Chapitre 6

Les processus

6.1 Introduction aux processus

Un processus est un ensemble d'octets (en langage machine) en cours d'exécution, en d'autres termes, c'est l'exécution d'un programme.
Un processus UNIX se décompose en :

1. un espace d'adressage (visible par l'utilisateur/programmeur)
2. Le bloc de contrôle du processus (BCP) lui-même décomposé en :
   - une entrée dans la table des processus du noyau `struct proc` définie dans `<sys/proc.h>`.
   - une structure `struct user` appelée `zone u` définie dans `<sys/user.h>`

Les processus sous Unix apportent :

- La multiplicité des exécutions
  Plusieurs processus peuvent être l'exécution d'un même programme.

- La protection des exécutions
  Un processus ne peut exécuter que ses instructions propres et ce de façon séquentielle ; il ne peut pas exécuter des instructions appartenant à un autre processus.
  Les processus sous UNIX communiquent entre eux et avec le reste du monde grâce aux appels système.

6.1.1 Création d'un processus - `fork()`

Sous UNIX la création de processus est réalisée par l'appel système :

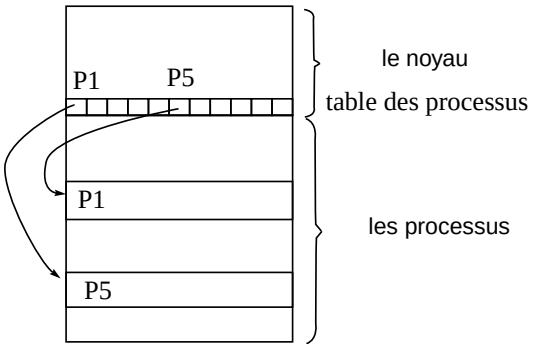

FIG. 6.1 – La table des processus est interne au noyau.

```
int fork(void);
```

Tous les processus sauf le processus d'identification 0, sont créés par un appel à *fork*.
Le processus qui appelle le fork est appelé processus *père*.
Le nouveau processus est appelé processus *fils*.

Tout processus a un seul processus père.
Tout processus peut avoir zéro ou plusieurs processus fils.

Chaque processus est identifié par un numéro unique, son **PID**.

Le processus de PID=0 est créé "manuellement" au démarrage de la machine, ce processus a toujours un rôle spécial[1] pour le système, de plus pour le bon fonctionement des programmes utilisant `fork()` il faut que le PID zéro reste toujours utilisé. Le processus zéro crée, grâce à un appel de fork, le processus init de PID=1.

Le processus de PID=1 de nom *init* est l'ancêtre de tous les autres processus (le processus 0 ne réalisant plus de `fork()`), c'est lui qui accueille tous les processus orphelins de père (ceci a fin de collecter les information à la mort de chaque processus).

6.2 Format d'un fichier exécutable

Les compilateurs nous permettent de créer des fichiers exécutables. Ces fichiers ont le format suivant qui permet au noyau de les transformer en processus :

- Une en-tête qui décrit l'ensemble du fichier, ses attributs et sa carte des sections.
- La taille à allouer pour les variables non initialisées.
- Une section TEXT qui contient le code (en langage machine)
- Une section données (DATA) codée en langage machine qui contient les données initialisées.

- Eventuellement d'autres sections : Table des symboles pour le débugeur, Images, ICONS, Table des chaînes, etc.

Pour plus d'informations se reporter au manuel a.out.h sur la machine.

6.3 Chargement/changement d'un exécutable

L'appel système `execve` change l'exécutable du processus courant en chargeant un nouvel exécutable. Les régions associée au processus sont préalablement libérées :

```
int execve(/* plusieurs formats */);
```

Pour chaque section de l'exécutable une région en mémoire est allouée.
Soit au moins les régions :

- le **code**
- les **données** initialisées

Mais aussi les régions :

- des **piles**
- du **tas**

La région de la **pile** :
C'est une pile de structures de pile qui sont empilées et dépilées lors de l'appel ou le retour de fonction. Le pointeur de pile, un des registres de l'unité centrale, indique la profondeur courante de la pile.

[1] swappeur,gestionnaire de pages

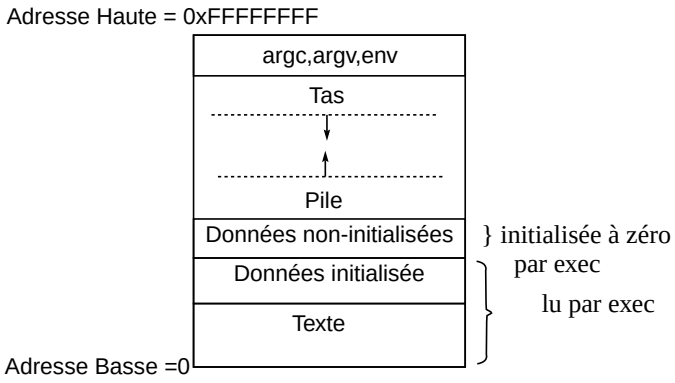

FIG. 6.2 – La structure interne des processus.

Le code du programme gère les extensions de pile (appel ou retour de fonction), c'est le noyau qui alloue l'espace nécessaire à ces extensions. Sur certains systèmes on trouve une fonction `alloca()` qui permet de faire des demandes de mémoire sur la pile.
Un processus UNIX pouvant s'exécuter en deux modes (noyau, utilisateur), une pile privée sera utilisée dans chaque mode.
La pile noyau sera vide quand le processus est en mode utilisateur.

Le **tas** est une zone où est réalisée l'allocation dynamique avec les fonctions `Xalloc()`.

6.4 zone u et table des processus

Tous les processus sont associés à une entrée dans la *table des processus* qui est interne au noyau. De plus, le noyau alloue pour chaque processus une structure appelée *zone u* , qui contient des données privées du processus, uniquement manipulables par le noyau. La *table des processus* nous permet d'accéder à la *table des régions par processus* qui permet d'accéder à la *table des régions*. Ce double niveau d'indirection permet de faire partager des régions.
Dans l'organisation avec une mémoire virtuelle, la table des régions est matérialisée logiquement dans la table de pages.

Les *structures de régions* de la table des régions contiennent des informations sur le type, les droits d'accès et la localisation (adresses en mémoire ou adresses sur disque) de la région.

Seule la **zone u** du processus courant est manipulable par le noyau, les autres sont **inaccessibles.** L'adresse de la zone u est placée dans le mot d'état du processus.

6.5 `fork` et `exec` (revisités)

Quand un processus réalise un `fork`, le contenu de l'entrée de la table des régions est dupliqué, chaque région est ensuite, en fonction de son type, partagée ou copiée.

Quand un processus réalise un `exec`, il y a libération des régions et réallocation de nouvelles régions en fonction des valeurs définies dans le nouvel exécutable.

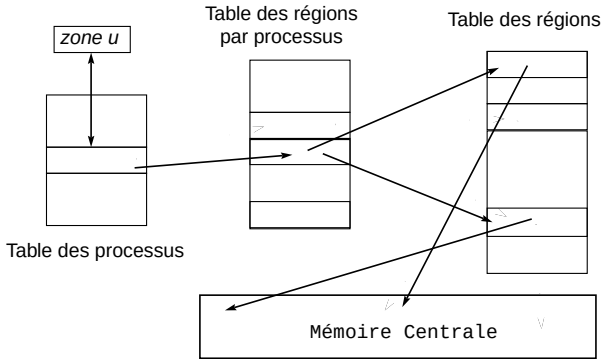

FIG. 6.3 – Table des régions, table des régions par processus

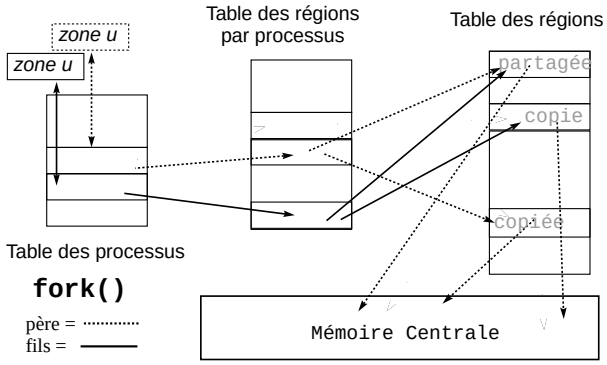

FIG. 6.4 – Changement de régions au cours d'un `fork`.

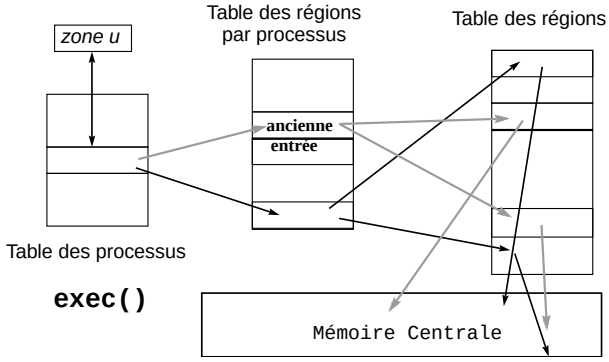

FIG. 6.5 – Changement de régions au cours d'un `exec`.

6.6 Le 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.

Le contexte d'un processus est l'ensemble de

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

Le noyau et ses variables ne font partie du contexte d'aucun processus !

L'exécution d'un processus se fait dans son contexte.

Quand il y a changement de processus courant, il y a réalisation d'une **commutation de mot d'état** et d'un **changement de contexte**. Le noyau s'exécute alors dans le nouveau contexte.

6.7 Commutation de mot d'état et interruptions.

Ces fonctions de très bas niveau sont fondamentales pour pouvoir programmer un système d'exploitation.

Pour être exécuté et donner naissance à un processus, un programme et ses données doivent être chargés en mémoire centrale. Les instructions du programme sont transférées une à une de la mémoire centrale sur l'unité centrale où elles sont exécutées.

L'unité centrale :
Elle comprend des circuits logiques et arithmétiques qui effectuent les instructions mais aussi des mémoires appelées registres.

Certains de ces registres sont spécialisés directement par les constructeurs de l'unité centrale, d'autres le sont par le programmeur du noyau. Quelques registres spécialisés :

L'accumulateur qui reçoit le résultat d'une instruction ; sur les machines à registres multiples, le jeu d'instructions permet souvent d'utiliser n'importe lequel des registres comme accumulateur.

le registre d'instruction (qui contient l'instruction en cours)

le compteur ordinal (adresse de l'instruction en mémoire) Ce compteur change au cours de la réalisation d'une instruction pour pointer sur la prochaine instruction à exécuter, la majorité des instructions ne font qu'incrémenter ce compteur, les instructions de branchement réalisent des opérations plus complexes sur ce compteur : affectation, incrémentation ou décrémentation plus importantes.

le registre d'adresse

les registres de données qui sont utilisés pour lire ou écrire une donnée à une adresse spécifiée en mémoire.

les registres d'état du processeur : (actif, mode (user/system), retenue, vecteur d'interruptions, etc)

les registres d'état du processus droits, adresses, priorités, etc

Ces registres forment le contexte **d'unité centrale d'un processus**. A tout moment, un processus est caractérisé par ces deux contextes : le contexte d'unité centrale qui est composé des mêmes données pour tous les processus et le contexte qui dépend du code du programme exécuté. Pour

	Nature de l'interruption	fonction de traitement
0	horloge	clockintr
1	disques	diskintr
2	console	ttyintr
3	autres peripheriques	devintr
4	appel system	sottintr
5	autre interruption	otherintr

FIG. 6.6 – Sous UNIX, on trouvera en général 6 niveaux d'interruption

pouvoir exécuter un nouveau processus, il faut pouvoir **sauvegarder** le contexte d'unité centrale du processus courant (mot d'état), puis charger le nouveau mot d'état du processus à exécuter. Cette opération délicate réalisée de façon matérielle est appelée **commutation de mot d'état.** Elle doit se faire de façon non interruptible ! Cette "Super instruction" utilise 2 adresses qui sont respectivement :
l'adresse de sauvegarde du mot d'état
l'adresse de lecture du nouveau mot d'état

Le compteur ordinal faisant partie du mot d'état, ce changement provoque l'exécution dans le nouveau processus.

C'est le nouveau processus qui devra réaliser la sauvegarde du contexte global. En général c'est le noyau qui réalise cette sauvegarde, le noyau n'ayant pas un contexte du même type.

Le processus interrompu pourra ainsi reprendre exactement où il avait abandonné.

Les fonctions `setjmp/longjmp` permettent de sauvegarder et de réinitialiser le contexte d'unité central du processus courant, en particulier le pointeur de pile.

6.8 Les interruptions

Une interruption est une commutation de mot d'état provoquée par un signal produit par le matériel.
Ce signal étant la conséquence d'un événement extérieur ou intérieur, il modifie l'état d'un indicateur qui est régulièrement testé par l'unité centrale.
Une fois que le signal est détecté, il faut déterminer la cause de l'interruption. Pour cela on utilise un indicateur, pour les différentes causes, On parle alors du **vecteur d'interruptions**.
Trois grands types d'interruptions :

- **externes** (indépendantes du processus) interventions de l'opérateur, pannes,etc
- **déroutements** erreur interne du processeur, débordement, division par zéro, page fault etc (causes qui entraine la réalisation d'une sauvegarde sur disque de l'image mémoire "core dumped" en général)
- **appels systèmes** demande d'entrée-sortie par exemple.

Suivant les machines et les systèmes un nombre variable de niveaux d'interruption est utilisé.

Ces différentes interruptions ne réalisent pas nécessairement un changement de contexte complet du processus courant.

Il est possible que plusieurs niveaux d'interruption soient positionnés quand le système les consulte. C'est le niveau des différentes interruptions qui va permettre au système de sélectionner l'interruption à traiter en priorité.

L'horloge est l'interruption la plus prioritaire sur un système Unix.

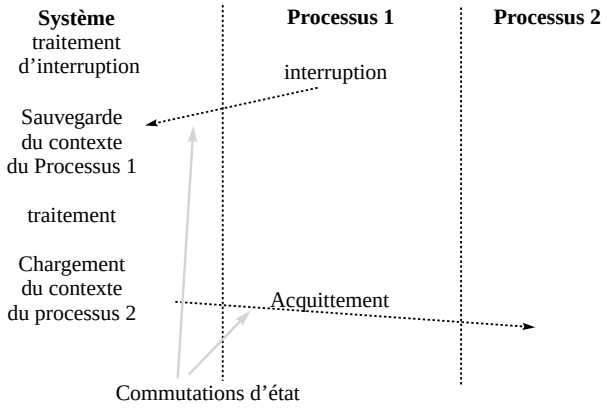

FIG. 6.7 – Le traitement d'une interruption.

6.9 Le problème des cascades d'interruptions

Si pendant le traitement d'une interruption, une autre interruption se produit, et que ceci se répète pendant le traitement de la nouvelle interruption, le système ne fait plus progresser les processus ni les interruptions en cours de traitement ...

Il est donc nécessaire de pouvoir retarder ou annuler la prise en compte d'un ou plusieurs signaux d'interruptions. C'est le rôle des deux mécanismes de **masquage** et de **désarmement** d'un niveau d'interruption. Masquer, c'est ignorer temporairement un niveau d'interruption.

Si ce masquage est fait dans le mot d'état d'un traitement d'interruption, à la nouvelle commutation d'état, le masquage disparaît ; les interruptions peuvent de nouveau être prises en compte. Désarmer, c'est rendre le positionnement de l'interruption caduque. (Il est clair que ceci ne peut s'appliquer aux déroutements).

6.9.1 Etats et transitions d'un processus

Nous nous plaçons dans le cas d'un système qui utilise un mécanisme de swap pour gérer la mémoire ; nous étudierons ensuite le cas des systèmes de gestion paginée de la mémoire (les couples d'états 3,5 et 4,6 y sont fusionnés).

6.9.2 Listes des états d'un processus

1. le processus s'exécute en mode utilisateur
2. le processus s'exécute en mode noyau
3. le processus ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. le processus est endormi en mémoire centrale
5. le processus est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible. (ce mode est différent dans un système à pagination).
6. le processus est endormi en zone de swap (sur disque par exemple).
7. le processus passe du mode noyau au mode utilisateur mais est préempté[2] et a effectué un changement de contexte pour élire un autre processus.
8. naissance d'un processus, ce processus n'est pas encore prêt et n'est pas endormi, c'est l'état initial de tous processus sauf le *swappeur*.

[2]Bien que le processus soit prêt, il est retiré de l'unité de traitement pour que les autres processus puissent avancer.

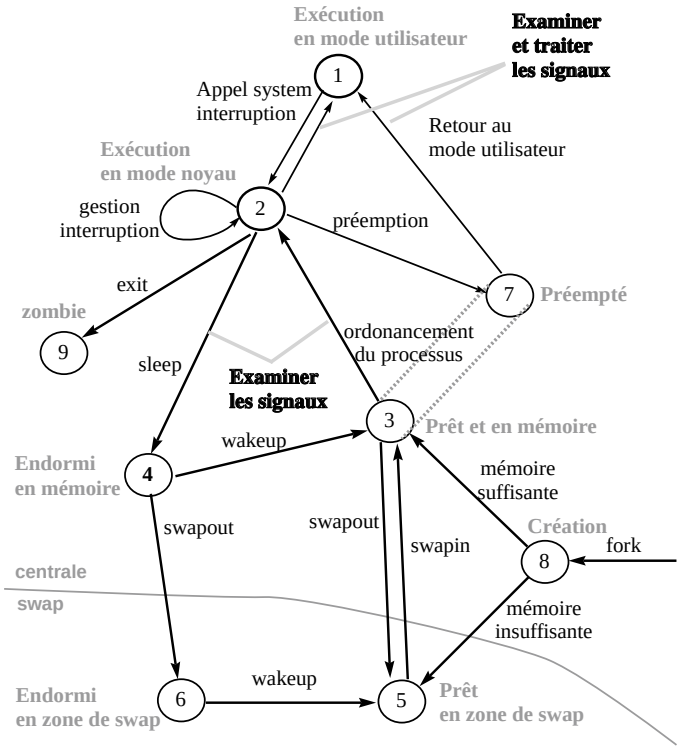

FIG. 6.8 – Diagramme d'état des processus

9. *zombie* le processus vient de réaliser un exit, il apparaît uniquement dans la table des processus où il est conservé le temps pour son processus père de récupèrer le code de retour et d'autres informations de gestion (coût de l'exécution sous forme de temps, et d'utilisation des ressources).

L'état *zombie* est l'état final des processus, les processus restent dans cet état jusqu'à ce que leur père lise leur valeur de retour (exit status).

6.10 Lecture du diagramme d'état.

Le diagramme des transitions d'état permet de décrire l'ensemble des états possibles d'un processus. Il est clair que tout processus ne passera pas nécessairement par tous ces différents états.

La naissance d'un processus a lieu dans l'état 8 après l'appel système `fork` exécuté par un autre processus. Il devient au bout d'un certain temps "prêt à s'exécuter". Il passe alors dans l'état "exécuté en mode noyau" où il termine sa partie de l'appel système `fork`. Puis le processus termine l'appel système et passe dans l'état "exécuté en mode utilisateur". Passé une certaine période de temps (variable d'un système à l'autre), l'horloge peut *interrompre* le processeur. Le processus rentre alors en mode noyau, l'interruption est alors réalisée avec le processus en mode noyau.

Au retour de l'interruption, le processus peut être `préempté` (étant resté tout son quantum de temps sur le cpu), c'est à dire, il reste prêt à s'exécuter mais un autre processus est élu. Cet état 7 est logiquement équivalent à l'état 3, mais il existe pour matérialiser le fait qu'un processus ne peut être préempté qu'au moment où il retourne du mode noyau au mode utilisateur. Quand un processus préempté est réélu, il retourne directement en mode utilisateur.

Un appel système ne peut être préempté. On peut détecter en pratique cette règle, en effet

on constate un ralentissement du débit de la machine pendant la réalisation d'un core de grande taille.

Quand un processus exécute un appel système, il passe du mode utilisateur au mode système. Supposons que l'appel système réalise une entrée-sortie sur le disque et que le processus doive attendre la fin de l'entrée-sortie. Le processus est mis en sommeil (sleep) et passe dans l'état endormi en mémoire. Quand l'entrée-sortie se termine, une interruption a lieu, le traitement de l'interruption consistant à faire passer le processus dans le mode prêt à s'exécuter (en mémoire).

6.11 Un exemple d'exécution

Plaçons-nous dans la situation suivante : l'ensemble de la mémoire est occupé par des processus, mais, le processus le plus prioritaire est un processus dans l'état 5, soit : "prêt à s'exécuter en zone de swap". Pour pouvoir exécuter ce processus, il faut le placer dans l'état 3, soit : "prêt à s'exécuter en mémoire". Pour cela le système doit libérer de la mémoire (faire de la place), en faisant passer des processus des états 3 ou 4 en zone de swap (swapout) donc les faire passer dans les états 5 et 6.
C'est au swappeur de réaliser les deux opérations :

- Sélectionner une victime (le processus le plus approprié), pour un transfert hors mémoire centrale (swapout).
- réaliser ce transfert.
- une fois qu'une place suffisante est libérée, le processus qui a provoqué le swapout est chargé en mémoire (swapin).

Le processus a un contrôle sur un nombre réduit de transitions : il peut faire un appel système, réaliser un `exit`, réaliser un `sleep`, les autres transitions lui sont dictées par les circonstances.

L'appel à `exit()` fait passer dans l'état zombie, il est possible de passer à l'état zombie sans que le processus ait explicitement appelé `exit()` (à la réception de certains signaux par exemple). Toutes les autres transitions d'état sont sélectionnées et réalisées par le noyau selon des règles bien précises. Une de ces règles est par exemple qu'un processus en mode noyau ne peut être préempté[3]. Certaines de ces règles sont définies par l'algorithme d'ordonnancement utilisé.

6.12 La table des processus

La table des processus est dans la mémoire du noyau. C'est un tableau de structure proc (`<sys/proc.h>`). Cette structure contient les informations qui doivent toujours être accessibles par le noyau.

état se reporter au diagramme, ce champ permet au noyau de prendre des décisions sur les changements d'état à effectuer sur le processus.

adresse de la zone u

adresses taille et localisation en mémoire (centrale, secondaire). Ces informations permettent de transférer un processus en ou hors mémoire centrale.

UID propriétaire du processus, permet de savoir si le processus est autorisé à envoyer des signaux et à qui il peut les envoyer.

PID,PPID l'identificateur du processus et de son père. Ces deux valeurs sont initialisées dans l'état 8, création pendant l'appel système `fork`.

évènement un descripteur de l'évènement attendu quand le processus est dans un mode endormi.

Priorités Plusieurs paramètres sont utilisés par l'ordonnanceur pour sélectionner l'élu parmi les processus prêts.

vecteur d'interruption du processus ensemble des signaux reçus par le processus mais pas encore traités.

[3]Exercice : Donner un exemple.

divers des compteurs utilisés pour la comptabilité (pour faire payer le temps CPU utilisé) et que l'on peut manipuler par la commande `alarm`, des données utilisées par l'implémentation effective du système, etc.

6.13 La zone u

La zone u de type `struct user` définie dans `<sys/user.h>` est la zone utilisée quand un processus s'exécute que ce soit en mode noyau ou mode utilisateur. Une unique zone u est accessible à la fois : celle de l'unique processus en cours d'exécution (dans un des états 1 ou 2).

Contenu de la zone u :

pointeur sur la structure de processus de la table des processus.

uid réel et effectif de l'utilisateur qui détermine les divers privilèges donnés au processus, tels que les droits d'accès à un fichier, les changements de priorité, etc.

Compteurs des temps (users et system) consommés par le processus

Masque de signaux Sur système V sous BSD dans la structure proc

Terminal terminal de contrôle du processus si celui-ci existe.

erreur stockage de la dernière erreur rencontrée pendant un appel système.

retour stockage de valeur de retour du dernier appel système.

E/S les structures associées aux entrées-sorties, les paramètres utilisés par la bibliothèque standard, adresses des buffers, tailles et adresses de zones à copier, etc.

"." et "/" le répertoire courant et la racine courante (c.f. chroot())

la table des descripteurs position variable d'un implémentation à l'autre.

limites de la taille des fichiers de la mémoire utilisable etc $\frac{1}{4}$(c.f. *ulimit* en Bourne shell et *limit* en Csh).

umask masque de création de fichiers.

6.14 Accès aux structures `proc` et `user` du processus courant

Les informations de la table des processus peuvent être lues grâce à la commande shell `ps`. Ou par des appels système. Par contre, les informations contenues dans la zone u ne sont accessibles que par une réponse du processus lui-même (en progammation objet, on dit que ce sont des variables d'instances privées), d'où les appels système suivants :
`times, chroot, chdir, fchdir, getuid, getgid, ..., setuid, ..., ulimit, nice, brk, sbrk`.
Qui permettent de lire ou de changer le contenu des deux structures.

6.14.1 Les informations temporelles.

```
#include <sys/times.h>
clock_t times(struct tms *buffer);
```

times remplit la structure pointée par `buffer` avec des informations sur le temps machine utilisé dans les état 1 et 2.
La structure**?** :

```
struct tms {
       clock_t    tms_utime;      /* user time */
       clock_t    tms_stime;      /* system time */
```

```
  clock_t    tms_cutime;     /* user time, children */
  clock_t    tms_cstime;     /* system time, children */
};
```

contient des temps indiqués en microsecondes 10-6 secondes, la précision de l'horloge est par defaut sur les HP9000 700/800 de 10 microsecondes.

6.14.2 Changement du répertoire racine pour un processus.

```
#include <unistd.h>
int chroot(const char *path);
```

permet de définir un nouveau point de départ pour les références absolues (commençant par /). La référence .. de ce répertoire racine est associée à lui-même, il n'est donc pas possible de sortir du sous-arbre défini par `chroot`. Cet appel est utilisé pour rsh et ftp, et les comptes pour invités.

Les appels suivants permettent de changer le répertoire de travail de référence "." et donc l'interprétation des références relatives :

```
int chdir(char *ref);
int fchdir(int descripteur);
```

6.14.3 Récupération du PID d'un processus

```
#include <unistd.h>
pid_t   getpid(void);
pid_t   getpgrp(void);
pid_t   getppid(void);
pid_t   getpgrp2(pid_t pid);
```

L'appel `getpid()` retourne le PID du processus courant, `getppid` le PID du processus père, `getpgrp` le PID du groupe du processus courant, `getpgrp2` le PID du groupe du processus pid (si pid=0 alors équivalent à `getpgrp`).

6.14.4 Positionement de l'euid, ruid et suid

L'uid d'un processus est l'identification de l'utilisateur exécutant le processus. Le système utilise trois uid qui sont :

euid uid effective utilisé pour les tests d'accès.

ruid uid réelle, uid à qui est facturé le temps de calcul.

suid uid sauvegardée, pour pouvoir revenir en arrière après un setuid.

```
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

Fonctionnement :
si euid == 0 (euid de root) les trois uid sont positionnés à la valeur de uid
sinon si uid est égal à ruid ou suid alors euid devient uid. ruid et suid ne changent pas. sinon rien ! pas de changements.
Syntaxe identique pour setgid et gid.

La commande *setreuid()* permet de changer le propiétaire réel du processus, elle est utilisé pendant le login, seul le super utilisateur peut l'exécuter avec succès.

6.15 Tailles limites d'un processus

```
#include <ulimit.h>
long ulimit(int cmd,...);
```

La commande *cmd* est

UL_GETFSIZE retourne le taille maximum des fichiers en blocs.

UL_SETFSIZE positionne cette valeur avec le deuxième argument.

UL_GETMAXBRK valeur maximale pour l'appel d'allocation dynamique de mémoire : brk.

Ces valeurs sont héritées du processus père.

La valeur FSIZE (taille maximum des fichiers sur disques en blocs) peut être changée en ksh avec ulimit [n].

6.15.1 Manipulation de la taille d'un processus.

```
#include <unistd.h>
int brk(const void *endds);
void *sbrk(int incr);
```

Les deux appels permettent de changer la taille du processus. L'adresse manipulée par les deux appels est la première adresse qui est en dehors du processus. Ainsi on réalise des augmentations de la taille du processus avec des appels à `sbrk` et on utilise les adresses retournées par `sbrk` pour les appels à `brk` pour réduire la taille du processus. On utilisera de préférence pour les appels à `sbrk` des valeurs de `incr` qui sont des multiples de la taille de page. Le système réalisant des déplacement du point de rupture par nombre entier de pages (ce qui est logique dans un système de mémoire paginé). A ne pas utiliser en conjonction avec les fonctions d'allocation standard `malloc, calloc, realloc, free`.

6.15.2 Manipulation de la valeur nice

Permet de changer la valeur de *nice* utilisée par le processus. Si l'on a des droits privilégiés la valeur peut être négative. La valeur de nice est toujours comprise entre -20 et +20 sous linux. Seul le super utilisateur pouvant utiliser une valeur négative.

```
#include <unistd.h>
int  nice(int valeur);
```

La commande shell `renice priorite -p pid -g pgrp -u user` permet de changer le nice d'un processus actif.

6.15.3 Manipulation de la valeur umask

L'appel `umask` permet de spécifier quels droits doivent être interdits en cas de création de fichier. cf. 5.1

```
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

la valeur retournée est l'ancienne valeur.

6.16 L'appel système `fork`

l'appel système `fork` permet le création d'un processus clône du processus courrant.

```
pid_t  fork(void);
```

DEUX valeurs de retour en cas de succès :
– Dans le processus père valeur de retour = le PID du fils,
– Dans le processus fils valeur de retour = zéro.
Sinon
– Dans le processus père valeur de retour = -1.
Les `PID` et `PPID` sont les seules informations différentes entre les deux processus.

6.17 L'appel système `exec`

```
#include <unistd.h>
extern char **environ;

int execl( const char *path, const char *arg0, ...,NULL);
int execv(const char *path, char * const argv[]);
int execle( const char *path, const char *arg0, ...,NULL,  char * const envp[]);

int execve(const char *file, char * const argv[], char * const envp[]);
int execlp( const char *file,const char *arg0, ... , NULL );
int execvp(const char *file, char * const argv[]);
```

Informations conservées par le processus : PID PPID PGID ruid suid (pour l'euid cf le `setuidbit` de `chmod`), nice, groupe d'accès, catalogue courant, catalogue "/", terminal de contrôle, utilisation et limites des ressources (temps machine, mémoire, etc), umask, masques des signaux, signaux en attente, table des descripteurs de fichiers, verrous, session.

Quand le processus exécute dans le nouvel exécutable la fonction :

```
main(int argc, char **argv,char **envp)
```

argv et *env* sont ceux qui ont été utilisés dans l'appel de `execve`.

Les différents noms des fonction `exec` sont des mnémoniques :

`l` liste d'arguments

`v` arguments sont forme d'un vecteur.

`p` recherche du fichier avec la variable d'environnement `PATH`.

`e` transmission d'un environnement en dernier paramètre, en remplacement de l'environnement courant.