# Système d'exploitation

## Cours 1

*Thierry Hamon*

Bureau H202
Institut Galilée - Université Paris 13
&
LIMSI-CNRS
hamon@limsi.fr
https://perso.limsi.fr/hamon/Teaching/P13/SE-AIR2-2020-2021/

AIR1/AIR2 - SE

# A propos de moi

- Maître de conférences en informatique - Enseignement : Institut Galilée (USPN) - Recherche : LIMSI (UPSay)
- Responsabilité : directeur de la spécialité Informatique, responsable des relations internationales de Sup Galilée
- Recherche : Traitement Automatique des Langues appliqué aux domaines de spécialite
  - Données textuelles en français, anglais, etc.
  - Constitution et enrichissement de ressources terminologiques
  - Application : fouille de texte, recherche d'information
- Enseignement :
  - BD et entreptôt de données
  - Gestion de projet
  - Programmation (C, système)
  - Modélisation et représentation des connaissances, fouille de texte

# Présentation du cours

- Contenu et objectif : programmation au niveau d'un système d'exploitation
  - Fonctionnement d'un système d'exploitation
  - Concepts clés des système d'exploitation (processus, mémoire, synchronisation, communication entre processus)
  - TP sous Linux, en C
- Organisation du cours
  - 12h de cours (8x1h30)
  - 21 de TP (3x2h + 4x1h30 + 3x3h)
- Evaluation :
  - Évaluation en TP (TP noté ou QCM)
  - Partiel sur table

# Plan

# Programmes

- Programme-source :
  suite d'instructions écrites dans un langage de programmation (Pascal, C, Java, Caml, Python ....).
- Compilation : obtention d'un exécutable ou code binaire.
  Remarques :
  - Le code binaire peut être complet ou incomplet
  - Code binaire incomplet : nécessite de faire appel appel à des codes externes (librairies dynamiques ....) en cours d'exécution
- Exécution du code binaire (processus) : Succession d'opération au niveau du système d'exploitation

# Exécution d'un code binaire

## Processus

Schématiquement, une succession d'opérations :

1. Le système d'exploitation (SE) réserve en mémoire centrale 3 zones : code, données, pile.
2. Le SE charge en zone de code l'exécutable.
3. L'exécution débute.
4. À la fin de l'exécution, les zones sont désallouées

NB : Selon les SE, certaines des étapes peuvent être modifiées ou même supprimées.

# Processus

- Processus : un exemplaire d'un programme, en cours d'exécution
  Abstraction d'un programme
- Traitement parallèle de tâches bien distinctes autorisé par tous les systèmes d'exploitation actuels
  Exemple : calcul proprement dit simultanément à l'affichage sur un terminal (ou plus généralement toute entrée-sortie).
- Processus : un ensemble de
  - codes chargés en mémoire
  - données,
  - une zone mémoire permettant de gérer les piles d'appels,
  - zone mémoire contenant les informations relatives au processus.

NB : Les zones de code et de données peuvent être partagées.

# Commutation de contexte

- Plusieurs processus peuvent à effectuer des calculs sur CPU par **commutation de contexte** (*context switch*)
- Le système d'exploitation
  - exécute pendant un certain temps un processus
  - commute d'un processus à un autre processus
    *i.e.* sauvegarde des informations permettant la reprise du processus arrêté,
  - charge les informations pour le nouveau processus
  - ... jusqu'à ce qu'il n'y ait plus de processus à exécuter

# Tâches

- Tâche : succession de calculs à effectuer
- Un programme peut être considéré comme un ensemble de tâches,
  le *découpage* en tâches n'est pas unique
- Plus généraleme : un ensemble de processus peut être considéré comme un ensemble de tâches.
- Le terme de **processus** est réservé pour l'exécution d'un programme n'ayant pas ou très peu de relations avec d'autres processus.
- On peut considérer comme une tâche, l'exécution d'un *programme* pouvant avoir des relations avec d'autres tâches
  Exemple : partage de données en lecture et/ou écriture

# Tâches

Programme en exécution est considéré comme un ensemble de tâches, si il existe

- **Possibilité 1** un langage de programmation adéquat
  - *i.e.* incluant des appels système
  - permettant de spécifier comment gérer une tâche (création, terminaison, ....),
  - le code de la tâche étant donné par le programmeur
- **Possibilité 2** un partage entre tous les processus de codes correspondant à des fonctionnalités standard (lecture/écriture de fichier par exemple).
  Un programme est écrit en faisant appel à ces fonctionnalités dont les codes exécutables sont
  - chargés en mémoire à la demande (librairies dynamiques)
  - ou directement inclus dans le code du SE lui-même
- **Possibilité 3** Possibilités 1et 2

# Tâches : exemple

Un programme de traitement de texte peut-il 'rendre la main' à l'utilisateur avant même qu'une commande de sauvegarde de fichier soit terminée ?

- la tâche 'sauvegarde' et la tâche 'édition' s'effectuent en parallèle
- L'exécution doit être cohérente : on ne peut lire et écrire au même endroit

# Evolutions des systèmes d'exploitation

- Par rapport à l'exécution de programmes : 4 étapes dans l'évolution des systèmes d'exploitation (SE) :
  - mono-tâche mono-processus
  - (mono-tâche) multi-processus
  - multi-tâches (mono-processus)
  - multi-tâches multi-processus
- En pratique, situations moins tranchées
  chaque type de SE a eu une bonne dizaine de variantes au cours des années.

# SE mono-tâche mono-processus

- Un programme est en exécution
- Alternance *SE / exécution complète d'un programme utilisateur / SE ....*
- Nécessaire lorsque
  - soit seul le temps d'exécution compte et pas le temps d'attente (cas de gros programmes),
    Mode 'temps réel' de certains SE : les programmes sont exécutées sans interruption
  - soit il n'y a qu'un seul utilisateur (premiers systèmes MS-DOS)
    Cas de figure obsolète

# SE (mono-tâche) multi-processus

Plusieurs programmes peuvent être en exécution au même instant

- Exécutions entremêlées
- Chaque programme en exécution (appelé processus) a sa propre zone mémoire.
- Indépendance et la sécurité des données : responsabilité du SE.
- Relations entre processus assez faibles
  - Variables non partagées
  - Possibilité d'utiliser les fichiers (ou les tubes) comme intermédiaires entre processus
- Tous les SE multi-utilisateurs étaient sous cette forme (Unix, VMS, ....),
  mais maintenant depuis quelques années les choses sont plus complexes

# SE multi-tâches (mono-processus)

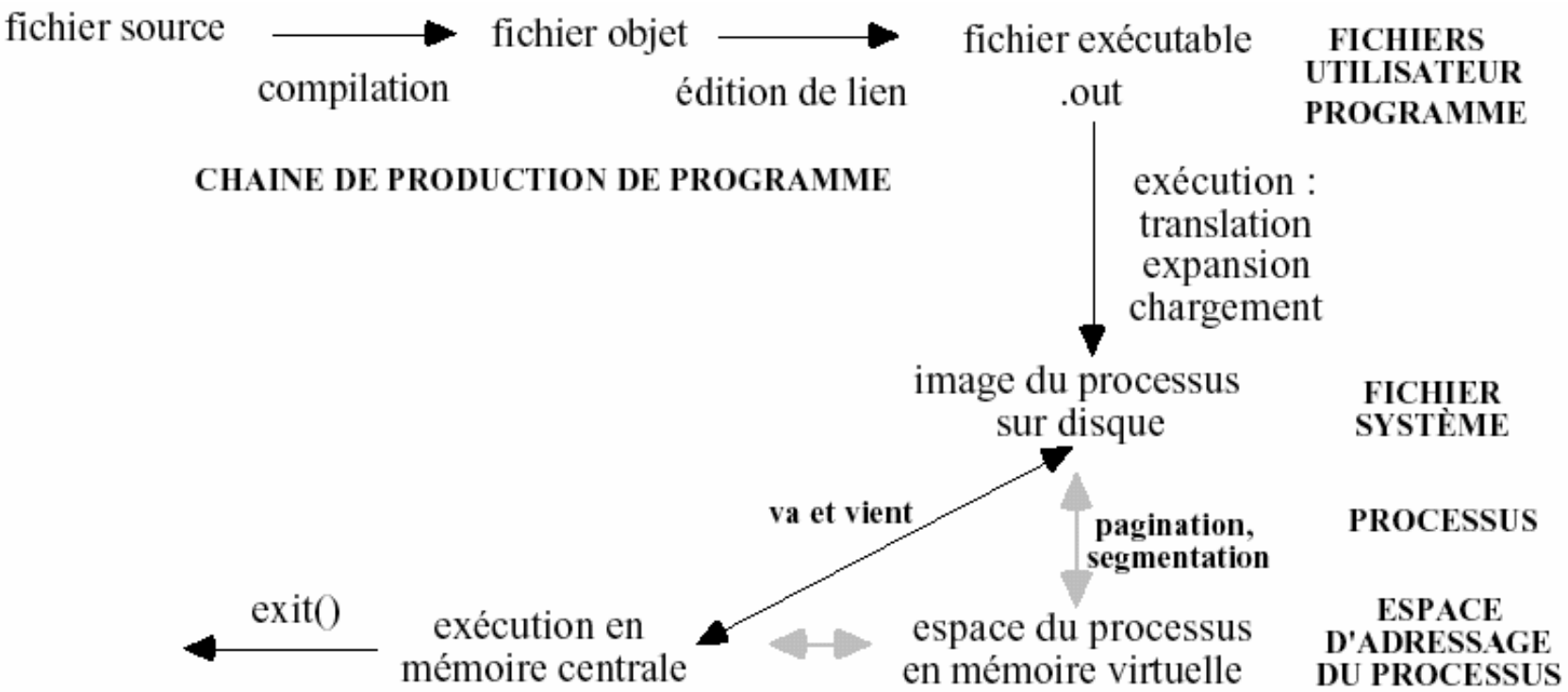
- Plusieurs programmes peuvent être chargés mais un seul en exécution
- Changement de tâche : du ressort de l'utilisateur

# SE multi-tâches multi-processus

- Hypothèse : possibilité de décomposer un programme en un ensemble de tâches
  ou code du programme constitué d'un ensemble de codes disjoints
- Toutes les tâches (de tous les processus) peuvent être en même temps en exécution.
- Relations entre tâches importantes
- Le contrôle des ressources (fichiers, données en mémoire, ....) doit être être soigneusement effectué.
- Tendance actuelle des SE (que ce soit Unix ou Windows)

# Du programme au processus

- Un processus peut être vu comme instance d'un programme mis sous une forme exécutable en mémoire



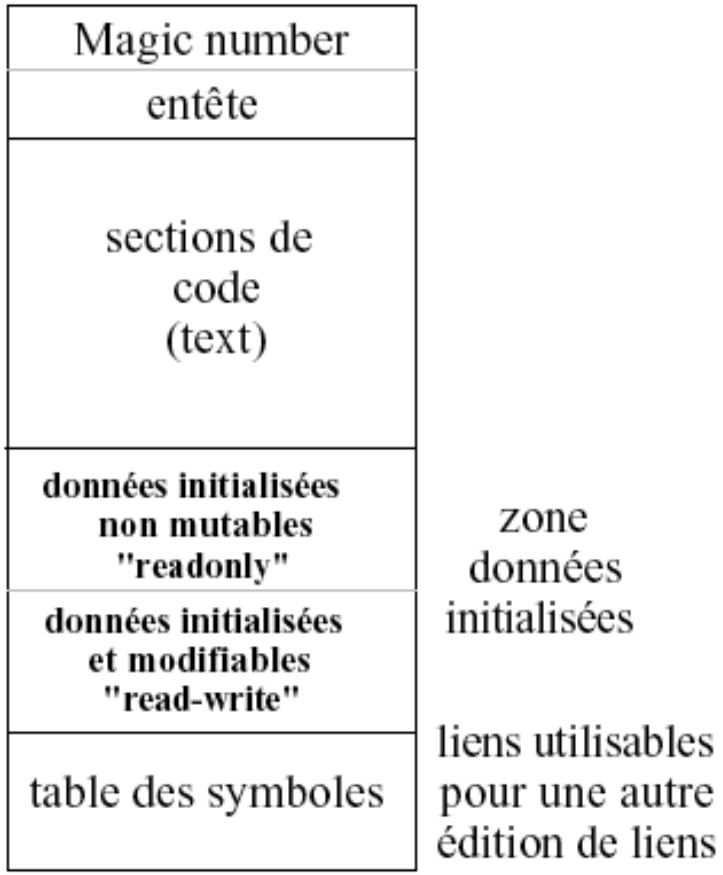
# Format d'un exécutable

## Magic number

- si #! : fichier exécutable par un interpréteur
  #! est alors suivi par le chemin d'accès à l'interpréteur :
  - `#!/bin/sh` pour le Bourne Shell
  - `#!/bin/csh` pour le C-Shell
  - `#!/bin/bash` pour le Bourne Again Shell (Bash)
  - ...
- sinon le fichier est directement exécutable : c'est le résultat d'une compilation et d'une édition de lien. Ce champ indique :
  - si la pagination est autorisée (stickybit),
    Le processus reste en mémoire même après la fin de son exécution, pour pouvoir être relancé plus rapidement.
  - si le code est partageable par plusieurs processus (code réentrant).

# Format d'un exécutable

## Format `a.out`




La structuration analogue pour le format ELF sous Linux

# Zones d'un processus

Pour exécuter un processus, il faut classiquement disposer en mémoire centrale

- d'une zone de code
- d'une zone de données
- d'une zone de pile
- d'un tas

Mais aussi

- Un compteur d'instruction
- Une description du processus
- Des données liées au processsus

# Zones d'un processus

## Remarques

- Morcellement des zones
  Possibilité de
  - Partage entre plusieurs processus
  - Gestion de la mémoire secondaire comme une partie (virtuelle) de la mémoire principale
- Référencement des zones par des adresses
  Association d'une table d'adresses pour chaque processus ou thread

# Zone de code

- Partage total ou partiel entre tous les processus utilisant
  - le même code
  - le même morceau de code

  - à défini lors de la compilation
  - Par exemple : une librairie dynamique par exemple),

# Zone de données

- Propre au processus
- Propre à l'ensemble des processus légers ou threads s'exécutant pour ce processus

# Zone de pile

- Propre à chaque processus léger ou thread
- Propre au processus s'il n'y a qu'une ligne d'exécution

# Compteur d'instruction

- Compteur d'instruction ($IC$) : Pointeur sur la prochaine instruction à exécuter
- Instruction immédiatement suivante (cas des instructions de calcul) ou non (cas des sauts)

# Descripteur de processus

- Exécution de plusieurs processus simultanément (en pratique et sur un système à monoprocesseur, alternativement)
  Besoin de conserver beaucoup d'information
- Représentation par le système de chaque processus à l'aide d'un descripteur contenant toutes les informations nécessaires à la commutation
  Commutation : passage du contrôle à un autre processus
- Les descripteurs sont contenus dans une table gérée par le noyau en mémoire centrale
- La taille de la table des descripteurs est fixe (de quelques dizaines d'entrée à plusieurs milliers d'entrées)

# Descripteur de processus

Informations principales dans le descripteur de processus :

- identificateur : nom ou numéro du processus
- état du CPU :
  - *IC*
  - compteurs et registres du CPU nécessaires à la reprise de l'exécution (sans valeur pendant l'exécution)

  Approximativement : les informations sauvées par une interruption lors d'un fonctionnement mono-tâche
- processeur : numéro du processeur sur lequel le processus s'exécute (sans valeur pendant l'exécution)
- occupation mémoire :
  - description de l'implantation en mémoire du processus
  - éventuellement, la correspondance entre les zones telles que le processus les voit (code1, code2, données1, ...) et leurs adresses réelles (privées ou partagées)
    aspect lié aux techniques de pagination, de segmentation, ...

# Descripteur de processus

Informations principales dans le descripteur de processus (suite) :

- ressources : liste des ressources allouées au processus (fichiers, imprimantes, lecteurs de bandes, terminal, etc. ) et droits associés
- processus parent
- processus fils
- événements asynchrones reçus
- état du processus
- priorités
- droits
- ...

# Données liées à un processus

4 types (MC = mémoire centrale) :

- Description dans la table des processus (*proc structure*), toujours en MC
- Description des segments partagés (*text structure*), toujours en MC
- Données utilisateur (*user structure*, *zone u*), en MC si élu
- Zones de données du processus (*process page tables*), en MC si élu

# Table des processus

- état ;
- taille du processus, localisation du processus et de sa *zone u*
- identificateurs d'utilisateur et de groupe réels et effectifs (détermination des droits)
- identificateur du processus père
- descripteur d'événements survenus (pour *sleep* et *wakeup*) et masques sur signaux
- paramètres d'ordonnancement (priorités, temps d'attente)
- champ des signaux en instance
- compteurs de temps utilisé

# Zone u

`user.h`

- pointeur vers la table des processus
- identificateurs d'utilisateurs réels et effectifs (modifiables par le processus)
- compteurs de temps
- tableau de réponses aux signaux
- terminal de connexion (s'il existe)
- identificateurs d'erreurs lors des appels système
- résultat des appels système
- paramètres pour appels d'entrée/sortie
- répertoire courant et racine courante
- table des fichiers ouverts par le processus (descripteurs)
- limites de taille du processus et des fichiers dans lesquels il peut écrire
- permission sur les fichiers créés (umask)

# Groupe et Session

- Groupe : raffinement POSIX des sessions
  - Chaque processus appartient à un groupe, dont il transmet le numéro à ses fils
  - Un processus peut changer de groupe (et créer un nouveau groupe) avec `setgrp()`
  - Objectif : Spécification de l'ensemble des processus actifs sur un terminal
    Ces processus ont alors la possibilité
    - d'écrire sur le terminal
    - d'être informé de la frappe des caractères de contrôle sur le terminal (`intr` – `CTRL-C`, `quit` – `CTRL-\`, `susp` – `CTRL-Z`)

# Terminal de contrôle

- Un processus est lié à un terminal par
  - ses E/S standard (stdin 0, stdout 1, stderr 2)
  - son terminal de contrôle
- Terminal de contrôle :
  - Au début de la session : le shell de login
    Symbolisé par la référence "/dev/tty"
  - Possibilité d'envoyer des signaux à tous les processus du groupe de ce shell (par `killpg(0,numéro de signal)`)
    en particulier les signaux provenant du clavier
  - Lorsque le procesus principal d'une session se termine, tous les processus de la session reçoivent le signal SIGHUP et sont alors interrompus

# Terminal de contrôle

Remarques :

- Un processus qui a changé de groupe est donc immunisé contre les signaux provenant du clavier (par exemple `SIGHUP`) mais a accès au terminal par ses descripteurs de fichiers standard 0, 1 et 2.
- Inversement : un processus peut fermer ses E/S standard et rester sous le contrôle du terminal
- Un démon (`daemon`) est un processus qui n'a accès au terminal ni par contrôle ni par les descripteurs de fichiers.

# Contexte d'un processus

Ensemble des données qui permettent de reprendre l'exécution d'un processus qui a été interrompu :

1. son état,
2. son mot d'état, en particulier :
   - la valeur des registres actifs,
   - le compteur ordinal,
3. les valeurs des variables globales statiques ou dynamiques,
4. son entrée dans la table des processus,
5. sa zone u,
6. Les piles user et system,
7. les zones de code et de données.

# Contexte d'un processus

Remarques :

- Le noyau et ses variables ne font partie du contexte d'aucun processus !
- L'exécution d'un processus se fait dans son contexte.
- Quand il y a changement de processus courant, il y a réalisation d'une commutation de mot d'état et d'un changement de contexte.
- Le noyau s'exécute alors dans le nouveau contexte.

# Commutation de contexte

A tout moment, un processus est caractérisé par deux contextes

- le contexte d'unité centrale
  données identiques pour tous les processus
- le contexte dépendant du code du programme exécuté

# Commutation de contexte

## Exécution de processus

- Pour exécuter un nouveau processus, il faut
  1. sauvegarder le contexte d'unité centrale du processus courant (mot d'état)
  2. charger le nouveau mot d'état du processus à exécuter

- Utilisation de 2 adresses :
  - adresse de sauvegarde du mot d'état
  - adresse de lecture du nouveau mot d'état

# Commutation de contexte

Remarques :

- Opération
  - délicate, réalisée matériellement
  - appelée : commutation de mot d'état
  - réalisée de manière non interruptible
- Le compteur ordinal faisant partie du mot d'état
- Le changement de contexte provoque l'exécution dans le nouveau processus

# Commutation de contexte

## Noyau

- Le nouveau processus qui devra réaliser la sauvegarde du contexte global
- Généralement : le noyau réalise cette sauvegarde, le noyau n'ayant pas un contexte du même type
- Le processus interrompu pourra ainsi reprendre exactement où il avait abandonné.
- Les fonctions `setjmp/longjmp` permettent de sauvegarder et de réinitialiser le contexte d'unité centrale du processus courant, en particulier le pointeur de pile.

# Commutation de contexte

## Exécution et création de processus

Pour être exécuté et donner naissance à un processus

- un programme et ses données doivent être chargés en mémoire centrale
- Les instructions du programme sont transférées une à une de la mémoire centrale sur l'unité centrale où elles sont exécutées

# Commutation de contexte

## Registres spécialisés

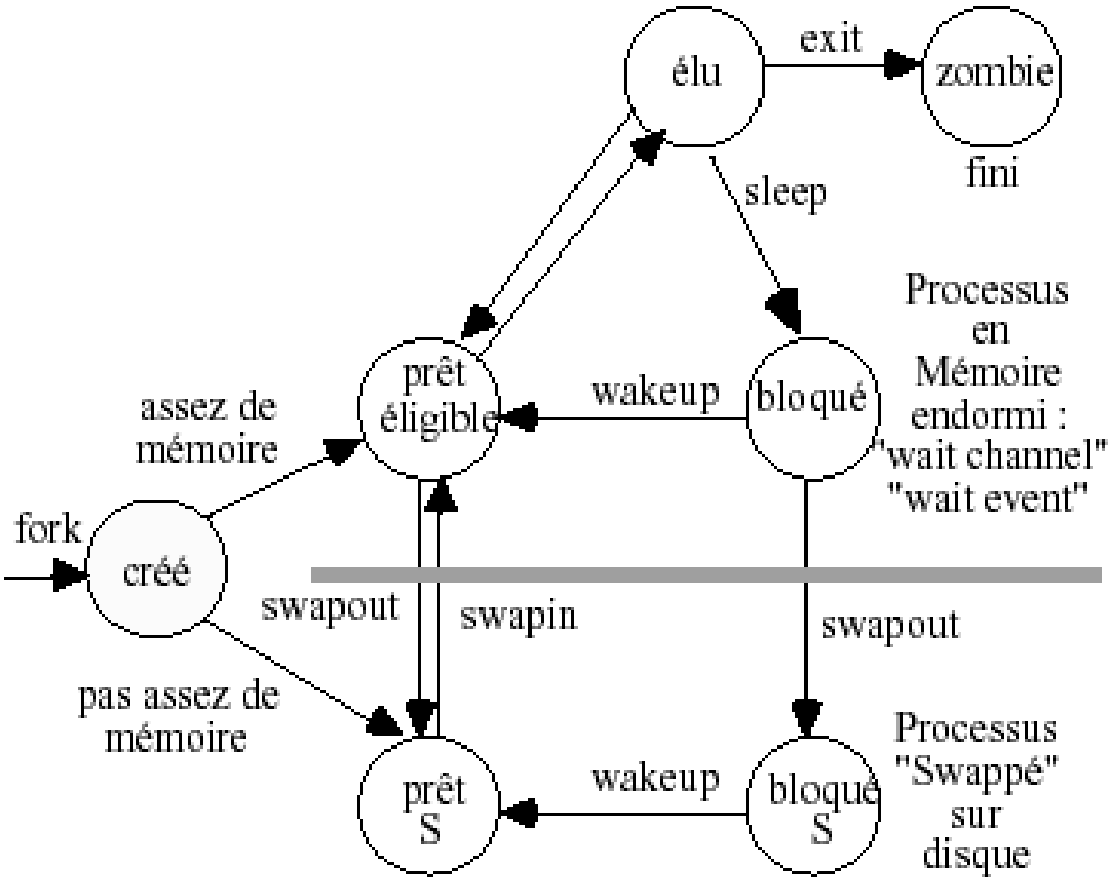
- Unité centrale : circuits logiques et arithmétiques qui effectuent les instructions mais aussi des mémoires appelées registres
- Il existe des registres spécialisés :
  - directement par les constructeurs de l'unité centrale
  - par le programmeur du noyau
- Les registres spécialisés forment le contexte d'unité centrale d'un processus

# Registres spécialisés

- **accumulateur** : reçoit le résultat d'une instruction
  NB : sur les machines à registres multiples, le jeu d'instructions permet souvent d'utiliser n'importe lequel des registres comme accumulateur
- **registre d'instruction** : contient l'instruction en cours
- **compteur ordinal** : adresse de l'instruction en mémoire
  - change au cours de la réalisation d'une instruction pour pointer sur la prochaine instruction à exécuter
  - la majorité des instructions ne font qu'incrémenter ce compteur
  - les instructions de branchement réalisent des opérations plus complexes sur ce compteur : affectation, incrémentation ou décrémentation plus importantes
- **registre d'adresse** et **registres de données** : utilisés pour lire ou écrire une donnée à une adresse spécifiée en mémoire
- **registres d'état du processeur** (actif, mode (user/system), retenue, vecteur d'interruptions, . . . )

# Etats d'un processus

Un processus prend différents états au cours de son exécution



# Etats d'un processus

Les états d'un processus sont les suivants :

- `IDLE` : Le processus en cours de création
- `RUN` : Le processus est en exécution. Il dispose du processeur.
- `Pret` : Le processus attend que le système lui attribue le processeur.
- `SLEEP` : Le processus est en attente d'un évènement particulier (entrée/sortie parexemple).
- `STOP` : Le processus est prêt mais ne demande pas l'accès au processeur.
- `ZOMBIE` : Le processus se termine. Il attend que son père prenne en compte sa terminaison et que le système libère ses ressrources.

# Tâches

- Objectif : analyser à quelles conditions on peut utiliser plusieurs tâches coopérant pour exécuter un programme
- Machine parallèle : possibilité de savoir utiliser plusieurs processeurs en coopération pour
  - réaliser un ensemble de tâches pour une seule application
  - améliorer un système multi-processus
- Intérêts :
  - Gain de temps appréciable
  - Fiabilité importante
    si un des processeurs n'est plus en état de fonctionner, le système peut encore travailler
- Utilisation d'un langage d'expression
- Formalisation : (Dijkstra 1968), Conway (1963)

# Tâches

Difficultés :

- Mode de programmation pas naturel
- Trop peu de systèmes parallèles réellement utilisés
- Programmes parallèles difficiles à déboguer

Mais :

- Montée en puissance des machines serveurs de stations de travail multiprocesseurs
- Mode de programmation par *processus légers* et *threads*

Exemple de parallélisation :

- tri fusion
- produit de matrices (autant de tâches que d'éléments dans la matrice résultat

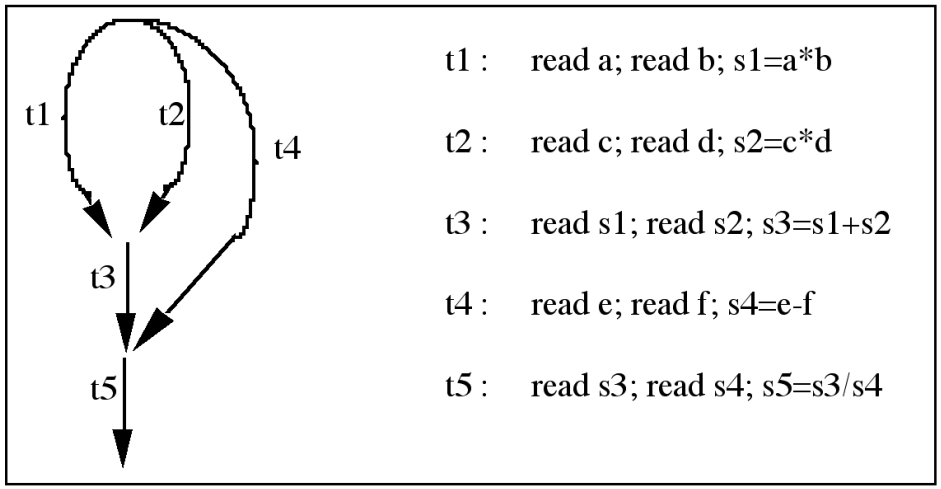
# Langages d'expression

- On suppose réalisé un découpage en tâches $(t_1, ...., t_n)$, $t_i$ de début $d_i$ et de fin $f_i$.
- Une relation $<$ entre tâche, $t_i < t_j$, c'est-à-dire $f_i$ doit être exécutée avant $d_j$
  Cet ordre peut se représenter dans un graphe de flot
- Le système de tâches peut encore être représenté par un graphe de précédence

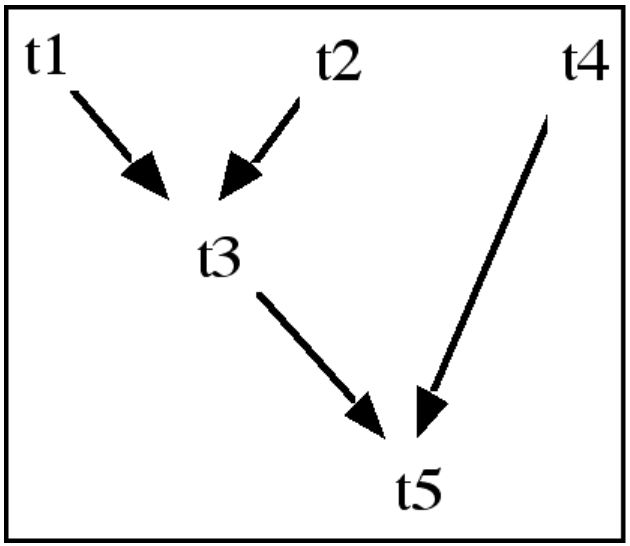
# Langages d'expression

## Exemple

Évaluation de `(a*b + c*d) / (e - f)`




graphe de flot




graphe de précédence

# Primitives `parbegin`, `parend`

(Dijkstra 1968)

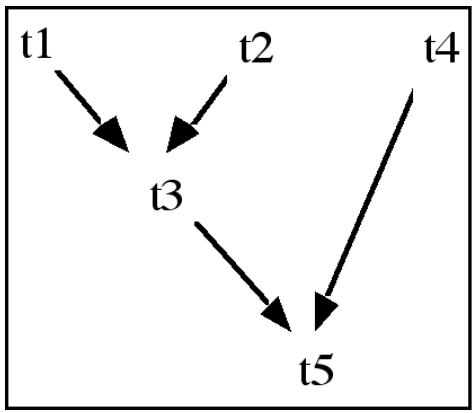
Soit l'instruction

```
parbegin
  code 1 | code 2 | .... .... | code n
parend ;
```

- Exécution en parallèle des $n$ séquences `code 1` à `code n`
- Instruction terminée quand tous les segments de code ont été exécutés

# Exemple

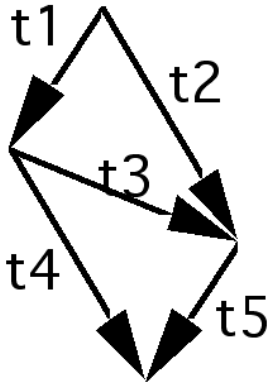
```
(a*b + c*d) / (e - f)
```




```
parbegin
        parbegin t1 | t2 parend ; t3
        t4
parend
t5
```

# Difficultés

- Langage simple mais ne permettant pas de décrire toutes les relations de précédence possibles entre tâches
- Exemple




- Recherche de l'expression la plus intérieure :
  - ce ne peut être `parbegin x | y parend` car il n'existe pas deux tâches ayant le même début et la même fin ;
  - ce ne peut être `x ; y` car il n'existe pas de tâches se succédant sans que `x` termine en même temps qu'une autre ou `y` commence en même temps qu'une autre.

# Primitives de Conway

## Conway (1963)

- On suppose un seul code et des variables partagées par les processus
- Trois primitives de Conway : *fork*, *join* et *quit*
  (où les instructions sont éventuellement étiquetées)
  - **fork x** : création d'un nouveau processus qui s'exécute à partir de la prochaine instruction étiquetée par x.
  - **quit** : fin du processus.
  - **join (t,x)** : instruction indivisible (où x est non partagée, par copie privée de la variable)

    `"t=t-1 ; if t=0 goto x".`

# Primitives de Conway

## Exemple

Calcul de (ab + cd) / (e - f)

```
    m := 2 ; fork p2 ; n := 2 ; fork p4 ;
p1: s1 := a*b ; join m,p3 ; quit ;
p2: s2 := c*d ; join m,p3 ; quit ;
p3: s3 := s1 + s2 ; join n,p5 ; quit ;
p4: s4 := e - f ; join n,p5 ; quit ;
p5: s5 := s3/s4 ;
```

# Primitives de Conway

## Remarques

- Méthode :
  - Identifier une variable pour chaque rencontre dans le graphe de flot (variable initialisée au nombre d'arcs se rencontrant)
  - Établir un fork par arc sortant d'un noeud de rencontre du graphe
  - Simplifier le programme en omettant les forks inutiles
- Le nombre d'instructions `join n,p` est égal à la valeur initiale de `n`, mais la syntaxe n'y oblige pas

# Bilan

- Possibilité d'optimiser la gestion des tâches

# Optimisation de la gestion des tâches

Soit $S = (T, <)$ un système de tâches : $T = (t_1, t_2, ...., t_n)$ et $<$ est un ordre partiel strict.
Soit $w = e_1 e_2 .... e_n$ tel que chaque $e_k$ est soit un $d_i$ soit un $f_i$ (i.e. le début ou la fin d'une tâche $t_i$).

- $d_i$ = lecture des paramètres d'entrée et acquisition des ressources nécessaires à l'exécution ;
- $f_i$ = écriture des résultats et libération des ressources.

# Comportement de $S$

## Définition

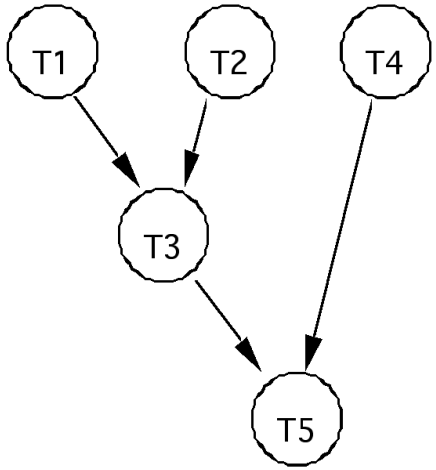
*$w$ est un comportement de $S$ si et seulement si les deux conditions ci-dessous sont vérifiées :*

- $\forall t_i \in T$, *$w$ comporte exactement une occurrence de $d_i$ et une de $f_i$, avec celle de $d_i$ à gauche de celle de $f_i$ ;*
- *si $t_i < t_j$ alors l'occurrence de $f_i$ est à gauche de celle de $d_j$.*

# Exemple de comportement

Exemple : calcul de (ab + cd) / (e - f).

$$
\begin{aligned}
t_1 &: a := a * b;\\
t_2 &: c := c * d;\\
t_3 &: a := a + c;\\
t_4 &: e := e - f;\\
t_5 &: a := a/e;
\end{aligned}
$$




Exemples de comportements :

$w_1 = d_1 d_2 f_1 d_4 f_2 d_3 f_3 f_4 d_5 f_5$

$w_2 = d_2 d_4 f_4 d_1 f_2 f_1 d_3 f_3 d_5 f_5$

# Etats d'un système de tâches

- Représentation de la mémoire centrale par une suite de $p$ cellules $C_1, C_2, ...., C_p$.
- Chaque tâche $t_i$ a
  - un domaine de lecture $L_i$
  - un domaine d'écriture $E_i$
- On considère une succession de $m$ ($m = 2n$) états du système :
  - quand une tâche $t_i$ écrit dans son domaine d'écriture $E_i$, elle exécute une fonction $Ft_i$ de son domaine de lecture (tel qu'il a été lu lors de $d_i$).

# Exemple d'états d'un système de tâches

- Exemple précédent avec le comportement $w1$

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C1 | — | a | a | a*b | a*b | a*b | a*b | a*b+c*d | a*b+c*d | a*b+c*d | Résultat |
| C2 | — | b | b | b | b | b | b | b | b | b | b |
| C3 | — | — | c | c | c | c*d | c*d | c*d | c*d | c*d | c*d |
| C4 | — | — | d | d | d | d | d | d | d | d | d |
| C5 | — | — | — | — | e | e | e | e | e-f | e-f | e-f |
| C6 | — | — | — | — | f | f | f | f | f | f | f |

Remarques :

- Information la plus importante : la suite (notée $V(Ci, w)$) des valeurs inscrites dans la cellule $C_i$ par le comportement $w$
  Par exemple ici : $V(C_3, w_1) = (-, c, c * d)$.
- Les conditions initiales ne figurent pas ici, puisqu'elles ne sont pas utilisées
  d'autres cas prévoient l'utilisation explicite des valeurs initiales.

# Système de tâches déterminé

## Définition

*Un système de tâches* $S = (T, <)$ *est déterminé si,*

> *pour tout couple de comportements* $w$ *et* $w'$ *et pour toute cellule* $C_i$ *de la mémoire,*
>
> $V(C_i, w) = V(C_i, w')$

# Paire de tâches non interférentes

Condition suffisante pour caractériser facilement un système déterminé : paire de tâches non interférentes relativement à un système

## Définition

*Deux tâches $t_i$ et $t_j$ sont non interférentes relativement à $S$ si (conditions de Bernstein) :*

- *soit* $t_i < t_j$ ;
- *soit* $t_j < t_i$ ;
- *soit* $L_i \cap Ej = Lj \cap Ei = Ei \cap Ej = \emptyset$.

# Système de tâches déterminé

## Théorème

*Soit $S = (T, <)$ un système de tâches :*

- *si $S$ est constitué de tâches deux à deux non interférentes, il est déterminé ;*
- *si $S$ est déterminé et si $\forall i, E_i \neq \emptyset$, alors les tâches de $S$ sont deux à deux non interférentes.*

Remarque :

- si $E_i = \emptyset$, $t_i$ peut interférer avec une autre tâche alors que le système reste déterminé, puisque $t_i$ n'écrit pas

  Cas où il a des conditions initiales indéterminées :
  $L_i \cap E_j = \emptyset$ et $\neg(t_i < t_j)$ et $\neg(t_j < t_i)$,

# Parallélisme maximal

## Définition

*Un système $S = (T, <)$ est de parallélisme maximal*

*s'il est déterminé*

*et si tout système $S' = (T, <')$ est non déterminé*

*où $<'$ est obtenu en supprimant du graphe de $<$ un couple qui est dans le graphe de précédence de $<$.*

# Systèmes équivalents

## Définition

*deux systèmes $S = (T, <)$ et $S' = (T, <')$, de même ensemble de tâches, mais munis de relations d'ordre différentes, sont équivalents*

> *si pour tout comportement $w$ de $S$, pour tout comportement $w'$ de $S'$ et pour toute cellule $C$, $V(C, w) = V(C, w')$.*

Remarque : un système déterminé est un système équivalent à lui-même

# Système de parallélisme maximal unique

## Théorème

*Pour tout système $S$ déterminé et tel que $\forall i, E_i \neq \emptyset$, il existe un unique système de parallélisme maximal équivalent à $S$*

# Système de parallélisme maximal unique

## Preuve

1. Il existe un système :
   - $T$ est fini,
   - donc aussi le graphe de $<$ et la collection des sous-graphes transitifs de $<$ qui sont déterminés.
   - Une collection finie d'ensembles a toujours au moins un élément minimal pour l'inclusion
     (car $\subset$ est une relation non circulaire).
   - Soit $S' = (T, <')$ un tel élément minimal
     tout comportement de $S$ est aussi un comportement de $S'$,
     donc $S$ et $S'$ sont équivalents.

# Système de parallélisme maximal unique

## Preuve

2. Ce système est unique :
   sinon
   - soit $S'' = (T, <'')$ un autre système de parallélisme maximal équivalent à $S$
   - Il existe deux tâches $t_1$ et $t_2$ telles que $t_1 <' t_2$ et $\neg(t_1 <'' t_2)$
   - $S''$ est déterminé et $\neg(t_1 <'' t_2)$ et pas de tâche de domaine d'écriture vide
   - donc $t_1$ et $t_2$ sont non interférentes.
   - Mais alors, on peut supprimer $t_1 <' t_2$ de $S'$, qui n'est donc pas de parallélisme maximal
     $\rightarrow$ Contradiction

# Système de parallélisme maximal : algorithme

Algorithme pour obtenir un système de parallélisme maximal à partir d'un système déterminé de domaines d'écriture non vides :

1. On garde seulement les couples indispensables, c'est à dire les couples $(t_i, t_j)$ tels que :
   $t_i < t_j$ et $E_i \neq \emptyset$ et $E_j \neq \emptyset$ et ($E_i \cap L_j \neq \emptyset$ ou $L_i \cap E_j \neq \emptyset$ ou $E_i \cap E_j \neq \emptyset$).
2. Ceci fait, on supprime les arcs redondants

Graphe obtenu : graphe de précédence du système de parallélisme maximal équivalent au système de départ

# Tâches sous Unix

- Avec les processus "classiques"
- Le système Unix offre deux primitives : `fork` et `wait`.
- Primitives complémentaires : `exec()`, `exit`
- Note : Il existe des variantes à ces fonctions système. Leur principe est toutefois celui que l'on vient d'énoncer.

# Création de processus `fork()`

**int fork()** :

- Création un deuxième processus (le fils) qui ne diffère du père que par la valeur de retour du fork :
  - 0 pour le fils
  - <numéro du fils> pour le père
- Pas d'association spécifique de `fork()` à une étiquette dans le programme
  mais simuler facilement par :

```
if (fork()=0) goto ? ;
```

# Création de processus `fork()`

La fonction `fork` réalise la séquence suivante :

- s'assurer qu'il y a assez de place en mémoire
- trouver une entrée libre dans la table de descriptions des processus
  et vérifier que le processus père ne dépasse pas une limite maximale de processus fils
- créer une copie du processus père, i.e. dupliquer l'entrée du père dans la table des processus
  incrémenter les compteurs associés aux fichiers ouverts, aux segments partagés
- mettre à jour des environnements des processus, i.e. valeur de retour de la fonction

# Recouvrement de processus `exec()`

**exec(char *filename, char *argv[], char *envp[])**

- Exécution du programme du fichier `filename`
- Fonctionnement :
  - Vérifications de droits d'accès au fichier
  - Désallocation des régions de mémoire
  - Allocation des régions nécessaires pour l'exécution du nouveau programme
  - Initialisation u nouvel environnement du processus (sauf les paramètres relatifs au processus comme l'horloge)
- Remarques :
  - La fonction ne crée donc pas de nouveau processus
  - Elle ne fait que substituer une exécution de programme à une autre pour un processus donné

# exit()

## exit(int status)

- Fonction système de terminaison d'un processus
- Fonctionnement :
  - Fermeture des fichiers ouverts, i.e. décrémentation ees compteurs associés aux entrées de la table des descriptions de fichiers
  - Libération du répertoire courant, le répertoire racine et le code, la mémoire.
  - Le processus
    - passe à l'état *zombie*
    - met à jour le descripteur de processus
    - envoie un signal de terminaison au processus père (cf. appel système `wait`)
    - associe ses processus fils au processus 1

# wait()

**wait(int *status)**

- Suspension de l'exécution du processus courant en le mettant dans un état *Sleep*
- L'état du processus ne sera modifié que par un événement (le fait qu'un fils se termine)
- La valeur de retour du processus fils est récupérée dans le pointeur status

# Remarques

Solution très couteuse en temps en raison des inconvénients suivants :

- `fork()` : "lourd" en raison des diverses vérifications et copies effectuées par le noyau pour sa réalisation
- Changement de contexte long, notamment pour les applications du type "temps réel" ou "multi média"
- Pas de partage de mémoire : communications lentes, problème dans le cas des architectures SMP [1]
- Manque d'outils de synchronisation
- Interface rudimentaire (`fork, exec, exit, wait`)

Autre solution : processus légers (threads)

# Signaux

## Introduction

- Signal : une interruption logicielle envoyée aux processus par le système
- Objectif : informer les processus sur des événements anormaux se déroulant dans leur environnement
  - violation mémoire,
  - erreur dans les entrées/sorties
- Également : moyen de communication entre les processus

# Traitement d'un signal

Trois types de traitement d'un signal (sauf **SIGKILL**) :

- **ignoré**
  - le programme peut ignorer les interruptions clavier générées par utilisateur
  - Exemple : processus lancé en background (bg)
- **Pris en compte**
  À la réception d'un signal :
  - détournement de l'exécution d'un processus vers une procédure spécifiée par l'utilisateur
  - puis reprise où l'exécution a été interrompue
- **Comportement par défaut**
  - restitué par un processus après réception de ce signal

# Identification des signaux

- Identification des signaux par le système par un nombre entier
- Liste des signaux accessibles : `/usr/include/signal.h` (`/usr/include/bits/signum-generic.h`)
- Chaque signal est caractérisé par un mnémonique

```
1 : SIGINT   /* Interruption du processus */
9 : SIGKILL  /* Terminaison du processus */
10 : SIGUSR1 /* Signal défini par l'utilisateur */
13 : SIGPPIPE /* Ecriture dans un tube fermé */
17 : SIGCHLD /* Terminaison d'un fils */
```

# Signaux particuliers

- Signal **SIGCHLD** : gestion des processus zombie
- Signal **SIGHUP** : gestion des applications longues

# Signal **SIGCLD** : gestion des processus zombie

**SIGCHLD** :

- Signal envoyé au processus père, par un processus fils pour l'informer de sa terminaison
  Processus zombie : processus *en cours de terminaison*, sans ressource allouée, mais toujours présent dans la table des processus
- Comportement différemment des autres :
  - Si ignoré
    c'est-à-dire, le processus père n'est pas en attente de la terminaison d'un fils (`wait()`, `waitpid()`)
  - La terminaison d'un processus fils n'entraîne pas la création de processus zombie

# Signal **SIGHUP** : gestion des applications longues

`SIGHUP`

- Terminaison du processus père ou du terminal de contrôle
  $\rightarrow$ par défaut, provoque la terminaison du processus
- Problème : Laisser un processus se poursuive après la fin de la session de travail (application longue)
  Si le processus ne traite pas ce signal : interruption par le système au moment de la déconnexion

# Signal **SIGHUP** : gestion des applications longues

Différentes solutions :

1. Commande shell `at`,
   - Exécution du programme une certaine date, via un processus du système (appelé *démon*)
   - Le processus n'est alors attaché à aucun terminal, le signal **SIGHUP** sera sans effet
   - `at -f $HOME/prog1 10:00 11/04/2020`
2. Lancer le programme en arrière-plan
   - un processus lancé en background traite automatiquement le signal **SIGHUP**
   - `$HOME/prog1 &`

# Signal **SIGHUP** : gestion des applications longues

Différentes solutions (suite) :

3. Lancer le programme sous le contrôle de la commande `nohup`
   - entraîne un appel à `trap`
   - redirige la sortie standard sur `nohup.out`
   - `nohup $HOME/prog1`
4. Inclure dans le code du programme, la réception du signal **SIGHUP**
   - signal ()
   - **struct** sigaction et sigaction ()

# Traitement des signaux

- Emission d'un signal : **kill()** et **alarm()**
- Signaux de base : **signal(), pause()**
- Signaix POSIX : **struct sigaction**, **sigaction()**, **sig*()**

# Emission d'un signal

## Primitive kill()

- Emission à destination du processus de numéro *pid* le signal de numéro *sig*
- Si l'entier *sig* est nul,
  - aucun signal n'est envoyé
  - la valeur de retour permet de savoir si le nombre *pid* est un numéro de processus ou non.
- Remarque : la primitive `kill()` est le plus souvent exécutée via la commande shell `kill`

```
#include <signal.h>

int kill(pid,sig)     /* emission d'un signal */
int pid ;             /* identificateur du processus
                         ou du groupe destinataire */
int sig ;             /* numero du signal */
```

**Valeur retournée :** 0 si le signal a été envoyé, -1 sinon.

# Primitive kill()

**Utilisation du paramètre pid :**

- Si *pid* $> 0$ : *pid* désigne le processus d'identificateur *pid*
- Si *pid* $= 0$ : le signal est envoyé à tous les processus du même groupe que l'émetteur

  Utilisée avec la commande shell `kill` pour tuer tous les processus en arrière-plan sans avoir à indiquer leurs identificateurs de processus : `kill -9 0`

# Primitive kill()

- Si *pid* = -1 :
  - Si le processus appartient au super-utilisateur,
    signal envoyé à tous les processus, sauf aux processus système et au processus qui envoie le signal
  - sinon,
    signal envoyé à tous les processus dont l'identificateur d'utilisateur réel est égal à l'identificateur d'utilisateur effectif du processus qui envoie le signal
    → Moyen de tuer tous les processus dont on est propriétaire, indépendamment du groupe de processus
- Si *pid* < -1 :
  signal envoyé à tous les processus dont l'identificateur de groupe de processus (*pgid*) est égal à la valeur absolue de *pid*

# Emission d'un signal

## Primitive alarm()

- Envoi d'un signal **SIGALRM** au processus appelant
  - après un laps de temps *secs* (en secondes) passé en argument,
  - puis réinitialisation l'horloge d'alarme.
- Si le paramètre *secs* est nul, toute requête est annulée.
- Utilisation : forcer la lecture au clavier dans un délai donné
- Le traitement du signal doit être prévu, sinon le processus est tué.

```
#include <signal.h>

unsigned alarm(secs)          /* envoi d'un signal SIGALRM */
unsigned secs ;               /* nombre de secondes */
```

**Valeur retournée :** temps restant dans l'horloge.

# Signaux de base

## Primitive signal()

- Interception du signal de numéro *sig*
- Possibilité de modifier le comportement par défaut (sauf **SIGKILL**)
  Second argument : pointeur de fonction sur une fonction *utilisateur* qui sera associée au signal

```
#include <signal.h>

int (*signal(sig,fcn)) ()     /* reception d'un signal */
int sig ;                     /* numero du signal */
int (*fcn)() ;                /* action apres reception */
```

**Valeur retournée :** adresse de la fonction spécifiant le comportement du processus vis-à-vis du signal considéré, -1 sinon.

# Primitive signal()

**int** (∗signal(in sig, **int** (∗fcn)())) ()

Second argument : pointeur sur une fonction prenant les valeurs :

1. **SIG_DFL** : Action par défaut pour le signal
   - Réception d'un signal : terminaison de ce processus,
   - Pour **SIGCHLD** et **SIGPWR** : ignorés par défaut
   - pour certains signaux : création d'un fichier image `core` sur disque
2. **SIG_IGN** : Indication que le signal doit être ignoré
   Immunisation du processus (sauf pour **SIGKILL**)
3. Pointeur sur une fonction (nom de la fonction) : captage du signal
   - Appel de la fonction quand le signal arrive
   - Après son exécution, reprise du traitement du processus où il a été interrompu
   - Sauf pour le signal **SIGKILL** puisque signal pas interceptable

# Primitive signal()

## Remarque

- Possibilité de modifier le comportement d'un processus à l'arrivée d'un signal donné
- Comportement habituel pour un certain processus standards : Affichage par le shell du prompt '**$**' À la réception d'un signal **SIGINT** (pas d'interruption)

# Signaux de base

## Primitive pause()

```
void pause()            /* attente d'un signal quelconque */
```

- Attente pure :
  - Aucune action
  - Pas d'attente
- `pause()` attend un signal puisque l'arrivée d'un signal interrompt toute primitive bloquée
  - Comportement de retour classique d'une primitive bloquée : positionnement de `errno` à **EINTR**.
- En général, le signal attendu par `pause()` est l'horloge d'`alarm()`.

# Signaux POSIX

## Norme POSIX

- Pas de définition du comportement d'interruption des appels systèmes
  à spécifier dans la structure de traitement du signal
- Introduction des ensembles de signaux
  - Dépassement de la contrainte classique qui veut que le nombre de signaux soit inférieur ou égal au nombre de bits des entiers de la machine
  - Fourniture des fonctions de manipulation de ces ensembles et définition de masques

# Ensembles de signaux POSIX

- Type `sigset_t`
- Manipulables grâce aux fonctions :

```
int sigemptyset(sigset_t *ens)          /* raz */
int sigfillset(sigset_t *ens)           /* ens = { 1,2,...,NSIG} */
int sigaddset(sigset_t *ens, int sig)   /* ens = ens + {sig} */
int sigdelset(sigset_t *ens, int sig)   /* ens = ens - {sig } */
```

Ces fonctions retournent -1 en cas d'échec et 0 sinon.

```
int sigismember(sigset_t *ens, int sig);    /* sig appartient
                                               a ens */
```

Cette fonction retourne vrai si le signal appartient à l'ensemble.

# Blocage des signaux

- Signal bloqué/masqué : signal dont la délivrance est différée jusqu'à la levée du blocage
- Masque : ensemble des signaux bloqués,
- Manipulation du masque de signaux du processus :

```
#include <signal.h>
int sigprocmask(int op, const sigset_t  *nouv,
                sigset_t *anc);
```

L'opération op :

- **SIG_SETMASK** : affectation du nouveau masque, recupération de la valeur de l'ancien masque.
- **SIG_BLOCK** : union des deux ensembles `nouv` et `anc`
- **SIG_UNBLOCK** : soustraction `anc - nouv`

# Signal pendant

Indication de signal pendant et donc bloqué :

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

- retourne -1 en cas d'échec
- retourne 0 sinon et l'ensemble des signaux pendants est stocké à l'adresse `ens`

# Structure sigaction

Description du comportement utilisé pour le traitement d'un signal

```
struct sigaction {
        void (*sa_handler) ();
        sigset_t sa_mask;
        int sa_flags;}
```

- **sa_handler** : fonction de traitement (ou **SIG_DFL** et **SIG_IGN**)
- **sa_mask** : ensemble de signaux supplémentaires à bloquer pendant le traitement
- **sa_flags** : différentes options

# Structure sigaction

- Options de **sa_flags** :
  - **SA_NOCLDSTOP** : pas d'envoi du signal SIGCHLD à un processus lorsque l'un de ses fils est stoppé
  - **SA_RESETHAND** :
    - Simulation de l'ancienne méthode de gestion des signaux
    - Pas de blocage du signal pendant le *handler*
    - Repositionnement du *handler* par défaut au lancement du *handler*.
  - **SA_RESTART** :
    - Relance des appels système interrompus par un signal capté au lieu de renvoyer -1.
    - L'indicateur joue le rôle de l'appel `siginterrupt(sig, 0)` des versions BSD.
  - **SA_NOCLDWAIT** :
    - si le signal est **SIGCHLD**, ses fils qui se terminent ne deviennent pas zombis.
    - L'indicateur correspond au comportement des processus pour **SIG_IGN** dans les versions ATT.

# Primitive **sigaction()**

- Positionnement du comportement de réception d'un signal avec la primitive `sigaction`
- La fonction réalise
  - une demande d'information.
    Si le pointeur `paction` est `null`, on obtient la structure `sigaction` courante.
  - une demande de modification du comportement.

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

# Primitive **sigaction()**

Remarques :

- Un *handler* positionné par `sigaction` reste jusqu'à ce qu'un autre *handler* soit positionné
- Dans les versions des versions ATT, le *handler* par défaut est repositionné automatiquement au début du traitement du signal

# Attente d'un signal

(en plus de l'appel `pause`)

- Appel POSIX `int sigsuspend(const sigset_t *ens);`
  Réalisation atomiques des actions :
  - installation du masque de blocage défini par `ens` (qui sera repositionné à sa valeur d'origine) à la fin de l'appel
  - mise en attente de la réception d'un signal non bloqué.

# Héritage des signaux par `fork()`

- le comportement vis-à-vis des signaux est hérité par les processus fils recevant l'image mémoire du père (`fork()`)
- Exemple : voir transparent suivant

# Exemple d'héritage des signaux par `fork()`

```c
/* heritage par le fils du comportement du pere vis-a-vis des signaux */

#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<signal.h>
#include<sys/types.h>
#include<unistd.h>

void fin(int c) ;

int main(int argc, char *argv[])
{
      signal(SIGQUIT,SIG_IGN) ;
      signal(SIGINT,fin) ;
      if (fork()>0){
            printf("processus pere : %d\n",getpid()) ;
            while(1) ;
      }
      else {
            printf("processus fils : %d\n",getpid()) ;
            while(1) ;
      }
}
void fin(int c)
{
      printf("SIGINT pour le processus %d\n",getpid()) ;
      exit(EXIT_SUCCESS) ;
}
```

# Exemple d'héritage des signaux par `fork()`

**Résultat de l'exécution :**

```
$ ./test_sign_fork
processus pere : 14584
processus fils : 14585
^D   <-----pas de reaction
^C
CSIGINT pour le processus 14585
SIGINT pour le processus 14584
```

# Conclusion

- Sauf **SIGCHLD**, les signaux qui arrivent ne sont pas mémorisés
- Comportement des signaux
  - Ils sont ignorés,
  - ils mettent fin aux processus, ou bien ils sont interceptés

# Inconvénients des signaux

1. Inadaptés pour la communication inter-processus :
   - Perte d'un message sous forme de signal s'il est reçu à un moment où ce type de signal est temporairement ignoré
   - Capture d'un signal par un processus : retour au comportement par défaut vis-à-vis de ce signal
     - Capter plusieurs fois un même signal plusieurs fois nécessite de redéfinir le comportement du processus par la primitive `signal()`
     - En général, réarmement de l'interception du signal le plus tôt possible
       $\rightarrow$ Première instruction effectuée dans la procédure de traitement du déroutement

# Inconvénients des signaux

2. Comportement brutal :
   - à leur arrivée, interruption du travail en cours
   - Exemple : réception d'un signal pendant que le processus est en attente d'un événement
     ( lors de l'utilisation des primitives `open()`, `read()`, `write()`, `msgrcv()`, `pause()`, `wait()` ...)
     - exécution de la fonction de déroutement
     - à son retour, la primitive interrompue renvoie un message d'erreur sans s'être exécutée totalement (`errno` est positionné à EINTR).

# Inconvénients des signaux

3. Gestion des signaux attendus
4. Par exemple
   1. Un processus père
      - intercepte les signaux d'interruption et d'abandon
      - en cours d'attente de la terminaison d'un fils
   2. Un signal d'interruption ou d'abandon éjecte le père hors du `wait()` avant que le fils n'ait terminé
      $\rightarrow$ Création d'un `<defunct>` (processus zombie)
5. Solution : ignorer certains signaux avant l'appel de telles primitives
   Autres problèmes : pas de traitement de ces signaux

# Tubes et tubes nommés

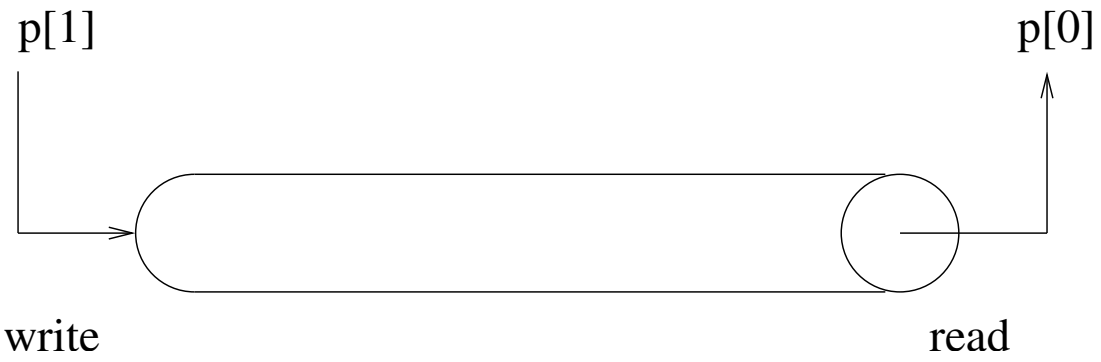
Tubes (ou *pipes*)

- Mécanisme fondamental de communication unidirectionnelle entre processus
- Files de caractères FIFO (First In First Out)

  *les informations y sont introduites à une extrêmité et en sont extraites à l'autre*

- Implémentation : comme des fichiers
  - tubes ordinaires : possèdent un i-node mais pas de nom dans le système)
  - tubes nommées : existent dans le système de fichiers (i-node, et nom)

# Tubes et tubes nommés




- Tube ordinaires : Technique fréquemment mise en œuvre dans le shell
  - Redirection de la sortie standard d'une commande sur l'entrée d'une autre (symbole |)
  - Exemple : ls | more

# Particularités des tubes

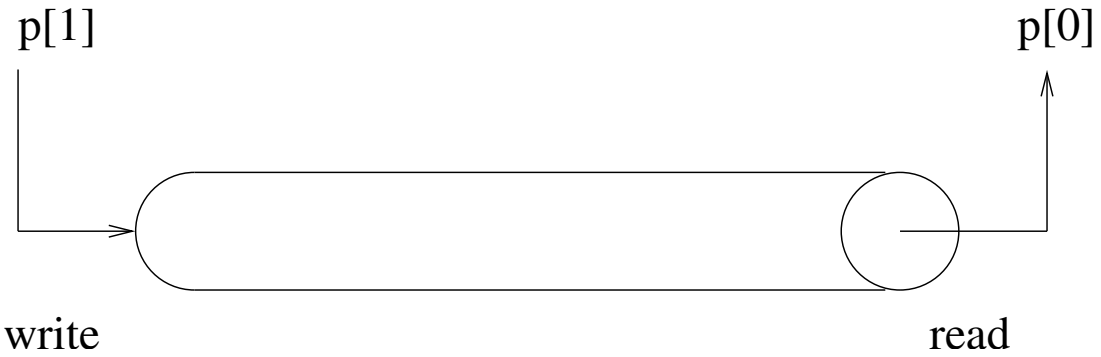
- Plusieurs processus peuvent écrire et lire sur un même tube,

  *mais pas de mécanisme de différenciation des informations à la sortie*

- Capacité limitée : en général à 4096 octets
- Impossible de se déplacer à l'intérieur d'un tube (FIFO)

# Tubes ordinaires (*pipe*)

- Matérialisation des tubes : 2 entrées de la table des fichiers ouverts
  - une entrée est ouverte en écriture (entrée du tube)
  - une entrée est ouverte en lecture (sortie du tube)



# Tubes ordinaires (*pipe*)

- Les deux entrées de la table des fichiers ouverts : nombre de descripteurs qui pointent sur elles
  - nombre de lecteurs : de descripteurs associés à l'entrée ouverte en lecture

    *On ne peut pas écrire dans un tube sans lecteur*
  - nombre d'écrivains : nombre de descripteurs associés à l'entrée ouverte en écriture

    *La nullité de ce nombre définit le comportement de la primitive read lorsque le tube est vide*

# Création de tubes ordinaires

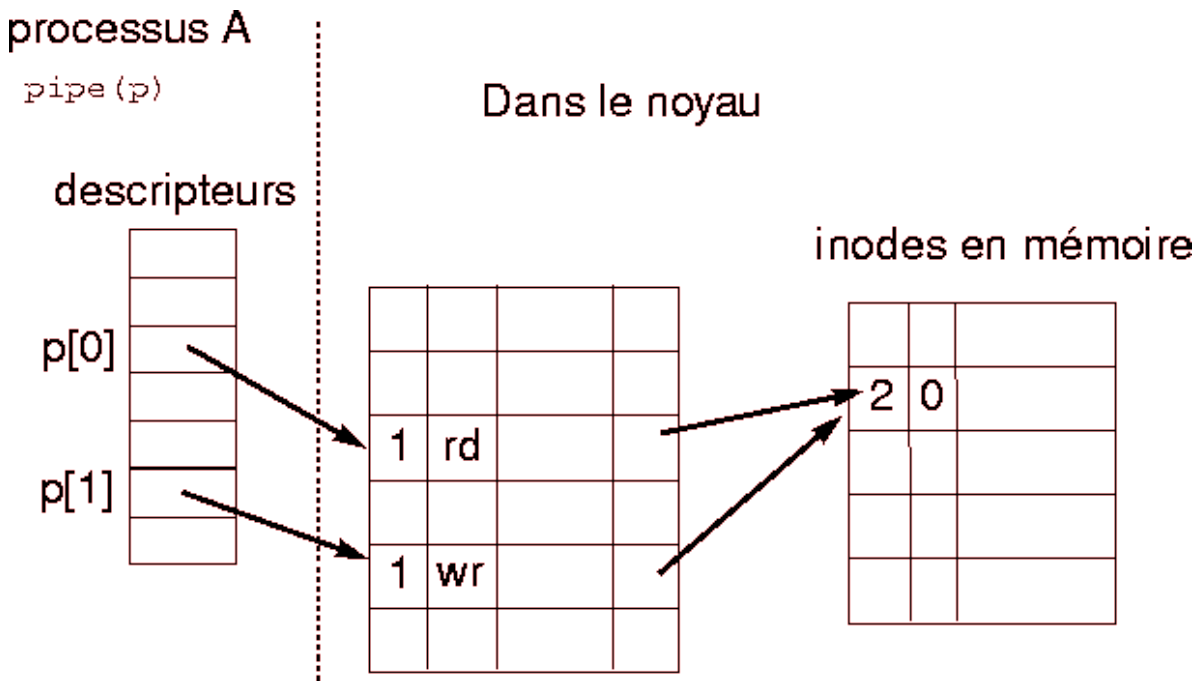
Un processus ne peut utiliser que les tubes

- qu'il a créés lui-même par la primitive `pipe()`

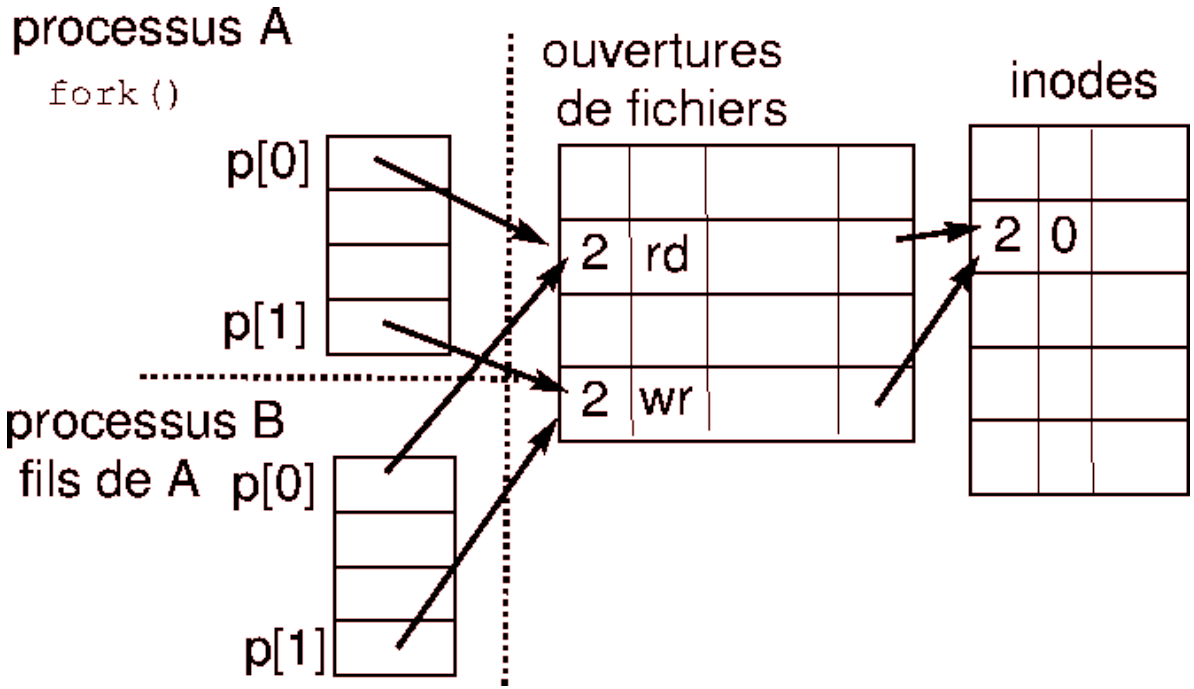
```
#include <unistd.h>
int pipe(int p[2]);
```

- qu'il a hérités de son père grâce à l'héritage des descripteurs à travers `fork` et `exec`

# Création de tubes ordinaires



# Héritage de tubes ordinaires




*Le processus B hérite des descripteurs ouverts par son père A et donc ici, du tube*

# Lecture dans un tube

- Appel système : `read()`

```
int nb_lu;
nb_lu = read(p[0], buffer, TAILLE_READ);
```

- La lecture se fait dans le descripteur p[0]

# Algorithme de lecture dans un tube

```
Si le tube n'est  pas vide et contient  taille caracteres:
    lecture de nb_lu = min(taille, TAILLE_READ) caracteres.
Si le tube est vide
    Si le nombre d'ecrivains est nul
        alors c'est la fin de fichier et nb_lu est nul.
    Si le nombre d''ecrivains est non nul
        Si lecture bloquante
            alors sommeil
        Si lecture non bloquante
            alors en fonction de l'indicateur
                O_NONBLOCK nb_lu= -1 et errno=EAGAIN.
                O_NDELAY nb_lu = 0.
```

# Ecriture dans un tube

- Appel système `write()`

```
nb_ecrit = write(p[1], buf, n);
```

- L'écriture est atomique si le nombre de caractères à écrire est inférieur à `PIPE_BUF` (la taille du tube sur le système)

# Algorithme d'écriture dans un tube

```
 Si le nombre de lecteurs est nul
    envoi du signal SIGPIPE à l'ecrivain.
Sinon
    Si l'ecriture est bloquante,
        retour que quand les n caracteres ont ete ecrits dans le tube.
    Si ecriture non bloquante
        Si n > PIPE_BUF
            retour avec un nombre inferieur a n eventuellement -1 !
        Si n  PIPE_BUF
            et si n emplacements libres, ecriture nb_ecrit = n
               sinon retour -1 ou 0.
```

# Tubes nommés

Tube nommé :

- tube (*pipe*) qui existe dans le système de fichiers
- Possibilité d'ouverture grâce à une référence (nom de fichier)
- Création et utilisation :

Etape 1 création dans le système de fichiers, grâce à la primitive `mknod()` (`mkfifo()`)

```
int mknod(reference, mode | S_IFIFO,0);
```

`mode` est construit comme le paramètre de mode de la fonction `open`

Etape 2 ouverture avec la primitive `open()`
Ouverture d'un tube nommé exclusivement

- soit en mode `O_RDONLY`
- soit en mode `O_WRONLY`

le nombre de lecteur et d'écrivain peut ainsi être comptabilisé

# Tubes nommés

- En POSIX, un appel simplifié :

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

- Création des FIFOs à partir du shell :

```
mkfifo [-p] [-m mode] ref ...
```

# Ouverture et synchronisation des ouvertures de tubes

- Synchronisation automatique des processus qui ouvrent en mode bloquant un tube
- Opération d'ouverture sur un tube : bloquante en lecture
  Le processus attend qu'un autre processus ouvre la fifo en écriture
- Ouverture en écriture : bloquante, avec attente qu'un autre processus ouvre le tube en lecture
  L'ouverture bloquante se termine de façons synchrone pour les deux processus
- Remarque : un unique processus ne peut ouvrir à la fois en lecture et écriture un tube

# Mode d'ouverture d'un tube

- Modification du mode d'ouverture par défaut à l'aide de `fcntl()`

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

int fcntl(int fd, F_SETFL, int flags );
```

`flags` :

  - `O_NONBLOCK` : lecture est non bloquante
  - `O_NDELAY` : lecture est non bloquante sans délai

- Pour paramétrer le tube lors de sa création

```
int pipe2(int p[2], int flags);
```

# Mode d'ouverture d'un tube

- En mode non bloquant (`O_NONBLOCK`, `O_NDELAY`) : seule l'ouverture en lecture réussit dans tous les cas
- Ouverture en écriture en mode non bloquant d'un tube ne fonctionne que si
  - un autre processus a déjà ouvert en mode non bloquant le tube en lecture
  - il est bloqué dans l'appel d'une ouverture en lecture en mode bloquant.

  Permet d'éviter
  - l'écriture dans le tube ouvert par le processus avant qu'il n'y ait de lecteur
  - et l'émission d'un signal `SIGPIPE` (tube détruit)
    $\rightarrow$ faux car le tube n'a pas encore été utilisé

# Suppression d'un tube nommé ou ordinaire

- `rm` ou `unlink` : destruction de la référence
- Si tous les liens par référence sont détruits, le tube nommé devient un tube ordinaire
- Fermeture d'un tube ordinaire : primitive `close()` (suppression d'un lien interne)
- Destruction réelle du tube lorsque son compteur de liens internes et externes est nul

To be continued...