# COURS SYSTEME D'EXPLOITATION (SE)

Refs : * Architecture des S.E
M. Griffith et H. Vayssade Hermès.
Lavoisier, 1991, 512 p

* UNIX Mécanismes base. Langage de commande Utilisation. H. Circas. Eyrolles 90.

* S.E 2e édition - A Tanenbaum
Pearson, 2003, 986 p

* Operating System Concepts, 7th Edition
A Silbershatz, P.B Galvin, G. Gagne
John Wiley & Sons Jan 2005, 944 p.

## I - GENERALITES :

### 1) Definition d'un SE :

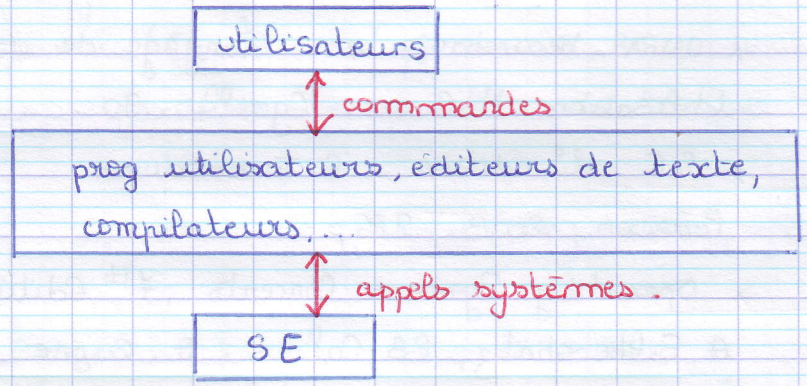
Ordinateur : machine complexe avec plusieurs dispositifs matériels :

- 1 ou plusieurs CPUs.
- Mémoires (RAM, ROM).
- Unités d'Entrée / Sortie.
- périphériques de stockage
- cartes (réseau, ...).

Machine sans SE :


⇒ les p.g.s doivent gérer directemt les périphériques
- nécessite de connaître finement les caractéristiques physiques et leurs spécificités.
- lourd, fastidieux, source d'erreurs.

idée: libérer les programmeurs de la gestion du matériel.
⇒ introduction d'1 couche logicielle sous la forme d'1 ens. de sous-progs (appels système) standards, génériques (open, close, read, write, ...) et faciles d'utilisation. SE interface entre le matériel et les progs utilisateurs.




S.E = machine "abstraite", indépendante du matériel.

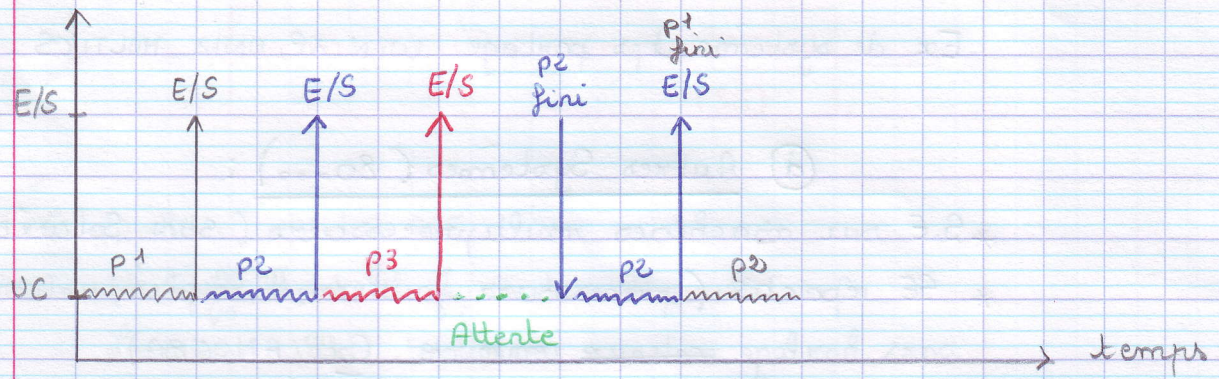
## 2) Evolution des SE:

### ⓐ 1ers systèmes (55-65): traitement par lots (batch):

1 prog chargé dans la machine, exécute seul (monoprogramat°) et ses résultats imprimés.
⇒ pour améliorer les tps de changement et d'impression, on a utilisé la technique du spooling.
- chargement des progs sur disque dur.
- en mê tps, exécution en séquence des progs déjà chargés.
- en mê tps, impression des résultats des progs finis.

### ⓑ Multiprogrammation (65-80):

- Le prog P1 en cours d'exécution comporte des tps morts (ex: lecture depuis le clavier).
- Pendant ce temps là, le processeur ne faisant rien.
- On exécute un autre prog P2.
- Si, de mê, P2 fait une E/S, on exécute P1 à sa place



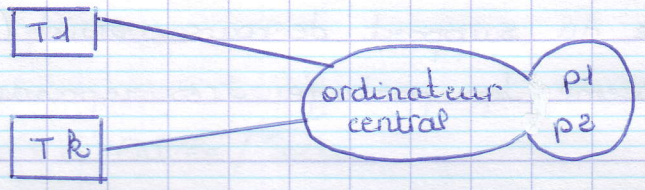

⇒ 3 prog à exécuter p1, p2, p3

Conséquence: Occupation maximale du processeur.

Problème: Si un gros prog de calcul monopolise l'UC, sans faire d'E/S, les autres progs ne peuvent pas se dérouler.

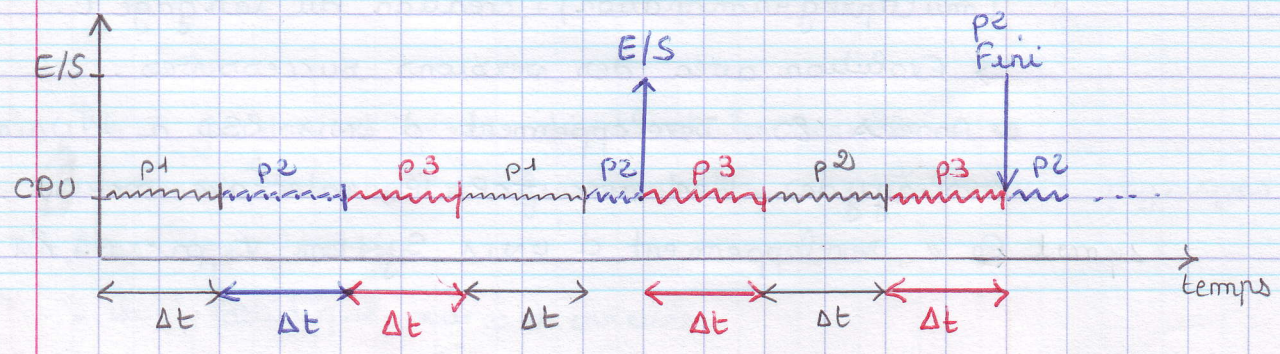
© Temps partagé (65-80):

Soumission interactive: fonctionnement pour plusieurs utilisateurs simultanés en mode question/réponse.




• Pr permettre l'intéractivité, allocation à tour de rôle de l'UC à chaque prog prêt à être exécuté, pour 1 quantum de tps $\Delta t$.

Ex: 3 progs p1, p2, p3



Ex de système tps partagé WIN XP, UNIX MULTICS.

### ⓓ Autres Systèmes (80...) :

* S.E sur machines multiprocesseurs (SUN Solaris).
* SE réparti (processeurs faiblement couplés, reliés en réseau, sans horloge ~~continue~~ commune. Ex: AMOEBA)
* SE tps réel (Ex: RT_MACH, CHORUS, RT_LINUX)

  2 types: • garanties temporelles sur les tps d'exécution des tâches
  • niveaux de priorité, systèmes réactifs.

### 3) Principales fonctionnalités d'1 SE :

Gestion: * des processus (prog en cours d'exécution)
* de la mémoire
* des E/S vers les périphériques (drivers, pilotes)
* des fichiers stockés sur disque.

Rq: Suite du support de cours : UNIX (System V)
⇒ Système tps partagé très répandu.

## II - SYSTEME UNIX :

### 1) Historique :

* 1ère version développée en 69 par K. Thompson au sein des labos Bell (AT&T) sur des machines PDP_7 et PDP_9 (monoprogrammation)
* nouvelle version par K. Thompson et D. Ritchié sur PDP_11 (multiprogrammation) + création du langage C.

→ Evolution avec des versions successives.

* Années 80. Développement d'UNIX BSD à l'Université de Berkeley, lien avec TCP_IP, interface sockets.

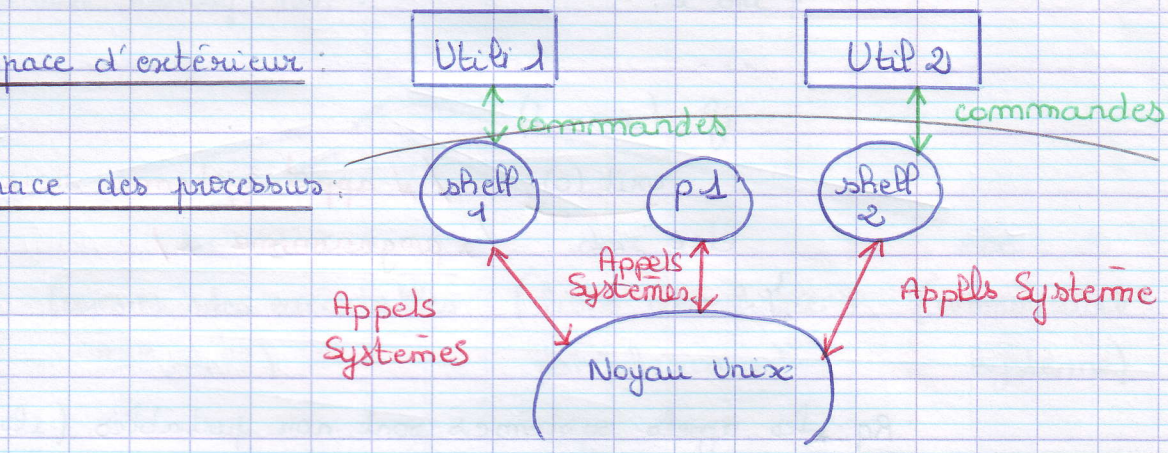
En //, Développement d'UNIX System V par AT&T

* Années 90, - nouvelles fonctionalités tps réel
  - noyaux dynamiques.
  - micro noyaux.
  - processus allégés. (threads)
  - autres interfaces de communication.

## 2) Architecture du Système:

2 niveaux : * les commandes
* les appels système

Schéma interaction:




Espace extérieur: - utilisateurs connectés.
- 1 utilisateur connecté est en relation avec 1 processus shell.

shell: interprète de commandes

Boucle qui : * lit 1 commande
* l'interprète
* la fait exécuter en créant 1 ou plusieurs autres process.

Ex: cd ...

Espace des process : 1 process ≡ 1 prog en cours d'exécution

Rq : 2 process ≠ permet exécuter le même code.

Noyau UNIX : code qui assure le fonctionnement du SE.
Ses fonctionnalités sont accessibles au travers des appels systèmes.

Appels systèmes : Sous. prog C, utilisable uniquement dans 1 prog.

Ex :
```
void main ()
{ int i;
  ...
  if (i == 1)
      exit (0);   /* appel système UNIX fin de */
  else ...        /* processus */
}
```

Rq : * les appels systèmes sont non portables (ils sont liés à chaque système).
(ex : exit ne marche pas sous DOS)

* pour éviter cela, utiliser les librairies de plus haut niveau (stdio.h) standardisées et portées dans tous les systèmes.

## III - SYSTEME DE FICHIERS (SF) :

### 1) Présentation :

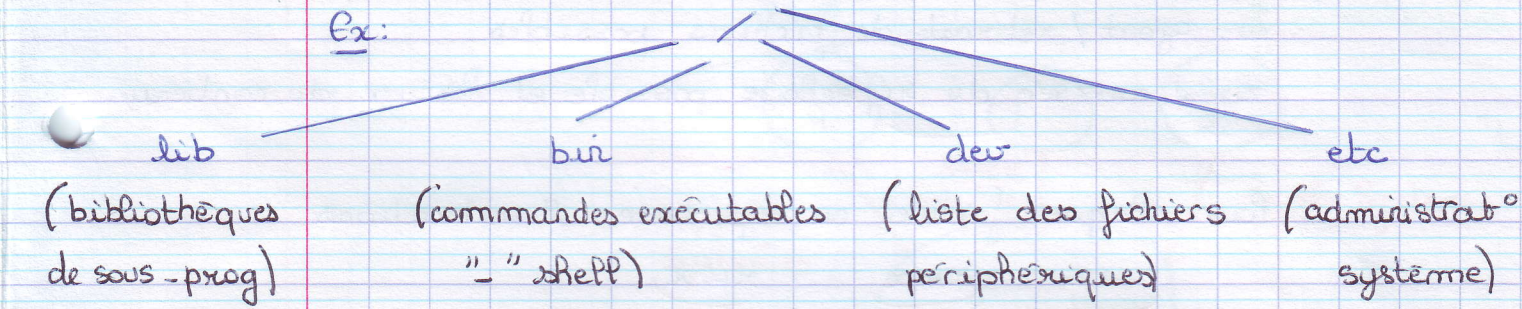
Déf : Fichier UNIX = Ensemble linéaire d'octets.
- possèdent un nom (extension non obligatoire)
- Majuscules ≠ minuscules
- ' ␣ ' à éviter

- souvent, les majuscules désignent les répertoires

⇒ Les fichiers sont rangés ds des répertoires (directories)
⇒ Sous UNIX, on a 1 structure arborescente globale et unique pouvant être réalisée avec plusieurs disques locaux ou accessibles au réseau.

- répertoire courant '.'
- répertoire père '..'
- racine '/' (slash)
- chemin absolu: commence tjrs par '/'
- chemin relatif: non.

Ex:




- Tout utilisateur connecté a 1 répertoire de travail (home directory).

## 2). Protection des fichiers et utilisateurs:

Types utilisateurs:

- superutilisateur (login: root) = Administrateur
- groupes d'utilisateurs (possibilités plus ou moins limitées)
- ds ces groupes, on a les utilisateurs standard (mot de passe + répertoire de travail)

3 types d'utilisateur pour 1 fichier ou 1 répertoire:

- propriétaire (user)

- les utilisateurs du m̂ groupe que le propriétaire (group)
- les utilisateurs du m̂ groupe que le propriétaire
- les autres étrangers au groupe (others)

Pour chaque type, 3 droits d'accès :

Fichiers :

r — fichier accessible en lecture
w — _____ écriture
x — _____ en exécution

Répertoires :

r : on voit la liste de ses fichiers
w : ajout / retrait de fichiers possibles.
x : traversée du répertoire possible et lecture du contenu.

Pour chaque fichier ou répertoire, 9 indications :

rwx rwx rwx
propriétaire groupe autres

⇒ les droits absents sont codés par '-'
⇒ on peut changer ces droits avec la commande chmod.

Δ Rem : pour les appels système, les 9 droits d'accès sont codés sous forme ~~binaire~~ binaire (0, droit refusé, 1 accepté)
Le chiffre obtenu est codé en base 8 (ou octal).

1 chiffre octal commence par 0.
Il représente des groupes de 3 bits.

Ex: création d'1 fichier avec les droits rwx rwx rwx

creat ("nom_fichier.txt", 0766)

rwx rwx rwx

## IV - GESTION DES RESSOURCES:

### 1) Définition:

Programme = Suite d'instructions

Process = Programme en cours d'exécution (ressources nécessaires) fournit l'état d'avancement de l'exécution du programme (image)

### 2) Caractéristiques des Process:

Principaux composants : tout ce qui est nécessaire à l'exécution du code

a) Zone de stockage des composants du process
- zone de mémoire pour le code à exécuter.
- zone mémoire pour les données.
- zone mémoire de pile pour l'exécution

b) Contexte d'exécution

Ensemble d'infos pour garder l'état courant du process
- Etat du process
- Priorité du process
- Etat des zones mémoires (code, données, pile)
- Valeur des registres du processeur
- Etat des fichiers ouverts...

Caractéristiques (attributs) d'1 Process:
- Identification (PID process Identifier)
- _____ de process créateur (process père ou PPID)
- _____ du propriétaire.

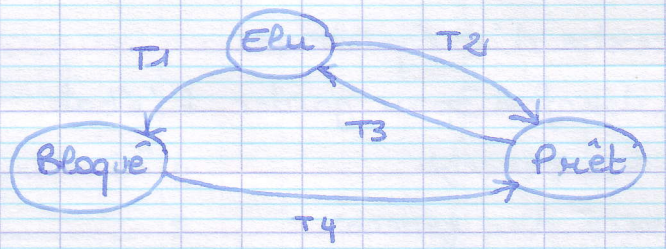
• Nom du processus = Nom du fichier qui contient le code.

* Process identifié par 1 entier unique (PID) compris entre 0 et 32767
Process originel = process swapper (numéro 0)
1er fils créé = process init (numéro 1)

Etat d'1 process:

Systèmes temps partagé = partage du processeur entre les divers actifs tous les quantums Δt.

Trois principaux états:
a) Elu (en cours d'exécution)
b) Prêt (suspendu, pour permettre l'exécution d'1 autre process)
c) Bloqué (attente d'1 evt extérieur (ex: Entrée / Sortie, signal, ...)




T1: Le process se bloque en attente de données d'entrée. ceci se produit lorsque la ressource attendue est non disponible.
• Sauvegarde des valeurs du contexte d'éxécut°.

T4: La ressource attendue est arrivée et le process redevient prêt à être éxecuter.

T2 et T3 : provoquées par l'ordonnanceur de process (scheduler)
T2 : le quantum $\Delta t$ expire
- Sauvegarde du contexte d'exécution et passage à l'état prêt.

T3 : Nouvelle exécution du process
- Restauration du contexte du process
- Exécution de ses instructions.

Rem : Chaque fois qu'un process quitte l'état élu, il sauvegarde son contexte, pour le restaurer lors de son retour élu.

## 3). Ordonnancement des process (scheduler)

Dans le noyau, on a la présence de 2 files d'attente.
- File de process prêts
- _____ bloqués

Scheduler : logiciel qui attribue la CPU avec process prêts et qui réordonne les process dans les files, suivant :
- les changements d'état
- la création de process
- la fin de _____

### politiques d'ordonnancement :

#### (a) circulaire (tourniquet) :

on gère la file des process prêts de façon circulaire. A chaque instant, le futur process élu est en tête de file.

*En cas de réalisation d'E/S, le process élu mis de la file des process bloqués, et on élit le process prêt suivant.

* Si fin de process élu, on le retire de la file et on élit le process prêt suivant.
* Si fin de $\Delta t$, on met le process élu en queue de file des prêts, et on élit le suivant.

Rem: Ordonnanceur préemptif car il prend le contrôle périodiquement.

Ex: 3 process, A, B, C.

temps ↓

| | | |
|---|---|---|
| Début | A élu | {B, C} prêts |
| fin $\Delta t$ | B élu | {C, A} prêts |
| fin $\Delta t$ | C élu | {A, B} prêts |
| fin de C | A élu | {B} prêt |
| fin $\Delta t$ | B élu | {A} prêt |
| E/S | A élu | {...} prêt |

(b) Ordonnancement avec priorité :

Idée: chaque process a 1 priorité et on ne lance que les process de priorité la plus élevée.
En cas de process de même priorité, on leur applique la politique du tourniquet.

Ex:

| priorité | Files de process. |
|---|---|
| 4 | {p1, p2, p3} |
| 3 | {p4, p5, p6, p7} |
| 2 | {p8, p9} |
| 1 | { } |

(seuls p1, p2, p3 peuvent être élus).

Calcul des priorités:

→ peut être fait de manière statique (inchangés par le système).

problème: risque de famine des process de basse priorité, qui ne s'exécutent jamais.
Pour éviter cela, le scheduler recalcule périodiquement les priorités et augmentent les priorités des process pour qu'ils accèdent tous à la CPU (priorités dynamiques: calculée à partir de la priorité de base et du tps processeur consommé réellement)
Politique d'ordonnancement avec priorités dynamiques appliquée dans le système UNIX et WINDOWS.

## 4) Appels système pour la gestion des process:

- Sous UNIX, tout process est créé à partir d'1 autre process → arborescence de process dont la racine est le process "swapper", et qui évolue dynamiquement au fur et à mesure des créations des process.

Appel systèmes: int getpid (); → renvoie le pid de ce process
int getppid (); → renvoie le PID du père de ce process.

### Création de process:

Appel système : int fork ()
permet à partir d'1 process, de créer deux process clones, qui éxécutent deux images indépendantes mais identiques.
→ 2 copies conformes avec duplication de ttes les données et de ttes les ressources.
→ Aucune mise en commun des données (2 copies indépendantes).
Distinction entre le père et le fils cloné:
le père a comme retour du fork, le PID du fils
le fils _____ 0

Si erreur, aucun clone n'est créé et le fork() retourne -1

Ex:
```
void main (void)
{ int pid;
  pid = fork();
  switch (pid);
  { case 0 :
      printf ("Je suis le fils \n");
      break;
    case -1 :
      printf ("Erreur : fils non crée \n");
      break;
    default :
      printf ("Je suis le père \n");
      break;
  }
}
```

Exécution d'un prog : Appel système int exec (char *path, char * argv0, ...) et les variantes execlp, execvp.
⇒ remplacent l'image (code + data + pile + ...) d'1 process par 1 autre image dont le code est dans 1 fichier dont le nom est passé en paramètre du "exec". Toutes les zones (code, data, pile) sont définitivement remplacées.

```
void main (void)
{ printf ("changement de code \n");
/* remplace le code courant par celui du fichier "ps" */
  execlp ("ps", "ps", NULL);
  printf ("Erreur : code non remplacé \n");
  exit (-1);
}
```

## Fin d'1 process:

Appel Système `exit (int status);`

Termine 1 process et renvoie le code de terminaison donné ds le paramètre 'status' vers le process père. La terminaison normale (faite automatiquement en fin de prog) est faite en renvoyant 0 (zéro). Toute autre valeur indique 1 terminaison anormale.

Ex:
```
void main (void)
{
    printf ("message 1\n");
    exit (0);
    printf ("message 2 non affiché\n");
}
```

## Synchronisation des process:

Appel système `int wait (int *ptr_status);` (passage par réf.)

- permet au père d'attendre la fin d'un de ses fils
- l'exécution du père est suspendue jusqu'à ce qu'un de ses fils termine (avec exit). Le père est réveillé ensuite.
  - → le résultat du wait fournit le PID du fils terminé.
  - → le paramètre par référence ptr_status renseigne sur la façon dont est mort le fils et sur sa valeur du "exit".

père

```
int pid_fils;
int stat;
   ⋮
pid_fils = wait (&stat);
   ⋮
```

1 fils de PID 12128

```
exit(3);
```

ici PID fils = 12128 et stat contient 3 et le père continue à s'exécuter

le père qui attendait, est réveillé

# I. ENTREES/SORTIES :

## 1) Lien Process - Extérieur :

⇒ Chaque process possède 1 table de fichiers propriétaire d'environ 20 valeurs (de 0 à 19)
Chaque entier désigne le nom interne d'1 fichier ou d'1 ressource de communication externe (ex: socket).
En général (on peut le changer)

0 connecté à stdin ( Standard input = clavier)
1 connecté à stdout ( _____ output = écran)
2 _____ stderr ( _____ error = msg d'erreur écran)

Ex: process P

| | |
|---|---|
| 1 | stdin |
| 2 | stdout |
| 3 | stderr |
| 4 | chemin accès fichier |
| 5 | Non attribué (vide) |
| 6 | tube |
| 7 | socket |
| 8 | autre fichier |
| | ... |

le process P: communique avec les ressources externes en utilisant leur n°- dans la table des descriptions.

Etapes nécessaires:
- création d'1 lien n°- interne vers objets extérieurs (on affecte 1 n°- interne)
- communiquer avec cet objet (on utilise son n° interne avec lecture / écriture dessus)
- suppression du lien (le n°- interne est libéré).

Rem: * tout nouveau lien créé se fait avec le plus petit n° interne de la table (ex: 5 ici)

* plusieurs n°s internes peuvent être associés à 1 même objet (ex: le même fichier)
* tout process fils hérite d'1 copie de la table du père (→ tous les objets avec lesquel le père communiquait sont accessibles par le fils)

## 2) Opérations sur les fichiers:

- correspondances n° internes et les objets extérieurs (fichiers, tubes, sockets...)

Création d'1 fichier: int creat (char * nom, int droits)
- Crée 1 fichier d'identificateur nom avec les droits avancés.
- Ouvre ce fichier en écriture
- Attribue 1 n° interne à ce fichier (plus petit n° libre)
- Retourne ce n° interne (ou -1 si erreur)

Destruction d'1 fichier: int unlink (char * nom)
- détruit le fichier d'identificateur nom
- retourne 0 si pas d'erreurs.

Ouverture d'1 fichier: int open (char * nom, int mode, int droits)
- ouvre le fichier considéré en lecture ou en écriture selon la valeur de mode.
- Attribut 1 n° interne au fichier.
- retourne ce n° interne (ou -1 si erreur)

mode : O_RDONLY (lecture seule)
O_WRONLY (écriture ____)
O_RDWR (lecture / écriture)

} * include <fontl.h>

Fermeture d'1 fichier : int close (int numero_interne)
- ferme la ressource ou le fichier associé au n° interne
- libère ce n° interne pour utilisation future.

Ecriture
Lecture d'1 ensemble de caractère depuis 1 fichier :

Rem : les fichiers sont gérés en mode production/consommateur, avec 1 curseur logique qui se déplace selon les lectures / écritures.

int ~~read~~ write (int num_interne, char * tampon, int taille)
- lit taille octets depuis le fichier ~~associé au n° interne~~ le tampon et les recopie ~~dans le tableau tampon~~ à la position courante du fichier
- déplace le curseur logique de taille octets.
- retourne le nbre d'octets effectivement ~~lus~~ écrits, ~~0 si fin de fichier~~, (<0 si erreur)

Positionnement dans 1 fichier :
- Déplace le curseur logique du fichier considéré.

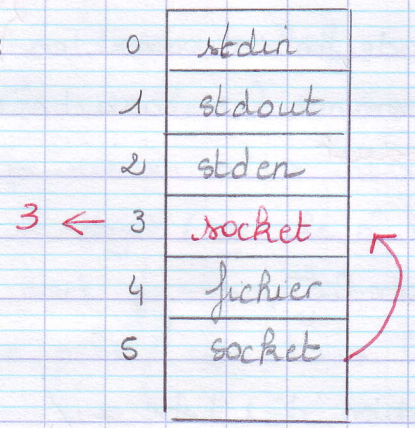
long lseek (int numero_interne, long deplacement, int base)
- 1 fois le curseur déplacé, 1 nouvelle lecture/écriture démarrera à partir de la nouvelle position.

base : origine du calcul de la nouvelle position.
SEEK_SET : à partir du début du fichier
SEEK_END : _____ de la fin _____
SEEK_CUR : à partir de la position courante.

déplacement : adresse relative à partir de la base
- retourne la nouvelle position (-1 si erreur)

Duplication : int dup (int numero_interne)
- duplique la ressource associée au numéro_interne dans la première entrée libre de la table.
- retourne le n° où a eu lieu la duplication.

Ex :




3 libres
dup(5) → retourne 3

Les 2 entrées 3 et 5 désignent le même objet physique.

Rem : close et dup servent à réaliser les mécanismes de redirection > < ! du shell

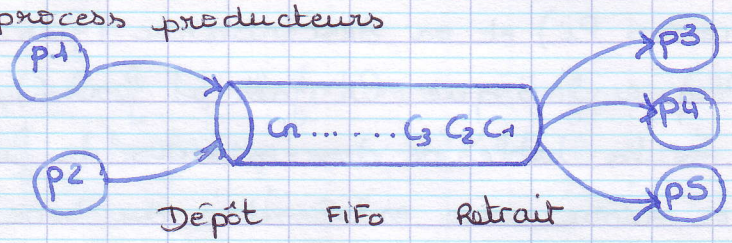
ls > toto.txt  close(1)
dup(fd)

- Appels systèmes précédents sont proches de ceux de la librairie <stdio.h> mais sont non portables en dehors de unix.

## 3) Communications entre process :

tubes de communications :

process : ne partagent aucune variable ni mémoire.
Pour faire communiquer 2 process, il est nécessaire de créer et d'utiliser 1 tube de communication (pipe)
tubes : canal unidirectionnel FIFO de caractères.

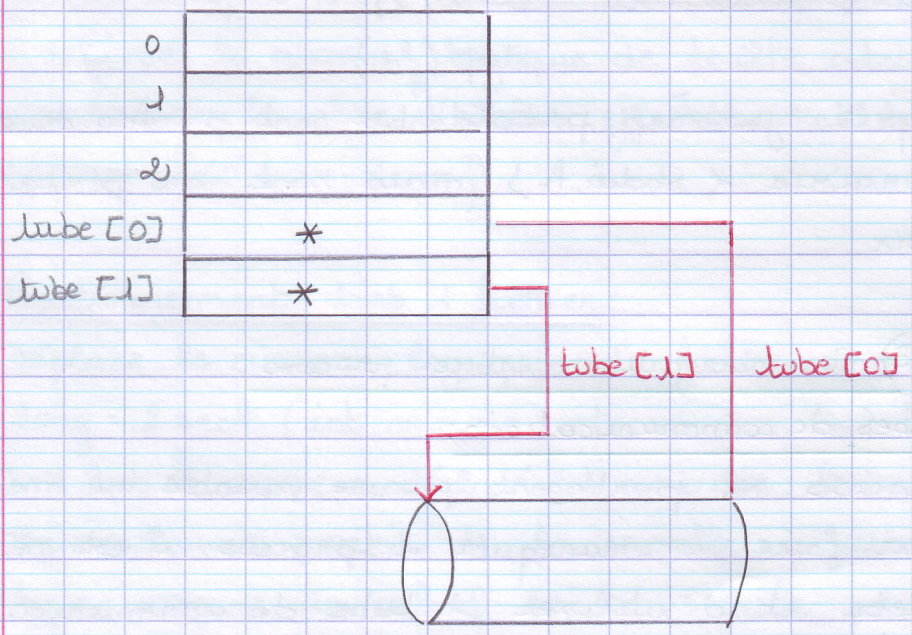



Création d'un tube : int pipe(int tube[2])
tube = paramètre par référence qui contient les 2 descripteurs de fichiers attachés à chaque extrémité du tube.

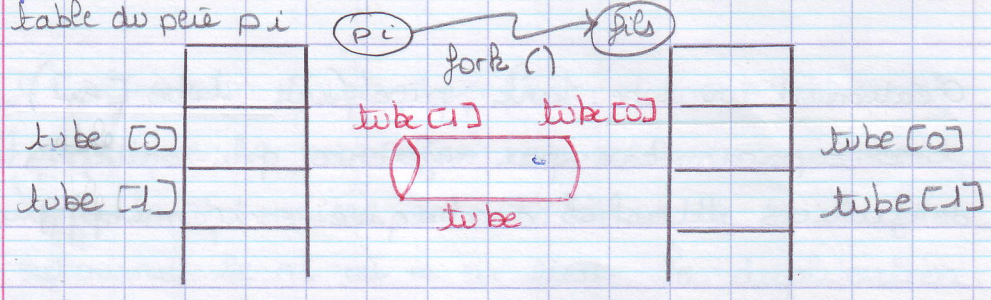
tube[1] : écriture
tube[0] : description de fichier ouvert en lecture (côté où l'on consomme les caractères).
produit

Process Pi : main()
{ int tube[2];
⋮
/* création d'1 tube */
pipe(tube);
}

table de pi :

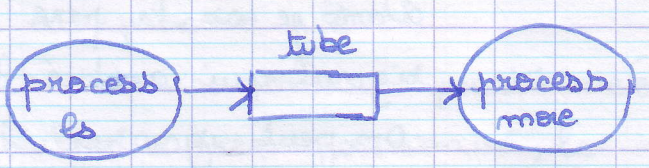



1 tube ne peut être utilisé qu'entre son créateur (ici le Process Pi) et sa descendance (fils, petit fils...)
En effet, chaque fois que Pi fait 1 fork(), le fils créé à la fin la table de descripteurs de fichiers. Il peut accéder ainsi aux extrémités du tube



Rem: pipe, utilisé avec dup et close sert à réaliser le mécanisme | du shell

ls | more




Signaux:

Signal: Interruption envoyée à 1 process

Moyen de signaler l'arrivée d'1 entrée externe au process: - réveiller 1 process
- endormir ———
- signaler l'arrivée de données, la mort d'1 fils.

2 types de signaux:

* engendrés par le matériel
  - bus error (accès interdit à la mémoire)
  - CTRC_C
  - horloge ...
* déclenchés par 1 process vers l'autre

  kill _9 1988 (envoie le signal 9 au process 1988)

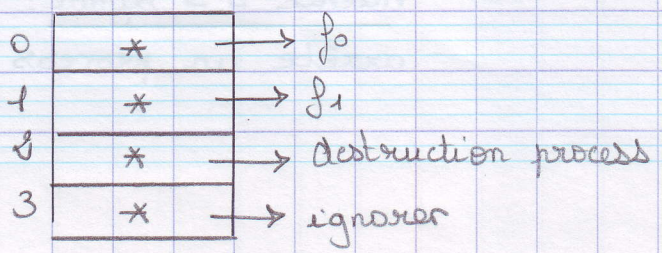
  kill _14 2222 ( ——— 14 ——— 2222)

Réception d'1 signal par 1 process:

- Ce process stoppe le traitement en cours et il va exécuter 1 sous-prog associé à l'interruption (ou signal) reçue
- 1 signal est identifié par 1 entier unique

→ chaque process possède 1 table des signaux propriétaire qui contient les adresses de sous-programmes associé à chaque interruption.

table de signaux:



Par défaut, le traitement associé détruit le process.
Dans ce cas, le père récupère dans le "wait" le n°. du signal qui a tué le fils.
- On peut ignorer 1 signal (routine SIG_IGN)

2 types de signaux:
Trappes: Signaux non déroutables qui tuent toujours le process (SIGBUS, SIGILL, SIGKILL)
Signaux classiques: peuvent être masqués, ou détournés sur 1 routine choisie.

principaux signaux:

| | |
|---|---|
| SIGINT | (associé au CTRL-C) |
| SIGALRA | (horloge) |
| SIGKILL | (valeur 9, arme fatale) |
| SIGUSR1 | (2 signaux utilisateur pour la synchro des process) |
| SIGUSR2 | |

Déroutement d'un signal:
signal (int signum, pteur_de_fonction)
signum: n°. du signal à dérouter
pteur_de_fonction: { Nouvelle fonction à exécuter
ou
SIG_IGN pour ignorer le signal
ou
SIG_DFL pour le remettre au traitement par défaut.

Attente d'1 signal: int pause ();
arrête un process jusqu'à ce qu'il reçoive 1 signal.

Envoi d'un signal : int kill (int pid, int signum)
Envoie le signal signum au process pid
le pid réagira en fonction de la routine positionnée chez lui pour signum (s'il n'a rien positionné, le pid est tué).

primitive alarm : int alarm (int seconds)
permet au process de recevoir le signal SIGALARM au bout du nb de secondes spécifiées.
alarm (0) désactive le comportement précédent.
Rem : # include <signal.h> pour les signaux.