IUT 'A' Paul SABATIER
Dpt Informatique

M3101 : Principes des systèmes d'exploitation

CM : Gestion des processus et des "threads" dans les systèmes UNIX/Linux

---

Introduction

Le dictionnaire définit un **processus** comme une suite continue et ordonnée d'opérations constituant la manière de fabriquer un objet ou d'effectuer une tâche.

Le **processus** est un concept clé de tout système d'exploitation.
Un programme objet décrit de façon statique, la manière de réaliser une tâche.
Pour réaliser une tâche, il faut faire exécuter par un processeur, le programme qui la décrit.
Donc tout programme en cours d'exécution par un processeur est un **processus**.

Dans les systèmes d'exploitations **multi-tâches**, plusieurs processus peuvent se dérouler **simultanément** suivant le principe de la **multiprogrammation**.

Sur une architecture **monoprocesseur**, cette simultanéité n'est qu'apparente car, il ne peut y avoir qu'un seul processus qui se déroule effectivement à un instant donné.
Les autres processus sont suspendus en attendant de pouvoir utiliser le processeur à leur tour.
Dans ce contexte, on dit que les processus se déroulent en **concurrence**.

Pour garantir un fonctionnement multi-tâche performant, le noyau du système doit fournir un mécanisme de régulation des tâches qu'on nomme **ordonnancement** (en anglais **scheduling**).

Ce mécanisme est mis en œuvre par une entité du noyau qu'on nomme l'ordonnanceur (en anglais **scheduler**).
C'est lui qui contrôle la répartition équitable de l'accès au processeur par les différents processus **concurrents**.

Sur une architecture **multiprocesseur**, le principe est le même, à la différence qu'à un instant donné, plusieurs processus peuvent se dérouler réellement en **parallèle** (un sur chacun des processeurs), mais pas tous car le nombre de processus existant à un instant donné, est en général nettement supérieur au nombre de processeurs de la machine.

Dans un système d'exploitation **multi-tâches**, il faut pouvoir contrôler la création de processus mais aussi leur terminaison.
De plus, plusieurs processus peuvent entrent en **conflit** pour accéder aux ressources dont ils ont besoin (données en mémoire, fichiers, ….), ou doivent échanger entre eux des informations pour **coopérer** à la réalisation d'une tâche.

Le noyau du système devra donc fournir des mécanismes de **création** et de **terminaison** de processus mais aussi des mécanismes de **synchronisation** et de **communication** entre processus.

Ces mécanismes sont fournis par une bibliothèque de sous-programmes du noyau (en anglais **system-calls**) auxquels les processus pourront faire appel suivant leurs besoins.

Le concept de "**thread**" (en français : processus léger, flux d'éxécution, …) est apparu pour simplifier la mise en œuvre d'actions en **parallèle** dans un programme.

Dans un programme **séquentiel**, les instructions doivent être exécutées une après l'autre (jamais plusieurs simultanément) en suivant l'enchaînement défini par les structures de contrôle du programme.
Dans un programme **parallèle**, il existe une ou plusieurs parties du programme, constituées de blocs d'instructions qui doivent s'exécuter en parallèle.

Un processus initié à partir d'un programme **séquentiel**, possède à tout moment un seul **flux d'exécution**.
Un processus initié à partir d'un programme **parallèle**, possédera un seul **flux d'exécution** dans les parties séquentielles du programme (s'il y en a) mais plusieurs **flux d'exécution** dans les parties parallèles.

Ces flux d'exécution sont appelés des "**thread**" en anglais.
Les **threads** permettent donc réaliser des sous tâches en parallèle à l'intérieur d'un processus et sont souvent qualifiés de sous-processus ou processus léger.

Un autre intérêt important des **threads** est que la même zone mémoire privée contenant les données d'un processus est commune à tous les **threads** internes à ce processus.

Le noyau d'un système qui implémente les **threads** devra donc fournir des mécanismes de contrôle des **threads** similaires à ceux fournis pour les processus.

L'objectif de ce cours est de présenter les principaux mécanismes de gestion des processus et des threads mis en œuvre dans les système UNIX /Linux.

CHAPITRE 1 : Contrôle des processus

1- Caractéristiques d'un processus

1.1- Modes de déroulement d'un processus

Un processus se déroule alternativement dans 2 modes :

- **le mode utilisateur** qui correspond à l'exécution d'instructions appartenant au programme objet.
- **le mode superviseur** appelé aussi mode **privilégié** ou **système** et qui correspond à l'exécution d'instructions appartenant à un sous-programme du noyau du système (appel système).

Le passage du mode utilisateur au mode superviseur est provoqué par l'occurrence d'un événement de type suivant :

- événement **interne synchrone** tel qu'un **appel système** (appel d'un sous-programme du noyau) ou qu'un **déroutement** lié à la levée d'une exception (erreur bus, violation de segment mémoire, division par zéro, …..).
- événement **externe asynchrone** pouvant survenir à tout moment tel qu'une **interruption matérielle** (générée par un contrôleur de périphériques ou l'horloge interne) ou qu'une **interruption logicielle** (signal émis par le noyau du système ou par un autre processus).

1.2- Structure d'un processus

1.2.1- Image mémoire d'un processus

Un processus est matérialisé dans le système par 2 ensembles d'informations stockés en mémoire centrale :

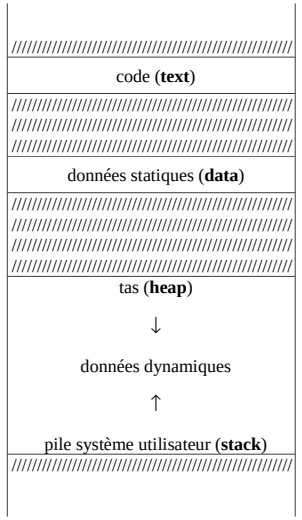
- le programme objet du processus
- l'environnement du processus (contexte d'exécution)

Dans la terminologie UNIX/Linux, ces informations sont appelées **image mémoire** du processus

1.2.2- Structure du programme objet

Le programme objet est mémorisé dans 3 segments alloués dans la partie de la mémoire centrale réservée aux utilisateurs (l'autre partie de la mémoire est occupée par le noyau du système et ses données) :

- un segment appelé **text** qui contient les instructions du programme objet. Ce segment est protégé en écriture et est partageable par tous les processus exécutant simultanément ce même programme objet
- un segment appelé **data** qui contient les données **statiques** du programme objet. En C par exemple, ces données correspondent aux constantes et variables globales et aux variables locales déclarées **static.**
- un segment contenant tête-bêche la pile système (**stack**) et le tas (**heap**). La pile est utilisée en mode utilisateur, pour gérer les appels et les retours des sous-programmes, pour la transmission des paramètres, pour allouer les variables locales non **static**, pour sauvegarder des registres, …).
  Le tas est utilisé pour allouer les variables **dynamiques** lorsque le processus fait de l'allocation dynamique.



1.2.3- Environnement du processus

L'environnement du processus est constitué par différentes structures de données mémorisées dans l'espace mémoire centrale réservé au système :

- **la table des processus**

  C'est un tableau d'enregistrements dont la taille est fixée par l'administrateur système ou déterminée automatiquement lors de la compilation du noyau du système. A chaque processus créé est alloué un élément dans la table qui contient des informations relatives à l'état du processus, à son identification, des pointeurs sur d'autres structures de données (structure U, table des textes, etc …).

- **la structure U**

  C'est un enregistrement qui contient des informations complémentaires sur le processus telles que l'identification des propriétaires, les descripteurs des fichiers ouverts, la taille et l'adresse des 3 segments du programme objet, le terminal de connexion, le répertoire courant et celui de connexion, le umask, …..
  La structure U est stockée dans l'espace mémoire centrale réservé au système mais peut être **swappée** si nécessaire.

- **la pile système**

  Cette pile n'est utilisée que lorsque le processus est en mode **superviseur**. Elle est allouée dans l'espace mémoire centrale réservé au système mais peut être **swappée** si nécessaire.

- **la table des textes**

  Elle sert à gérer le partage des segments **text** par plusieurs processus.

- **les tables des régions et prérégions**

  Ces tables servent à gérer la mémoire virtuelle (correspondances entre adresse virtuelles et adresses physiques).

1.3- Etats d'un processus

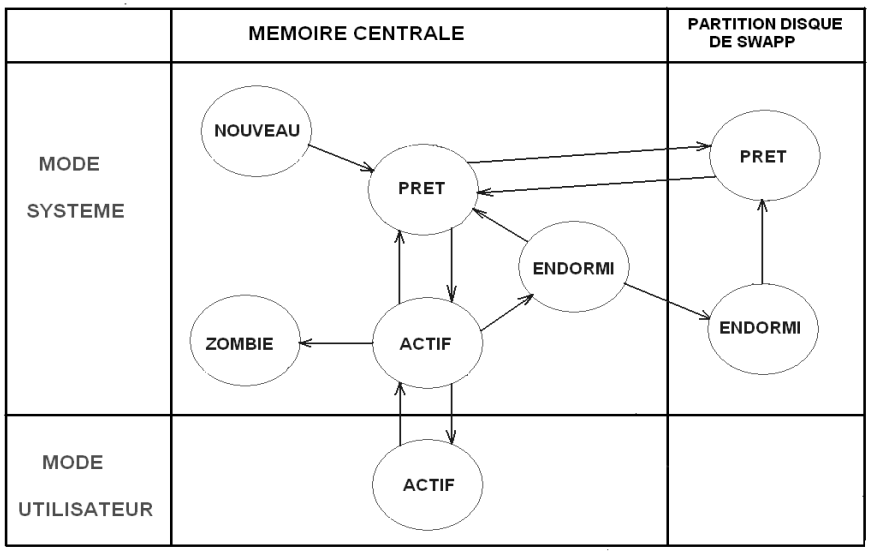
Durant son existence un processus peut se trouver dans différents états :

- suivant qu'il se déroule en mode système ou en mode utilisateur

- suivant qu'il est suspendu ou endormi en l'attente d'un événement qui le réveillera

- lorsqu'il est endormi il pourra être transféré en zone de swap

La figure ci-dessous représente le diagramme d'état d'un processus.



2- Création, continuation et terminaison d'un processus

2.1- Identification d'un processus

Un processus est créé par un autre processus (processus parent) dont il est l'enfant.
Seul le processus **init** n'a pas de parent car il est créé par le noyau au démarrage du système.
C'est le premier processus à partir duquel seront créés tous les autres processus.

Un processus est identifié par 3 attributs :

PID : numéro du processus.
PPID : numéro du processus parent.
PGID : numéro du groupe auquel appartient le processus.

Par exemple :

Pour le processus **init** => PID=1, PPID=0, PGID=0

Appels système permettant d'obtenir ces attributs :

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

Fonction retournant le numéro de processus (PID) :

pid_t **getpid**(void) ;

Fonction retournant le numéro de processus parent (PPID) :

pid_t **getppid**(void) ;

Fonction retournant le numéro du groupe du processus (PGID) :

pid_t **getpgrp**(void) ;

En cas d'erreurs ces fonctions retournent le code d'erreur **-1**.

2.2- Identification des propriétaires d'un processus

L'utilisateur qui provoque la création d'un processus, est appelé propriétaire **réel** du processus. Le groupe auquel appartient cet utilisateur est appelé groupe propriétaire **réel**.

Lorsque le fichier **exécutable**, à partir duquel le processus à été créé, n'appartient pas au propriétaire **réel**, la question se pose de savoir si le processus possède les **droits** d'accès de ce dernier ou ceux du propriétaire du fichier.

Le système autorise les deux possibilités car certaines commandes doivent s'exécuter obligatoirement avec les **droits** d'accès du propriétaire du fichier et pas ceux du propriétaire **réel** du processus.
Le choix est fait en fonction d'un indicateur binaire (**setUID bit**) associé au fichier **exécutable**.

Le même principe est appliqué pour les **droits** d'accès du groupe propriétaire (**setGID bit**).

Le système attribue donc au processus un second propriétaire et un second groupe propriétaire, appelés respectivement propriétaire **effectif** et groupe **effectif**, par rapport auxquels sont gérés les droits du processus.

Si le **setUID bit** vaut **0** (cas général), le propriétaire **effectif** est le même que le propriétaire **réel**,
si le **setUID bit** vaut **1**, le propriétaire **effectif** est le propriétaire du fichier **exécutable**.

Le même principe est appliqué au groupe propriétaire **effectif** en fonction de la valeur du **setGID bit**.

Donc pour le système, les droits d'accès d'un processus sont ceux du propriétaire **effectif** et du groupe propriétaire **effectif** et non ceux du propriétaire **réel** et du groupe propriétaire **réel**.

Les propriétaires d'un processus sont donc identifiés par 4 attributs :

UID : numéro du propriétaire **réel**.
GID : numéro du groupe propriétaire **réel**.
EUID : numéro du propriétaire **effectif**.
EGID : numéro du groupe propriétaire **effectif**.

Appels système permettant d'obtenir ces attributs :

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

Fonction retournant le numéro du propriétaire **réel** (UID) :

```
uid_t  getuid( void ) ;
```

Fonction retournant le numéro du groupe propriétaire **réel** (GID) :

```
uid_t  getgid( void ) ;
```

Fonction retournant le numéro du propriétaire **effectif** (EUID) :

```
uid_t  geteuid( void ) ;
```

Fonction retournant le numéro du groupe propriétaire **effectif** (EGID) :

```
uid_t  getegid( void ) ;
```

En cas d'erreurs ces fonctions retournent le code d'erreur **-1**.

2.3- Création d'un processus

La création d'un processus est réalisée par l'appel de la fonction **fork** dans le processus **parent**.
Le processus **enfant** ainsi crée est la copie conforme du processus parent (mécanisme de **mitose** ou de **clonage**).

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

    pid_t   fork( void ) ;
```

Si la création s'effectue correctement la fonction retourne :

- dans le processus **enfant** : **0**.
- dans le processus **parent** : le **PID** de l'enfant.

Sinon la fonction retourne le code d'erreur **-1**.

Déroulement de l'appel système fork

Les opérations effectuées par le noyau à la suite de l'appel de **fork** sont les suivantes :

- vérifier que le nombre maximum de processus autorisés par utilisateur n'est pas dépassé puis que le nombre maximum de processus autorisés par le noyau n'est pas dépassé (existence d'au moins une place libre dans la table des processus).
- vérifier l'état des ressources système (espaces mémoire centrale suffisants, place disponible dans la zone de swap sur disque, …).

- affecter un PID au processus enfant puis lui allouer une entrée libre dans la table des processus et y recopier les informations de l'entrée du processus parent (sauf le PID et le PPID).
- dupliquer l'image mémoire du parent dans les segments mémoires alloués au enfant (sauf le segment text).
- mettre à jour les compteurs de références de la table des inodes (cf TP Admin)et de celle des fichiers ouverts.
- placer le processus enfant ainsi créé en queue de la file d'attente du processeur (run queue).
- retourner la valeur 0 dans le code de l'enfant et le PID de l'enfant dans le code du parent.

Héritage du processus enfant

Les conséquences du mécanisme de **clonage** utilisé sont que l(enfant hérite du contexte d'exécution du parent tel qu'il était juste avant l'appel fork (segment text et data, pile et tas) et des informations stockées dans l'entrée de la table des processus et dans la structure U :

- identification des propriétaires et groupes propriétaires réels et effectifs.
- environnement du shell.
- descripteurs de fichiers ouverts.
- traitement des signaux.
- taille limite des fichiers (ulimit).
- masque de création des droits des fichiers (umask).
- terminal de connexion.
- répertoire de connexion et répertoire courant.
- etc …

2.4- Terminaison d'un processus

Un processus peut se terminer de 2 façons :

- **normalement** par l'appel d'une des fonctions **exit()** ou **_exit()**.

- **anormalement** parce qu'il est avorté par le système ou par l'utilisateur propriétaire (touche **<^c>** ou <**del**>, …).
  Dans les deux cas, le processus reçoit un **signal** qui le déroute vers un **traitement d'exception** dans le noyau qui se termine aussi par l'appel de la fonction **_exit().**
  Le mécanisme des signaux sera abordé au chapitre 2.

```
# include <unistd.h>

    void  _exit( int statut) ;

# include <stdlib.h>

    void  exit( int statut ) ;
```

La fonction **_exit()** ferme tous les fichiers ouverts, envoie le signal **SIGCHLD** au processus parent, puis libère toutes les structures de données représentant le processus dans le système.

La fonction **exit()** recopie les tampons de sortie dans les fichiers ouverts **en sortie** ou **en mise à jour** puis appelle la fonction **_exit()**.

Dans le cas d'une terminaison **normale,** le paramètre *statut* peut être utilisé comme un code d'erreur retourné par le processus avec la convention suivante sous UNIX/Linux :

- **0** si le processus s'est déroulé entièrement sans erreur.
- une valeur comprise entre **1** et **255** si le processus doit s'arrêter suite à une erreur qui ne lui permet pas de poursuivre son traitement.

Exemple : création et terminaison de processus

```
…..
pid_t  idProc ;
…..
idProc = fork() ;
switch  (idProc )
{
     case -1 : /* la création du processus enfant a échoué*/
                /* afficher un message d'erreur et arrêter le processus parent*/
                  perror("echec fork") ; exit(1) ;

     case 0 : /* la création du processus enfant a réussi*/
               /* le processus enfant va poursuivre son exécution en appelant une fonction */
                /* dans laquelle est codé son traitement*/
                  traitement_enfant( ) ;
                  exit(0) ;    /* on peut aussi placer un exit(0) à la fin de la fonction traitement_enfant */
}
/* suite du traitement du parent */
```


2.5- Synchronisation de processus parents

Un processus parent peut **suspendre** son exécution en attendant qu'un de ses enfant se termine, en appelant la fonction **wait().**

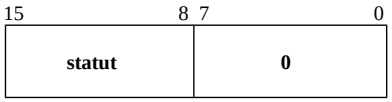
```
# include <sys/types.h>
# include <unistd.h>

        pid_t  wait( int * pRapport ) ;
```

Dés qu'au moins un des enfant est terminé, la fonction retourne le **PID** de ce processus enfant et affecte à la variable entière pointée par ***pRapport***, des informations indiquant de quelle façon il s'est terminé.
Le processus enfant correspondant est alors **détruit** par le système.

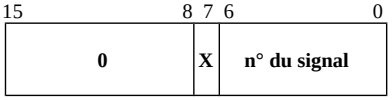
La fonction retourne **–1** si tous les processus enfant ont été détruits.

Les informations affectées à la variable entière pointée par ***pRapport,*** sont codées sur les 16 bits faibles poids de la façon suivante :

- **terminaison normale**



- **terminaison anormale** (réception d'un signal)



Le bit 7 vaut **1** si un fichier **core** à été généré(core dumped), et 0 sinon.

Exemple : boucle d'attente de la terminaison de plusieurs processus enfant

```
……...
pid_t  idProc ;
int  rapport, numSig, statut ;

……..

idProc = wait( &rapport ) ;

while  ( idProc != -1 )
{
    numSig = rapport & 0x7F ;

    switch ( numSig )
    {
        case 0 :   /* fin normale */
                   statut = ( rapport >> 8 )  & 0xFF  ;
                   printf("fin normale de l'enfant PID= %d , statut= %d \n ", idProc, statut) ;
                   break ;

        default :   /* fin anormale */
                   printf("fin anormale de l'enfant PID= %d , numéro du signal = %d \n ", idProc, numSig) ;
                   break ;
    }
    idProc = wait( &rapport ) ;
}
…….
```

Lorsqu'un processus se termine, le système supprime en mémoire toutes les structures de données qui matérialisaient ce processus sauf l'enregistrement qui lui est associé dans la table des processus.

On dit que le processus est dans l'état **zombie** (mort vivant).
Le système envoie ensuite le signal **SIGCHLD** au processus parent.

Si le processus parent prend acte de la terminaison de ce processus enfant en appelant la fonction **wait**, alors le système supprime l'enregistrement qui lui est associé dans la table des processus.

Si le processus parent se termine sans faire de **wait** ou s'il s'est déjà terminé avant que le enfant se termine, le processus enfant devient **orphelin**.

Afin de ne pas laisser 'traîner' des processus orphelins qui encombreraient inutilement la table des processus, le système les fait adopter par **init** qui régulièrement fait des **wait** pour les supprimer.

Le processus **init** est le seul processus qui ne se termine jamais sauf quand on arrête le système, c'est pour cette raison que c'est lui qui adopte les processus orphelins pour pouvoir les supprimer.
D'où son surnom de **bourreau des orphelins** dans la mythologie UNIX.

2.6- <u>Exécution d'un autre programme par un processus (continuation)</u>

Après sa création, un processus enfant commence son exécution à la première instruction qui suit l'appel de la fonction **fork()** et possède le même fichier exécutable que son parent.

Ceci est très contraignant, notamment si le traitement de l'enfant est différent de celui du parent ou très volumineux. Le système permet à un processus de remplacer le programme en cours par un autre programme, chargé à partir d'un nouveau fichier exécutable, en appelant une des 6 variantes de la fonction **exec()** :

```
# include <unistd.h>

   int execl ( const char * fichProg, const char * arg0,……, const char * argn, NULL ) ;

   int execlp ( const char * fichProg, const char * arg0,……, const char * argn, NULL ) ;

   int execv ( const char * fichProg, const char * tabArg[ ] ) ;

   int execvp ( const char * fichProg, const char * tabArg[ ] ) ;

   int execle ( const char * fichProg, const char * arg0,……, const char * argn, NULL, const char * tabEnv[ ] ) ;

   int execve ( const char * fichProg, const char * tabArg[ ], const char * tabEnv[ ] ) ;
```


Le premier paramètre *fichProg* désigne le nouveau fichier exécutable.

Si le nom du fichier n'est pas préfixé par un chemin d'accès **absolu**, la recherche du fichier sera faite à partir :

- du répertoire **courant** pour les fonctions **execl**, **execv**, **execle** et **execve**
- des répertoires successifs de la variable **PATH** pour les fonctions **execvp** et **execlp**

Le fichier peut être un programme exécutable binaire ou un programme de commandes **script shell**.

En cas d'erreur la fonction retourne le code **–1**, sinon le processus commence à exécuter ce nouveau programme.

Les paramètres du programme peuvent être transmis sous la forme :

- d'une liste de chaînes de caractères (*arg0*,…., *argn*, NULL) pour les fonctions **execl**, **execlp**, **execle**
- d'un tableau de chaînes de caractères (*tabArg*[ ]) pour les fonctions **execv**, **execvp**, **execve**

Par convention *arg0* ou *tabArg*[0] doivent être présents et contenir le nom du fichier même s'il n'y a aucun paramètre.

Dans les fonctions **execle** et **execve** le tableau de chaînes de caractères *tabEnv*[ ] permet de définir de nouvelles variables d'environnement ou de modifier celles exportées par le shell, sous la forme :

variable=valeur

2.7- Codage d'un programme en langage C avec paramètres

Un programme exécutable (script shell ou compilé binaire) peut être appelé **avec des paramètres effectifs**

- soit directement par un shell:

```
prog  param_1  param_2 … ↵
```

- soit indirectement par un appel système **exec** comme décrit au §2.6

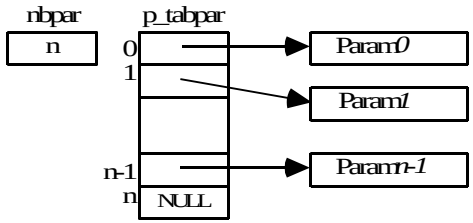
Les paramètres effectifs sont transmis uniquement en mode **entrée** sous forme de **chaînes de caractères**.

Dans le cas d'un programme en langage C qui sera appelé avec des paramètres, la déclaration de la fonction **`main`** doit être de la forme suivante :

```
int main( int nbpar, char * p_tabpar[] ) ↵
```

`nbpar` nombre de paramètres effectifs + 1 (le nom du programme est aussi transmis)

`p_tabpar` tableau de pointeurs sur les chaînes de caractères représentant les paramètres effectifs



Remarques importantes :

p_tabpar[0] pointe sur une chaîne contenant le nom du programme (équivalent à **$0** en shell)

Même si le programme est appelé sans aucun paramètre **p_tabpar[0]** est défini et **nbpar** vaut **1**.

Si un paramètre effectif représente un **nombre**, il devra être converti dans la représentation interne binaire avec une des fonctions de conversion **`atoi`** , **`atof`** ou **`sscanf`**.

Exemple1 : utilisation de execl

```
/* un processus parent crée un processus enfant pour exécuter la commande rm afin de supprimer
2 fichiers, fic1 et fic2, dans le répertoire courant */
…..
pid_t  idProc ;
int  err ;
…..
idProc = fork() ;
switch  (idProc )
{
     case -1 : /* afficher un message d'erreur et arrêter le processus parent */
                perror("echec fork") ; exit(1) ;

     case 0 : /* exécution du programme rm par le processus enfant*/

          err = execl( ″/bin/rm″ , ″rm″ , ″fic1″ , ″fic2″ , NULL ) ;

          if ( err == -1 )
          { perror( ″echec execl″ ); exit(2); }
}
/* suite du traitement du parent */
……
```


Exemple2 : utilisation de execvp

```
/* un processus parent crée un processus enfant pour exécuter la commande rm afin de supprimer 2 fichiers, fic1 et
fic2, dans le répertoire courant */
…..
pid_t  idProc ;
int  err ;
char* tabArg[ ] = { ″rm″, ″fic1″, ″fic2″, NULL };
…..
idProc = fork() ;
switch  (idProc )
{
     case -1 : /* afficher un message d'erreur et arrêter le processus parent */
                perror("echec fork") ; exit(1) ;

     case 0 : /* exécution du programme rm par le processus enfant*/
               /* on suppose que la variable PATH contient le chemin /bin */

          err = execvp( ″rm″ , tabArg ) ;

          if ( err == -1 )
          { perror( ″echec execvp″ ); exit(2); }
}
/* suite du traitement du parent */
……..
```


2.8- Les points de reprises

Au cours de son exécution, un programme peut détecter des **exceptions** (erreurs, signaux, etc..) dont le traitement conduit à reprendre partiellement le traitement déjà réalisé à partir d'un endroit précis dans le programme, appelé **point de reprise**.

Dans ce but, des informations décrivant un état d'avancement de la tâche doivent être mémorisées. Ces informations (pile, registres, …) représentent le **contexte** du processus en ce point du programme et seront restaurées à chaque reprise en ce point.

La gestion des points de reprise est réalisée par les 2 fonctions suivantes :

```
# include <setjmp.h>

     int  setjmp( jmp_buf  ptRep ) ;
```

à son **premier appel**, cette fonction permet de positionner un point de reprise.
Elle mémorise dans la variable *ptRep* le contexte du processus et retourne la valeur **0**.

```
     void  longjmp( jmp_buf  ptRep , int codeRep ) ;
```

cette fonction réalise un branchement vers le point de reprise matérialisé dans le programme par la fonction **setjmp** qui a pour paramètre *ptRep*.

La fonction **setjmp** est appelée une nouvelle fois, elle restaure le contexte du processus en ce point et retourne la valeur du paramètre *codeRep* (qui évidemment doit être différent de 0).

Remarques :

- Les variables pointeurs du type **jmp_buf** devront être déclarées **globales** si elles ne sont pas transmises en paramètre des fonctions qui les utilisent dans des appels **longjmp**.

- Le paramètre *codeRep* de **longjmp** doit être différent de **0**, pour distinguer le cas du retour du **setjmp** après la pose du point de reprise, du cas du retour après une reprise.

- La valeur de *codeRep* doit permettre de distinguer l'endroit du programme où a été fait le **longjmp**.

- Lorsqu'une fonction se termine par un **longjmp**, les variables **locales** de cette fonction ne sont plus visibles.

- Les valeurs des variables visibles ne sont pas **restaurées** suite à une reprise par un **longjmp**.
  A la reprise sur le **setjmp**, elles ont la valeur qu'elles avaient juste avant l'appel du **longjmp**.

- Il est impossible de revenir sur un point de reprise situé dans une fonction qui n'est pas en cours d'exécution. Donc on ne peut faire une reprise que dans la fonction courante ou dans une fonction appelante de niveau supérieur.

Exemple :

```
# include <setjmp.h>
……
jmp_buf  ptRep ;
int  ret ;
……
/* pose d'un point de reprise */
ret=setjmp( ptRep ) ;

if ( ret == 0 )
{
     /* traitement initial au premier passage sur le setjmp */
     …… ;
}
else
{
     /* retour après un longjmp */
     /* traitement de la reprise */
     ……. ;
}
…….
…….
/* reprise du traitement au point de reprise défini par ptRep */
longjmp( ptRep, 3 ) ;
…….
```