Programmation systèmes et reseaux

Plan

- Préambule
- Systèmes d’exploitation – Définitions
- Systèmes d’exploitation – Fonctions de base
- Appels système
  - Appels système et gestion de fichiers
  - Appels système et gestion des processus
  - Les signaux
  - Les threads
  - Appels systèmes et communications inter-machines

Préambule (Objectifs du cours)

- Ce cours a pour objet …
  - de décrire le rôle et le fonctionnement d'un système d'exploitation **monoprocesseur** et **multi-tâches**
  - d'apprendre à manipuler certains concepts de base
- Ces concepts de base seront essentiellement illustrés au travers du système **UNIX/Linux** qui est un système
  - très homogène,
  - très riche,
  - très souple.

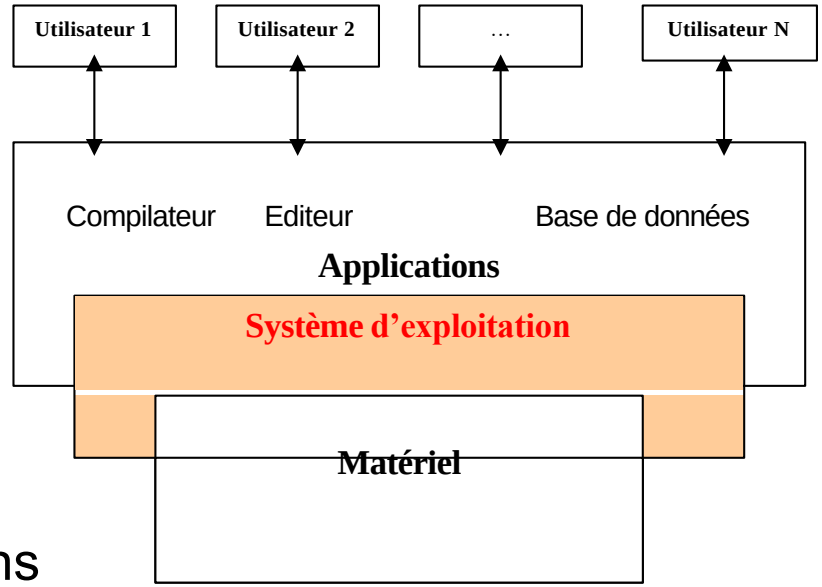
Systèmes d’exploitation – Définitions

- **Le système d'exploitation** (SE, en anglais Operating System ou OS) est un ensemble de programmes responsables de la liaison entre les ressources matérielles d'un ordinateur et les applications de l'utilisateur (traitement de texte, jeu vidéo, …)

- Il fournit aux programmes applicatifs des points d'entrée génériques pour **les périphériques**

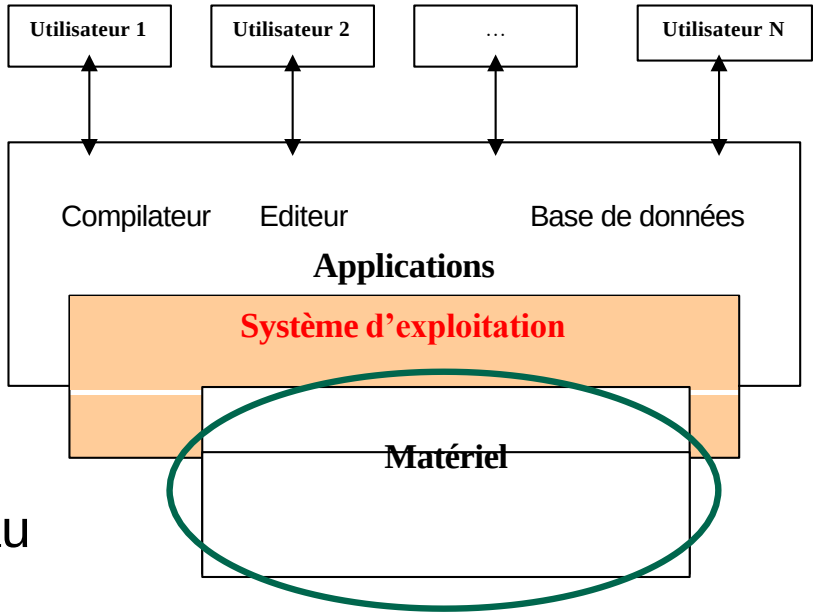
Systèmes d'exploitation – Définitions

- **Un ordinateur est constitué …**
  - **du matériel**
    - ✓ Dispositifs physiques
    - ✓ Langage machine
  - **d'un système d'exploitation**
  - **de programmes**
    - ✓ Programmes système
    - ✓ Programmes d'applications



Systèmes d'exploitation – Définitions

- **Le matériel (Hardware): Les dispositifs physiques constituent la couche la plus basse.**
  - le processeur,
  - la mémoire principale,
  - des disques,
  - des imprimantes,
  - des interfaces de connexion réseau
  - …

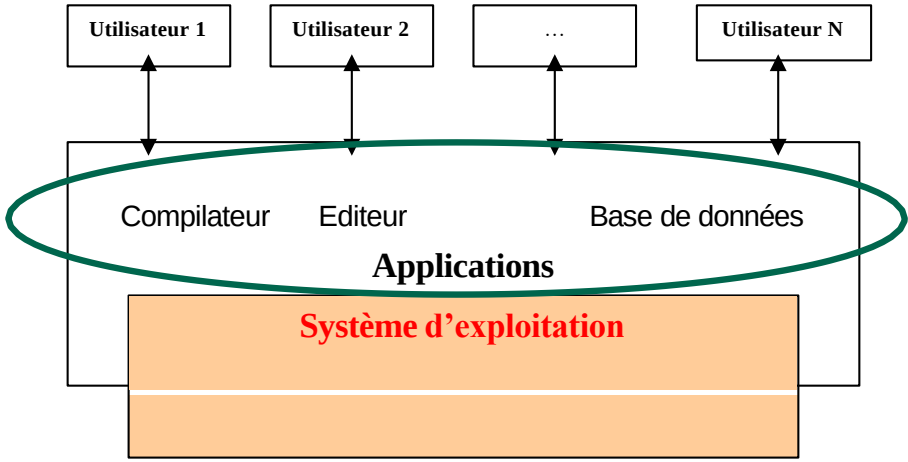


Systèmes d'exploitation – Définitions

- **Le système d'exploitation (Operating System ou OS ou SE) :** c'est le plus important des programmes systèmes
  - Il contrôle les ressources de l'ordinateur.
  - Il libère le programmeur de la complexité du matériel.
- Il se compose :
  - **D'un noyau (kernel) :** partie la plus critique d'un OS. Il permet aux éléments matériel et logiciel de communiquer entre eux, de fonctionner ensemble et de former un tout. Pour ces raisons, il est le premier logiciel chargé en mémoire.
  - **Des outils système :** partie permettant à l'utilisateur de tirer profit de l'OS, de gérer les périphériques, les configurer … En bref, ils fournissent une interface d'accès au système.
- Exemple de tâche: **LIRE UN BLOC DU FICHIER,**

Systèmes d'exploitation – Définitions

- Les programmes (Software, applications) : ils sont écrits
  - par les utilisateurs
  - ou par les éditeurs de logiciels
- But : résoudre des problèmes spécifiques tels que …
  - le traitement commerciales,
  - les calculs scientifiques,
  - etc.



Systèmes d'exploitation – Définitions

La double fonction d'un système d'exploitation

- Un système d'exploitation permet de répondre à deux besoins qui **ne sont pas forcément liés** :
  - Le système d'exploitation en tant que **machine étendue** (ou « machine virtuelle »)
  - Le système d'exploitation en tant que **gestionnaire de ressources**

Exemple de systèmes d'exploitation :

Linux, Mac OS, Windows 9X, Me, 2000, XP, MS-DOS, MINIX, etc

Systèmes d'exploitation – Définitions

Le SE en tant que machine étendue

- Le système d'exploitation correspond à «**l'interface**» entre les applications et le matériel.
- De ce point de vue, le système d'exploitation peut être assimilé à une machine étendue ou virtuelle **plus facile à programmer ou à utiliser que le matériel**
  - Un programmeur va utiliser le système d'exploitation par l'intermédiaire **"d'appels système".**
  - Un utilisateur peut lui aussi – dans une certaine mesure – manipuler un système d'exploitation, sans pour autant avoir à créer un programme (**commandes shell**).

Systèmes d'exploitation – Définitions

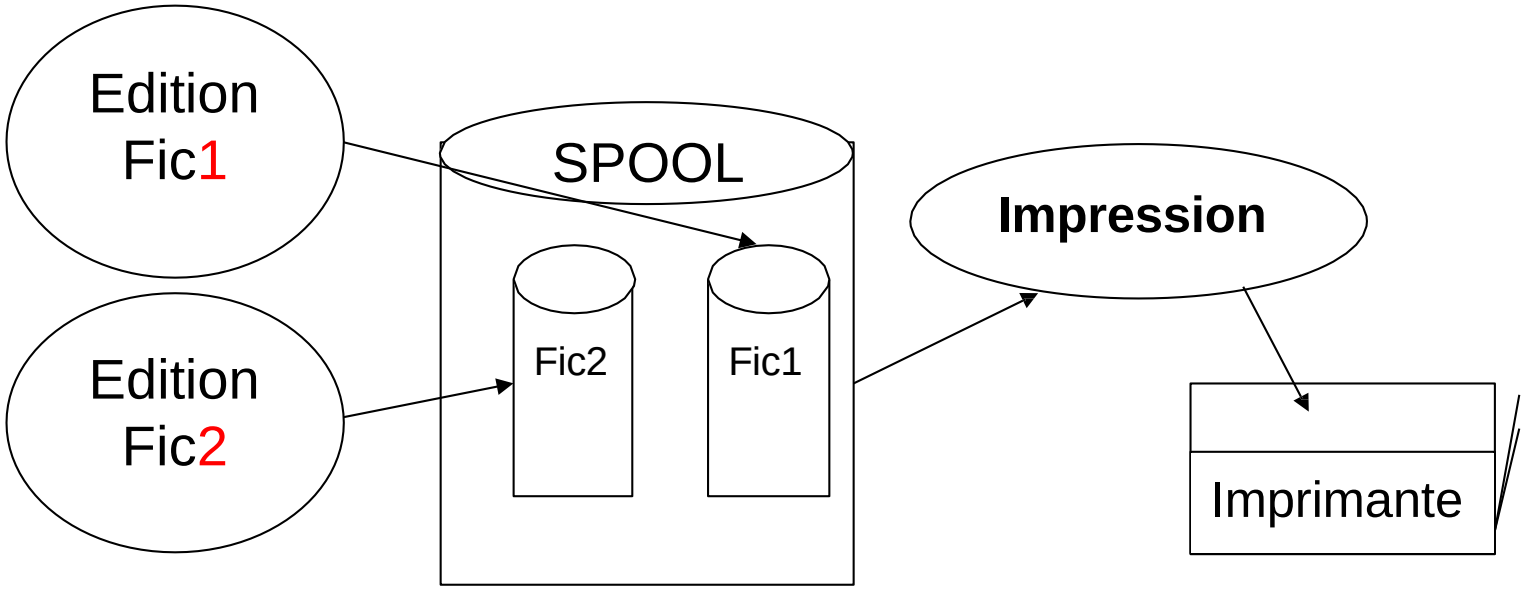
Le SE en tant que gestionnaire de ressources

- Les différents composants d'un ordinateur doivent coopérer et partager des ressources
- Dans cette optique, le travail du système d'exploitation consiste à
  - ordonnancer,
  - contrôler l'allocation des ressources :
    - processeurs,
    - mémoires,
    - périphériques d'E/S,

  entre les différents programmes qui y font appel

Systèmes d'exploitation – Définitions

Exemple de gestion de ressources : gestion des impressions



Systèmes d'exploitation – Définitions

Le SE en tant que gestionnaire de ressources

- Pour chacune des ressources d'un ordinateur, le système d'exploitation doit :
  - Connaître à tout moment l'utilisateur de la ressource,
  - en accorder l'usage de manière équitable,
  - Éviter les conflits d'accès entre les différents programmes ou utilisateurs.

- Les deux tâches essentiels du système d'exploitation en tant que gestionnaire des ressources sont :
  - Le partage des ressources.
  - La protection de l'accès aux ressources.

Systèmes d'exploitation – Fonctions de bases

Un SE à quoi ca sert ?

1. La gestion des processus
   - qui correspondent à l'exécution des programmes.
2. La gestion de la mémoire
   - qui permet de gérer les transferts entre la mémoire principale et secondaire.
3. Le système de fichiers
   - qui offre à l'utilisateur une vision homogène et structurée des données et des ressources : disques, périphériques.
4. Les entrées-sorties
   - qui correspondent aux mécanismes qu'utilisent les processus pour communiquer avec l'extérieur

Systèmes d'exploitation – Fonctions de bases

Un SE à quoi ca sert ?

5. Les réseaux d'ordinateurs
   - avec les protocoles de communication, d'interconnexion et d'application.

6. Les systèmes répartis
   - avec les protocoles d'appels de procédures à distance (RPC).
   - ou les objets distribués.

Les appels systèmes

Qu'est ce que c'est ?

Les appels système

Pour accéder aux services du système d'exploitation, un programme utilisateur doit effectuer **un appel système** qui consiste en :

- Basculer en mode noyau
- Invoquer le système d'exploitation
- Revenir en mode utilisateur
- Retourner le contrôle au programme utilisateur

Exemple : **Lecture ou écriture sur le disque dur**

Les appels systèmes

Un exemple avec le compilateur C

cpt = read (df, tampon, nboctets)

1. Empiler nboctets
2. Empiler &tampon
3. Empiler df
4. Appeler read()
5. Placement du code du *read* dans un registre
6. Déroutement vers le noyau - basculer en mode noyau,
7. Le code noyau examine le numéro d'appel système, utilise une table de pointeurs indexée sur les numéros d'appels système
8. Exécution de l'appel système
9. Rendre le contrôle a la fonction *read* de bibliothèque
10. Rendre le contrôle au programme utilisateur

Les appels systèmes POSIX

A quoi ca sert ?

- Gestion des Fichiers
  - open, close, read, write, dup, dup2, …
- Gestion des Processus
  - fork, pipe, exec, execve, execlp, exit, …
- Communications inter-machines
  - socket(), bind(), listen(), connect(), accept(), send(), recv(), write(), read(), sendto(), recvfrom(), close(), gethostbyname(), gethostbyaddr(), …

Les appels systèmes POSIX

A quoi ca sert ?

- Gestion des Fichiers
  - open,
  - close,
  - read,
  - write,
  - dup,
  - dup2,

Ouvrir un fichier (open)

Ouvre un fichier, qui restera ouvert jusqu'à la fin du processus ou **close**

int open(const char *pathname, int typeOpen);

int open(const char *pathname, int typeOpen, mode_t droitsFic);

- **Pathname (chemin)**=chaine de caractère correspondant au chemin d'accès du fichier à ouvrir.
- **typeOpen** = mode d'ouverture:
  - **O_RDONLY** (lecture seule), **O_WRONLY** (écriture seule) **et O_RDWR** (lecture et écriture)
  - Peut se combiner avec un OU binaire avec un attribut de création

✓ **O_CREAT**: créer si nécessaire le fichier

✓ **O_APPEND** : ouvrir en mode ajout

✓ **O_TRUNC** : Si le fichier existe, est un fichier régulier, et est ouvert en écriture (O_RDWR ou O_WRONLY), il sera tronqué à une longueur nulle (vider avant de pouvoir y écrire de nouvelles données), Si le fichier n'existe pas, il ne sera pas créé sauf si **O_CREAT** est spécifié aussi.

Ouvrir un fichier (open)

- ✓ **O_EXCL** : S'assurer que cet appel crée le fichier : si cet attribut est spécifié en conjonction avec O_CREAT et si le fichier existe déjà, open() échouera.

- **droitsFic** = droits d'accès à utiliser si le fichier
  - doit être créé grâce à **O_CREAT** fonctionne avec les mêmes valeurs octales que la commande **chmod (0777** = tous les droits pour tous)
  - se combinent avec le **umask**
  - si on oublie, open donnera des résultats bizarres.

Remarque

open retourne **-1** si le fichier n'a pas pu être ouvert et un entier positif ou nul sinon. Ce nombre correspond à un index dans le tableau des fichiers ouverts ou chaque case contient le descripteur d'un fichier ouvert.

Ouvrir un fichier (open)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *chemin,
         int typeOpen,
         mode_t droitsFic);
```

Nom du fichier à ouvrir

Mode d'ouverture :
O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_CREAT

Si création : droits appliqués à la création (p. ex. 0750)

Descripteur du fichier ou **–1** en cas d'erreur

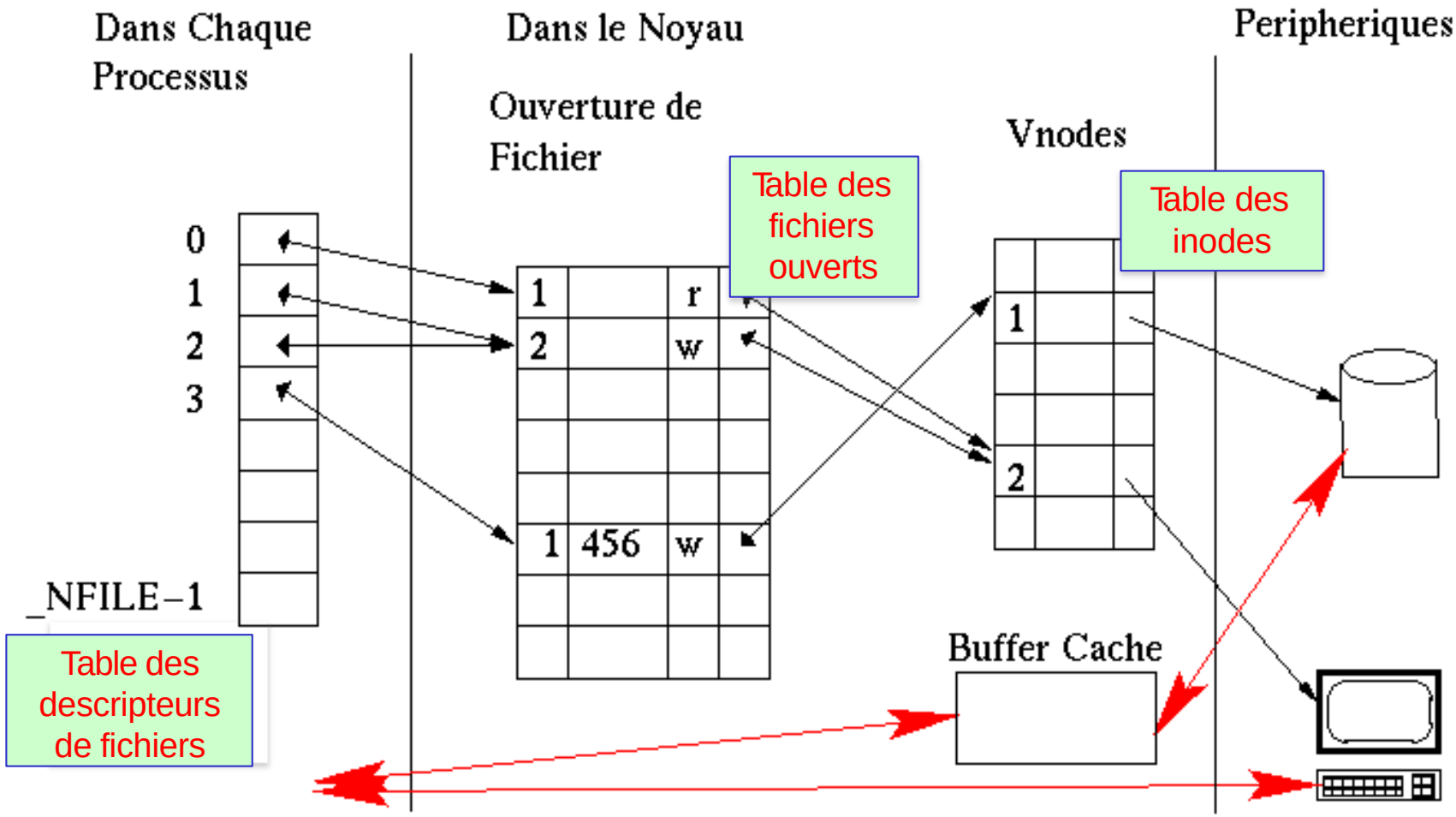
Ouvrir un fichier (open)

Etapes d'un open

- Trouver l'inode du fichier (dans la table des inodes)
  - charger l'inode en mémoire si elle n'y est pas
- Vérifier les droits d'accès
  - la vérification n'est faite qu'à l'ouverture
  - si on enlève les droits sur un fichier ouvert, ça ne bloque pas le processus qui utilise le fichier !
- Allouer une entrée dans la table des fichiers ouverts
- Positionner l'offset courant (0 ou fin de fichier en cas d'ouverture en ajout)
- allouer une place dans la table des descripteurs de fichiers processus
- renvoyer au processus appelant le descripteur obtenu

Ouvrir un fichier (open)

open et les tables (3 tables)



Fermer un fichier ouvert (close)

close

- Ferme le fichier ouvert et libère le descripteur

int close (int fd);

- si **fd** était le dernier descripteur sur le fichier (à l'intérieur du processus) :
  - le compteur associé à l'inode en mémoire est décrémenté; s'il vaut 0, l'inode est déchargée
  - les ressources associées à l'ouverture du fichier sont libérées

Lire à partir d'un fichier ouvert (read)

int read(int fd, void *buf, size_t count);

```
#include <unistd.h>
int read(int desc,
         void *buffer,
         size_t nb);
```

Descripteur du fichier

Adresse de la variable dans laquelle on stocke le résultat de la lecture

Nombre d'octets à lire

Nb d'octets lu
-1 en cas d'erreur
0 si fin de fichier

Attention : **read** lit et déplace le curseur !

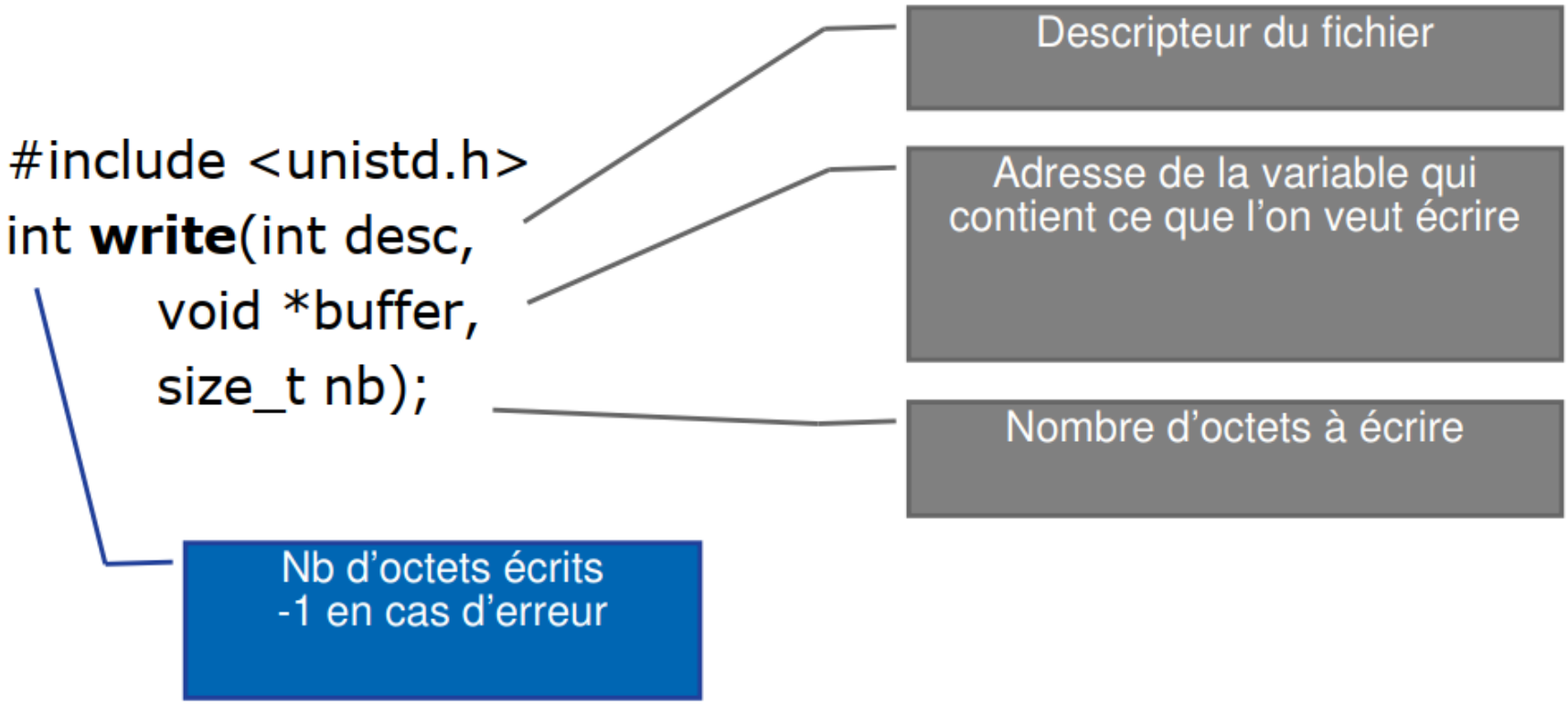
Lire à partir d'un fichier ouvert (read)

int read(int fd, void *buf, size_t count);

- Peut retourner moins que demandé en cas :
  - de fin de fichier
  - de lecture sur un tube ou un terminal
  - d'interruption par un signal
- Vérification de la validité du descripteur et du bon mode d'ouverture
- mais les droits d'accès de l'utilisateur ne sont pas testés

Ecrire dans un fichier ouvert (write)

ssize_t write(int fd, const void *buf, size_t count);



Attention : **write** écrit et déplace le curseur !

Ecrire dans un fichier ouvert (write)

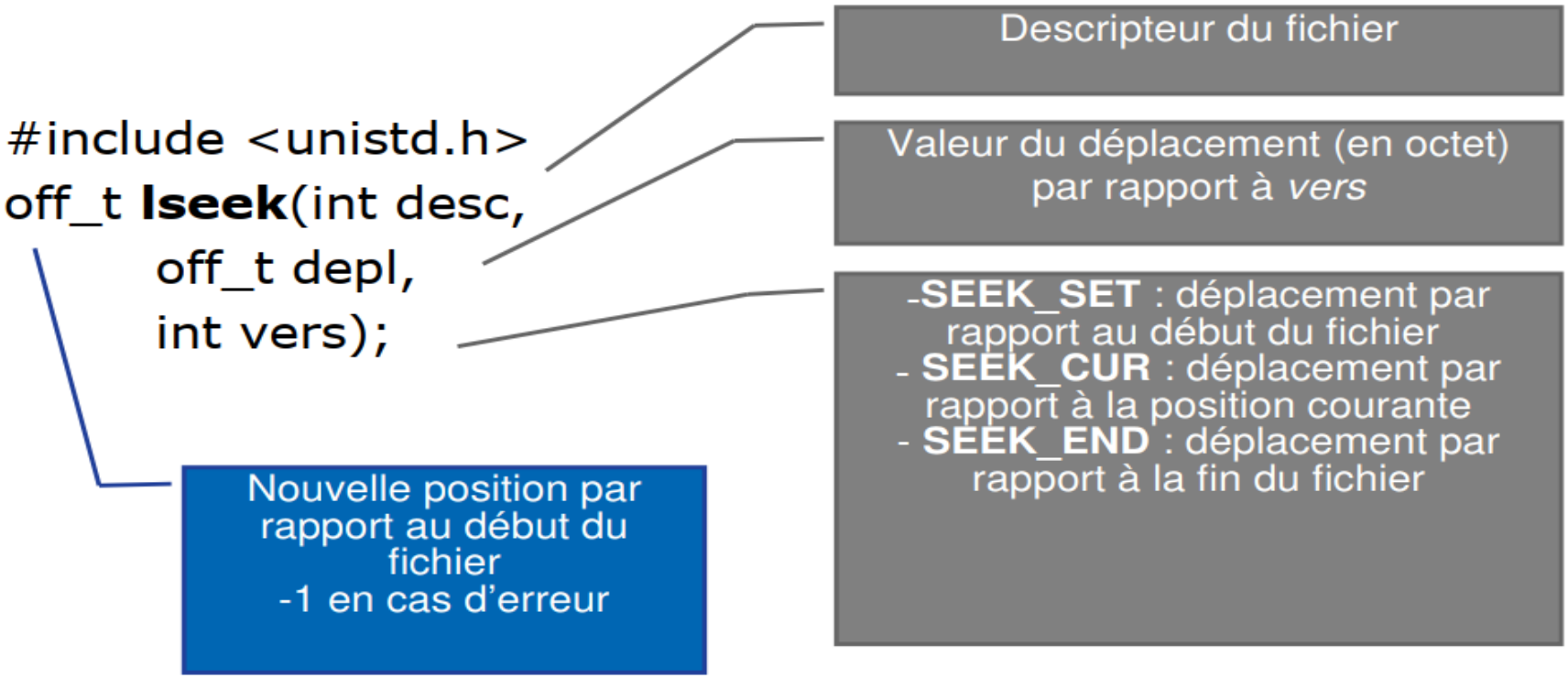
```
ssize_t write(int fd, const void *buf, size_t count);
```

- Peut écrire moins que demandé en cas :
  - plus de place sur le périphérique
  - quota disque du processus dépassé
  - interruption par un signal
- Dans un fichier, la modification de la position et l'écriture sont faites de façon atomique

Se positionner dans un fichier ouvert (lseek)

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

- Déplacement de la position courante : **lseek**



Se positionner dans un fichier ouvert (lseek)

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

- **whence**=SEEK_SET, SEEK_CUR ou SEEK_END
- peut aller au-delà de la taille du fichier (sans la modifier); une écriture modifiera alors la taille réelle

Dupliquer le descripteur d'un fichier ouvert(dup)

int dup(int oldfd);

- Duplique un descripteur de fichier valide
- La copie reçoit le plus **petit** numéro de descripteur **disponible**
- Pénible quand on veut un numéro précis

Dupliquer le descripteur d'un fichier ouvert(dup2)

int dup2(int oldfd, int newfd);

- Duplique un descripteur de fichier valide
- La copie reçoit le numéro **newfd**
- Si **newfd** était ouvert, il est fermé automatiquement
- Très utile pour les **redirections**

```
int fd=open("toto",O_RDONLY);
if (fd!=-1) {
    dup2(fd,0); /* utilise "toto" comme entrée standard */
}
```

Dupliquer le descripteur d'un fichier ouvert

dup et dup2 et lseek

- Le descripteur et sa copie partagent la position modifiable par **lseek**
- Risque de concurrence
- Après une duplication, il faut fermer deux fois le descripteur pour fermer complètement

Dupliquer le descripteur d'un fichier ouvert

dup et dup2

Par défaut un processus dispose de:

- **stdin** qui représente l'entrée standard dont le **fd** est **0**
- **stdout** qui représente la sortie standard dont le **fd** est **1**
- **stderr** qui représente la sortie d'erreur dont le **fd** est **2**

Les appels systèmes POSIX

A quoi ca sert ?

- Gestion des Fichiers
  - open, close, read, write, dup, dup2, stat, …
- Gestion des Processus
  - fork, pipe, exec, execve, execlp, exit, …
- Communications inter-machines
  - socket(), bind(), listen(), connect(), accept(), send(), recv(), write(), read(), sendto(), recvfrom(), close(), gethostbyname(), gethostbyaddr(), …

Le processus

- **Création d'un processus**
  - Lorsqu'on entre une commande, le shell lance un processus pour l'exécuter.
  - Le shell attend la fin du processus, puis attend la commande suivante.
  - Un programme qui s'exécute avec un jeu de données particulier, est un processus.
- **Arborescence de processus.**
  - Chaque processus a un père, celui qui l'a lancé.
  - Un processus a un numéro permettant son identification, **PID**.
  - Le numéro du processus du père est le **PPID**.
  - Un **processus père** peut avoir plusieurs **processus fils**.
  - Sous UNIX, un premier processus **init (ou systemd)** est créé avec un **PID** égal à **1**. C'est l'ancêtre de tous les processus

Duplication d'un processus

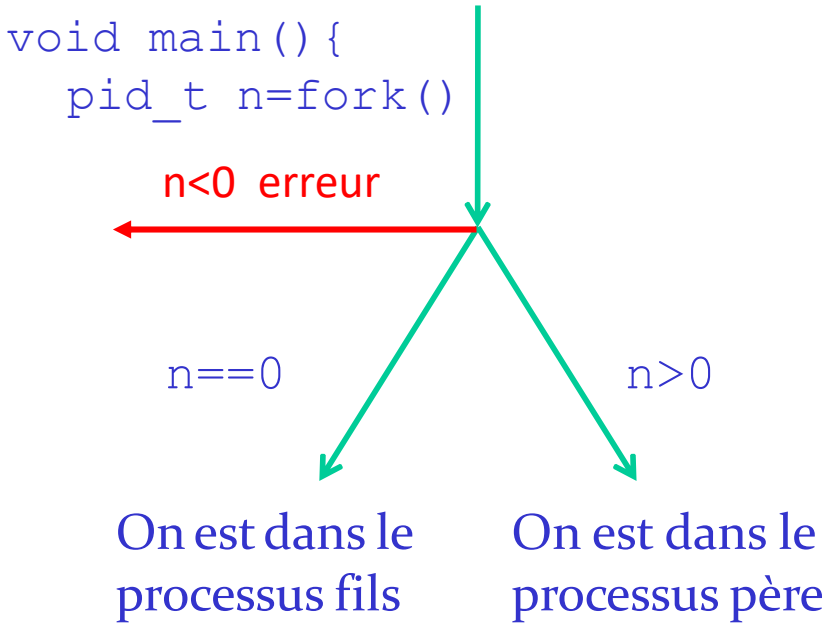
pid_t fork();

- L'appel système **fork** est le seul moyen de créer un processus à bas niveau
  - Crée un processus fils de ce processus, et le lance.
  - L'exécution continue dans les deux processus, à l'instruction qui suit le **fork**
  - Les **PID** et **PPID** sont les seules informations différentes entre les deux processus
- Valeur retournée par **fork()** :
  - Dans le processus père, retourne le PID du fils.
  - Dans le processus fils, retourne 0.
  - Si le fork échoue, renvoie -1 (seul le père existe, alors).

Duplication d'un processus

La manière la plus courante d'utiliser fork

- L'appel système **fork** duplique / clone le processus appelant




```
void main(){
  pid_t
  n=fork();
  if(n!=0){
     if(n<0){
   //gestion des erreurs
     }else{
       //programme du papa
     }
  }else{
    //programme du fils
  }
}
```

Récupération du PID

getpid, getppid, getgrp
pid_t getpid (void); pid_t getppid (void); pid_t getpgrp (void);

- L'appel à **getpid** retourne le **pid** du processus **courant**
- **getppid** retourne le **pid** du processus **père.**
- **getpgrp** retourne le **pid** du **groupe** du processus courant

Duplication du PID

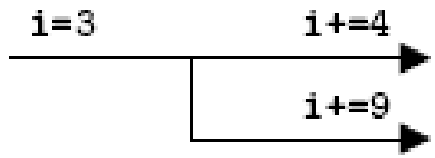
Un exemple d'utilisation de fork, getpid, getppid

- Clonage, c'est une recopie complète et indépendante

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
   pid_t pid;
   int i=3;
   pid = fork();

   if(pid<0)
      {
        printf("erreur de creation");
        exit(EXIT_FAILURE);
      }
      else
      {
          if(pid==0)
          {
          printf("je suis le fils, mon PID est %d ; mon pere est %d\n",  getpid(), getppid()) ;
           i += 9 ;
          }
          else
          {
          printf("je suis le pere, mon PID est %d\n", getpid());  i += 4 ;

          }

      }
}
```

je suis le fils, mon PID est 10271 ; mon pere est 10270
pour le pid 10271, i = 12

je suis le pere, mon PID est 10270
Pour le pid 10270, i = 7



L’attente de la mort d’un fils (wait)

pid_t wait(int *status)

- La primitive **wait** permet à un processus d’attendre la fin d’un de ses fils.
  - S’il n’y a **pas** de **fils** ou erreur, retourne **-1.**
  - Sinon, retourne le **PID du fils** qui s’est terminé.
- La fonction **wait** suspend l’exécution du processus courant jusqu’à ce que l’un de ses fils se termine (ou jusqu’à la réception d’un signal)
- Si un processus fils est déjà mort la fonction revient immédiatement
- Toutes les ressources utilisées par le fils sont libérées

L’attente de la mort d’un fils (wait)

pid_t wait(int *status)

- Si **status ≠ NULL**, les informations sur la terminaison du fils y sont stockées
  - **WIFEXITED(status)**: est non nul (retourne une valeur différentes à 0) si le fils s’est terminé normalement, sinon 0.

Macron	Signification
WIFEXITED(status)	vrai si le fils s’est terminé normalement (exit)
WEXITSTATUS(status)	code de retour du fils (si normal)
WIFSIGNALED(status)	vrai si le fils a été tué par un signal
WTERMSIG(status)	numéro du signal qui a tué le fils

L’attente de la mort d’un fils (wait)

void exit(int status)

- **status** permet d’indiquer au processus père qu’une erreur s’est produite.
- **status=0** si pas d’erreur.

L'attente de la mort d'un fils (waitpid)

pid_t waitpid(pid_t pid, int* status, int options)

- La fonction **waitpid** suspend l'exécution du processus courant jusqu'à ce que le processus fils de numéro **pid** se termine (ou jusqu'à réception d'un signal)
- La valeur de **pid** peut être :
  - < -1 : attendre la fin de n'importe quel processus fils appartenant au groupe d'ID pid (n'importe quel processus fils dont GID égal à pid)
  - -1 : attendre n'importe quel fils (identique au **wait**)
  - 0 : attendre la fin de n'importe quel processus fils du même groupe appelant
  - > 0 : attendre la fin du processus de numéro **pid**
- Il existe différentes options, la plus utile : **WNOHANG** qui permet de ne pas bloquer si aucun fils n'est mort

Temporisation (sleep)

int sleep(int seconds)

- Similaire à la commande shell **sleep**.
  - Le processus qui appel **sleep** est bloqué pendant le nombre de secondes spécifiées.

- La différence avec **wait()** :
  - **wait** : bloque le processus jusqu'à la fin d'un fils, alors que sleep() bloque pour un temps spécifié.

La temporisation Vs. l'attente

Un exemple d'utilisation de wait et sleep

```
int pidfils=fork();

if(pidfils!=0) {
  printf("Je suis le père.\n");
  wait(0);
  printf("mon fils a terminé.\n");
} else {
  printf("je suis le fils.\n");
  sleep(2);
  printf("arrêt du fils.\n");
  exit(0);
}
```

Résultat

- $ fork
  Je suis le père
  je suis le fils
  arrêt du fils
  mon fils a terminé
  $

Lancer un autre processus

Appel système exec sous Unix

- Le système UNIX offre une famille d'appels système **exec** qui permettent à un processus de remplacer son code exécutable par un autre programme spécifié par **path** ou **file**. sans changer de numéro de processus (**PID**).
- Valeur de retour : en cas de succès **AUCUNE** le code ayant fait l'`exec` n'existe plus, en cas d'échec –1

```
int execl(char* path,char* arg0,char* arg1, …, NULL);
int execv(char* path,char* arg[]);
int execle(char* path,char* arg0,…, NULL, char* envp[]);
int execve(char* path,char* arg[],char* envp[]);
int execlp(char* file,char* arg0,char* arg1, …, NULL);
int execvp(char* file,char* arg[]);
```

Lancer un autre processus

Appel système exec sous Unix

- La fonction **execl** prend en paramètre une liste des arguments à passer au programme (liste terminée par **NULL)**.
- Le premier paramètre est une chaîne qui doit contenir le chemin d'accès complet au fichier exécutable ou au script shell à exécuter.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
/* dernier élément NULL, OBLIGATOIRE */
execl("/usr/bin/emacs", "emacs", "fichier.c", "fichier.h", NULL);
perror("Problème : cette partie du code ne doit jamais être exécutée");
return 0;
}
```

Lancer un autre processus

Appel système exec sous Unix

- La fonction **execv** a différence avec execl est que l'on n'a pas besoin de connaître la liste des arguments à l'avance (ni même leur nombre). Cette fonction a pour prototype :

int execv(const char* application, const char* argv[]);

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
char * argv[] = {"emacs", "fichier.c", "fichier.h", NULL}
/* dernier élément NULL, obligatoire */
execv("/usr/bin/emacs", argv);
puts("Problème : cette partie du code ne doit jamais être exécutée");
return 0;
}
```

Lancer un autre processus

execl, execv, execle, execve, execlp, execle

Primitives	Format d'argument	Passage d'environnement	Recherche avec PATH
execl	Liste	Automatique	Non
execv	Tableau	Automatique	Non
execle	Liste	Manuel	Non
execve	Tableau	Manuel	Non
execlp	Liste	Automatique	Oui
execvp	Tableau	Automatique	Oui

- Les différents noms des fonctions **exec** sont mnémoniques :
  - **l :** liste d'arguments
  - **v :** arguments sous forme de vecteur
  - **p :** recherche du fichier avec la variable d'environnement PATH
  - **e :** transmission d'un environnement en dernier paramètre, en remplacement de l'environnement courant

Lancement d'une commande (system)

int system(const char * commande)

- La fonction system de la bibliothèque **stdlib.h** permet directement de lancer un programme dans un programme **C** sans utiliser **fork** et **exec**.
- Crée un nouveau processus « **/bin/sh** » qui exécute la commande **(/bin/sh –c** commande)

- Le processus appelant cette fonction reste bloqué jusqu'à la fin de l'exécution du processus

Exemple:

```
system("clear");
```