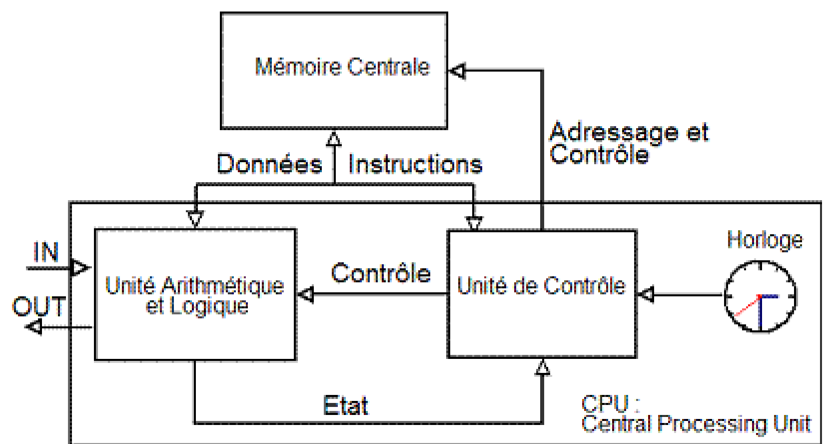
Systèmes d'Exploitation

Mécanismes de base

version 1



Y. Challal, H. Bettahar, M. Vayssade

Paternité - Pas d'Utilisation Commerciale :
http://creativecommons.org/licenses/by-nc/2.0/fr/

Table des matières

Objectifs 5

I - Mécanismes de base 7

A. Architecture matérielle d'une machine Von Neumann.............................7

B. Cheminement d'un programme dans un système................................10

C. Modèle de processus...16

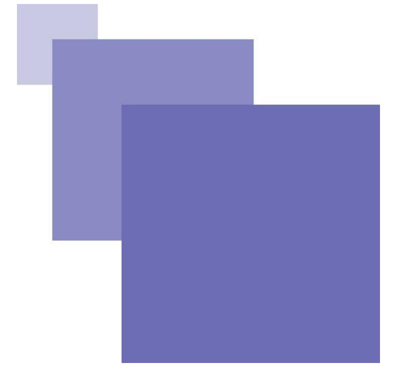
D. Processus sous UNIX...19

E. Commutation de contexte..24

F. Appels système...25

G. Le système d'interruption..31

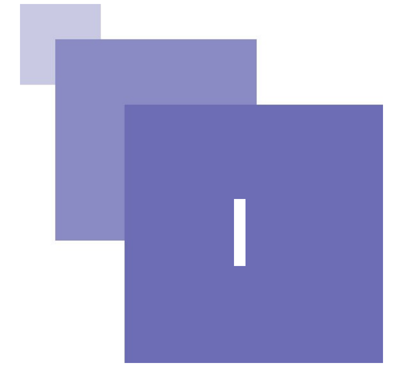
H. Les signaux sous UNIX...36



Objectifs

- Exploiter les mécanismes de base d'exécution de programmes
- Utiliser l'API UNIX pour créer des processus
- Utiliser l'API UNIX pour faire communiquer des processus à travers des signaux

Mécanismes de base



Architecture matérielle d'une machine Von Neumann	7
Cheminement d'un programme dans un système	10
Modèle de processus	16
Processus sous UNIX	19
Commutation de contexte	24
Appels système	25
Le système d'interruption	31
Les signaux sous UNIX	36

Dans cette partie du cours, nous étudierons les mécanismes de base utilisés par les systèmes d'exploitation pour réaliser leurs fonctions fondamentales.

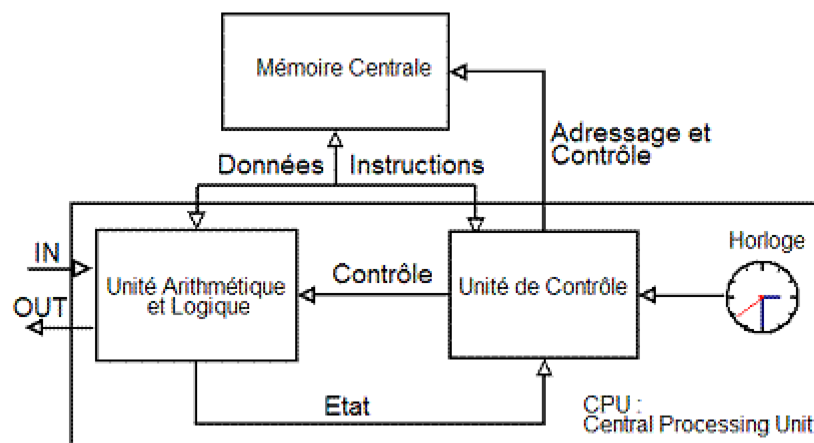
A. Architecture matérielle d'une machine Von Neumann

Machine de Von Neumann

Von Neumann (cf. 'Von Neumann' p 8) proposa l'architecture d'un calculateur qui inspire toujours les processeurs les plus performant d'aujourd'hui.

Von Neumann

L'architecture (cf. 'Architecture de Von Neumann' p 8), dite architecture de Von Neumann, est un modèle pour un ordinateur qui utilise une unique structure de stockage pour conserver à la fois les instructions et les données requises ou générées par le calcul. L'architecture de Von Neumann est composée d'une mémoire centrale, d'une unité centrale ou CPU (Central Processing Unit), et de dispositifs d'entrées/sorties qui permettent de communiquer avec le monde extérieur.



Architecture de Von Neumann

Définition : L'unité centrale

L'unité centrale (appelée aussi processeur) est à son tour composée d'une unité arithmétique et logique (ALU) et d'une unité de contrôle ou de commande (UC). L'unité arithmétique et logique réalise une opération élémentaire (addition, soustraction, multiplication, ...) du processeur à chaque top d'horloge. L'unité de commande contrôle les opérations sur la mémoire (lecture/écriture) et les opérations à réaliser par l'ALU selon l'instruction en cours d'exécution.

Les registres du processeur

Le processeur central dispose d'un certain nombre de registres physiques :

- **Le compteur ordinal (CO) :** pointe vers la prochaine instruction à exécuter.

- **Le registre instruction (RI) :** contient l'instruction en cours d'exécution.
- **Les registres généraux :** registre accumulateur, registre index, etc.
- **Le mot d'état (ou PSW pour Program Status Word) :** indique l'état du processeur (actif, attente, etc.), le mode d'exécution (maître/esclave), le compteur ordinal, masque d'interruption, etc..
- etc.

Attention : PSW

L'équivalent du PSW sur les machines actuelles peut être composé de plusieurs registres.

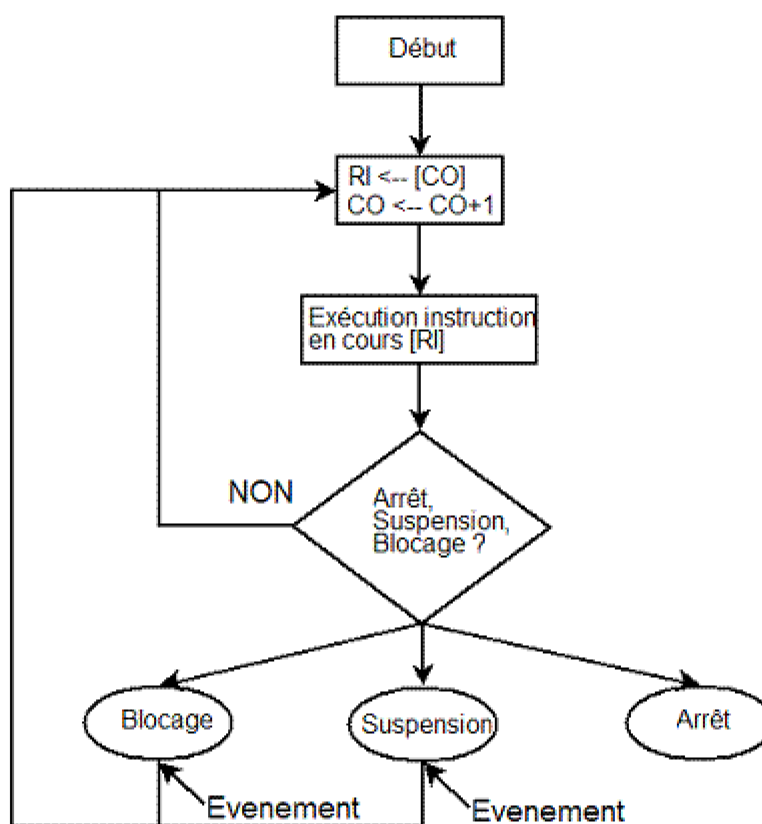
Sur une machine Intel, PSW=PC+PS

Méthode : Cycle d'exécution du processeur

Pratiquement, toutes les instructions se déroulent en quatre étapes principales qui sont :

- la recherche de l'instruction en mémoire
- le décodage de l'instruction
- l'exécution de l'instruction
- le passage à l'instruction suivante

La figure (cf. 'Cycle d'exécution d'une instruction' p 9) résume ces quatre étapes.

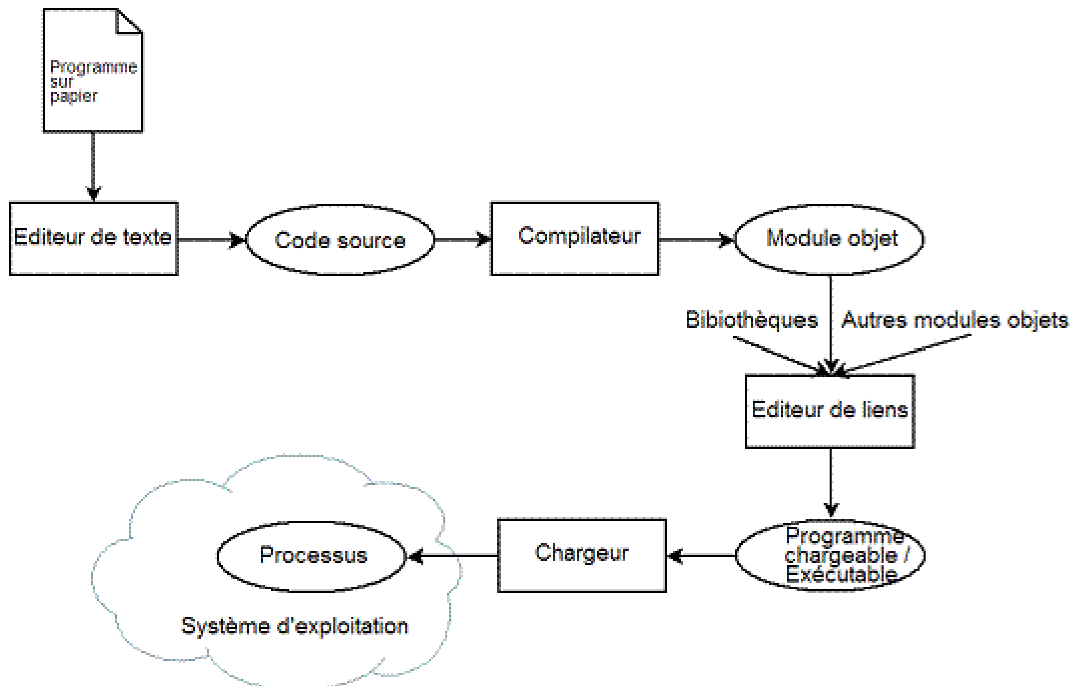


Cycle d'exécution d'une instruction

B. Cheminement d'un programme dans un système

Du code source au procesus

Un programme est généralement écrit dans un langage évolué (Pascal, C, VB, Java, etc.). Pour faire exécuter un programme par une machine, on passe généralement par les étapes suivantes (cf. 'Cheminement d'un programme dans un système' p 10):



Cheminement d'un programme dans un système

1. **Édition :** on utilise généralement un éditeur de texte pour éditer un programme et le sauvegarder dans un fichier.
2. **Compilation :** un compilateur est un programme qui convertit le code source écrit dans un langage donné, en un programme écrit dans un langage machine. Un programme compilé est dit module objet.
3. **Édition de liens :** un éditeur de liens est un logiciel qui permet de combiner plusieurs programmes objets en un seul appelé généralement module chargeable.
4. **Chargement :** le rôle du chargeur est de charger le module objet, obtenu après édition de liens, en mémoire centrale afin de l'exécuter.

Exemple : Edition du code source

Considérons le programme suivant en langage C :

```
#include <stdio.h>
main(){
  int i, r=0 ;
  for(i=0  ; i<100 ; i++) r=r+i ;
  printf("r=%d\n",r) ;
}
```

Pour l'exécuter il va d'abord falloir le traduire en langage machine.

En effet la machine ne sait exécuter que des opérations simples.

Exemple : Compilation

Regardons comment ce programme est traduit par le compilateur C. On obtient la traduction assembleur par la commande :

gcc -S -O3 prog.c qui produit prog.s.

On extrait les instructions qui implémentent la boucle "for" :

```
xorl %edx, %edx # initialiser r=0
xorl %eax, %eax # i=0
.p2align 2, , 3
.L2
addl %eax, %edx # r=r+i
incl %eax # i++
cmpl $100, %eax # test fin boucle i<100
jne .L2 # si pas fini goto .L2
```

Cette version en langage assembleur du programme est directement traduisible en binaire. Par exemple :

addl <source>, <destination>

exécute <destination>=<destination> + <source>

En particulier : addl %eax, %edx

sera traduite en 0x01C2. En effet :

```
<---addl---> ex dx (ex=0, dx=2)
000000011100 00 10 code binaire sur 16 bits
on regroupe les bits par 4 pour passer en hexa-décimal
0000 0001 1100 0010
---0 ---1 ---C ---2
```

Vérifions que c'est bien ce que fait le compilateur :

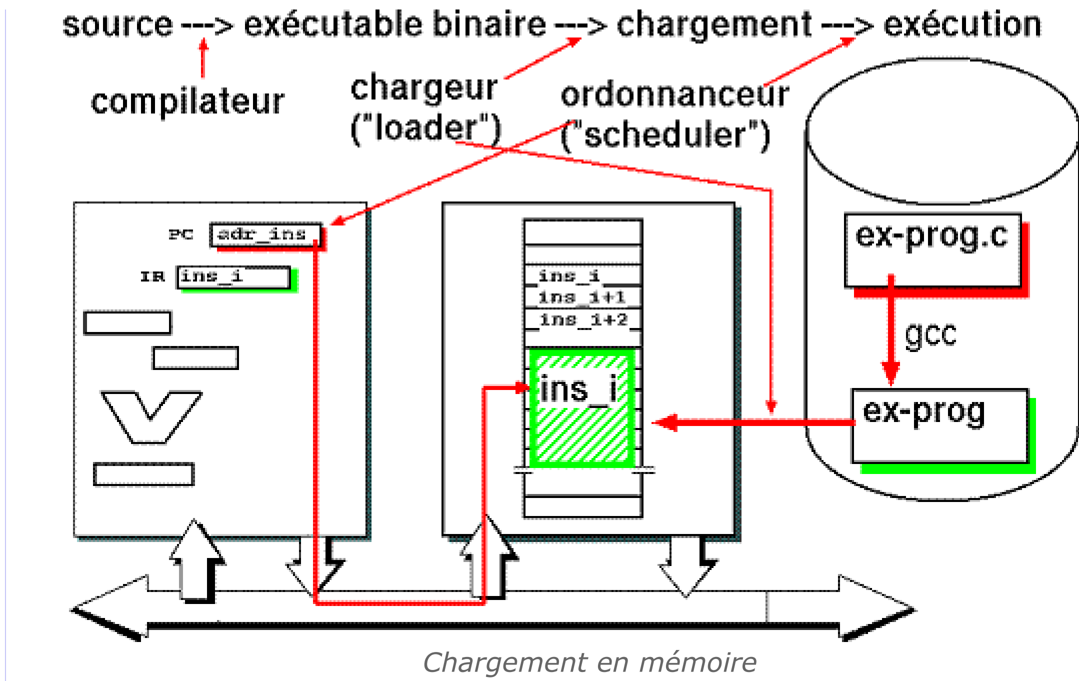
$ gcc -c -O3 prog.c -> prog.o

$ objdump -d prog.o

```
............
c  : 31 d2    xor %edx, %edx
e  : 31 c0    xor %eax, %eax
10 : 01 c2    add %eax, %edx
12 : 40       inc %eax
13 : 83 f8 64 cmp $0x64, %eax
16 : 75 f8    jne 10 <main+0x10>
```

Exemple : Chargement

On a vu que l'UC ne sait exécuter que des instructions binaires stockées dans la mémoire. L'étape suivante va donc consister à charger le programme binaire dans la mémoire. Le système contient donc un programme de chargement (*loader*).



Chargement en mémoire

Exemple : Exécution

Examinons ce qui se passe en mémoire une fois le programme en exécution :

$ gcc -o prog -03 prog.c -> prog

$ gdb prog

```
.........
0x0804838a <main+14>: xor %eax,%eax
0x0804838c <main+16>: add %eax,%edx
0x0804838e <main+18>: inc %eax
0x0804838f <main+19>: cmp $0x64,%eax
0x08048392 <main+22>: jne 0x804838c <main+16>
(gdb) x/4b 0x0804838c
0x804838c <main+16>: 0x01 0xc2 0x40 0x83
```

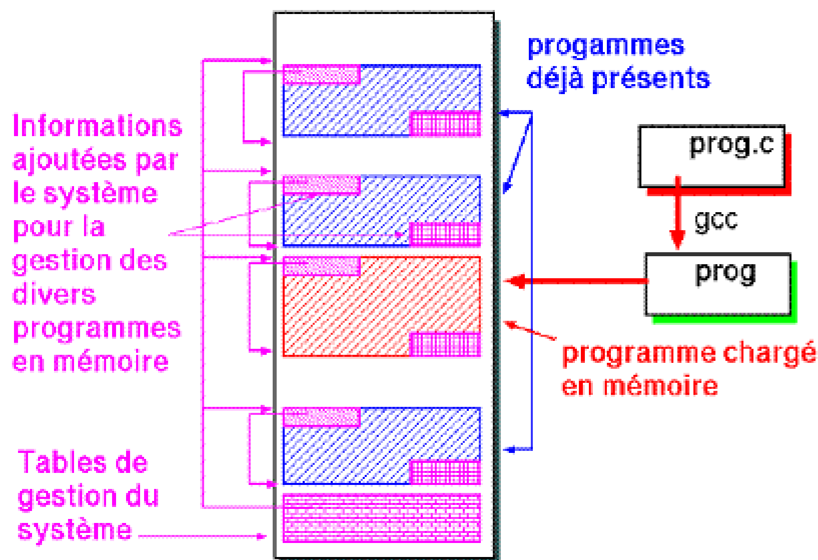
L'UC va chercher en mémoire le contenu de l'adresse 0x0804838c. Elle récupère la suite 0x01, 0xc2 et la décode comme étant l'instruction

add %eax, %edx

qu'elle exécute : ajouter au contenu du registre edx le contenu du registre eax et ranger le résultat dans le registre edx.

Exemple : Processus

Quand on charge un programme en mémoire, il y déjà des programmes et il en viendra d'autres. Le système a donc besoin d'ajouter des informations autour du code binaire constituant le programme stricto-sensu. Au minimum pour savoir où commence et fini chaque programme présent en mémoire, et pour pouvoir facilement passer de l'exécution de l'un d'eux à celle d'un autre.



Cohabitation de plusieurs programmes en mémoire

Le système construit donc en mémoire une structure transitoire dont la durée de vie est le temps d'exécution du programme et qui contient le code binaire du programme. Cette structure est appelée **processus** .

C. Modèle de processus

Définition : Processus

Un processus est un programme qui s'exécute. Un processus est alors une entité dynamique qui naît, qui vit et qui meurt, par opposition à la notion de programme qui est une entité statique qui occupe un espace en mémoire ou sur disque et qui, telle qu'elle n'est pas exécutée, ne produit aucune action *[bouzefrane03]*.

Remarque : Exécution parallèle sur architecture monoprocesseur

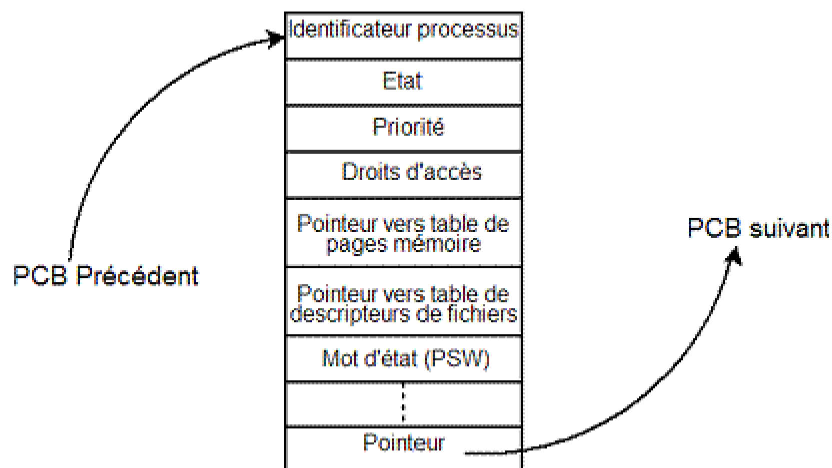
Sur une architecture monoprocesseur, l'exécution parallèle est réalisée par l'exécution de plusieurs processus en alternance sur un seul processeur. Ce dernier commute entre plusieurs processus. La rapidité de commutation donne l'impression à l'utilisateur que les processus s'exécutent simultanément.

Définition : Contexte d'un processus

Le contexte d'un processus est son image en un point interruptible. Le contexte d'un processus est représenté dans le système d'exploitation par une structure regroupant les informations essentielles pour l'exécution du processus comme le contenu des registres, sa pile d'exécution, son masque d'interruption, son état, son compteur ordinal, des pointeurs sur ses ressources (mémoire, fichiers, etc.), etc.

Définition : Process Control Bloc (PCB)

L'ensemble des processus du système est représenté en général par une liste chaînée, dont chaque élément est un bloc de contrôle de processus : PCB (Process Control Bloc). Le PCB (cf. 'Contexte d'un processus / Process Control Bloc (PCB)' p 14) n'est autre que la structure qui héberge le contexte du processus.



Contexte d'un processus / Process Control Bloc (PCB)

Droits d'un processus

Les droits d'un processus peuvent être définis par la description des ressources qui lui sont accessibles et des modes d'accès à ces ressources. Les principaux champs liés aux droits d'un processus sont les suivants *[bouzefrane03]* :

- **le mode d'exécution master / slave (système/utilisateur) :** traduit la possibilité d'exécuter des instructions privilégiées ou pas. Les instructions machine privilégiées accèdent directement à des éléments vitaux de la machine (contrôleurs d'unités de disques, toute la mémoire, table des pages mémoires, etc.). Elles ne sont exécutées que par des processus du système pour le compte du système lui-même ou de processus utilisateur qui ont fait pour cela une demande qui a été contrôlée et validée par un processus du noyau ;
- **le niveau de privilège ou clé d'écriture** donne les droits d'accès à la mémoire ;
- **l'indicateur de masquage des interruptions** traduit le fait qu'un processus peut être interrompu ou non par certaines interruptions ;
- etc.

Image mémoire d'un processus

Le résultat de la compilation se traduit par différentes zones dans le programme binaire. Dans un système unix on distingue :

- data-rw : une zone pour variables déclarées en statique,
- data-ro : une zone pour les données non modifiables (chaines des printf, ...)
- texte : le code binaire des instructions du programme,
- en-tête : des informations sur les bibliothèques utilisées par le programme.

L'ensemble de ces informations permettent au système de construire une structure en mémoire qui sépare bien les différentes fonctions.

La figure suivante illustre l'image mémoire d'un processus unix. Nous remarquons que le PCB d'un processus fait partie de l'espace virtuel système.

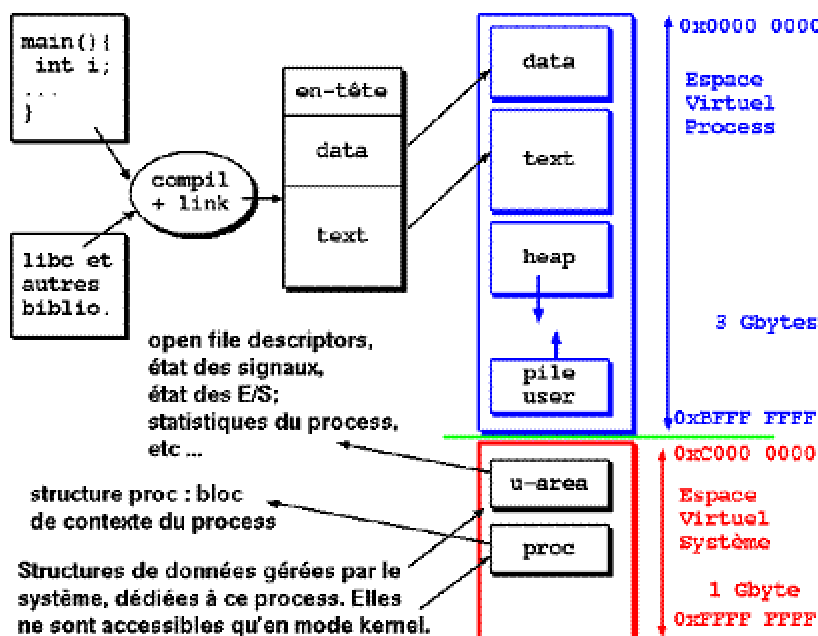
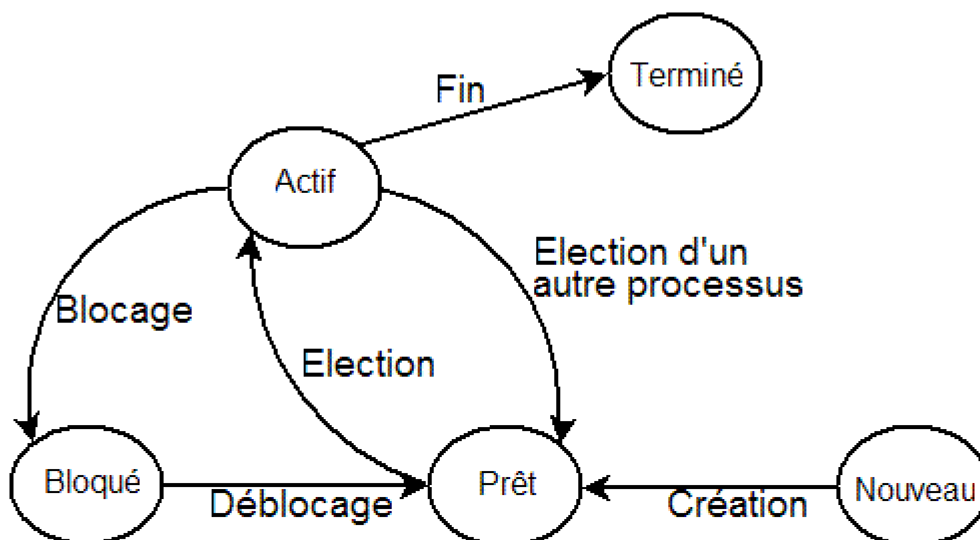


Image mémoire d'un processus UNIX

États d'un processus

Depuis sa création jusqu'à sa destruction (Fin), chaque processus va transiter par un certain nombre d'états (cf. 'États d'un processus' p 15).



États d'un processus

- **Actif :** processus en cours d'exécution;
- **Bloqué :** Processus en attente d'un événement extérieur qui puisse le débloquer. C'est le cas par exemple, quand un processus fait une opération d'entrée/sortie;
- **Prêt :** processus prêt pour exécution.

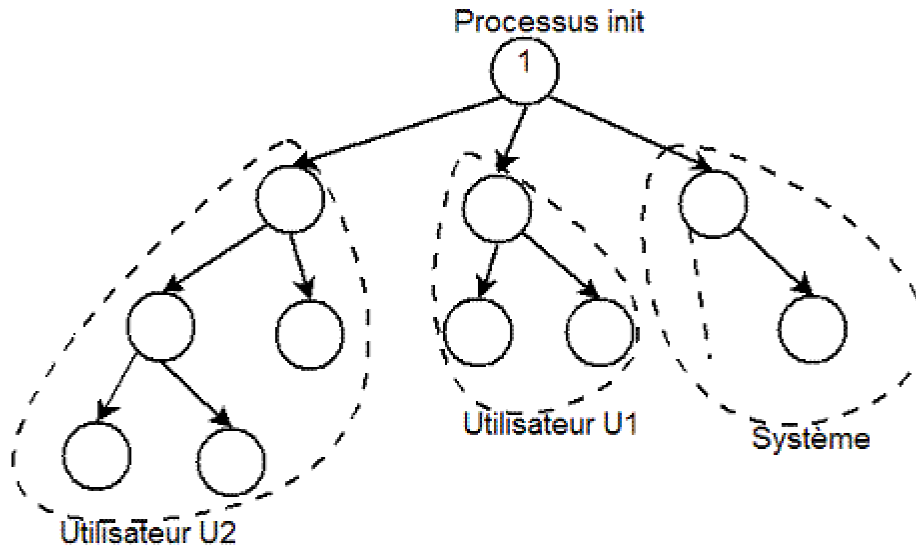
Un processus passe de l'état actif à l'état bloqué quand il ne peut plus continuer son exécution, à moins qu'une condition / événement particulier se réalise (terminaison d'un certain processus, fin d'une opération d'entrée / sortie, etc.). Les transitions actif -> prêt, prêt -> actif sont provoquées par l'ordonnanceur des processus. La transition bloqué -> prêt est provoquée quand la condition / événement attendu par le processus bloqué est réalisé.

D. Processus sous UNIX

Hiérarchie de processus

Sous UNIX, les processus sont organisés sous forme d'une hiérarchie (cf. 'Hiérarchie de Processus' p 16) (chaque processus crée ses processus fils).

La commande UNIX ps permet de lister les processus en cours avec leurs propriétés (identifiant du processus, identifiant du processus père, taille, priorité, propriétaire, état, etc.)



Hiérarchie de Processus

Définition : Identifiant d'un processus

Un identifiant unique est attribué par le système à la création de tout processus. Il s'agit d'un numéro d'ordre et on l'appelle couramment le pid pour *process identifier*. Le tout premier processus créé possède le pid 1 ; c'est le processus init.

Les primitive suivantes permettent de récupérer les pid d'un processus et celui de son père (processus créateur) :

```
#include <unistd.h>
pid_t getpid(void) ; //retourne le pid du processus
appelant
pid_t getppid(void) ; //retourne le pid du père du
processus appelant
```

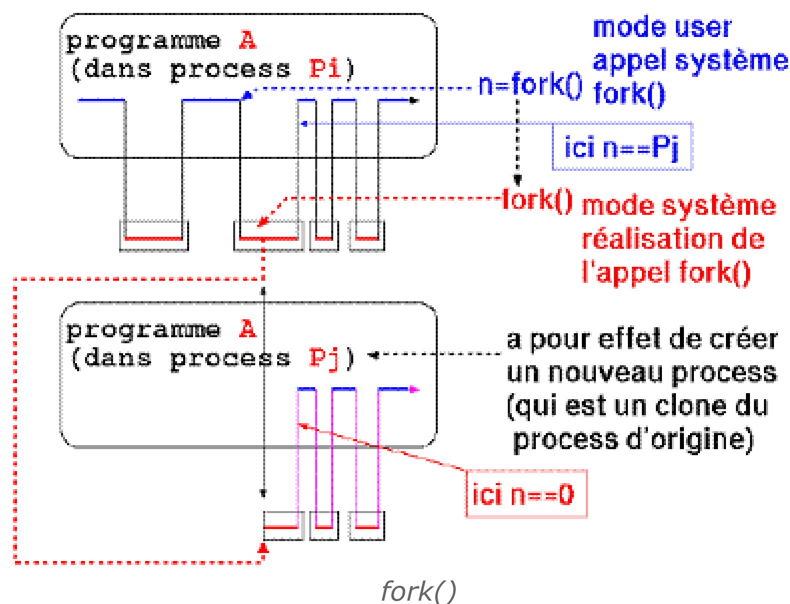
Méthode : Création d'un processus UNIX

Sous UNIX, un processus crée un processus fils en utilisant la primitive suivante :

```
#include <unistd.h>
pid_t fork(void);
```

fork() crée un nouveau processus (le fils) par duplication du processus appelant (le père) : le fils aura le même espace adressable que son père. Afin de distinguer le père du fils, la fonction retourne :

- 0 au processus fils ;
- l'identifiant (ou *pid*) du processus fils créé, au processus père.
- -1, en cas d'erreur.



fork()

Exemple

Le code suivant illustre un usage typique de la primitive fork():

```
#include <unistd.h>
#include <stdio.h>
main(){
  pid_t pid;
  switch(pid=fork()){
    case -1: /* CAS ERREUR DE CREATION */
      perror("creation de processus echouee");
      exit(2);
    case 0: /* CORPS DU PROCESSUS FILS */
      printf("je suis le fils %d, de pere %d\n", getpid(),
getppid());
      printf("fin processus fils\n");
      exit(0);
    default: /* CORPS DU PROCESSUS PERE */
      printf("je suis le processus %d, de pere %d\n,
getpid(), getppid());
      sleep(1);
      printf("fin processus pere\n");
  }
}
```

Un exemple d'exécution de ce programme sera le suivant :

```
je suis le processus 1234 de pere 1235
je suis le fils 1236, de pere 1234
fin processus fils
fin processus pere
```

Remarque : Le code retour d'un processus

Afin de contrôler la bonne exécution d'un processus, on impose à celui-ci de retourner au processus qui l'a lancé, autrement dit son père, un code de retour (entier) lorsqu'il se termine. Par convention, le code est 0 en cas de fin normale. Un code différent de zéro indique une terminaison anormale.

Synchronisation père-fils

Tout processus qui se termine passe dans l'état zombi jusqu'à ce que le processus père prenne connaissance de sa terminaison ; cela pour que le processus père puisse accéder au code retourné par son fils. Un processus zombi affiché avec la commande ps donne Z comme état du processus et 0 pour sa taille. Les primitives wait et waitpid permettent d'éviter les processus zombi.

Méthode : La primitive wait()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int* etat);
```

Si le processus appelant possède au moins un processus fils zombi, la primitive renvoie le pid de l'un d'eux, et la valeur *etat fournit des informations sur la terminaison de ce processus, et notamment le code retour. Si le processus appelant possède des procesus fils mais qu'aucun n'est à l'état zombi, il sera alors bloqué jusqu'à ce que l'un d'eux devienne zombi. Si le processus appelant ne possède aucun processus fils, wait retourne -1.

Méthode : La primitive waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int* etat, int option);
```

Cette primitive permet de tester la terminaison d'un processus particulier (paramètre pid) en bloquant ou non l'appelant (selon paramètre option). Les différentes possibilités du paramètre pid sont :

- <-1 : Tout processus fils dans le groupe |pid|
- -1 : Tout procesus fils
- 0 : Tout processus fils du même groupe que l'appelant
- >0 : Le processus fils d'identité pid

Le paramètre options est une combinaison bit à bit des valeurs suivantes :

- WNOHANG : Le processus appelant n'est pas bloqué si le processus spécifié n'est pas terminé
- WUNTRACED : Si le processus spécifié est arrêté, permet de récupérer la raison de l'arrêt

Remarque : Valeur de retour de wait et waitpid

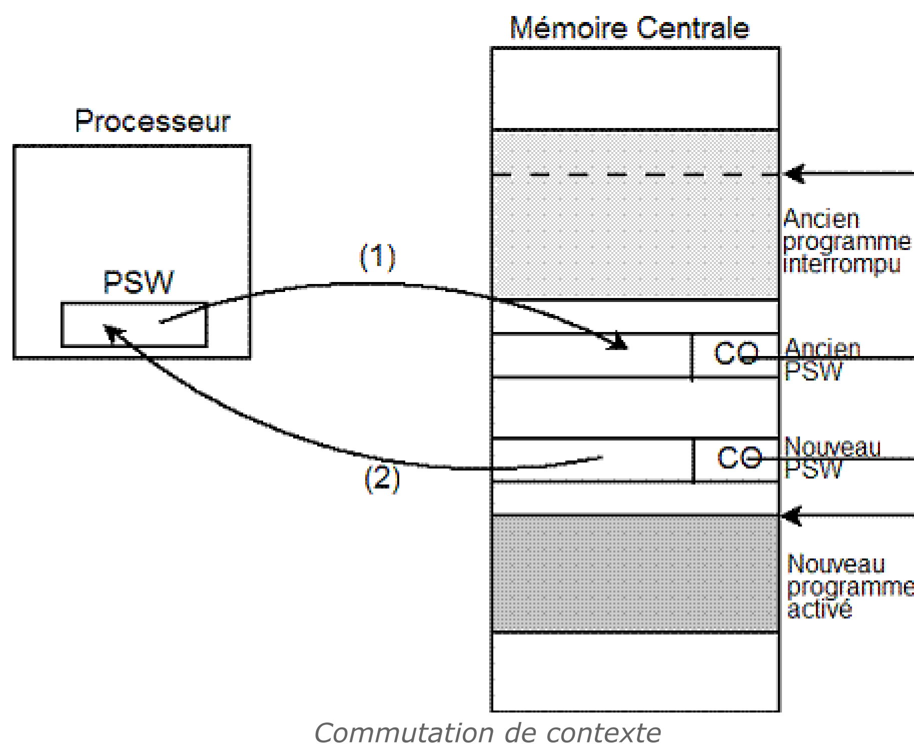
La valeur retournée par wait ou waitpid se trouve dans *etat sur 2 octets, le premier octet contient le code de retour du processus et le second octet contient un ensemble d'indicateurs renseignant sur la terminaison du processus et pouvant être testés par des macros (voir manuel UNIX : commande man).

E. Commutation de contexte

Définition : Commutation de contexte

Le mécanisme de commutation de contexte (cf. 'Commutation de contexte' p 19) permet en une seule opération indivisible de:

- ranger dans un emplacement spécifié de la mémoire, le contenu courant du mot d'état du processeur (PSW) ;
- charger dans le mot d'état un nouveau contenu préparé à l'avance dans un emplacement spécifié de la mémoire.



Commutation de contexte

La commutation du contexte est déclenchée suivant un indicateur (dans le PSW) consulté par le processeur après l'exécution de chaque instruction. Ce mécanisme est également appelé changement d'état.

F. Appels système

Limites du mode utilisateur

Les programmes, au cours de leur exécution ont besoin :

- d'obtenir des informations du système (e.g. heure)
- d'échanger des données ou des signaux avec d'autres programmes,
- d'accéder aux périphériques d'entrées-sorties, ...

Nombre de ces opérations ne peuvent être réalisées qu'en mode système, or, les programmes s'exécutent en mode utilisateur.

Donc il faut une méthode pour qu'un programme "utilisateur" puisse demander au système une action à faire en mode système.

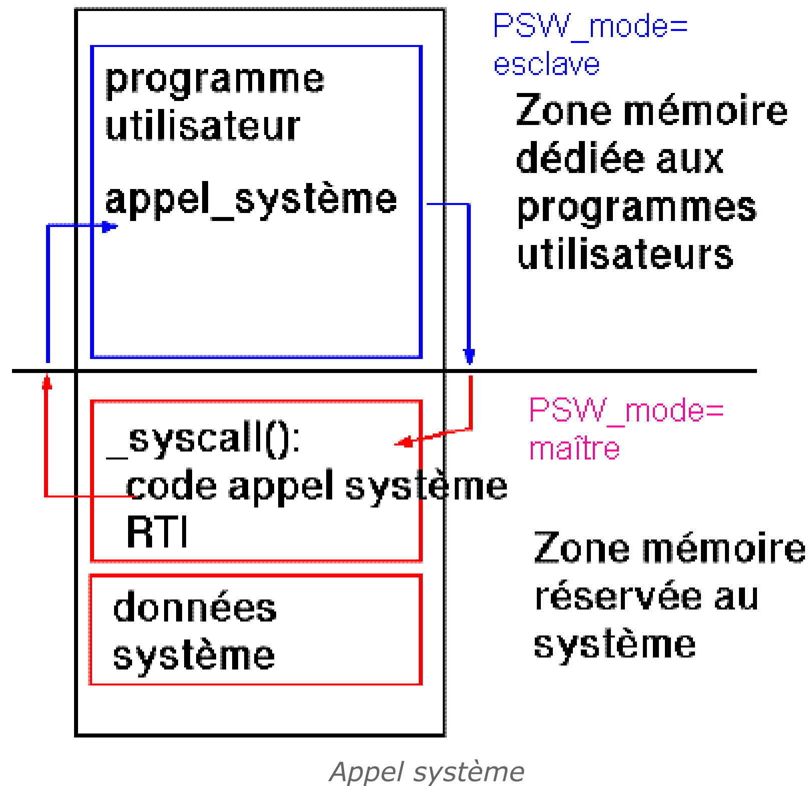
Cette possibilité est dénommée un "appel système" ou "appel au superviseur".

Définition : Appel au superviseur (appel système)

Un appel système (appel au superviseur, ou SVC SuperVisor Call) est une instruction utilisée pour réaliser un appel, à partir d'un processus qui s'exécute en mode esclave (utilisateur), à une fonction qui s'exécute en mode maître (système) (par exemple une fonction d'accès à un fichier : ouvrir, lire, fermer, ...). L'appel au superviseur provoque une commutation de contexte.

Méthode : Appel système

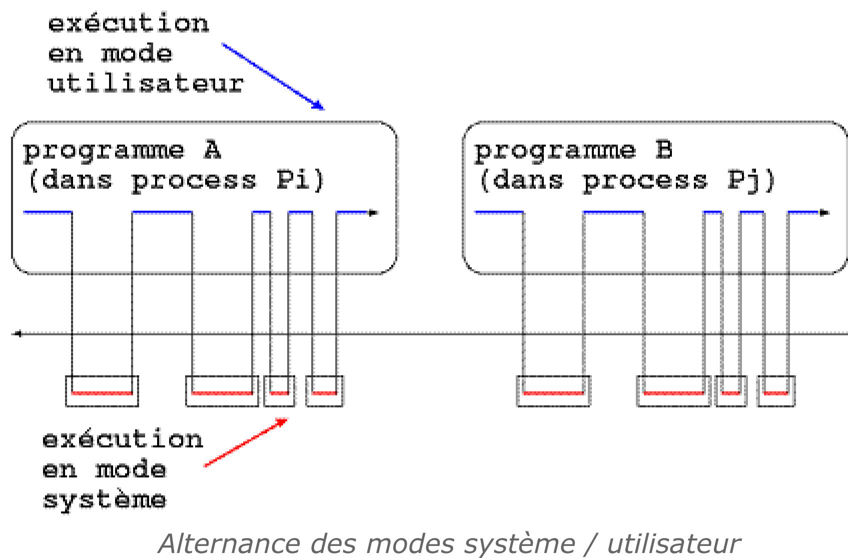
Cette possibilité d'appel système est réalisée par une paire d'instructions :
system_call # appel au système
RTI # retour au programme utilisateur



Appel système

Remarque : Alternance mode système / mode utilisateur

Tout au long de son exécution un processus alterne entre les deux modes : système / utilisateur



Alternance des modes système / utilisateur

Exemple : Mise en oeuvre

La mise en oeuvre des appels systèmes peut différer d'un processeur à un autre.

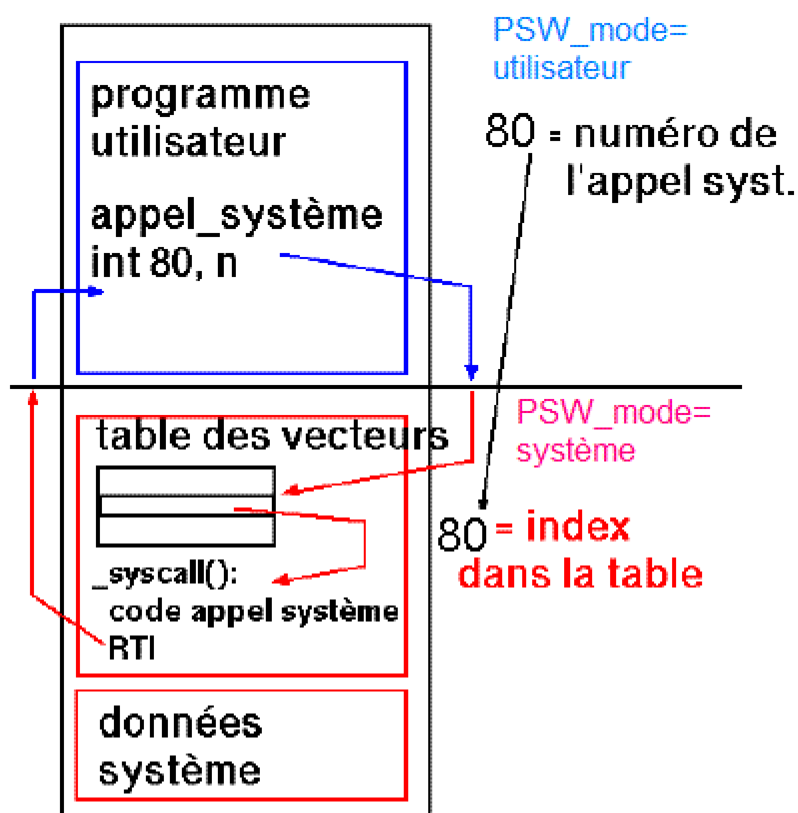
Dans les processeur Intel on définit l'instruction INT qui permet de faire appel au système pour exécuter un service privilégié identifié par un paramètre N :

INT N

Les paramètre du service demandé au système seront passés par les registres du processeur.

Le numéro de l'appel identifie une entrée dans une table des vecteurs des INT, qui contient le PC et PS (PSW) pour la commutation de contexte provoquée par l'appel système.

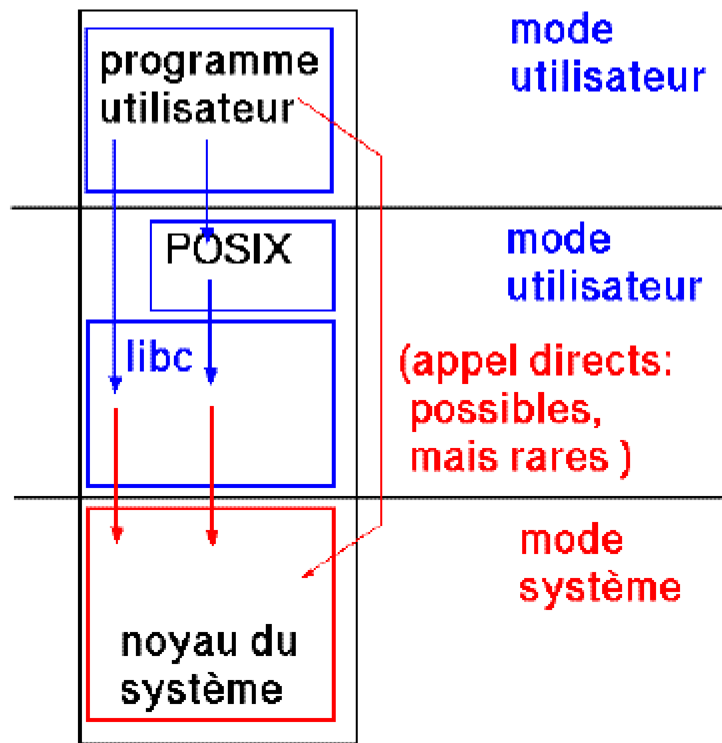
- exécution de l'appel INT i
- sauver pc et ps sur la pile système (kernel stack)
- charger pc, ps avec le contenu de l'entrée i de la table
  - charger nouvelle valeur dans ps : passe en mode système
  - charger nouvelle valeur dans pc : branchement sur la fonction système demandée, dont l'adresse a été préalablement rangée dans l'entrée i de la table des appels systèmes
- exécution de l'appel système
- retour en mode normal en terminant la fonction système par RTI



Exemple de mise en oeuvre d'un SVC

Remarque : Bibliothèque système

En général le système d'exploitation offre au développeur des librairies qui facilitent les appels système. C'est la cas vace la norme POSIX sous UNIX :



Libraire d'appels systèmes

G. Le système d'interruption

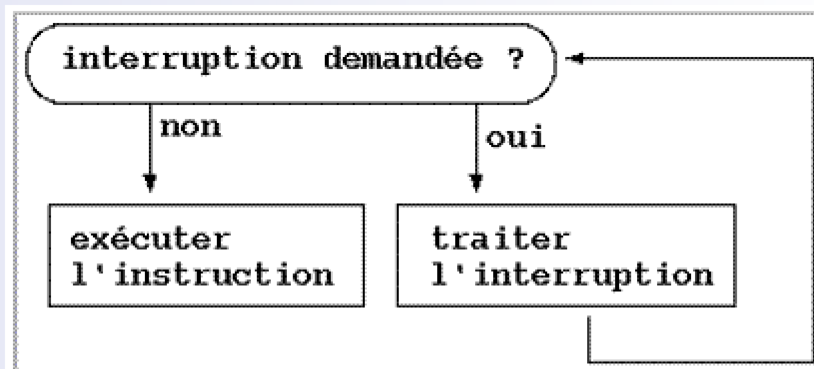
Définition : Interruption

Une interruption a pour fonction de forcer un processeur à réagir à un événement. L'exécution du programme en cours est suspendu et, un programme d'interruption est exécuté. Quand l'indicateur d'interruption du PSW est modifié à partir de l'extérieur du processeur, on dit qu'il y a interruption *[belkhir96]*.

Fondamental : Point observable et demande d'interruption

L'arrivée d'une interruption positionne un bit particulier dans le PSW qui signifie qu'une ineterruption est arrivée.

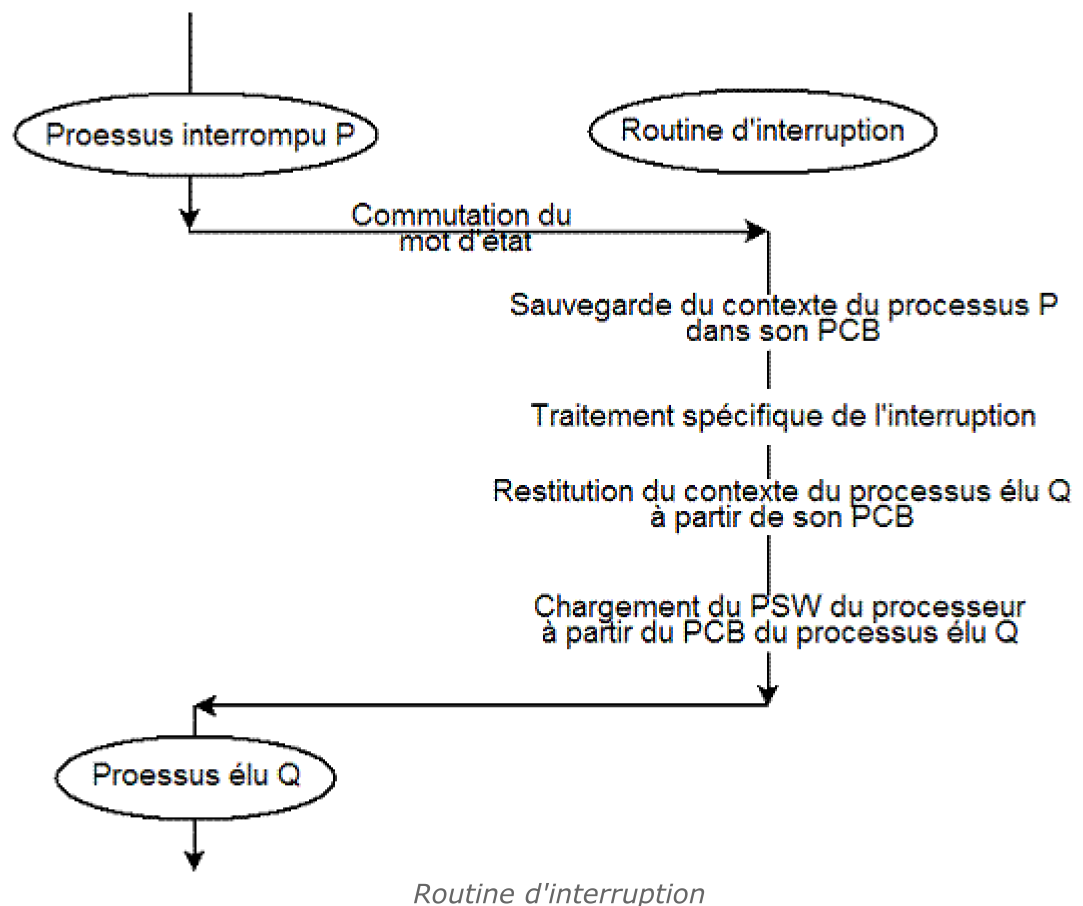
L'UC consulte alors, dans un point observable (entre l'exécution de deux instructions machine), le PSW pour traiter une éventuelle interruption :



Arrivée d'une interruption et point observable

Schéma général d'une routine d'interruption

La figure (cf. 'Routine d'interruption' p 23) illustre les différentes étapes d'une routine d'interruption.

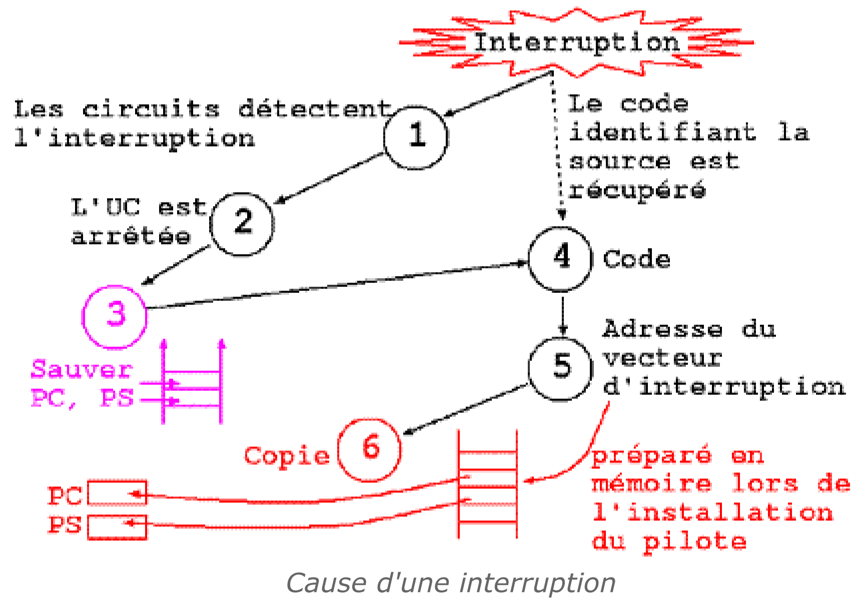


Routine d'interruption

L'interruption force le processeur à interrompre l'exécution du processus en cours P. Son contexte est sauvegardé, et la routine d'interruption est exécutée. Quand l'exécution de de la routine d'interruption est terminée, un processus est élu pour s'exécuter. Pour cela, son contexte est repris et le PSW du processeur est chargé à partir du contexte du processus élu.

Cause d'une interruption

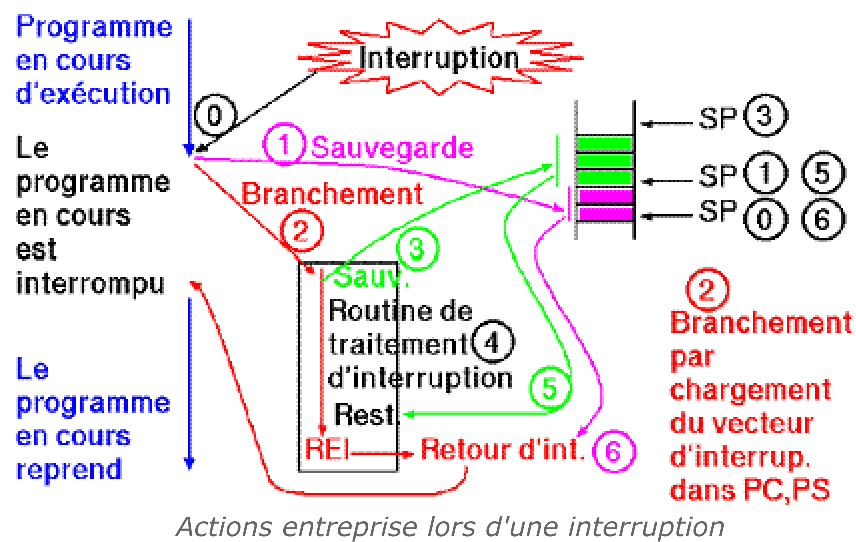
Un code d'interruption contenu dans le PSW (ou un emplacement conventionnel de la mémoire) permet de distinguer la cause de l'interruption. Ce code permettra de déterminer le vecteur d'interruption et de charger le PSW correspondant pour exécuter la routine d'interruption correspondante :



Cause d'une interruption

Remarque : Commutation de contexte et sauvegarde du contexte

Après l'occurrence d'une interruption, le matériel se charge de sauvegarder le PSW (on utilise une pile en général). Le reste du contexte du processus interrompu est généralement sauvegardé par le système :



Actions entreprise lors d'une interruption

Attention : Actions réalisées par le matériel

Dans cette séquence d'opérations, le maximum d'actions sont des séquences précâblées dans le matériel et non pas réalisées par des exécutions. En particulier, la séquence "sauver PSW (PC, PS) sur la pile du noyau", "brancher en chargeant PSW (PC, PS) avec le contenu du vecteur d'interruption est câblée. En effet, cette séquence peut être exécutée des milliers de fois par seconde.

Remarque : Priorité d'interruption

Quand il existe plusieurs niveaux d'interruption, il se peut que deux indicateurs correspondant à deux niveaux différents soient activés en même temps. Le conflit est réglé en établissant un ordre de priorité entre les différents niveaux d'interruption.

Masquage des interruptions

Parfois, il est utile de protéger contre certaines interruptions l'exécution d'une suite d'instructions (par exemple la séquence de sauvegarde du contexte d'un processus interrompu par une interruption). Cela revient à retarder la commutation de contexte déclenchée par l'activation de l'indicateur correspondant à un niveau d'interruption. On dit alors que ce niveau est masqué. Le masquage des interruption figure dans le PSW.

Désarmement des interruptions

On peut supprimer complètement et non seulement retarder une interruption. On dit que le niveau d'interruption est désarmé. Le niveau peut être réarmé (réactivé).

Exemple : Temps partagé

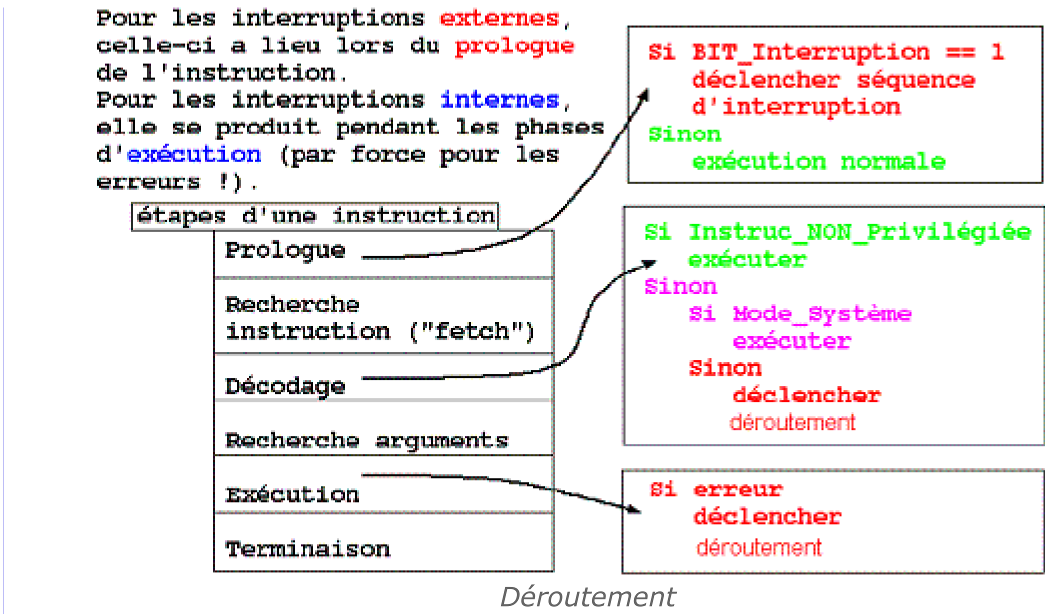
Soit N processus notés 0 à N-1 s'exécutant sur une machine en temps partagé, chacun l'utilisant pendant un quantum de temps égal à Q unités avant d'être interrompu et remplacé par un autre processus. Chaque processus Pi a un PCB[i] dans lequel est rangé son contexte. Le compteur est initialisé à Q : chaque fois que Q unités de temps se sont écoulées, l'interruption d'ordonnancement est générée et provoque l'exécution de la routine d'interruption appropriée qui comporte entre autres les actions suivantes *[bouzefrane03]*:

- sauver le contexte actuel (le PSW entre autres) dans PCB[i] du processus interrompu Pi ;
- mettre PCB[i] dans la file des processus prêts ;
- choisir un nouveau processus Pj à exécuter, par exemple j:=i+1 modulo N ;
- réinitialiser le compteur à Q ;
- charger le contexte de Pj à partir de son PCB (PCB[j]) ;
- charger le PSW du processeur à partir de PCB[j].

Définition : Déroutement

Quand un indicateur de changement d'état est modifié par une cause liée à l'exécution d'une instruction en cours, on dit qu'il y a déroutement (trap). Le mécanisme de déroutement a essentiellement pour rôle de traiter une anomalie dans le déroulement d'une instruction : division par 0, débordement arithmétique, violation de la mémoire, exécution d'une instruction privilégiée en mode esclave, etc.

La figure suivante situe le déroutement dans les différentes étapes d'exécution d'une instruction machine.



Déroutement

H. Les signaux sous UNIX

Définition : Signal UNIX

Dans le système UNIX, un signal est un mécanisme d'interruption logicielle, qui permet de communiquer à un processus l'occurrence d'une interruption matérielle, un déroutement, ou un ensemble d'événements systèmes ou provoqués par d'autres processus.

Un processus qui reçoit un signal est terminé, à moins que ce signal soit intercepté ou masqué. Comme pour les interruptions, on peut mettre en attente certains signaux en les masquant. Un signal pendant est un signal en attente d'être pris en compte. Le bit associé dans le registre des signaux pendant est à 1. Si un autre signal du même type arrive, alors qu'il en existe un pendant, il est perdu. Si le signal n'est pas masqué, alors sa prise en compte a lieu lorsque le processus passe à l'état **actif utilisateur** : un processus en mode noyau (maître) ne peut être interrompu par un signal. La prise en compte d'un signal entraîne l'exécution d'une fonction spécifique appelée *handler*. Celle-ci peut être la routine prédéfinie dans le système (traitement par défaut) ou une routine mise par l'utilisateur pour personnaliser le traitement de ce signal. Dans le PCB du processus, il existe une structure pour assurer la gestion des signaux qui regroupe la mémorisation des signaux en attente, le masquage et les pointeurs vers les handlers.

Types de signaux

Il existe un nombre NSIG de signaux différents, chacun étant identifié par un numéro de 1 à NSIG. NSIG étant une constante définie dans le fichier <signal.h>. La commande "kill -l" donne la liste des signaux définis dans le système. Le tableau (cf. 'Quelque signaux UNIX' p 27) illustre certains signaux principaux.

Nom du signal	Événement associé	Traitement par défaut
SIGALRM	Fin d'une temporisation (fonction alarm)	Terminaison du processus
SIGCHLD	Terminaison d'un fils	Signal ignoré
SIGFPE	Erreur arithmétique (division par zéro, ...)	Terminaison du processus
SIGINT	Frappe de intr sur le terminal de contrôle	Terminaison du processus
SIGKILL	Signal de terminaison	Terminaison du processus
SIGQUIT	Frappe de quit sur le terminal de contrôle	Terminaison du processus avec fichier core
SIGSEGV	Violation mémoire	Terminaison du processus avec fichier core

Tableau 1 : Quelque signaux UNIX

Envoi d'un signal : la primitive kill

Des processus ayant le même propriétaire peuvent communiquer par le biais des signaux. La primitive kill permet d'envoyer un signal à un processus ou un groupe de processus.

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

sig désigne le signal émis par son nom ou son numéro (entier entre 1 et NSIG), et pid désigne le ou les processus qui recevront le signal:

- >0 : Le processus d'identité pid
- 0 : Tous les processus dans le même groupe que le processus émetteur
- <-1 : Tous les processus du groupe de numéro |pid|

La primitive kill renvoie 0 si le signal a pu être envoyé et -1 en cas d'echec.

Attente d'un signal : la primitive pause

En exécutant la primitive pause, un processus peut suspendre son exécution pour se mettre en attente de l'arrivée d'un signal quelconque.

```
#include <unistd.h>
int pause(void);
```

Installation de nouveaux handlers

A chaque type de signal est associé un handler par défaut appelé SIG_DFL. Un processus peut ignorer un signal en lui associant le handler SIG_IGN.

Autrement, la structure sigaction est utilisée par la primitive sigaction qui permet d'installer un handler pour personnaliser le traitement d'un signal.

```
struct sigaction{
  void (*sa_handler)(); /* pointeur sur handler ou SIG_DFL
ou SIG_IGN */
  sigset_t sa_mask; /* liste signaux supplémentaites à
bloquer éventuellement*/
  int sa_flags; /* indicateurs optionnels */
}
```

Le champ sa_handler est le seul obligatoire ; c'est un pointeur sur la fonction qui servira de handler. Les indicateurs optionnels sont peu utilisés et définissent des comportements spécifiques du handler à la reception de certains signaux.

La primitive sigaction permet d'installer un nouveau handler pour le traitement d'un signal.

```
#include <signal.h>
int sigaction(int sig, struct sigaction *new_handler,
struct sigaction *old_handler);
```

- sig désigne le signal pour lequel on veut installer un nouveau handler et
- new_handler pointe sur la structure sigaction à utiliser. La prise en compte du signal entraînera l'exécution de la fonction new_handler->sa_handler. Si de plus, la fonction n'est ni SIG_DFL, ni SIG_IGN, ce signal ainsi que ceux contenus dans la liste new_handler->sa_mask}, seront masqués pendant le traitement.
- old_handler pointe sur une structure sigaction qui contient les anciennes options du traitement du signal.