisé dans `mode`.
Les permissions effectives seront calculés en appliquant le `umask`

Exemple

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(void) {
    int fd;
    fd = open("/tmp/toto", O_WRONLY | O_CREAT, 0666);
    if (fd == -1)
        printf(" Cration du fichier /tmp/toto impossible");
    return 0;
}
```

Traitement des erreurs

- Lors du erreur d'un appel système, la variable globale `errno` est fixée à une valeur correspondante à l'erreur
- Avec `perror`, on peut afficher un message correspond à cette erreur sur la sortie d'erreur

```
if (open("/monfichier", O_RDONLY) == -1) {
    perror("/monfichier");
}
```

A l'exécution :

```
/monfichier: No such file or directory
```

Vous devez traiter les erreurs pour tous les appels système !

Lecture

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- Lit au plus `nbytes` octets dans le fichier `fildes` et les place dans la mémoire à l'adresse indiquée par `buf`
- `ssize_t` et `size_t` sont simplement des entiers (signés et non signés respectivement) dont la taille dépend du système (généralement 64 bits sur les systèmes modernes)

Retourne

- -1 en cas d'erreur
- 0 si la fin de fichier est atteinte
- le nombre d'octets effectivement lus sinon

Une lecture avance le curseur du fichier. La prochaine lecture commencera là ou la précédente s'est arrêtée

Écriture

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Écrit les `nbyte` octets contenus en mémoire à l'adresse `buf` dans le fichier `fildes`

Retourne

- -1 en cas d'erreur
- le nombre d'octets effectivement écrits sinon

Fermeture

```
int close(int fildes);
```

Le nombre de fichiers ouvert simultanément par un processus est limité. Il faut fermer les fichiers que l'on utilise plus
Ferme le fichier `fildes`.
Retourne :

- -1 en cas d'erreur
- 0 sinon

Déplacement du curseur

On peut déplacer le curseur courant (pour lire ou écrire ailleur dans le fichier)

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

Déplace le curseur de `fildes` de `offset` octets par rapport à la position `whence`

- `off_t` est simplement un entier signé (à l'image de `ssize_t`)
- `offset` peut être positif ou négatif
- `whence` peut valoir :
  - `SEEK_SET` : déplacement par rapport au début du fichier
  - `SEEK_CUR` : déplacement par rapport à la position courante
  - `SEEK_END` : déplacement par rapport à la fin du fichier

Retourne

- -1 en cas d'erreur
- la position après déplacement par rapport au début du fichier (en octets)

Plan du cours

Philosophie

- Sous UNIX : tout est fichier
- Le terme *fichier* désigne les ressources
  - matérielles (disque dur, terminal,...)
  - logicielles (image, son, texte,...)
- Chaque fichier est associé à une structure décrivant ses caractéristiques : l'inode

Inode

Un fichier est identifié de manière unique par

- son numéro de périphérique (sur quel système de fichiers est situé le fichier)
- son numéro d'inode (relatif au périphérique)

Pour accéder aux informations de l'inode d'un fichier, on utilise l'appel système

```
int stat(const char *restrict path,
         struct stat *restrict buf);
```

Dans la structure de type `struct stat`, on retrouve, en autres :

`st_dev`	numéro de périphérique
`st_ino`	numéro d'inode
`st_mode`	permissions et type de fichier
`st_nlink`	nombre de noms de cet inode (liens physiques)
`st_uid`, `st_gid`	utilisateur et groupe
`st_size`	taille du fichier

voir manuel `stat(2)` pour les autres champs

Types de fichiers

Il existe plusieurs type de fichiers (qu'on retrouve dans le champ `st_mode`)

- fichier réguliers : le contenu est une suite d'octets non struturée classique. La taille est connue et permet de trouver la fin du fichier
- répertoires : le contenu est structuré comme une liste d'entrées
- fichiers spéciaux : le contenu correspond à une ressource. Ils permettent des accès par blocs (disques, etc.) ou par octets (terminaux, imprimantes, etc.)
- liens symboliques : le contenu est interprété comme le chemin vers un autre fichier
- tubes : ils permettent la communication uni-directionnelle entre processus
- socket : ils permettent la communication bi-directionnelle entre processus

Test du type de fichier

Différentes macros sont définies pour interpréter le champ `st_mode`

```
S_ISBLK(s.st_mode)      /* block special */
    S_ISCHR(s.st_mode)      /* char special */
    S_ISDIR(s.st_mode)      /* directory */
    S_ISFIFO(s.st_mode)     /* fifo or socket */
    S_ISREG(s.st_mode)      /* regular file */
    S_ISLNK(s.st_mode)      /* symbolic link */
    S_ISSOCK(s.st_mode)     /* socket */
```

Lecture des fichiers répertoires

Un fichier répertoire contient une liste de couple

- nom : une chaîne de caractère
- numéro d'inode : un entier

La structure de cette liste dépend du système de fichiers.
Pour permettre une utilisation portable, on utilise les fonctions de la librairie C

- `opendir` : ouvrir un répertoire
- `readdir` : récupérer la prochaine entrée d'un répertoire ouvert
- ...

  Pages de manuel `opendir(3)`, `readdir(3)`,...

Exemple live - une implémentation de `ls`

Plan du cours

Rappels

Un programme est une suite d'instructions que le système doit faire accomplir au processeur . Ces instructions sont rangées dans un fichier.
Un processus correspond au déroulement (*l'exécution*) d'un programme dans un environnement particulier.

Définition

Un processus :

- est un objet dynamique correspondant à l'exécution d'un programme
- possède un espace d'adressage propre
- peut s'exécuter dans deux modes différents :
  - en mode utilisateur, le processus exécute les instructions du programme et accède aux données de son espace d'adressage
  - en mode noyau, le processus exécute les instructions du noyau et a accès à l'ensemble des données du système

Plusieurs processus peuvent partager le même programme.

Généralités

- Tout processus peut créer de nouveaux processus
- Tout processus est créé par un appel à `fork` (sauf le premier)
- `fork` duplique le processus appelant
- Les processus sont organisés en arborescence en fonction de leur processus créateur que l'on nomme père
- Le noyau a en charge la gestion des processus et le partage des ressources entre eux.
  - en particulier l'ordonnancement

Caractéristiques 1/2

- Son identité : `pid`
- L'identité de son père : `ppid`
- Un lien avec les utilisateurs :
  - son propriétaire réel/effectif
  - son groupe propriétaire réel/effectif

Caractéristiques 2/2

- Son répertoire de travail
- Sa date de création
- Les temps CPU consommés en mode utilisateur et mode noyau
- La table des descripteurs de fichiers
- Son état
- ...

Accès aux caractéristiques

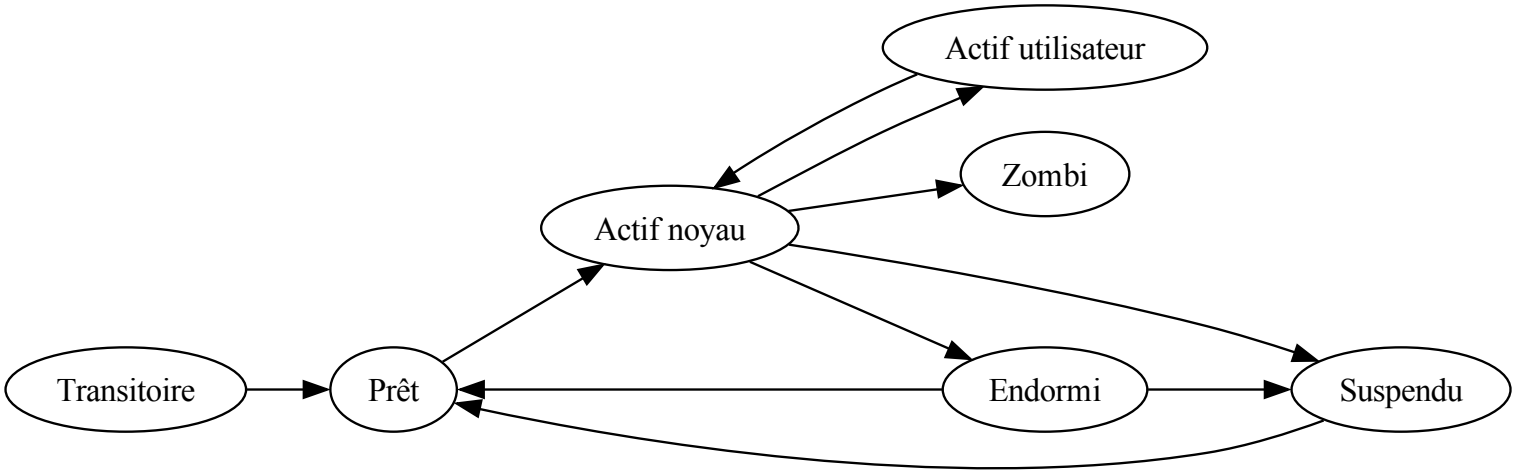
```c
pid_t getpid(void);          // identité
pid_t getppid(void);         // identité du père

uid_t getuid(void);          // propriétaire réel
uid_t geteuid(void);         // propriétaire effectif

gid_t getgid(void);          // propriétaire réel
gid_t getegid(void);         // propriétaire effectif

int chdir(const char *path); // Changement répertoire
                             // de travail

// récupération du répertoire de travail
char *getcwd(char *buf, size_t size);
```

État d'un processus

Au cours de sa vie, un processus passe dans plusieurs états

- Transitoire : à sa création
- Prêt (`R`) : prête à s'exécuter
- Actif (`R`) : en cours d'exécution
- Endormi (`S` ou `D`) : en attente d'un événement (entrées/sorties,...)
- Suspendu (`T`)
- Zombi (`Z`) : processus terminé, mais dont le père n'a pas encore récupérer les informations concernant la termination

Changement d'état



fork

```
pid_t fork(void);
```

- Créé un processus fils
- Après la création, les deux processus semblent avoir exécuté l'appel à `fork`
- Chacun des processus continue son exécution à partir de l'instruction qui suit le `fork`
- La valeur de retour de `fork` permet de différencier le père du fils
  - 0 dans le fils
  - le `pid` du fils nouvellement créé dans le père

Exemple

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

int main(void) {
    printf("Processus %d: je suis tout seul\n", getpid());
    fork();
    printf("Processus %d: nous sommes deux\n", getpid());
    return 0;
}
```

```
Processus 71720: je suis tout seul
Processus 71720: nous sommes deux
Processus 71721: nous sommes deux
```

Génétique

Le processus fils hérite des caractéristiques de son père sauf :

- son `pid`
- le `pid` du père
- les temps CPU (remis à zéro dans le fils)
- les verrous sur les fichiers
- les signaux pendants
- la priorité utilisée pour l'ordonnancement (remise à une valeur standard dans le fils)

Terminaison

- Le processus père peut accéder aux informations relatives à la terminaison du fils
  - S'est-il terminé correctement ? (a-t'il fini son programme)
  - A-t'il été arrêté par un signal (une faute, un Ctrl-C. . .)
- Le processus qui se termine passe par l'état de Zombi et y reste tant que le père n'a pas pris connaissance de l'information

Remarque si le père se termine avant le fils, le fils est *adopté* par le processus de `pid 1`

Récupération des informations de terminaison 1/2

```
pid_t wait(int *pstatus);
```

- à appeler par le processus père
- retourne -1 s'il n'a pas de fils
- si le processus a des fils qui ne sont pas zombi, bloque le processus appelant jusqu'à ce qu'un des fils passe zombi
- si le processus possède au moins un fils à l'état de zombi, retourne le `pid` de l'un des zombi (choisi par le système)
  - le processus zombi disparait
  - si `pstatus` est différent de `NULL`, l'entier pointé par `pstatus` reçoit les informations sur la terminaison du processus zombi.

Récupération des informations de terminaison 2/2

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

- Identique à `wait` mais permet un contrôle plus fin
- `pid` peut être
  - -1 attends tous les processus fils (comportement de `wait`)
  - nombre positif : attends un processus en particulier
  - nombre négatif ou 0 : lien avec les groupes de processus, voir `man waitpid`
- `options` est une combinaison (|) d'options, en particulier `WNOHANG` qui permet de faire un appel non bloquant
- retourne
  - -1 en cas d'erreur
  - 0 si aucun fils à l'état de zombi en appel non bloquant
  - `pid` du fils zombi qui a été supprimé

Plan du cours

Exécuter un programme

Pour pouvoir être exécuté, un programme a besoin :

1. d'un processus
2. d'être placé en mémoire, dans l'espace d'adressage du processus

Lorsque l'on créé un processus avec `fork()`, le processus père est dupliqué pour constituer le processus fils, y compris le code de son programme.

Il faut donc charger un nouveau programme exécutable dans le nouveau processus

Chargement d'un programme

Sous linux, un programme est stocké dans un fichier au format ELF (*Executable and Linkable Format*).
Il s'agit d'un format de fichier qui contient plusieurs sections correspondantes aux différents partie du programme.

Quelques sections standards

Parmis ces sections, on peut citer par exemple :

- `.text` : le code du programme, *i.e.* les instructions machine du programme, celles-ci sont dépendantes du jeu instruction de l'architecture cible
  - `x86_64` : Intel 64 bits
  - `aarch64` : ARM 64 bits
  - ...
- `.bss` : les données globales non initialisées
- `.data` : les données globales initialisées, c'est à dire dont les valeurs initiales sont données dans le fichier
- `.rodata` : les données constantes (comme les chaînes constantes par exemple)
- ...

Inspection d'un fichier ELF : objdump

La commande objdump permet d'obtenir des informations sur un fichier exécutable au format ELF.

Analysons le programme suivant

```c
#include <stdio.h>

int bss_var;
int data_var = 3;
const int rodata_var = 12;


void une_fonction(void) {
    puts("Bonjour");
}


int main(void) {
    une_fonction();
    return 0;
}
```

Inspection d'un fichier ELF : les symboles

Les symboles sont des alias vers des adresses. Il correspondent aux nom de vos variables, fonctions...

Section .text

```
$ objdump -t test-objdump
[...]
00401040 l    d  .text 0000   .text
00401080 l     F .text 0000   deregister_tm_clones
004010b0 l     F .text 0000   register_tm_clones
004010f0 l     F .text 0000   __do_global_dtors_aux
00401120 l     F .text 0000   frame_dummy
004011b0 g     F .text 0001   __libc_csu_fini
00401126 g     F .text 0011   une_fonction
00401150 g     F .text 005d   __libc_csu_init
00401070 g     F .text 0001   .hidden _dl_relocate_static_pie
00401040 g     F .text 002b   _start
00401137 g     F .text 0010   main
[...]
```

Inspection d'un fichier ELF : les symboles

Section .data

```
00404020 l    d  .data 0000   .data
00404020  w      .data 0000   data_start
00404034 g       .data 0000   _edata
00404020 g       .data 0000   __data_start
00404030 g     O .data 0004   data_var
00404028 g     O .data 0000   .hidden __dso_handle
00404038 g     O .data 0000   .hidden __TMC_END__
```

Section .rodata

```
00402000 l  d  .rodata 0000   .rodata
00402004 g    O .rodata 0004   rodata_var
00402000 g    O .rodata 0004   _IO_stdin_used
```

Inspection d'un fichier ELF : le code machine

```
$ objdump -d test-objdump
[...]
0000000000401137 <main>:
  401137: 55                    push   %rbp
  401138: 48 89 e5              mov    %rsp,%rbp
  40113b: e8 e6 ff ff ff        callq  401126 <une_fonction>
  401140: b8 00 00 00 00        mov    $0x0,%eax
  401145: 5d                    pop    %rbp
  401146: c3                    retq
  401147: 66 0f 1f 84 00 00 00  nopw   0x0(%rax,%rax,1)
  40114e: 00 00
[...]
```

Chargement d'un fichier programme dans un processus

Le chargement du programme se fait à l'aide des commandes de la famille `exec*` :

- `execl`
- `execv`
- `execlp`
- `execvp`
- `...`

Le caractère de terminaison `p` indique si la commande doit chercher le fichier exécutable dans le `PATH` de l'utilisateur
Les caractères `v` ou `l` indiquent le mode de passage des arguments au programme que l'on charge

- `l` : les paramètres de la fonction seront les arguments du programme
- `v` : on doit fournir les arguments sous la forme d'un tableau de type `argv` pré-construit

Chargement d'un fichier programme dans un processus

- Un appel à `exec*` remplace le programme du processus courant
- Le processus continue son exécution dans le `main` du nouveau programme

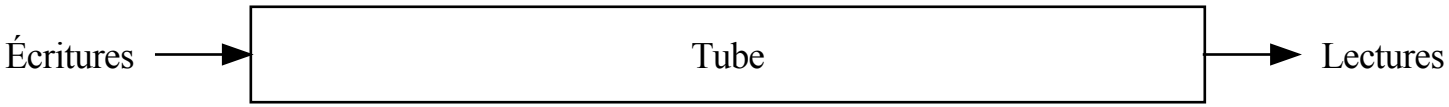
```c
#include <unistd.h>
#include <stdio.h>

int main(void) {
    printf("Je m'affiche\n"); fflush(stdout);
    execlp("echo", "echo", "Bonjour", NULL);
    printf("Je ne m'affiche pas\n"); fflush(stdout);
    return 0;
}
```

```
Je m'affiche
Bonjour
```

Plan du cours

Généralités 1/3

- Les tubes sont des mécanismes permettant aux processus de communiquer entre-eux
- Ils appartiennent au système de fichier *i.e.* ils sont décrits par un inode
- On peut les manipuler via un descripteur de fichier et, par exemple, `read` ou `write`
- Les tubes sont unidirectionnels



Généralités 2/3

- Un tube correspond à deux inœuds
  - une sortie, (utilisée en lecture)
  - une entrée, (utilisée en écriture)
- Un tube est une file : les premières données écrites sont les premières données lues (FIFO)
- Un tube a une capacité finie
  - une écriture *ajoute des données* au le tube
  - une lecture *retire des données* du tube

Généralités 3/3

Le système associe un nombre de lecteurs et d'écrivains pour chaque tube

- Le nombre de lecteurs correspond au nombre de descripteurs associés à la sortie du tube
- Le nombre d'écrivains correspond au nombre de descripteurs associés à l'entrée du tube

Le nombre de lecteurs et d'écrivains est important

- Si le nombre de lecteurs est nul, il est impossible d'écrire dans le tube
  - provoque, par défaut, l'arrêt du processus qui tente d'écrire (`SIGPIPE`)
- Si le nombre d'écrivains est nul, l'utilisation de `read` détecte une fin de fichier (*i.e.* retourne 0)

Création d'un tube

```
int pipe(int pipefd[2]);
```

Créé un tube, l'ouvre, et retourne, via `pipefd` deux descripteurs

- `pipefd[0]` correspond à la sortie du tube. On peut utiliser `read` sur ce descripteur
- `pipefd[1]` correspond à l'entrée du tube. On peut utilise `write` sur ce descripteur

Retourne 0 en cas de succès, -1 en cas d'erreur.

Exemple

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

int main(void) {
    int t[2];

    if (pipe(t) == -1) {
        perror("tube");
        return 1;
    }

    int source = 42;
    int destination = 0;

    // On écrit des données dans le tube en
    // utilisant t[1]
    if (write(t[1], &source, sizeof(source)) == -1) {
        perror("write");
        return 1;
    }
    // On a plus besoin d'écrire dans le tube, on ferme
    // le descripteur. Il n'y a plus d'écrivain
    close(t[1]);

    // On peut les lires avec t[0]
    if (read(t[0], &destination, sizeof(destination)) == -1) {
        perror("read");
    }
    printf("Lu: %d\n", destination);

    // Si on tente de relire, read va retourner 0
    printf("Retour de read: %zd\n", read(t[0], &destination, sizeof(destination)));
    return 0;
}
```

```
Lu: 42
Retour de read: 0
```

Lectures et écritures

Les lectures sont *effaçantes*

- Il s'agit d'une extraction des données et non d'une consultation
- Les fonction comme `lseek` n'ont pas de sens avec les tubes
- Une écriture dans un tube plein a pour effet de bloquer le processus
- Une lecture dans un tube vide a également pour effet de bloquer le processus

Partage de tube entre processus

Un tube n'a pas de nom

- On ne peut pas l'ouvrir avec `open`
  - un processus ne peut obtenir les descripteurs d'un tube qu'en le créant ou par héritage
- Un tube créé par un processus sera donc accessible à ses fils
  - s'il a été créé avant les fils
  - si ses descripteurs n'ont pas été fermés avant la création des fils
- Si un processus perd le descripteur (par un appel à `close`), il ne peut plus le récupérer

Attention au nombre d'écrivains !

- Tant qu'il y a des écrivains, le tube n'est jamais terminé
- Même si l'écrivain restant est le processus qui tente de lire
  - Un processus peut se bloquer lui même
- Toujours fermer l'entrée d'un tube dont on a plus besoin

Attention au nombre d'écrivains

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

int main(void) {
    int t[2];

    if (pipe(t) == -1) {
        perror("tube");
        return 1;
    }

    int source = 42;
    int destination = 0;

    // On écrit des données dans le tube en
    // utilisant t[1]
    if (write(t[1], &source, sizeof(source)) == -1) {
        perror("write");
        return 1;
    }
    // On oublie de fermer l'entrée du tube
    // close(t[1]);

    // On peut les lires avec t[0]
    if (read(t[0], &destination, sizeof(destination)) == -1) {
        perror("read");
    }
    printf("Lu: %d\n", destination);

    // Si on tente de relire, comme il y a toujours un écrivain
    // potentiel (nous), l'appel a read est bloquant
    printf("Retour de read: %zd\n", read(t[0], &destination, sizeof(destination)));
    return 0;
}
```

Même exemple avec deux processus

les traitements d'erreurs sont omis pour la lisibilité de la diapo

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(void) {
    int t[2];

    // le tube DOIT être créé avant le processus
    pipe(t);

    // Après le fork on a 2 écrivains : le père et le fils
    if (fork() == 0) { // Dans le fils, on hérite du tube
        int source = 42;
        write(t[1], &source, sizeof(source));
        // On termine le fils. Quitter le processus ferme automatiquement les descripteurs
        // Il ne reste plus qu'un écrivain : le père
        exit(0);
    }
    // Dans le père, on ferme immédiatement l'entrée du tube
    // Un écrivain en moins, il n'y en a plus
    close(t[1]);

    int destination = 0;
    read(t[0], &destination, sizeof(destination));
    printf("Lu: %d\n", destination);

    // Si on tente de relire, comme il n'y a plus d'écrivain, on détecte une fin de fichier
    printf("Retour de read: %zd\n", read(t[0], &destination, sizeof(destination)));
    // on attends la fin du fils pour éliminer le zombie
    wait(NULL);
    return 0;
}
```

```
Lu: 42
Retour de read: 0
```

Plan du cours

Cas d'usage de tubes et de processus

Lorsqu'un shell veut exécuter la commande

```
ls | wc
```

il doit :

1. Créer un tube
2. Créer deux processus (un pour `ls` et un pour `wc`)
3. Rediriger la sortie du processus `ls` vers l'entrée du tube
4. Rediriger l'entrée du processus `wc` vers la sortie du tube
5. Dans chaque processus, exécuter la commande correspondante

Redirections : la primitive dup

```
int dup(int oldfd);
```

permet de copier l'entrée de la table des descripteurs de fichiers `oldfd` vers une nouvelle entrée.
Retourne le descripteur vers la nouvelle entrée ou -1 en cas d'erreur.

- les deux descripteurs (le retour et oldfd) sont utilisables de façon interchangeable
- dans le cas d'un tube, cela implique un écrivain (ou un lecteur) supplémentaire
- `dup` choisi la première entrée libre comme destination de la copie : le descripteur retourné est donc le plus petit descripteur libre

Redirections : la primitive dup2

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

Effectue exactement la même opération mais, au lieu d'utiliser le plus petit descripteur disponible, copie `oldfd` dans l'entrée `newfd`.

- si `newfd` était ouvert (*i.e* non libre), l'entrée est fermée *silencieusement*
- l'opération est atomique, utiliser `dup2` permet d'éviter des erreurs de *race condition* (autre thread ou traitement dans un signal)

Retourne `newfd` ou `-1` en cas d'erreur

Exemple : `ls | wc`

le traitement d'erreurs est omis pour raison de lisibilité

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

int main(void) {
    int t[2];

    pipe(t);

    // processus pour ls
    if (fork() == 0) {
        // On copie le descripteur de l'entrée du tube vers 1
        // le tube devient la sortie standard du processus
        dup2(t[1], 1);
        // on peut exécuter ls
        execlp("ls", "ls", NULL);
        exit(1);
    }
    // Meme principe pour wc
    if (fork() == 0) {
        // Attention, wc est un écrivain, il DOIT fermer l'entrée du tube
        close(t[1]);
        // cette fois, c'est le descripteur 0 qu'on utilise
        dup2(t[0], 0);
        execlp("wc", "wc", NULL);
        exit(1);
    }
    // le père est également un écrivain, il DOIT fermer l'entrée du tube
    close(t[1]);
    // le père n'a plus qu'a attendre la fin des deux processus
    wait(NULL); wait(NULL);
    return 0;
}
```

Sortie du programme

```
26      26      299
```