

CHAPITRE II : MECANISMES DE BASE D'EXECUTION DES PROGRAMMES

1 Machine de VON-NEUMANN

1.1 Définition

Une machine de **Von-Neumann** est un ordinateur électronique à **base de mémoire** dont les composants sont :

- Mémoire Centrale (MC).
- Processeur ou Unité Centrale (UC) ; pour effectuer les calculs et exécuter les instructions.
- Unités périphériques ou d'Entrée/Sortie (E/S).

1.2 Unité Centrale de Traitement (CPU, Central Process Unit)

Appelée aussi "**Processeur Central**" (PC), elle est composée de :

- L'Unité de Commande (UC).
- L'Unité Arithmétique et Logique (UAL).
- Les Registres (Données, Adresses, Contrôle, Instruction).
- Bus Interne.

Le processeur exécute des instructions ou actions d'un programme. C'est le cerveau de l'ordinateur. Une **action** fait passer, en un temps fini, le processeur et son environnement d'un **état initial** à un **état final**. Les actions sont constituées de suites d'instructions **élémentaires**.

1. Registres du processeur central

Les registres sont une sorte de mémoire interne à la CPU, à accès très rapide qui permettent de stocker des résultats temporaires ou des informations de contrôle. Le nombre et le type des registres implantés dans une unité centrale font partie de son architecture et ont une influence importante sur la programmation et les performances de la machine. Le processeur central dispose d'un certain nombre de registres physiques :

- **Le Compteur Ordinal (CO, Program Counter (PC), Instruction Pointer (IP))** : Il contient l'adresse de la prochaine instruction à exécuter.
- **Le Registre d'Instruction (RI, Instruction Register (IR))** : Il contient l'instruction en cours d'exécution.
- **Les Registres Généraux (General-purpose Registers)**: Ils permettent de faire le calcul, la sauvegarde temporaire de résultats, ...etc. Il s'agit de :
 - **Registres Accumulateurs (Accumulator Registers)** qui servent aux opérations arithmétiques.
 - **Registres d'Index (Index Registers)** qui sont utilisés pour l'**adressage indexé**. Dans ce cas, l'adresse effective d'un opérande est obtenue en ajoutant le contenu du registre d'index à l'adresse contenue dans l'instruction. Ce type d'adressage et de registre est très utile pour la manipulation des tableaux.

- **Registre de Base (Base Register)** qui permet le calcul des adresses effectives. Un registre de base contient une adresse de référence, par exemple l'adresse physique correspondant à l'adresse virtuelle 0. L'adresse physique est obtenue en ajoutant au champ adresse de l'instruction le contenu du registre de base.
- **Registres Banalisés** qui sont des registres généraux pouvant servir à diverses opérations telles que stockage des résultats intermédiaires, sauvegarde des informations fréquemment utilisées, ...etc. Ils permettent de limiter les accès à la mémoire, ce qui accélère l'exécution d'un programme.
- **Le Registre Pile (Stack Pointer, SP)** : Il pointe vers la tête de la pile du processeur. Cette pile est une pile particulière (appelée **pile système**) est réservée à l'usage de l'unité centrale, en particulier pour sauvegarder les registres et l'adresse de retour en cas d'interruption ou lors de l'appel d'une procédure. Le pointeur de pile est accessible au programmeur, ce qui est souvent source d'erreur.
- **Le Registre Mot d'Etat (PSW, Program Status Word)** : Il contient plusieurs types d'informations ; à savoir :
 1. Les valeurs courantes des **codes conditions (Flags)** qui sont des bits utilisés dans les opérations arithmétiques et comparaisons des opérandes.
 2. Le **mode d'exécution**. Pour des raisons de protection, l'exécution des instructions et l'accès aux ressources se fait suivant des modes d'exécution. Ceci est nécessaire pour pouvoir réserver aux seuls programmes du S.E. l'exécution de certaines instructions. Deux modes d'exécution existent généralement :
 - Mode **privilégié** ou **maître** (ou **superviseur**). Il permet :
 - L'exécution de tout type d'instruction. Les instructions réservées au mode maître sont dites **privilégiées** (Ex. instructions d'E/S, protection, ...etc.).
 - L'accès illimité aux données.
 - Mode **non privilégié** ou **esclave** (ou **usager**). Il permet :
 - Exécution d'un répertoire limité des instructions. C'est un sous ensemble du répertoire correspondant au mode maître.
 - Accès limité aux données d'utilisateur.
 3. masque d'interruptions (seront détaillés dans la suite).

2. Cycle d'exécution du processeur

Un programme séquentiel est composé d'une suite d'instructions. L'exécution du programme fait évoluer l'état de la machine d'un état à un autre. Cette évolution est **discrète** : l'état n'est observable qu'en des instants particuliers appelés "**points observables**", qui correspondent en général aux débuts et fins d'instructions.

Le processeur central exécute continuellement le cycle suivant :

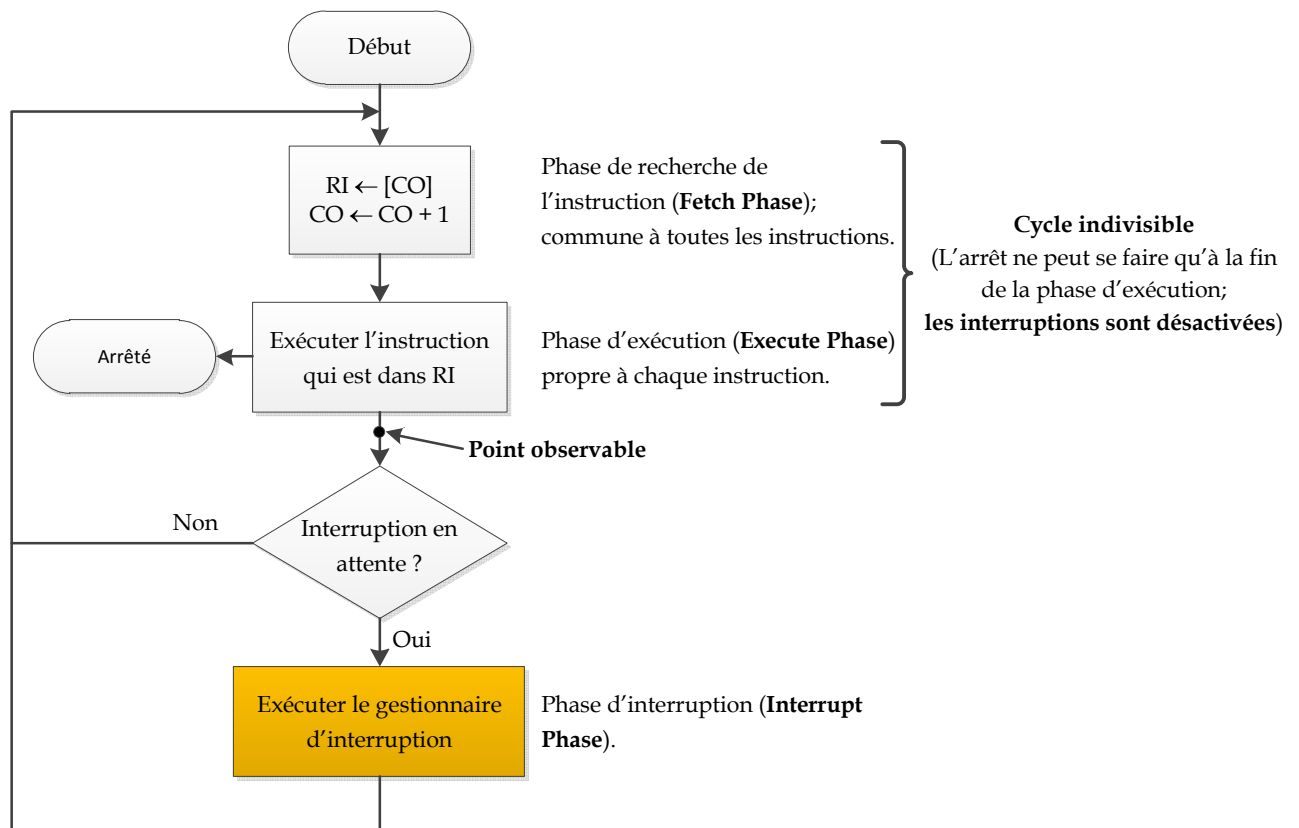


Figure 2.1 : Cycle d'exécution du processeur

Remarque

Le cycle de processeur est **indivisible** et ne peut être interrompu qu'à la fin de la phase **Execute** (i.e. on exécute au moins une instruction élémentaire d'un programme pour pouvoir interrompre).

3. Etat du processeur (Processor State Information)

Il est décrit par le **contenu** des **registres** du processeur. Le contenu des registres concerne le processus en cours d'exécution. Quand un processus est interrompu, l'information contenue dans les registres du processeur doit être sauvegardée afin qu'elle puisse être restaurée quand le processus interrompu reprend son exécution.

Remarque

L'état d'un processeur n'est observable qu'entre deux cycles du processeur.

2 Cheminement d'un Programme dans un Système

Le passage d'un programme de la forme externe à la forme interne se fait en plusieurs étapes. 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 (Voir Figure 2.2) :

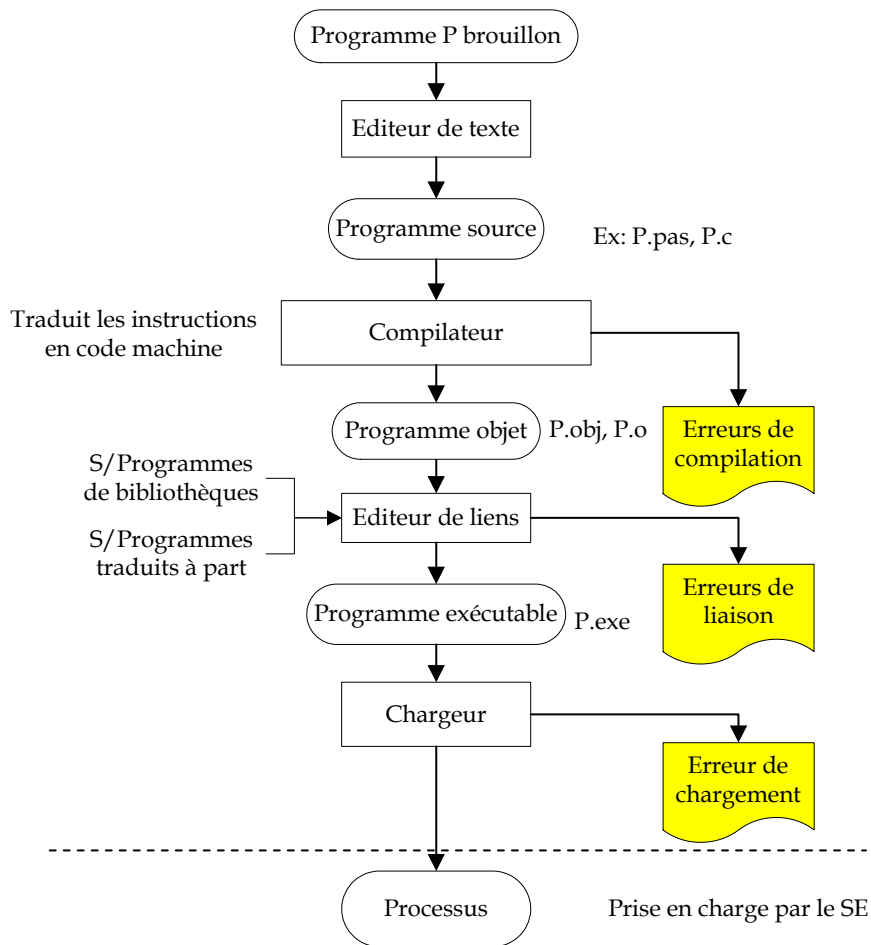


Figure 2.2 : Cheminement d'un programme dans un système

2.1 Editeur de texte (Text Editor)

C'est un logiciel interactif qui permet de saisir du texte à partir d'un clavier, et de l'enregistrer dans un fichier.

Exemple : Bloc-Notes, vi, vim, emacs, ...etc.

2.2 Compilateur (Compiler)

Un compilateur est un programme qui traduit des programmes écrits dans des langages évolués (Pascal, C, Ada, Java, ...etc.) en programmes binaires ou en langage machine, appelés aussi **objets**.

Le **langage machine** est un ensemble de codes binaires directement décodables et exécutables par la machine.

2.3 Editeur de liens (Linker)

Un programme source peut être constitué :

1. des instructions,
2. des données localement définies,
3. des données externes, et

4. des procédures ou sous-programmes externes (bibliothèques de fonctions).

Avant d'exécuter le programme objet, il est nécessaire de rassembler les parties non locales et les procédures externes avec le programme.

L'éditeur de liens est donc un logiciel permettant de combiner plusieurs programmes objets en un seul programme.

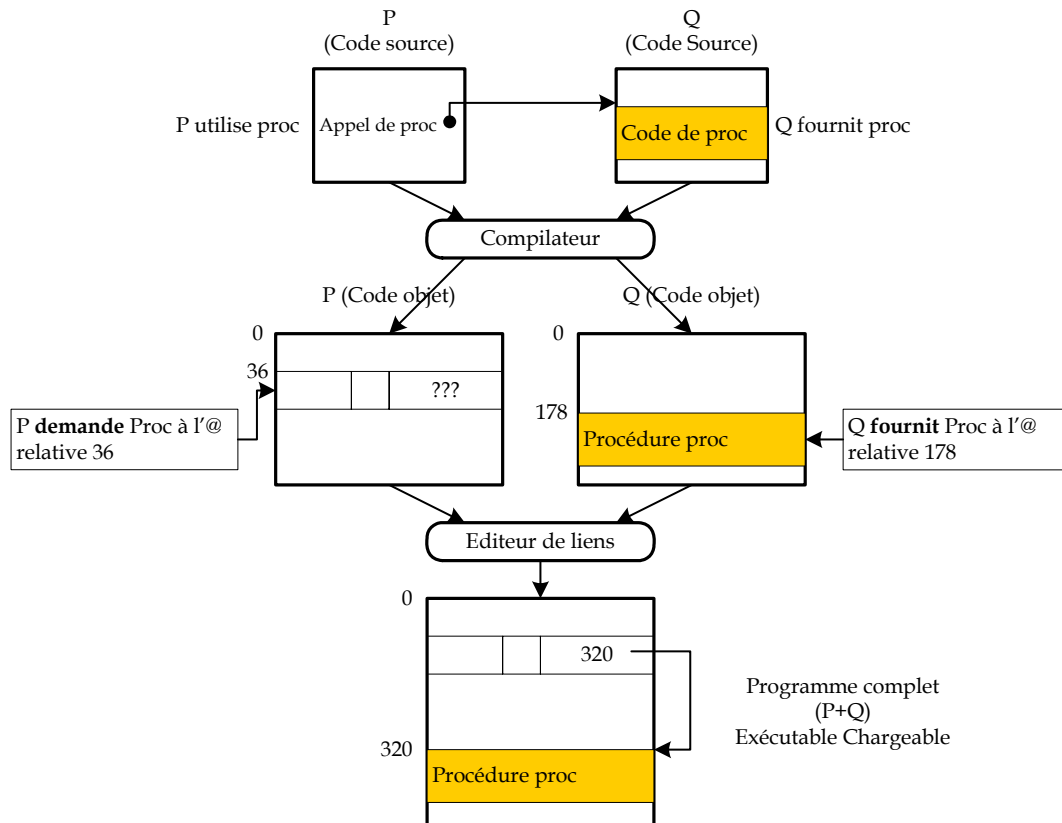


Figure 2.3 : Principe d'édition de liens

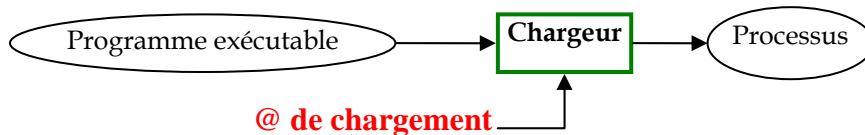
Pour pouvoir développer de gros programmes, on structure ceux-ci en modules que l'on traduit indépendamment. Ainsi, un programme peut être constitué de plusieurs fichiers dont l'un est le programme principal. Ce dernier fait appel aux sous-programmes, ce qui donne lieu à des références extérieures.

Lors de la compilation, le compilateur ne peut pas remplacer les références extérieures par les adresses correspondantes puisqu'il ne les connaît pas. Le programme objet généré contient donc le code machine des instructions et la liste des références extérieures, ainsi que la liste des adresses potentiellement référencables depuis l'extérieur du module. Après la compilation, chaque sous-programme commence à l'adresse **logique 0**.

L'éditeur de liens ordonne les sous-programmes et le programme appelant, modifie les adresses de chacun d'eux et produit un programme final dont l'origine est à l'adresse 0.

2.4 Le chargeur (Loader)

Après l'édition de liens, le programme exécutable doit se trouver en MC pour être exécuté. Le chargeur est un programme qui installe ou charge un exécutable en MC à partir d'une adresse déterminée par le S.E.



3 Concepts de Processus et de Multiprogrammation

3.1 Définition

Un **processus (Process)** est un **programme en cours d'exécution**. Tout processus possède des **caractéristiques** propres (Ex. un numéro d'identification), des **ressources** qu'il utilise (comme des fichiers ouverts) et se trouve à tout moment dans un **état** (en exécution ou en attente ...).

Un processus est constitué d' :

- Un **code exécutable** du programme en exécution.
- Un **contexte** qui est une image décrivant l'environnement du processus.

3.2 Contexte d'un processus

Le contexte d'un processus est l'ensemble des données qui permettent de reprendre l'exécution d'un processus qui a été interrompu. Il est formé des contenus de :

- Compteur Ordinal (CO)
- Mot d'état PSW
- Registres généraux
- Pile

Le CO et le PSW représentent le **petit contexte**, et les registres généraux et la pile représentent le **grand contexte**.

3.3 Image mémoire d'un processus

L'espace mémoire alloué à un processus, dit **image mémoire (Memory Map)** du processus, est divisé en un ensemble de parties :

1. **Code (Text)** ; qui correspond au code des instructions du programme à exécuter. L'accès à cette zone se fait en **lecture seulement (Read-Only)**.
2. **Données (Data)** ; qui contient l'ensemble des constantes et variables déclarées.
3. **Pile (Stack)** ; qui permet de stocker
 - Les valeurs des registres,
 - Variables locales et paramètres de fonctions,
 - Adresse de retour de fonctions.
4. **Tas (Heap)** ; une zone à partir de laquelle l'espace peut être alloué dynamiquement **en cours d'exécution (Runtime)**, en utilisant par exemple les fonctions **new** et **malloc**.

3.4 Descripteur de Processus (PCB)

Chaque processus est représenté dans le SE par un **bloc de contrôle de processus (Process Control Bloc, PCB)**. Le contenu du PCB (Figure 2.4) varie d'un système à un autre suivant sa complexité. Il peut contenir :

1. **Identité du processus** ; chaque processus possède deux noms pour son identification :
 - Un **nom externe** sous forme de chaîne de caractères fourni par l'utilisateur (c'est le nom du fichier exécutable).
 - Un **nom interne** sous forme d'un entier fourni par le système. Toute référence au processus à l'intérieur du système se fait par le nom interne pour des raisons de facilité de manipulation.
2. **Etat du processus** ; l'état peut être nouveau, prêt, en exécution, en attente, arrêté, ...etc.
3. **Contexte du processus** ; compteur ordinal, mot d'état, registres ;
4. **Informations sur le scheduling de l'UC** ; ces informations comprennent la priorité du processus, des pointeurs sur les files d'attente de scheduling, ...etc.
5. **Informations sur la gestion mémoire** ; ces informations peuvent inclure les valeurs des registres base et limite, les tables de pages ou les tables de segments.
6. **Information sur l'état des E/S** ; l'information englobe la liste de périphériques d'E/S alloués à ce processus, une liste de fichiers ouverts, ...etc.
7. **Informations de Comptabilisation** ; ces informations concernent l'utilisation des ressources par le processus pour facturation du travail effectué par la machine (chaque chose a un coût).

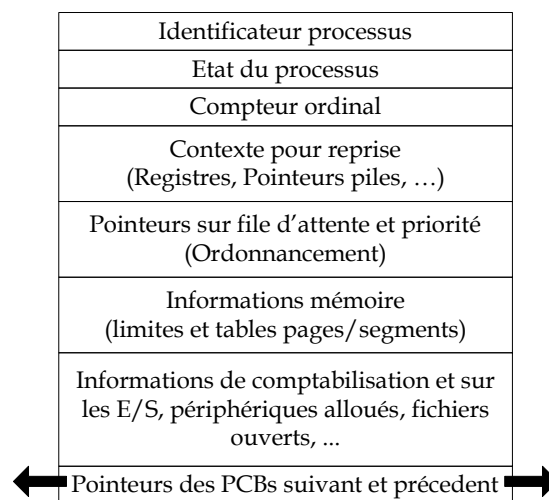


Figure 2.4 : Bloc de contrôle de processus (PCB)

Le système d'exploitation maintient dans une table appelée «**table des processus**» les informations sur tous les processus créés (une entrée par processus : Bloc de Contrôle de Processus PCB). Cette table permet au SE de localiser et gérer tous les processus.

Donc, un processus en cours d'exécution peut être schématisé comme suit (Figure 2.5) :

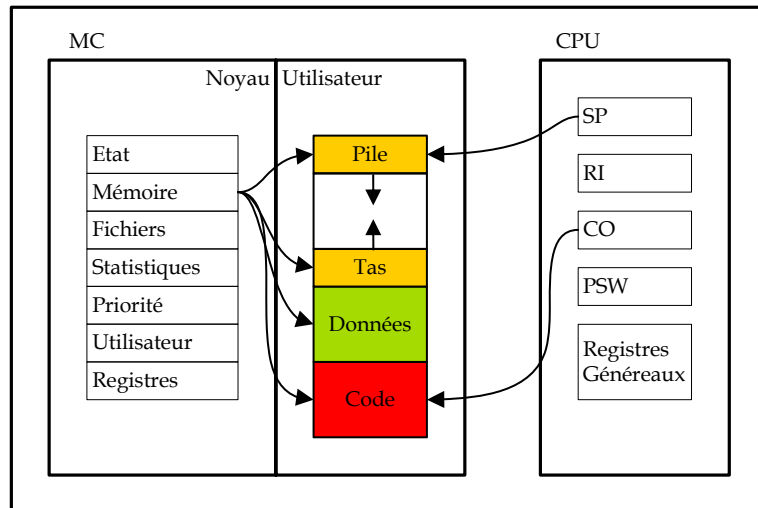


Figure 2.5 : Schéma général d'un processus en cours d'exécution

3.5 Etat d'un processus

Un processus prend un certain nombre d'états durant son exécution, déterminant sa situation dans le système vis-à-vis de ses ressources. Les trois principaux états d'un processus sont (Voir Figure 2.6) :

- **Prêt (Ready)** : le processus attend la libération du processeur pour s'exécuter.
- **Actif (Running)** : le processus est en exécution.
- **Bloqué (Waiting)** : le processus attend une ressource physique ou logique autre que le processeur pour s'exécuter (mémoire, fin d'E/S, ...etc.).

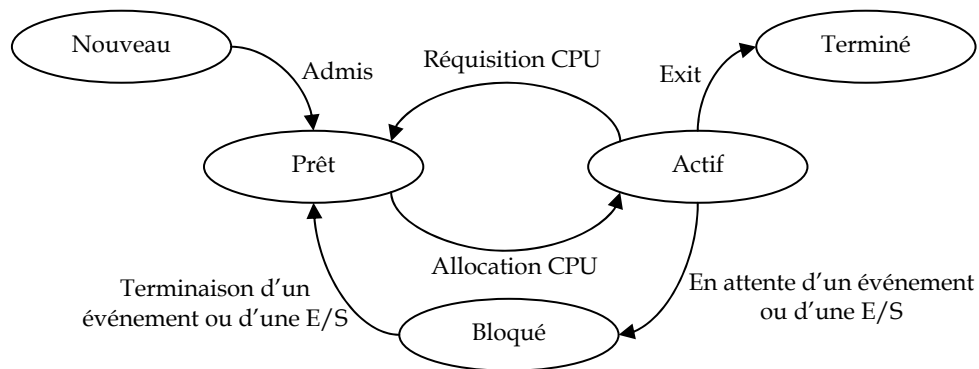


Figure 2.6 : Diagramme de transition des états d'un processus

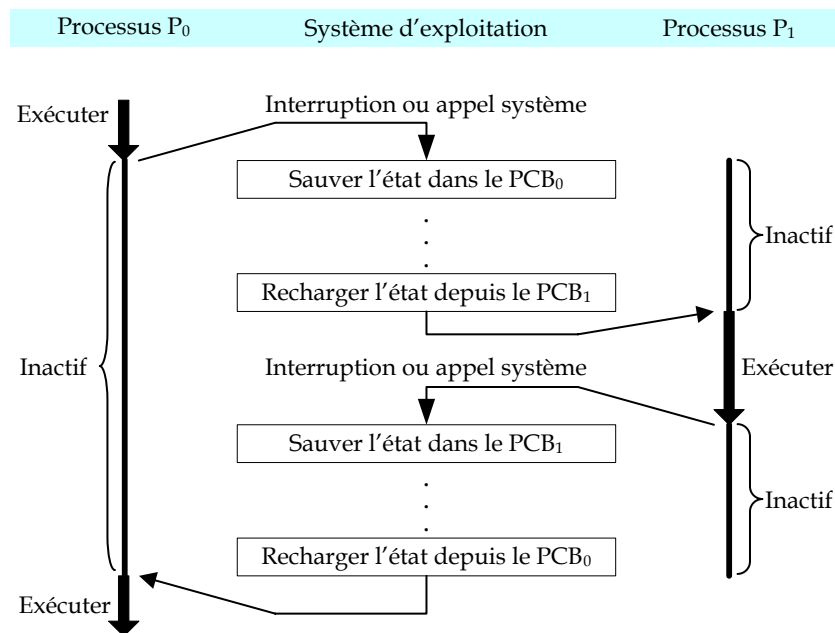
3.6 Multiprogrammation

Dans un système multiprogrammé, le processeur assure l'exécution de plusieurs processus en parallèle (pseudo-parallélisme).

Le passage dans l'exécution d'un processus à un autre nécessite une opération de sauvegarde du contexte du processus arrêté, et le chargement de celui du nouveau processus. Ceci s'appelle la **commutation du contexte (Context Switch)**.

A. Mécanisme de commutation de contexte

La commutation de contexte consiste à changer les contenus des registres du processeur central par les informations de contexte du nouveau processus à exécuter.



La commutation du contexte se fait en deux phases (Voir Figure 2.8) :

- La **première phase** consiste à commuter le petit contexte (CO, PSW) par une instruction **indivisible**.
- La **deuxième phase** consiste quant à elle à commuter le grand contexte par celui du nouveau processus.



Figure 2.8 : Les phases d'une commutation de contexte

Remarque

La commutation du contexte est déclenchée suivant l'état d'un indicateur qui est consulté par le processeur à chaque point observable.

3.7 Processus sous UNIX

A. Identification d'un processus

Un processus est identifié de manière unique par un numéro (**pid**: Process IDentifier).

- La commande **ps** (voir **man ps**) donne la liste des processus en cours d'exécution avec leurs propriétés (identifiant du processus, identifiant du processus père, taille, priorité, propriétaire, état, etc.)
- La fonction **getpid()** indique le numéro du processus qui l'exécute

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

```
pid_t getpid(void);
```

B. Création d'un processus

Un processus est créé par une instruction spéciale **fork**. Le processus créé (le **fils**) est un **clone (copie conforme)** du processus créateur (le **père**).

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

Le père (**Parent**) et le fils (**Child**) ne se distinguent que par le résultat retourné par **fork**. Pour le père, cette fonction renvoie le **numéro du fils** (ou **-1** si création impossible) et pour le fils, elle renvoie **0**.

```
pid_t pid=fork();
switch(pid){
    case -1: /* Erreur de création */
        ...
    case 0: /* Programme du fils */
        ...
    default: /* Programme du père */
        ...
}
```

C. Hiérarchie des processus

Sous UNIX, les processus forment une hiérarchie (père-fils) (Figure 2.9).

La commande UNIX **ps -aeH** permet d'afficher la hiérarchie complète des processus, y compris le processus **init**.

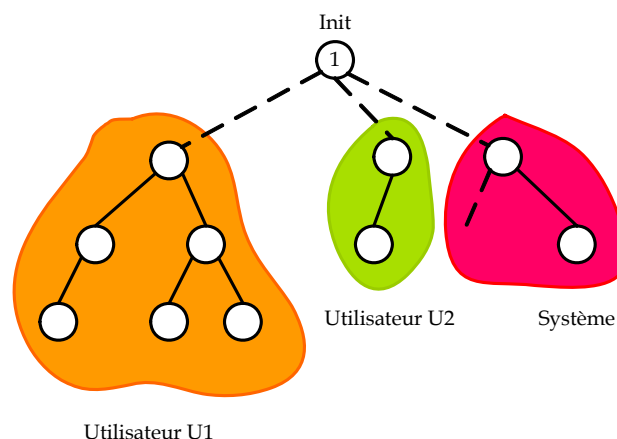


Figure 2.9 : Hiérarchie des processus

D. Terminaison d'un processus

- Dans des conditions normales, un processus peut se terminer de deux façons:
 - Soit la fonction **main** du programme se termine,
 - Soit le programme appelle la fonction **exit**.

```
#include <stdlib.h>
void exit (int status);
```

- Comme il peut être explicitement éliminé par un autre processus, avec l'envoi d'un signal du type **kill**. Ceci peut se faire de deux façons :
- Soit à partir d'un programme, en utilisant la fonction **kill()**.

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
```

kill() envoie le signal numéro **sig** à un processus identifié par son **pid**. En cas d'erreur elle retourne **-1**, et **0** autrement.

```
pid_t pid=fork();
switch(pid){
    case -1: /* Erreur de création */
        ...
    case 0: /* Programme du fils */
        ...
    default: /* Programme du père */
        ...
        /*Envoi d'un signal de terminaison au fils*/
        kill(pid, SIGTERM);
}
```

- Soit à partir du shell, en invoquant la commande kill.

```
$ kill pid
```

La commande **kill** envoie par défaut un signal **SIGTERM**, ou de terminaison au processus d'identité **pid**.

E. Code de sortie d'un processus

Chaque processus a un code de sortie (**Exit Code**) : un nombre que le processus renvoie à son parent. Le code de sortie est l'argument passé à la fonction **exit** ou la valeur retournée depuis **main**.

Par convention, le code de sortie est utilisé pour indiquer si le programme s'est exécuté correctement :

- Un code de sortie à **zéro** indique une **exécution correcte**,
- Un code **différent de zéro** indique qu'une **erreur** est survenue.

```
pid_t pid=fork();
switch(pid){
    case -1: /* Erreur de création */
        ...
        exit(1);
    case 0: /* Programme du fils */
```

```

    ...
    exit(0);
    default: /* Programme du père */
    ...
}

```

F. Synchronisation père-fils

Lorsqu'un processus fils se termine, il devient un processus **zombie**. Un processus zombie ne peut plus s'exécuter, mais consomme encore des ressources (son entrée dans la table des processus n'est pas enlevée). Il restera dans cet état jusqu'à ce que son processus père ait récupéré son code de sortie. Les appels systèmes **wait** et **waitpid** permettent au processus père de récupérer ce code. A ce moment le processus termine et disparaît complètement du système.

- L'appel système **wait()**

```

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

```

L'appel système **wait** suspend l'exécution du processus appelant jusqu'à ce qu'un de ses fils se termine. Si un fils est déjà terminé, **wait** renvoie le PID du fils immédiatement sans bloquer. Il retourne l'identifiant du processus fils et son état de terminaison dans **status** (si **status** est différent de **NULL**). Si par contre le processus appelant ne possède **aucun fils**, **wait** retourne **-1**.

```

pid_t pid=fork();
switch(pid){
    case -1: /* Erreur de création */
        ...
    case 0: /* Programme du fils */
        printf("processus fils %d\n", getpid());
        exit(10);
    default: /* Programme du père */
        printf("processus père %d\n", getpid()) ;
        pid_t id = wait (&status);
        printf("fin processus fils %d\n", id);
        exit(0);
}

```

- L'appel système **waitpid()**

```

#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid , int* status , int options);

```

L'appel système **waitpid()** permet à un processus père d'attendre un fils particulier d'identité **pid**, de façon **bloquante** (**options=0**) ou **non bloquante** (**options=WNOHANG**).

Si l'appel réussit, il renvoie l'identifiant du processus fils et son état de terminaison dans **status** (si **status** n'est pas NULL). En cas d'erreur **-1** est renvoyé. Si l'option **WNOHANG** est utilisée et aucun fils n'a changé d'état, la valeur de retour est **0**.

```
pid_t id;
pid_t pid=fork();
switch(pid){
    case -1: /* Erreur de création */
        ...
    case 0: /* Programme du fils */
        ...;
        exit(10);
    default: /* Programme du père */
        ...
        While((id=waitpid(pid,&status,WNOHANG))==0)
            printf("processus fils non encore terminé\n");
        printf("fin processus fils %d\n", id) ;
        ...
}
```

L'appel **waitpid(-1, &status, 0)** est équivalent à l'appel **wait(&status)**

– **Code de sortie retourné** par **wait()** et **waitpid()**

Pour extraire le code de retour du processus fils de **status**, il faut d'abord tester que le processus c'est terminé normalement. Pour cela, on utilise la macro **WIFEXITED(status)**, qui renvoie **vrai** si le processus fils c'est terminé normalement. Ensuite pour obtenir son code de retour, on utilise la macro **WEXITSTATUS(status)**.

```
pid_t id;
pid_t pid=fork();
switch(pid){
    case -1: /* Erreur de création */
        ...
    case 0: /* Programme du fils */
        ...;
        exit(10);
    default: /* Programme du père */
        ...
        while((id=waitpid(pid,&status,WNOHANG))==0)
            printf("processus fils non encore terminé\n");
        if(WIFEXITED(status))
            printf("fils %d terminé par exit(%d)\n",id,WEXITSTATUS(status));
        ...
}
```

4 Les Systèmes d'Interruption

4.1 Problématique

Dans un ordinateur, ils coexistent deux types de programmes :

- Les **programmes usagers** qui font un calcul utile.
- Les **programmes du S.E.** qui font un travail de **superviseur** de tous les événements qui arrivent à la machine.

Ces deux types de programmes se partagent les ressources communes de la machine et plus particulièrement le processeur.

Lorsqu'un programme est en cours d'exécution, plusieurs événements peuvent arriver :

a. Les événements synchrones qui sont liés à l'exécution du programme en cours, comme :

1. Division par zéro.
2. Exécution d'une instruction inexistante ou interdite.
3. Tentative d'accès à une zone protégée.
4. Appel à une fonction du S.E.

b. Les événements asynchrones qui ne sont pas liés à l'exécution du programme en cours :

1. Fin d'opération d'E/S.
2. Signal d'horloge.

La supervision de ces deux types d'événements se fait par des contrôles continus sur l'arrivée de ceux-ci.

Question

Par quel mécanisme peut-on réduire le temps de supervision du S.E. ?

Solution

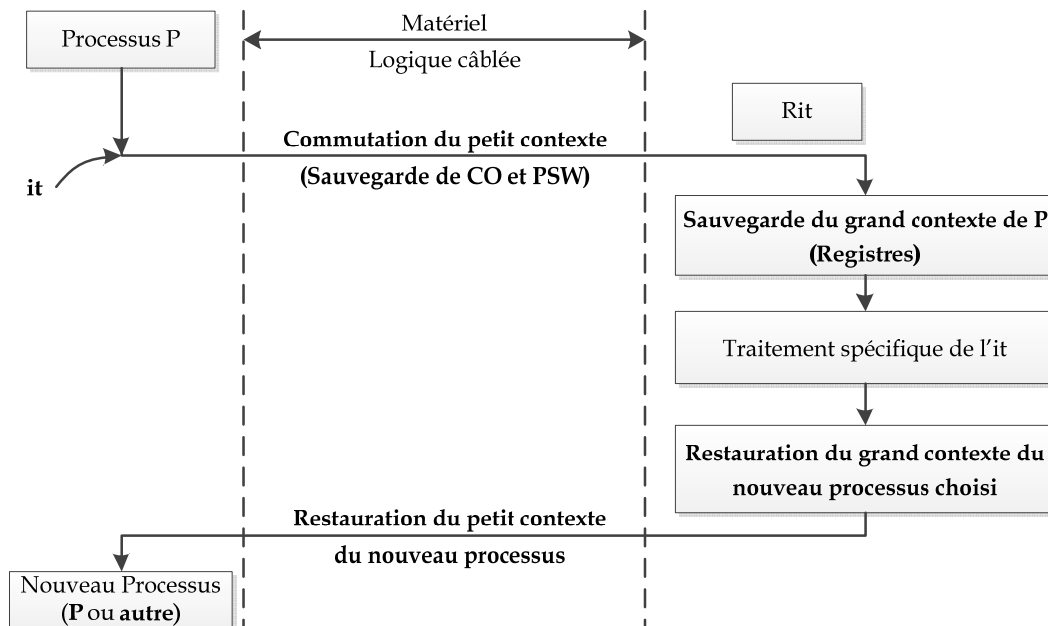
Au lieu que ça soit le processeur qui contrôle continuellement l'état d'une ressource, c'est plutôt à la ressource d'informer le processeur central sur son état au moment significatif. C'est le **principe des interruptions** de programmes.

4.2 Définition (Interruption)

Une **interruption** est une réponse à un événement qui interrompt l'exécution du programme en cours à un **point observable** (interruptible) du processeur central. Physiquement, l'interruption se traduit par un signal envoyé au processeur. Elle permet de forcer le processeur à **suspendre** l'exécution du programme en cours, et à déclencher l'exécution d'un programme prédéfini, spécifique à l'événement, appelé **routine d'interruption** (Rit).

4.3 Mécanismes de gestion des interruptions

A. Organigramme général



Remarque

Le programme repris après le traitement de l'interruption peut être le programme interrompu ou autre.

B. Conditions d'arrivée d'une interruption

Une interruption ne peut arriver au processeur que dans les conditions suivantes :

1. Le système d'interruption est **actif**.
2. L'UC est à un point observable (interruptible).
3. L'interruption est **armée**.
4. L'interruption est **démasquée**.
5. L'interruption est plus **prioritaire** que le programme en cours.

– Système d'interruption actif

Dans certains cas, le processeur a besoin d'interdire toute interruption possible. Pour cela, il dispose d'un **mécanisme d'activation/désactivation globale des interruptions**.

Dans ces conditions, aucune interruption ne peut interrompre l'UC, et toute interruption est retardée à la prochaine activation du système d'interruption.

– L'interruption est armée

Une interruption désarmée ne peut interrompre l'UC. Ceci se passe comme si la cause de l'interruption était supprimée. Toute demande d'interruption faite durant son désarmement est perdue.

On utilise ce procédé quand on désire qu'un élément ne doive plus interrompre.

– L'interruption est démasquée

Parfois, il est utile de protéger, contre certaines interruptions, l'exécution de certaines instructions (par exemple, les programmes d'interruption eux-mêmes). Une interruption masquée ne peut alors interrompre l'UC, mais toute demande d'interruption faite durant le masquage est retardée (mémoire) pour être traitée à la levée du masquage.

Les informations concernant l'état masqué des interruptions figurent dans le mot d'état du processeur.

On utilise le procédé de masquage pour définir des règles de priorité entre différentes causes d'interruption. Ainsi, les interruptions de même niveau de priorité ou d'un niveau plus bas peuvent être masquées, alors qu'une interruption de priorité supérieure est en cours d'exécution.

Remarque

Le masquage porte sur un niveau ou une cause d'interruption, contrairement à l'activation qui porte sur l'ensemble du système d'interruption.

C. Types d'interruption

Les interruptions sont classées en deux grandes classes :

- Les interruptions **externes** ou **matérielles**.
- Les interruptions **internes** ou **logicielles**.

1. Interruptions externes

Ce sont les interruptions causées par des organes externes au processeur central, comme les horloges de temps, les périphériques d'E/S, ...etc.

Ces interruptions **asynchrones** (c'est-à-dire, peuvent arriver à tout moment indépendamment de l'exécution du programme en cours) sont dues à :

- Périphérique prêt.
- Erreur durant l'E/S.
- Fin d'E/S.
- Ecoulement d'un délai de garde (horloge).
- Réinitialisation du système.
- ...etc.

2. Interruptions internes

Ce sont des interruptions causées par l'exécution du programme. Ces interruptions sont synchrones et se divisent en deux sous-classes :

- Les **déroutements (trap ou exception)** qui sont dus à des erreurs lors de l'exécution d'un programme et en empêchent la poursuite de son exécution. Ces erreurs peuvent avoir diverses causes :
 - Tentative d'exécution d'une opération interdite ou invalide.
 - Violation d'accès à une zone mémoire protégée ou inexistante.
 - Division par zéro.
 - Débordement arithmétique.
 - ...etc.

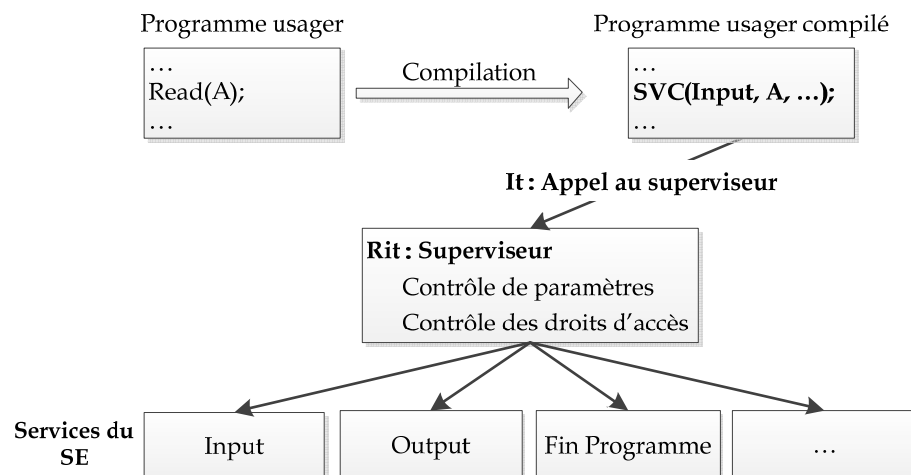
Remarque

Un déroutement ne peut être masqué ou retardé, mais il peut être supprimé (comme pour les déroutements liés aux opérations arithmétiques).

- Les **appels au superviseur (SuperVisor Call, SVC)** qui est une instruction permettant, à partir d'un programme utilisateur d'accéder à un service du S.E. (**Ex.** demande d'E/S, allocation dynamique de la mémoire, fin de programme, accès à un fichier, ...etc.).

Cette façon de procéder permet au système de :

- Se protéger des usagers.
- Vérifier les droits d'accès au service demandé.



L'appel au superviseur est en fait un appel de procédure système avec un passage de paramètres. Le service demandé par l'utilisateur sera choisi au moyen d'un paramètre supplémentaire désignant la cause de l'appel.

Toutefois, l'appel au superviseur d'un usager est traité comme interruption pour permettre une commutation du contexte usager par le contexte superviseur, car l'utilisation d'un service de superviseur nécessite un changement d'état (droits d'accès, mode d'exécution, priorité, ...etc.).

Remarque

La distinction entre interruption, déroutement et appel au superviseur se base sur la fonction, mais le mécanisme est commun.

D. Requêtes simultanées d'interruption

Durant un cycle processeur, plusieurs demandes d'interruption peuvent arriver simultanément.

Le traitement des requêtes simultanées se fait suivant un certain ordre de priorité établi entre les niveaux d'interruption. La priorité peut être fixe ou modifiable.

E. Requêtes imbriquées

Pour des raisons de temps critiques, certaines demandes d'interruption doivent être prises en compte, même quand le processeur est en train de traiter une interruption, on parle alors d'interruptions imbriquées.

Du point de vue fonctionnement, l'interruption d'une interruption est traitée comme une interruption classique de programme. En réalité, le processeur ignore même qu'il s'agit d'interruptions imbriquées.

5 Les signaux sous UNIX

5.1 Définition

Un signal est une **interruption logicielle** asynchrone d'un processus. Le signal peut être envoyé par un **processus** ou par le **noyau**.

5.2 Identification des signaux

Les signaux sont identifiés par un **numéro entier** et un **nom symbolique** décrit dans **signal.h**. Il est important d'utiliser les noms symboliques des signaux, les valeurs entières associées étant dépendantes de l'implémentation.

La liste des signaux peut être obtenue à l'aide de la commande **kill -l**. Le tableau ci-après présente quelques principaux signaux :

Nom du signal	Événement associé
SIGTERM	Terminaison d'un processus.
SIGINT	(interrupt) signal d'interruption émis à tous les processus associés à un terminal lorsque le caractère (Ctrl - c) est tapé.
SIGKILL	Terminaison d'un processus (non déroutable)
SIGQUIT	Signal d'interruption émis à tous les processus associés à un terminal lorsque le caractère (Ctrl - \) est tapé.
SIGTSTP	Arrêt temporaire d'un processus (Ctrl - z).
SIGSTOP	Suspension du processus (non déroutable).
SIGCONT	Signal de continuation d'un processus stoppé
SIGCHLD	Terminaison d'un processus fils
SIGABRT	Terminaison anormale provoquée par l'exécution de la fonction abort .
SIGALRM	Expiration d'un timer (fonction alarm)
SIGUSR1	Signal utilisateur 1 (disponible pour les applications)
SIGUSR2	Signal utilisateur 2 (disponible pour les applications)
SIGSEGV	Violation des protections mémoire (Segmentation Fault)
SIGFPE	Erreur arithmétique (Ex. division par zéro). (Floating-Point Arithmetic Error)
SIGILL	Détection d'une instruction illégale. (ILLegal)
SIGTRAP	Emis après chaque instruction en cas d'exécution en mode trace, par un débogueur.
SIGPIPE	Ecriture dans un tube sans lecteur.

5.3 Origines d'un signal

Les signaux Unix ont des origines diverses, ils peuvent être :

- transmis par le **noyau** (division par zéro, overflow, instruction interdite, ...etc.),
- envoyés depuis le **clavier** par l'utilisateur (touches **Ctrl - z**, **Ctrl - c**, ...etc.),
- émis par la **commande kill** depuis le shell,

Exemples

```
$ kill -KILL 2400
```

La commande ci-dessus envoie le signal SIGKILL au processus d'identité 2400.

```
$ kill -9 2400
```

La commande ci-dessus est identique à la commande précédente, à la différence que le signal est identifié par son numéro au lieu que ça soit par son nom symbolique.

- ou émis par la **primitive kill** dans un programme C/C++.

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
```

La primitive **kill** permet d'émettre le signal **sig** (désigné par son nom symbolique ou son numéro) à :

- processus d'identité **pid**, si **pid > 0**,
 - tous les processus du même groupe que le processus appelant, si **pid = 0**,
 - tous les processus du groupe dont le **gid** est la **valeur absolue de pid**, si **pid < -1**,
- à condition que les processus émetteur et destinataire(s) soient du **même propriétaire**, ou que le **premier** soit le **super-utilisateur** (root).

Cette primitive renvoie **0** en cas de succès et **-1** sinon.

5.4 Réponse à un signal

Il existe trois manières de répondre à un signal :

- **Exécution de l'action par défaut** pour le signal. Cinq traitements par défaut sont possibles:
 - **exit** : provoque la terminaison du processus (Ex. SIGINT, SIGKILL, SIGALRM)
 - **core** : sauvegarde l'état de la mémoire et termine le processus (Ex. SIGFPE, SIGBUS, SIGSEGV)
 - **stop** : suspend l'exécution du processus (Ex. SIGSTOP)
 - **ignore** : le signal est ignoré (Ex. SIGCHLD)
 - **continue** : le processus suspendu reprend son exécution ou le signal est ignoré (Ex. SIGCONT)
- **Ignorance** du signal. On peut ignorer un certain nombre de signaux (sauf par exemple SIGKILL, SIGSTOP) ;
- **Interception** puis **invocation** d'une fonction en réaction à l'événement. Le signal peut être rattrapé par le processus destinataire, ce qui provoque un **déroutement** et le lancement d'une routine spécifique de traitement (**handler**). Après le traitement, le processus reprend où il a été interrompu.

Un signal peut être dérouté soit à travers les appels systèmes **signal()** et **sigaction()** à partir d'un programme C/C++ ou en utilisant la commande **trap** à partir du shell.

- **L'appel système signal()**

```
#include <signal.h>
void (* signal(int signum, void (* handler)(int))) (int);
```

L'appel système **signal()** permet d'installer **handler** comme fonction de traitement lors de la réception du signal **signum**. Cette fonction ne pourra prendre qu'un argument de type entier : le **numéro du signal** qui l'aura appelée. La fonction renvoie **SIG_ERR** en cas d'échec.

Exemple

La fonction de déroutement (handler) permet d'afficher le numéro et le nom symbolique du signal dérouté. S'il s'agit du signal SIGINT, on arrête le programme.

La fonction **strsignal** permet de renvoyer le nom symbolique du signal numéro **sig**.

```
#include <string.h>
char *strsignal (int sig)

#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
//fonction de déroutement
void handler (int sig) {
    printf("Bien reçu %d %s\n", sig, strsignal(sig));
    if (sig == SIGINT) {
        printf ("Fin volontaire\n");
        exit (1);
    }
}
void main () {
    signal (SIGINT, handler); //Ctrl-c : signal 2
    signal (SIGQUIT, handler); //Ctrl-\ : signal 3
    signal (SIGTSTP, handler); //Ctrl-z : signal 20
    //SIGKILL est non déroutable
    if (signal (SIGKILL, handler) == SIG_ERR) perror("SIGKILL");

    for (;;)
}
```

Remarque

La fonction de traitement peut être remplacée par une des constantes suivantes :

- **SIG_IGN**, qui indique que le signal doit être ignoré.

```
/*Rendre Ctrl-C (SIGINT) inopérant*/
/*l'arrêt de ce programme doit se faire avec kill -9 pid*/
#include <signal.h>
void main () {
    signal (SIGINT, SIG_IGN);
    for (;;)
}
```

- **SIG_DFL**, qui indique de rétablir l'action par défaut pour le signal.

- **L'appel système sigaction()**

La primitive sigaction() associe un signal à un contexte de déroutement, représenté par une structure **sigaction**.

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

Le champ **sa_handler** est le **seul obligatoire** ; c'est un pointeur sur la fonction qui servira de handler.

```
#include <signal.h>
int sigaction (int signum, const struct sigaction* act,
struct sigaction* oldact)
```

Exemple

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <stdlib.h>
void handler (int sig) {
    printf ("Bien reçu %d %s\n", sig, strsignal(sig));
    if (sig == SIGINT) {
        printf ("Fin volontaire\n");
        exit (1);
    }
}
void main () {
    struct sigaction act;
    act.sa_handler = handler;
    sigaction (SIGINT, &act, NULL); //Ctrl-c : signal 2
    sigaction (SIGQUIT, &act, NULL); //Ctrl-\ : signal 3
    sigaction (SIGTSTP, &act, NULL); //Ctrl-z : signal 20
    //SIGKILL est non déroutable
    if (sigaction(SIGKILL, &act, NULL) == -1) perror ("SIGKILL");

    for (;;)
}
```

- **La commande trap**

La commande trap permet d'associer un traitement à un ou plusieurs signaux directement à partir du shell.

```
$ trap 'liste de commandes' sig1 sig2 ...
```

Exemple

La commande :

```
$ trap `rm /tmp/* ; exit` 2 3
```

a pour effet d'exécuter la commande **rm /tmp/* ; exit** à la réception des signaux SIGINT ou SIGQUIT.

Pour désactiver un ou plusieurs signaux, il suffit de transmettre la chaîne vide comme argument de la commande trap.

Exemple

La commande :

```
$ trap "" 2 3 15
```

permet de masquer les signaux SIGINT, SIGQUIT et SIGTERM.

Pour rétablir l'action par défaut pour un ou plusieurs signaux, il suffit de transmettre les numéros de ces signaux comme seul argument de la commande trap.

Exemple

La commande :

```
$ trap 2 3 15
```

permet de rétablir les actions par défaut associée aux signaux SIGINT, SIGQUIT et SIGTERM.

BIBLIOGRAPHIE

- W. STALLINGS, "*Operating Systems: Internals and Design Principles*", 7th Edition, Prentice Hall, Pearson Education, 2011.
- N. SALMI, "*Principes des Systèmes d'Exploitation*", Pages Bleues, les Manuels de l'Étudiant, 2007.
- B. LAMIROY, L. NAJMAN, H. TALBOT, "*Systèmes d'exploitation*", Collection Synthex, Pearson Education, 2006.
- A. BELKHIR, "*Système d'Exploitation, Mécanismes de Base*", OPU, 2005.
- A. Silberschatz, P.B. Galvin, G. Gagne, "*Operating System Concepts*", 7th Edition, John Wiley & Sons Editions, 2005, 921 p.
- A. TANENBAUM, "*Systèmes d'Exploitation : Systèmes Centralisés – Systèmes Distribués*", 3^{ème} édition, Editons DUNOD, Prentice Hall, 2001.
- A. Silberschatz, P. B. Galvin, "*Principes des Systèmes d'Exploitation*", traduit par M. Gatumel, 4^{ème} édition, Editions Addison-Wesly France, SA, 1994.
- M. GRIFFITHS, M. VAYSSADE, "*Architecture des Systèmes d'Exploitation*", Edition Hermès, 1990.
- S. KRAKOWIAK, "*Principes des Systèmes d'Exploitation des Ordinateurs*", Editions DUNOD, 1987.
- A. TANENBAUM, "*Architecture de l'Ordinateur*", Editions Pearson Education, 2006.
- J. M. LERY, "*Linux*", Collection Synthex, Pearson Education, 2006.
- J. Delacroix, "*LINUX, Programmation Système et Réseau, Cours et Exercices Corrigés*", Editions DUNOD, 2003.
- M. J. BACH, "*Conception du système UNIX*", Editions Masson, 1989.