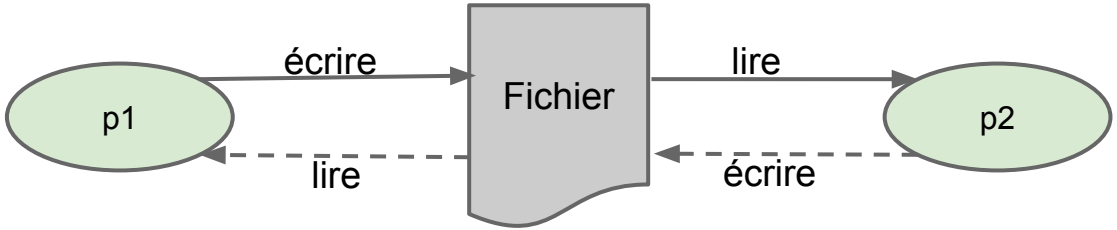
Principes des Systèmes d'exploitation

IUT de Villetaneuse
D. Buscaldi

4. Communication Inter-Processus: Les Tubes

Échange d'informations entre processus

- Idée: utiliser un fichier intermédiaire où écrire les données



- Principe très simple mais:
  - Problème de synchronisation
    - Exemple: p1 veut lire mais p2 n'a pas encore écrit
- Solution: **tube**

Tube

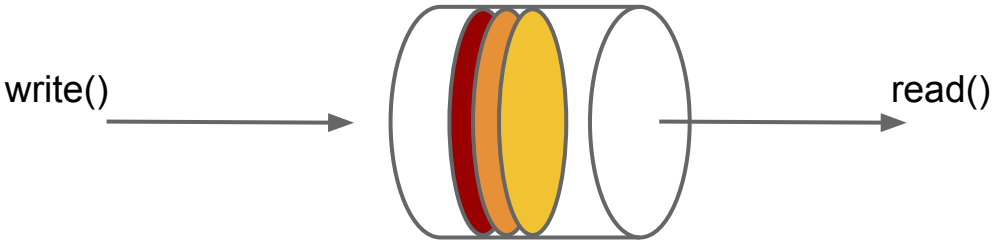
- Fichier:
  - Sans nom (impossible de l'ouvrir avec `open()`)
  - De capacité limitée
  - Géré comme une file FIFO (pas d'accès random avec lseek)

- C'est le système de gestion des fichiers qui gère les tubes
  - Les tubes peuvent être associés aux entrées/sorties stdin et stdout

- Lecture/écriture
  - Avec les primitives `read()` et `write()`

Tube

- Mécanisme de communication unidirectionnelle



- 2 extrémités: une pour écrire et une pour lire - donc:
  - 2 descripteurs de fichier (un pour le processus lecteur et un autre pour l'écrivain)
  - 2 entrées dans la table des fichiers ouverts
- Lecture possible une seule fois (pas de relecture)

Tube

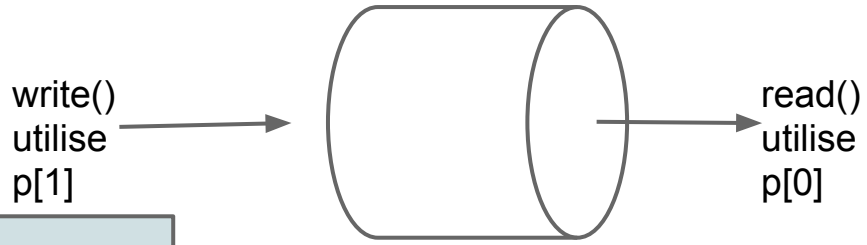
- On ne peut pas écrire s'il n'y a pas au moins un lecteur
  - Signal SIGPIPE
- S'il n'y a pas d'écrivain, la `read()` donnera 0
- Accès possible uniquement par descripteurs
- Communication seulement entre les processus descendants du créateur du tube

Création d'un tube

- Primitive `int pipe(int p[2])`:

  p[0] (ou p[STDIN_FILENO]) pour la lecture

  p[1] (ou p[STDOUT_FILENO]) pour l'écriture




```
#include <stdlib.h>
#include <unistd.h>
/* test capacité */
int main(void) {
    int p[2]; /* descripteurs pour le tube */
    int i;
    if (-1==pipe(p)) {
       exit(1); /* erreur de création */
    } else {
      for (i=1;;i++) {
       write(p[1],"",1); /* Écrit un '\0' */
       printf("octets écrits avant blocage=%d\n",
      }
    }
    return 0;
}
```

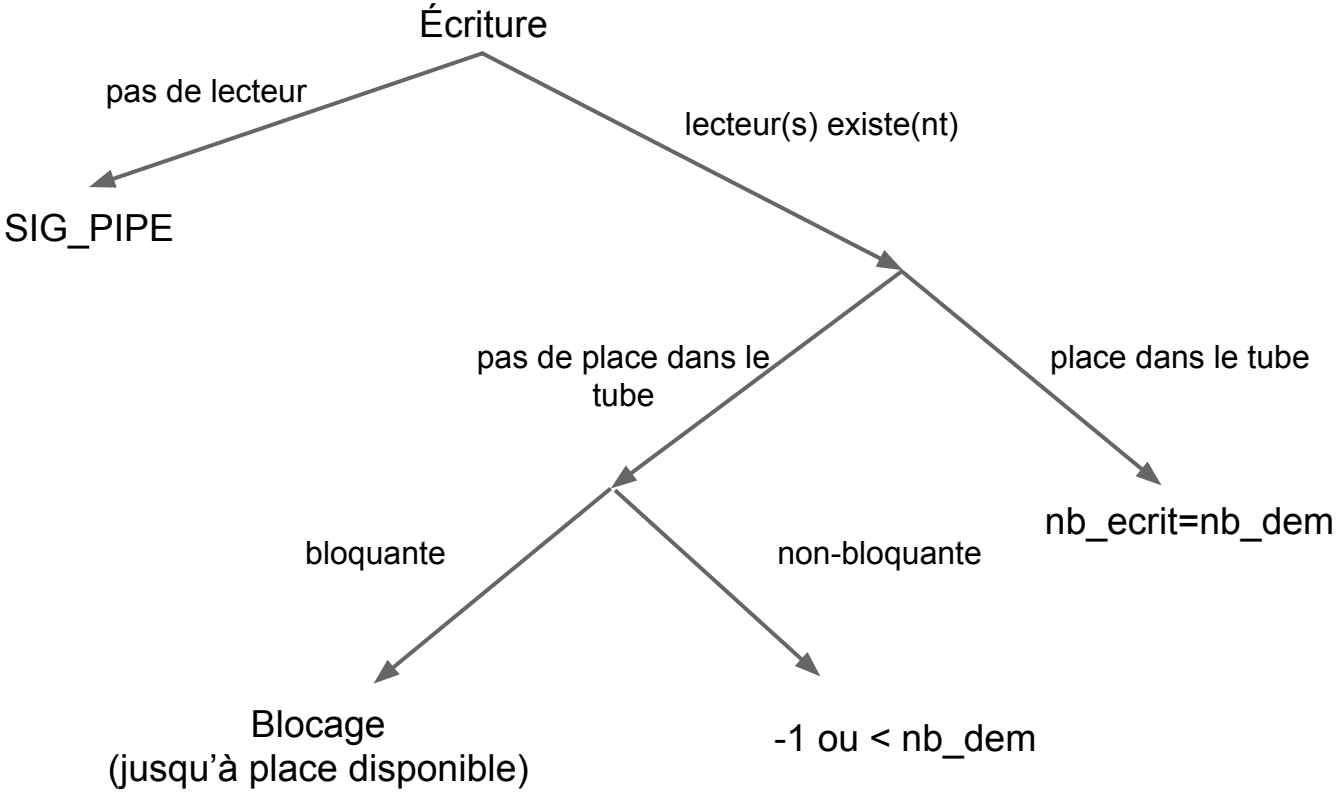
descripteur du tube, buffer d'écriture, nombre d'octets

```
octets ecrits avant blocage=65534
octets ecrits avant blocage=65535
octets ecrits avant blocage=65536
^Z
[1]+  Stoppé                 ./pipe_capacity
$ lsof -p 6169
COMMAND     PID   USER   FD   TYPE DEVICE SIZE/OFF    NODE
NAME
pipe_capa 6169 dbuscaldi      3r  FIFO    0,8   0t0 2611672
pipe
pipe_capa 6169 dbuscaldi      4w  FIFO    0,8   0t0 2611672
pipe
```

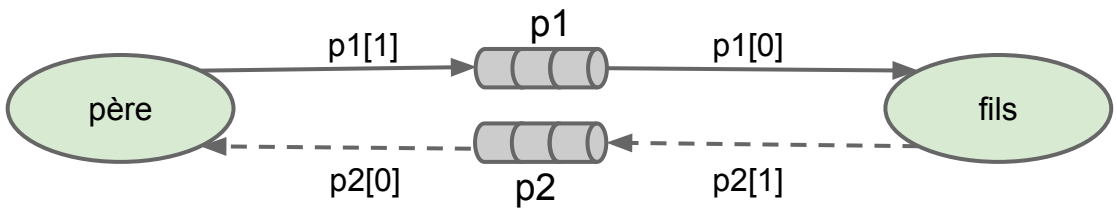
Lecture/Écriture d'un tube

```
/*Possibilité d'écriture/lecture non bloquante*/
fcntl( p[0], F_SETFL, O_NONBLOCK) ;

char buf[taille] ; /* buffer */
int nb_lu ; /* Nb. d'octets réellement lus */
int nb_dem ; /* Nb. d'octets demandés au plus */
int nb_écrit; /* Nb. d'octets écrits */
/* lecture */
nb_lu = read (p[0] , buf , nb_dem ) ;

/* écriture */
nb_ecrit = write(p[1], buf, nb_dem) ; /* ATOMIQUE si nb_dem < PIPE_BUF (1 à 8k) */

/* fermeture */
close(p[0]);
```

Lecture/Écriture d'un tube



Risque de blocage



Attention à l'ordre des opérations!!!

```
int p1[2], p2[2];
pipe(p1); pipe(p2);
if (fork()==0) {
     read (p1[0], ch, 1);
     write(p2[1], ch, 1);
} else {
     read (p2[0], ch, 1);
     write (p1[1], ch, 1);
}
```

Blocage!

Inversion d'ordre evite le blocage

Tubes Nommées

- Tube ordinaire + nom dans un répertoire
- Existe même après l'usage
  - (Mais vide)
- Création:
  - Primitive:

```
int mkfifo (const char *ref, mode_t mode);
```

  - Commande:

```
$> mkfifo -m mode ref
```

- Ouverture avec open():

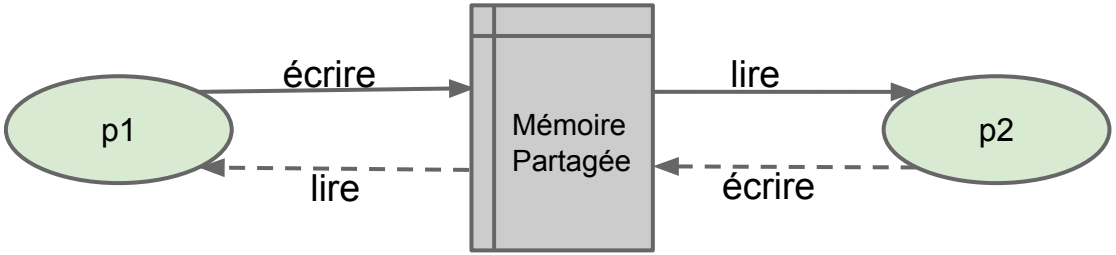
```
open("tube_n", O_WRONLY); /* écriture seule: sans lecteur se bloque */
open("tube_n", O_RDONLY); /* lecture seule: sans écrivain se bloque */

/* lecture seule, non-bloquant */
desc = open("tube_n", O_RDONLY | O_NONBLOCK);
```

Exemple

```
void main ( ) {
      int i , nb ;
      int pf[2] , fp[2] ; char chaine[80]; pid_t pif ;
      pipe (pf) ; pipe (fp); //pf: père-> fils, fp: fils -> père
      pif = fork() ;
      if ( pif > 0 ) { // Père: ne lit pas de pf; n'écrit pas dans fp
            close(pf[0]) ; close(fp[1]) ;
            printf(" Entrez une séquence d'entiers :\n");
            while ( scanf("%d",&i) != EOF )
                  write (pf[1],&i,sizeof(int));
            close(pf[1]); // fin de la séquence
            read(fp[0], &chaine , 80 ) ; // lecture du nb.
            printf ("%s",chaine);
      } else { // Fils
            close(pf[1]) ; close(fp[0]) ;
            while ( read( pf[0], &i , sizeof(i)) != 0 ) nb++;
            close(pf[0]);
            sprintf(chaine," %d entiers reçus \n\0",nb);
            write(fp[1], chaine , strlen(chaine)+1 );
            close(fp[1]);
      }
}
```

Autres Mécanismes de Communication

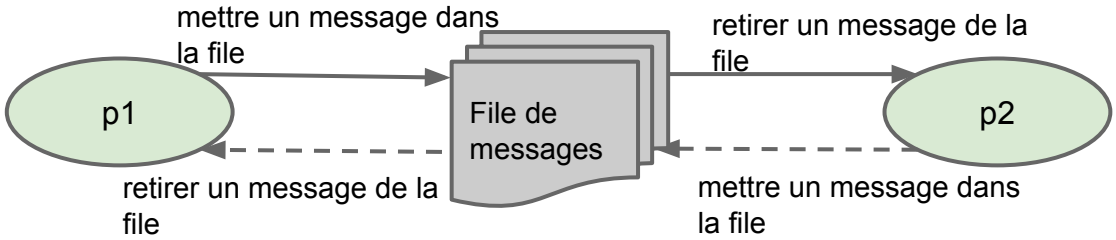
- La communication par tubes est un cas particulier de communication par mémoire commune:
- Communication par la mémoire partagée:



- Besoin de créer/attacher/detacher des segments de mémoire partagée
  - shared memory (shm)
  - En-tête shm.h, appels système shm…(...)

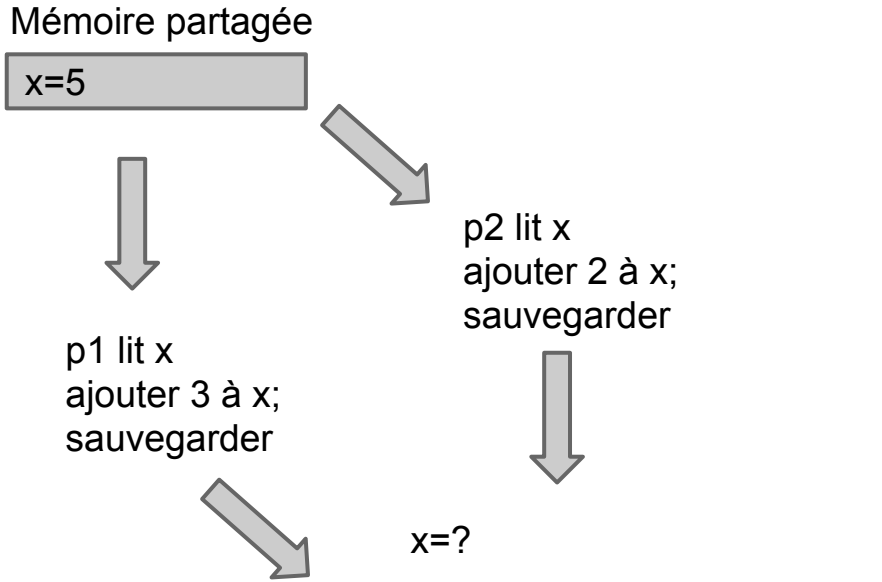
Autres Mécanismes de Communication

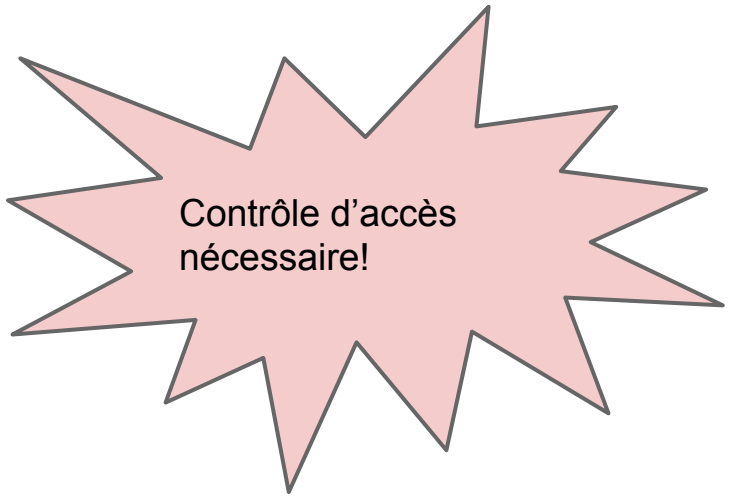
- Communication par échange de messages:



- Utilisable aussi à travers d'un réseau
- Besoin de créer une file de messages, envoyer/recevoir un message
  - En-tête msg.h, appels système msg…(...)

Accès Concurrent





Exclusion Mutuelle

- Pour la mémoire partagée: sémaphores
  - dans une table du SE
- Pour les fichiers: verrous
  - objets du SF
- IPC: Sous-système de communication inter-processus
  - {shm, sem, msg} = IPC
- Commandes linux:
  - ipcs (info)
  - ipcrm (remove)

5. Système de Fichiers

Système de fichiers

- Quantité de données sur un disque dur: toujours plus grand
  - Il faut un moyen de les organiser
- Un système de fichier est une façon d'organiser les fichiers dans une périphérique de mémoire secondaire (disque dur, USB, etc…)
  - Structures à **arbre** (b-tree)
  - Structures à **tableau** (flat)
- Chaque système définit en général:
  - Taille maximale des fichiers et des partitions
  - Gestion des droits d'accès
  - Journalisation: utilisation d'un journal qui trace les opérations d'écriture tant qu'elles ne sont pas terminées
    - Objectif: garantir l'intégrité des données en cas d'arrêt brutal
  - Autres caractéristiques genre le type de caractères permis dans les noms de fichiers, attributs, etc.

Exemples de SF

Nom	Taille max fichier	Taille max partition	Journalisé?	Géstion des droits?	Notes
ext2fs	2Tio	4Tio	Non	Oui	Linux (1993)
ext3fs	2Tio	4Tio	Oui	Oui	ext2 + journalisation
ext4fs	16Tio	1Eio	Oui	Oui	Amélioration d'ext3 en attente de Btrfs
ReiserFS	8Tio	16Tio	Non	Oui	(2001) Optimisé pour petit fichiers, permet la modification "à chaud" de la table de partitions
FAT	2Gio	2Gio	Non	Non	MS-DOS (1984)
FAT32	4Gio	8Tio	Non	Non	Windows95 (1996)
NTFS	16Tio	256Tio	Oui	Oui	WindowsNT (1993), conçu pour multi-utilisateur
HFS+	8Eio	8Eio	Oui (10.3)	Oui	MacOSX (1998)
ZFS	16Zio	256Zio	Oui	Oui	Solaris (2005) réparation automatique, copy-on-write, compression, chiffrement...

Notion de Fichier

- Unité de stockage d'information dans le système

- Dans UNIX les fichiers ordinaires sont une suite de code:
  - Fichier de texte: code ASCII
  - Fichier binaire: tout autre code

- Information stockée sur le disque (Mémoire Secondaire) et échangée fréquemment avec les périphériques
  - Les SE ont tendance à représenter les périphériques et les ressources comme des fichiers
    - Uniformisation et facilité de représentation

Types de fichiers en Linux

- **Fichiers réguliers** : stockage d'information
- **Répertoires** : collection de références
- **Liens symboliques** : chemin d'accès d'une référence

```
-rw-r--r-- ... /etc/passwd
drwxr-xr-x ... /etc
```

```
lrwxrwxrwx ... S03xinetd -> ../init.d/xinetd
```

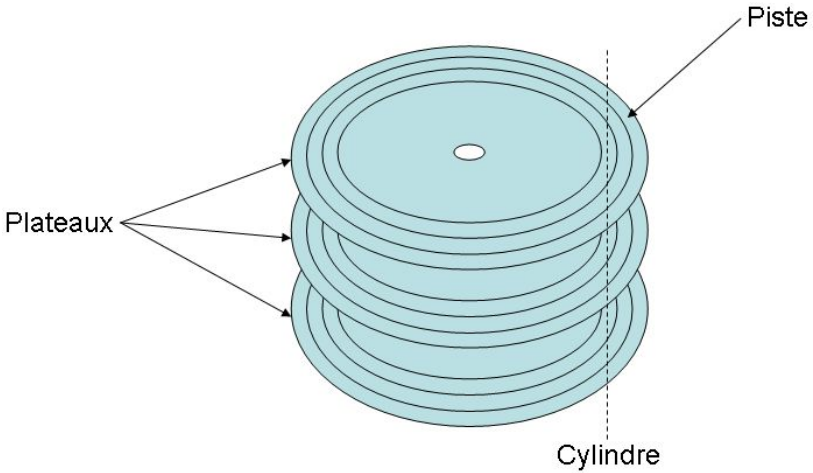
- **Tubes** (nommés)
- **Sockets**

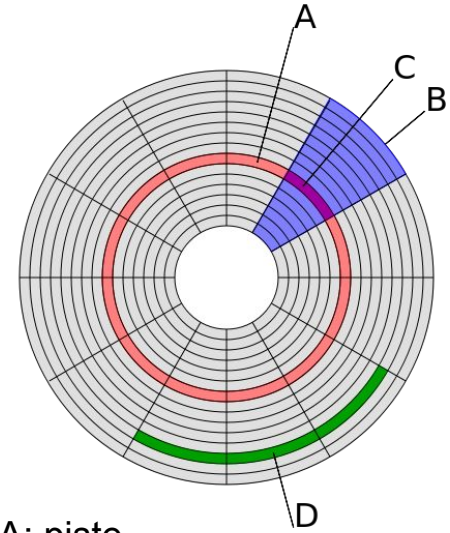
```
prw-rw---- ...  fifo
srwxrwxrwx ... /tmp/.X11-unix/X0
```

- **Fichiers spéciaux (Device Files)**
  - En mode **bloc** : disques
  - En mode **caractère** : terminaux, mémoire

```
brw-rw---- ... /dev/sda
crw-rw-rw- ... /dev/tty
```

Structure d'un disque dur





- A: piste
- B: secteur géométrique
- C: secteur (de disque) = bloc physique
- D: bloc (logique)

Secteur: la plus petite unité physique de stockage sur un support

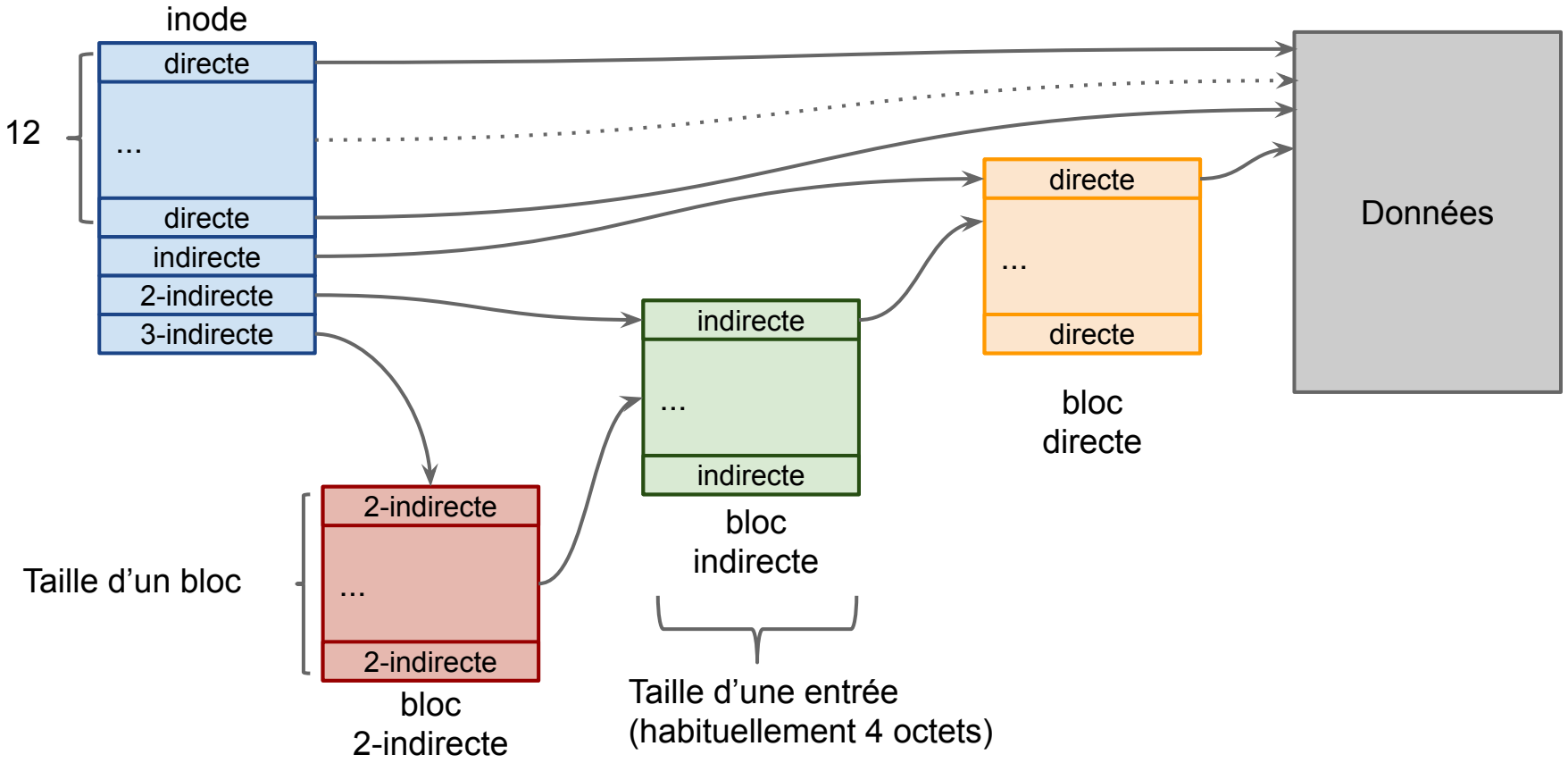
Bloc (logique): la plus petite unité de stockage d'un système de fichiers

Inode

- Structure qui contient toutes les informations sur un fichier donné
  - le type du fichier ;
  - les droits d'accès au fichier ;
  - le propriétaire, le groupe ;
  - la date du plus récent accès en lecture et écriture ;
  - la date la plus récente de mise à jour de l'inode ;
  - la taille exprimée en octet ;
  - Le nombre de blocs physiques utilisé par le fichier (incluant les éventuels blocs indirects contenant les pointeurs vers les blocs de données) ;
  - le nombre de références sur l'inode ;
  - le tableau des pointeurs référençant les blocs de données du fichier
  - …
- Pour avoir des information sur les inodes:

```
$> stat nom_fichier
```

Inode- Adresses des blocs



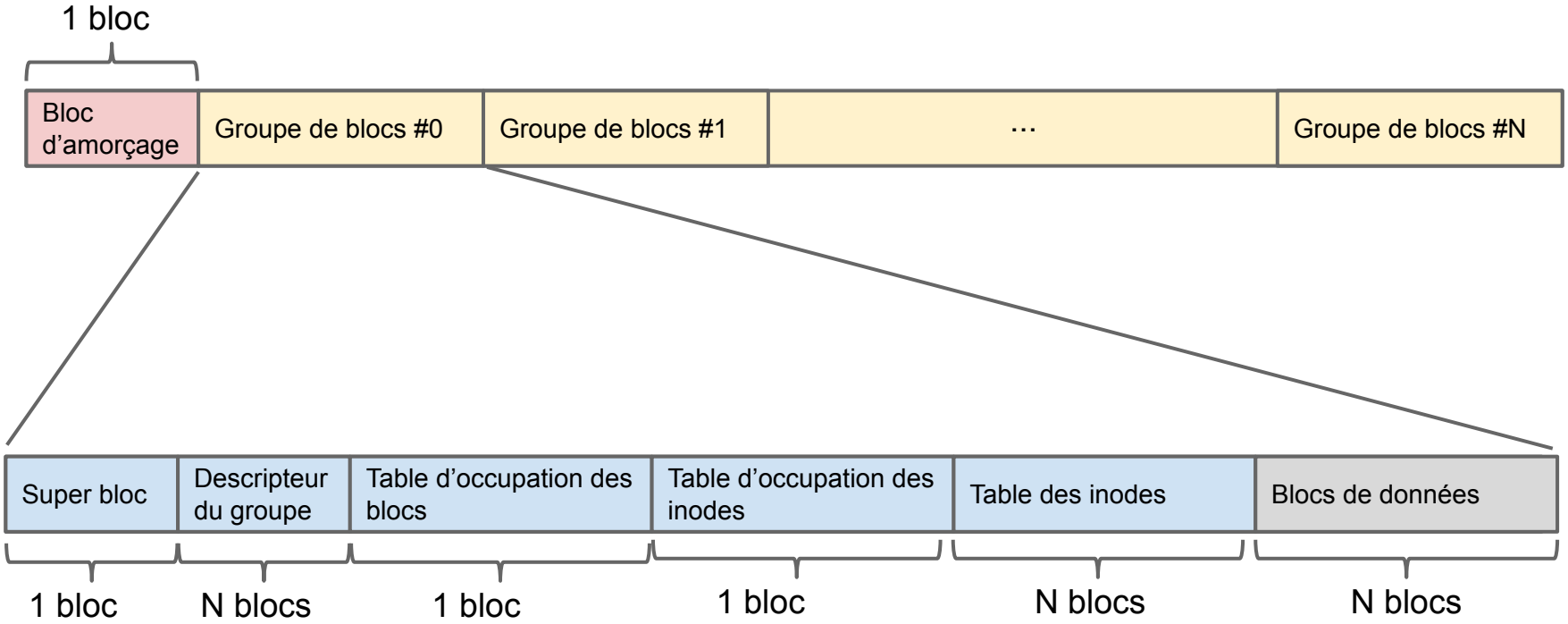
Taille max des fichiers selon adressage

- On considère chaque entrée d'un bloc sur 4 octets:

Taille d'un bloc	T_max d'un fichier (Adr. directe)	T_max d'un fichier (Adr. indirecte)	T_max d'un fichier (Adr. 2-indirecte)	T_max d'un fichier (Adr. 3-indirecte)
1024 octets (1Kio)	12*1024 = 12Kio	12Kio + (1024/4) = 268 Kio	256*256 + 268 Kio = 64,26 Mio	256^3 + 64,26Mio = 16,06Gio
2048 octets (2Kio)	24Kio	1,02Mio	513,02Mio	256,5Gio
4096 octets (4Kio)	48Kio	4,04Mio	4Gio	~4Tio

Tout cela en théorie car les développeurs ont utilisé un entier signé… donc perte d'un bit d'information...

Organisation des partitions ext2



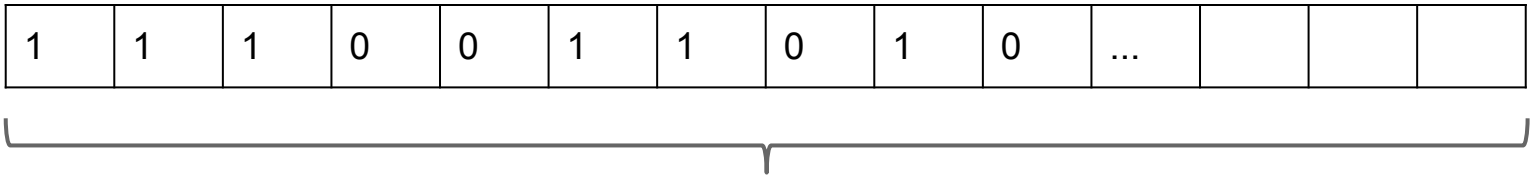
Super bloc

- Contient les informations relatives au Système de Fichiers:
  - Nombre de blocs sur le disque, taille des blocs, inodes libres, etc.…

```
struct ext2_super_block {
	__le32	s_inodes_count;		/* Inodes count */
	__le32	s_blocks_count;		/* Blocks count */
	__le32	s_r_blocks_count;	/* Reserved blocks count */
	__le32	s_free_blocks_count;	/* Free blocks count */
	__le32	s_free_inodes_count;	/* Free inodes count */
	__le32	s_first_data_block; /* First Data Block */
	__le32	s_log_block_size;	/* Block size */
	__le32	s_blocks_per_group; /* # Blocks per group */
	__le32	s_frags_per_group;	/* # Fragments per group */
	__le32	s_inodes_per_group; /* # Inodes per group */
...
};
```

Table d’occupation des blocs et des inodes

- Indiques les blocs (ou inodes) occupés
- Implémentées comme des bitmaps (1=occupé, 0=libre)



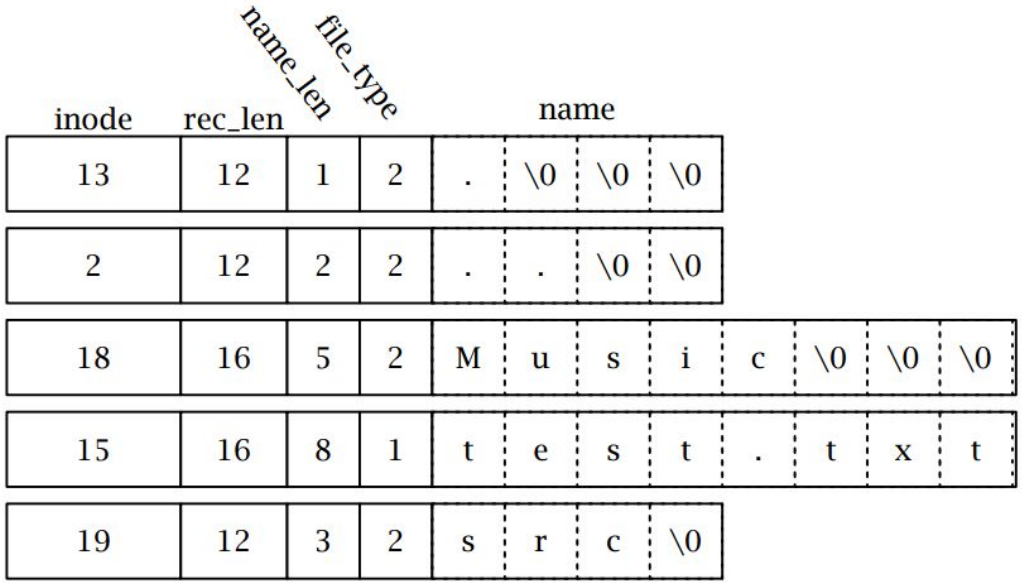
- Si bloc = 1Ko, alors taille bitmap 8192 bits

=> Le groupe ne peut pas avoir plus de 8192 blocs

Répertoires

- Utilisés pour l'accès au contenu d'un fichier
- Fichier qui contient une liste de fichiers contenus dans le répertoire

```
struct ext2_dir_entry_2 {
__le32 inode; /* Inode number */
__le16 rec_len; /* Directory entry length */
__u8 name_len; /* Name length */
__u8 file_type;
charname[EXT2_NAME_LEN]; /* File name */
};
```



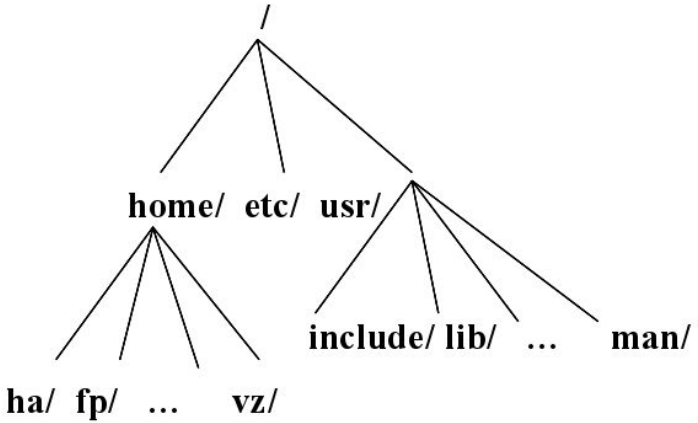
Journaling

- Idée : marquer les changements à faire dans un journal avant de les effectuer.
- Formes physiques possibles d'un journal :
  - Fichier standard
  - Zone réservée du disque
- Types de journaux :
  - Journal complet : contient les modifications faites aux données et aux méta-données
  - Méta-journal : contient seulement les modifications faites aux méta-données

Montage/démontage des disques logiques

Organisation générale des SF :

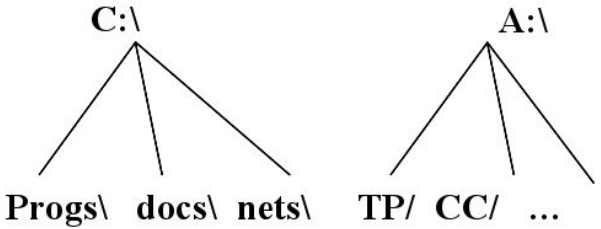
sous Unix



Les fichiers ou les SF
forment une arborescence.
(en réalité un graphe orienté acyclique)

⇒ Nécessité d 'attacher / de détacher
des sous arbres

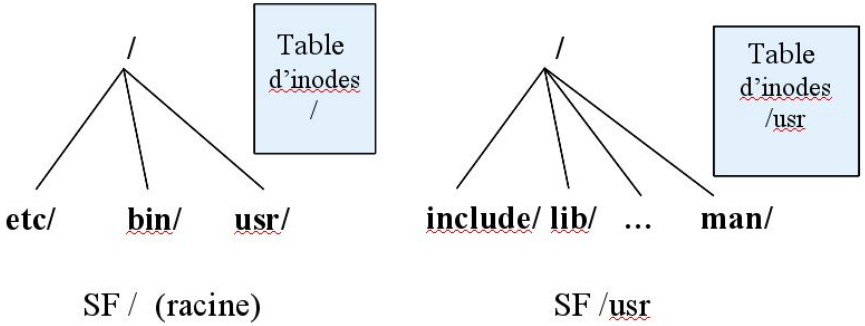
sous MSDOS

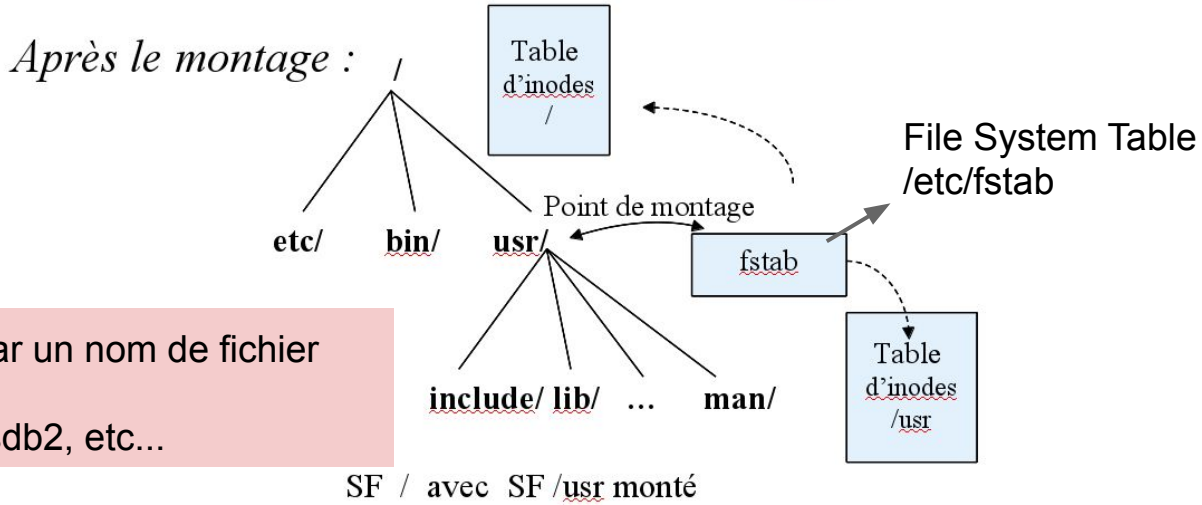


Sous DOS le systéme du fichier
est une forêt

⇒ Obligation de connaître le
périphérique

Montage/démontage des disques logiques





Le SF est désigné par un nom de fichier spécial
ex: /dev/hda1, /dev/sdb2, etc...

6. Signaux

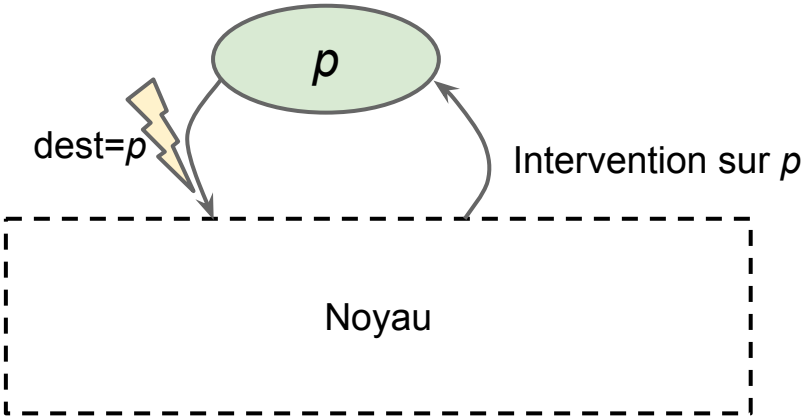
Signal

- Mécanisme asynchrone permettant d'alerter un processus

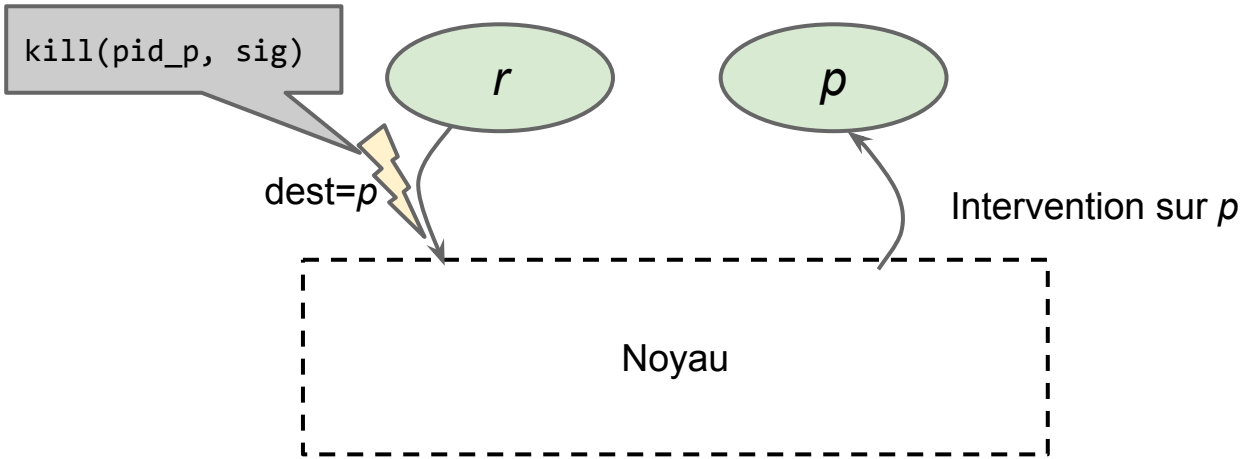
- Pourquoi on alerte un processus?
  - Événement *extérieur* au processus:
    - fils terminé, caractère d'interruption (Ctrl-D, Ctrl-C, ...) frappé…
  - Événement *intérieur* au processus:
    - instruction illégale, violation de segment, erreur en format à virgule flottante…
  - Pour *synchronsation*

Événement interne

Violation de segmente, floating p., exception...



Événement extérieur ou Synchronisation



Envoi de signaux

- Si le propriétaire est le même:
  - Commande:
  - Primitive:

```
$ kill -signum pid
```

```
int kill (pid_t pid, int sig);
```

- Si pid > 0, signal envoyé au processus avec ce pid
- Si pid=0, signal envoyé à tous les processus du même groupe de l'appelant
- Si pid < 0, signal envoyé à tous les processus du groupe ABS(pid)

- Liste des signaux disponibles:

```
$ kill -l
```

Liste des Signaux

- Identification par un entier
- Noms symboliques : SIGxx comme SIGINT …
  - Suffixe évoquant l'événement auquel le signal est associé

- Comportements par défaut:
  - **Terminer**
  - **Core Dump**
  - **Ignorer**
  - **Suspendre**
  - **Reprendre**

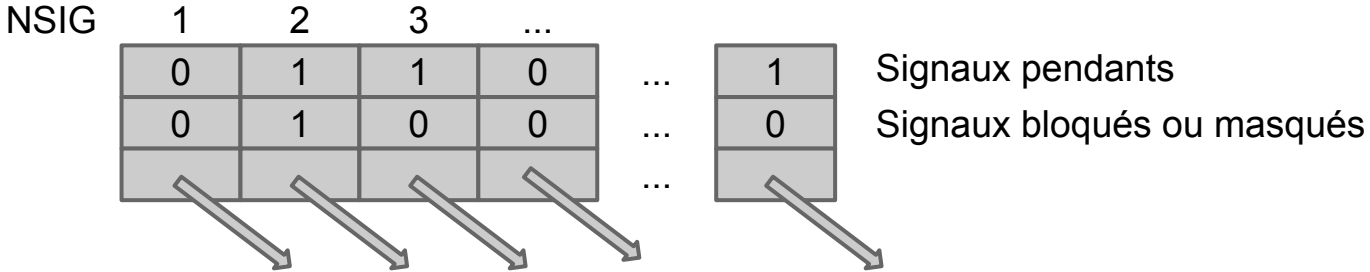
- **SIGTERM** (valeur 15) : (*termination*) demande de fin du processus, envoyé par la commande kill;
- **SIGINT** (valeur 2) : (*interrupt*) fin du processus, envoyé par CTRL-C;
- **SIGQUIT** (valeur 3) : terminaison du processus, génère un core dump;
- **SIGKILL** (valeur 9) : fin immédiate; il ne peut pas être ignoré ou bloqué ou masqué;
- **SIGHUP** (valeur 1) : le terminal dont le processus dépend a été fermé;
- **SIGPIPE** (valeur 13) : écriture dans un tube sans lecteur;
- **SIGSTOP** (valeur 19) : suspend l'exécution du processus;
- **SIGTSTP** (valeur 20) : suspend l'exécution du processus, envoyé par CTRL-Z;
- **SIGCONT** (valeur 18) : si le processus qui reçoit le signal était suspendu, alors celui-ci reprend son exécution;
- **SIGSEGV** (valeur 11) : (*segmentation violation*) problème d'adressage dans l'exécution du programme;
- **SIGFPE** (valeur 8) : Exception arithmétique (virgule flottante)
- **SIGCHLD** (valeur 17) : Terminaison d'un fils

Signaux non prédéfinis, donc utilisables :

- **SIGUSR1** (valeur 10) : (Signal défini par l'utilisateur, numéro 1),
- **SIGUSR2** (valeur 12) : (Signal défini par l'utilisateur, numéro 2),

Représentation des signaux

- Structure dans le BCP de chaque processus
  - 2 vecteurs binaires + un tableau de pointeurs vers la fonction de prise en compte (*handlers*)



- Signal pendant: notifié mais pas encore pris en compte
- Signal bloqué ou masqué: la prise en compte est différée en tant que le masque est à 1

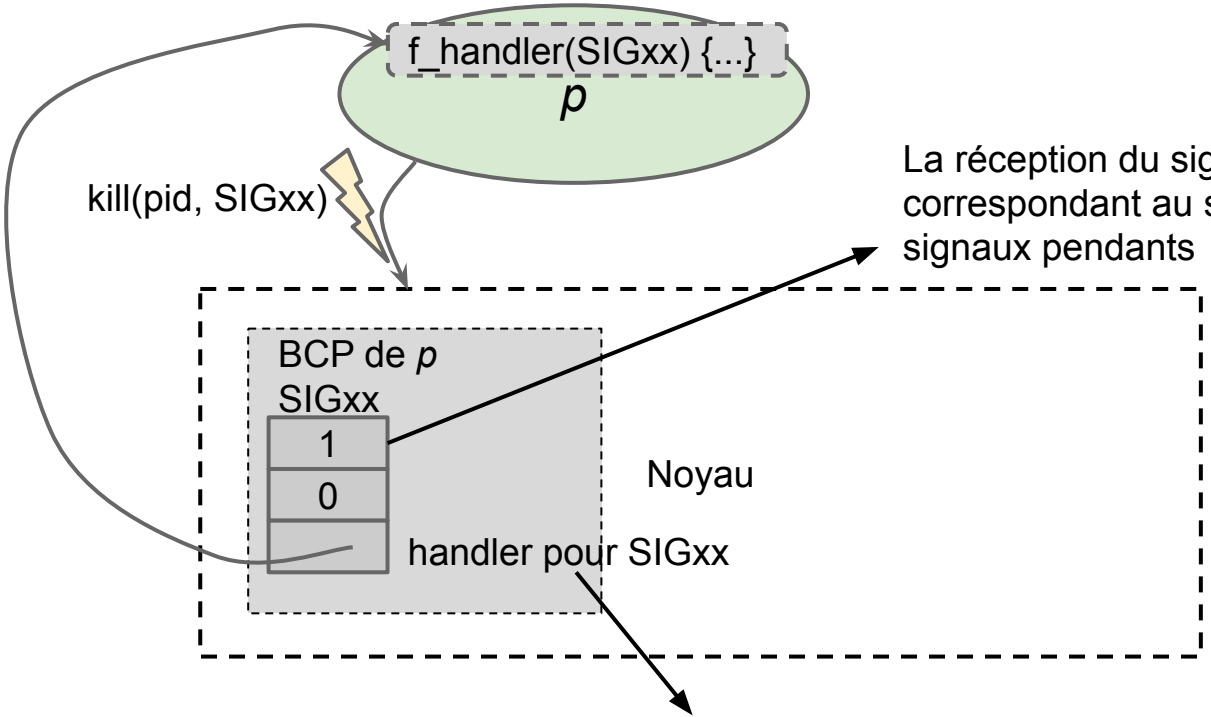
Handlers

- Fonctions à exécuter comme réponse à un signal
  - Pas de valeurs de retour (**void**)
  - Récupèrent le numéro du signal comme paramètre
  - On ne les appelle pas:
    - C'est le SE qui les lance comme réponse au signal
- On peut associer un handler sur mesure pour un signal donné

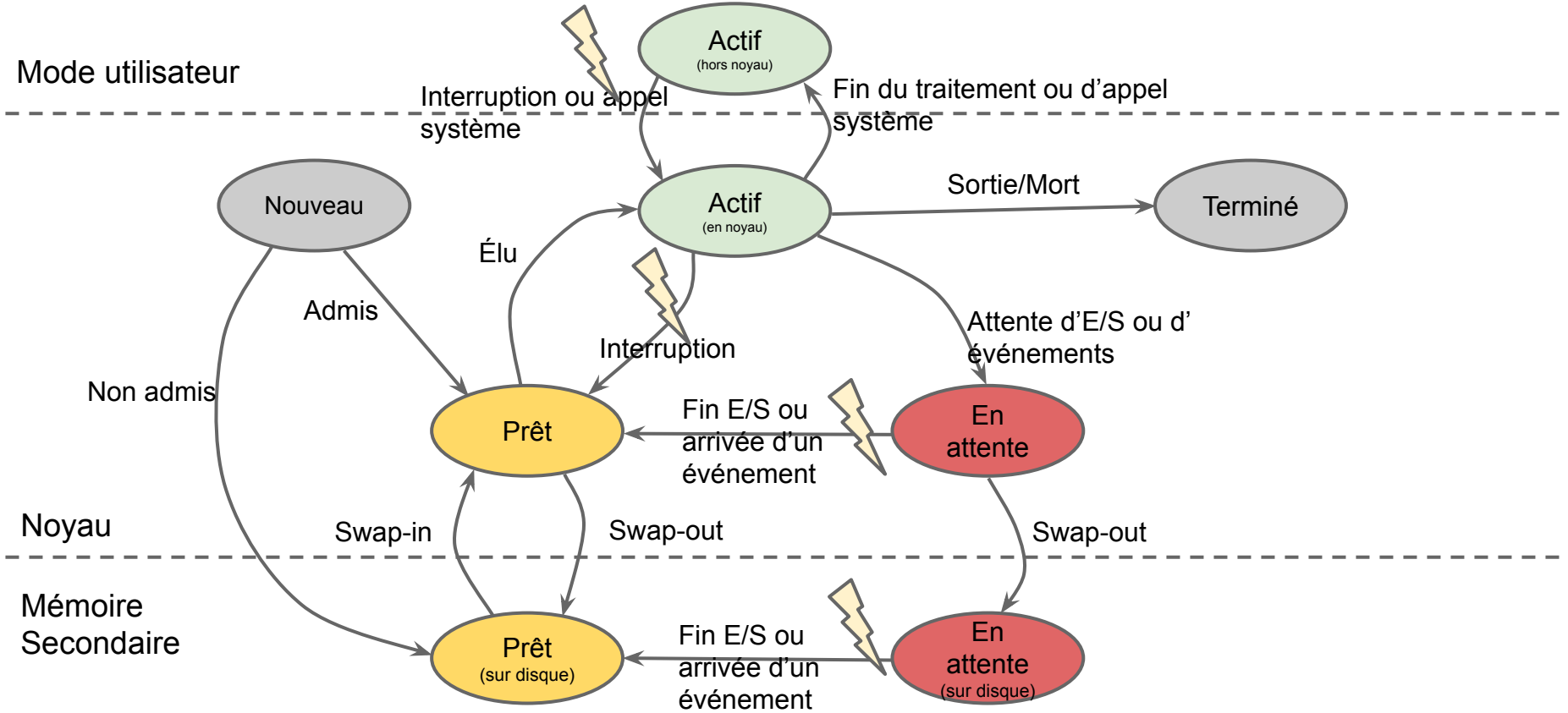
```
/* définition d'un handler: */
void f_handler(int sig) {
     …
}

/* Association handler <-> signal: */ NOTE: version obsolète mais simple
signal (sig, f_handler);
```

Attention: certain signaux ne peuvent pas ni être associés à des handlers différents, ni bloqués:
SIGKILL, SIGSTOP, SIGTSTP...

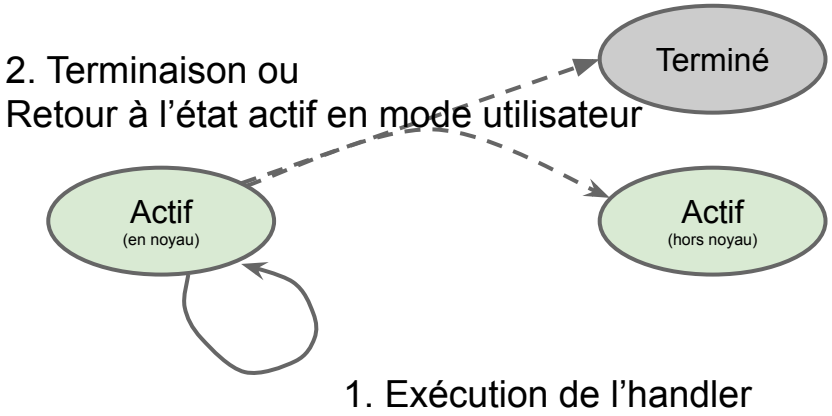
Prise en compte



États d'un processus

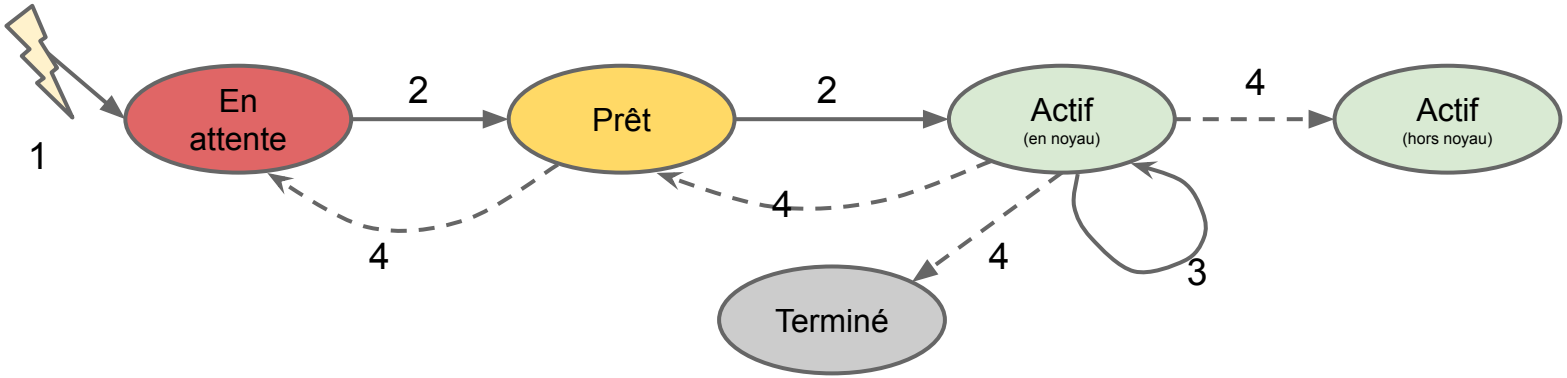


Prise en compte - processus actif



Prise en compte - processus en attente

1. Signal arrive
2. On active le processus
3. Exécution de l'handler
4. Terminaison ou retour en attente ou activité en mode user



Installation d'un handler

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

void confirme ( int sig ) {
      char c;
      printf(" Signal %d reçu - Confirmation ?
      (o/n) ",sig );
      scanf(" %c",&c);
      if ( c == 'o' || c == 'O' ) {
            signal ( sig, SIG_DFL) ;
            kill ( getpid() , sig ) ;
            /* == raise(sig) */
      }
}
```

notre handler

```
main ( ) {
      pid_t fils ;
      int f_etat;

      signal ( SIGINT , confirme ) ; /* binding du handler */
      signal ( SIGQUIT, confirme ) ; /* binding du handler */

      fils = fork () ;
      if ( fils == 0 ) {
            sleep ( 120 ) ;
            exit (0) ; /*fils se termine ici*/
      }
      signal( SIGINT , SIG_IGN ) ; /* exécuté seulement par le père
*/
      signal( SIGQUIT , SIG_IGN ) ;/* exécuté seulement par le père
*/

      wait(&f_etat) ;
      printf(" Mon fils %d est terminé ",fils);
      switch ( f_etat ) {
            case (int) 0: { printf(" normalement \n "); break ; }
            case (int) 2: { printf(" par SIGINT \n "); break ; }
            case (int) 3:
            case (int) 131 : { printf(" par SIGQUIT \n "); break ; }
            default : printf("d'une façon inattendue ! \n ");
      }
}
```

Interface POSIX

- La fonction signal() permet d'associer un handler à un signal uniquement pour une occurrence du signal:
  - Pour associer de façon permanente un handler, il faut appeler à nouveaux la signal() dans l'handler
    - (Après l'appel à l'handler customisé, le SE réinstalle l'handler par défaut)
  - Consequence: risque de comportement *impredictible* (si un deuxième appel à l'handler arrive avant de l'appel à signal())
- signal est *déconseillé*
- On utilise l'interface POSIX: structure **`sigaction`** `(pas de temps dans ce cours)`