Chapitre 4 LES PROCESSUS AVEC UNIX

1. GENERALITES SUR LES PROCESSUS SOUS UNIX

1.1 Contexte d'un processus

On distingue des **processus utilisateurs** et des **processus système**. Ces derniers :

- ne sont sous le contrôle d'aucun terminal
- ont comme propriétaire le super utilisateur (processus **démons**). Ils restent résidents en MC en attente d'une requête
- ils assurent des services généraux accessibles à tous les utilisateurs
- ils peuvent être créés au lancement du système ou à des dates fixées par l'administrateur.

Exemples de processus système :

cron lance à des dates spécifiées des commandes particulières
lpsched assure l'ordonnancement des requêtes d'impression

Le contexte d'un processus, défini à un instant donné de l'exécution, comprend l'ensemble des 3 environnements suivants nécessaires à son exécution :

- **environnement utilisateur :** (il se trouve dans le fichier exécutable)
  zone programme (code à exécuter) ou TEXT
  zone données (variables) ou DATA
  zone pile utilisateur (pour les variables locales et les appels de fonctions) ou BSS, dont la taille peut être modifiée dynamiquement.

La zone programme peut être partagée entre plusieurs processus. En outre, un processus peut partager des segments de mémoire (données) avec d'autres processus,

- **environnement machine :** ensemble des registres utilisés pour l'exécution (compteur ordinal, pointeur de pile, registres de travail, registres de données, registres pour la mémoire virtuelle, etc...)

- **environnement système :** situé dans le noyau, il comprend au moins les 3 structures suivantes :

  * *table des processus :* entité globale; chaque processus correspond à une ligne de la table (pid, état, priorité, utilisateur réel [celui qui l'a créé], utilisateur effectif [celui qui précise les droits d'accès], etc...)

  * *structure U* : il en existe une par processus. Elle contient des informations de contrôle : répertoire courant; pointeurs vers les fichiers ouverts; variables d'environnement)

  * *table des textes :* utilisée par la gestion du code partageable. Chaque ligne contient les informations suivantes : taille du texte, pointeur vers la structure U, vers l'i-node du fichier exécutable, nombre de processus qui exécutent ce texte, etc...)

L'environnement système initial est créé par le shell lors de la création du processus.

Un processus peut s'exécuter dans deux modes différents :

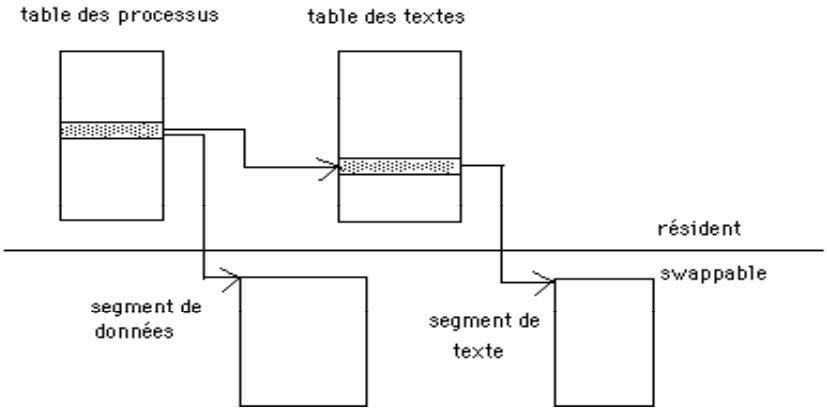
- **mode utilisateur (user mode) :** le processus n'accède qu'à son espace d'adressage et n'exécute que des instructions ordinaires du programme chargé

- **mode noyau ou système (kernel mode) :** le processus exécute des instructions n'appartenant pas au programme, mais au noyau. Il accède à des données externes à son espace d'adressage (tables systèmes). On peut passer en mode noyau soit par une interruption appropriée, soit par la réalisation d'un appel système par le processus lui-même (exemple : requête de lecture au clavier dans le programme)

1.2 Structure d'un processus sous UNIX

Le noyau gère une *table des processus* , listable par la commande **ps** avec une entrée par processus, et contenant des informations sur celui-ci. Cette entrée est allouée à la création du processus, désallouée à son achèvement.

Un processus est identifié par un *pid* (*process identifier*), entier de 0 à 32767 retourné par le noyau. Il est caractérisé aussi par sa priorité, son propriétaire et son groupe propriétaire, son terminal de rattachement.



1.3 Initialisation

A l'initialisation du système, s'exécute un bootstrap primaire résidant sur la piste 0 du disque qui charge un boostrap secondaire (incluant les fonctions d'un **échangeur**), et exécutant le noyau d'UNIX, de pid 0. C'est l'échangeur (**swapper)** qui assure la gestion du va-et-vient (overlay) en mémoire en remplissant les rôles suivants :

- *chargement* : trouver parmi les processus activables celui qui attend depuis le plus longtemps et qui peut s'implanter dans l'espace libre de la mémoire.

- *élimination :* on élimine d'abord les processus bloqués dans l'ordre de leur durée de résidence en mémoire centrale, et éventuellement d'autres processus.

Le noyau découpe la mémoire centrale, teste les volumes, vérifie leur cohérence, charge les tables systèmes nécessaires. Le noyau utilise l'échangeur pour créer 4 processus dont l'exécution de /etc/ **init, de pid 1**. **init** est l'ancêtre de tous les processus. Il consulte des tables (dont /sbin/inittab qui contient les conditions initiales du système). **init** crée par filiation autant de processus **getty** qu'il y a de lignes dans /etc/inittab.

Chaque utilisateur est identifié par un numéro individuel **UID** (*user identity*). La correspondance entre le nom et **UID** est assurée par le fichier **/etc/passwd**. Un groupe est un ensemble d'utilisateurs ayant des points communs. Un utilisateur appartient à un ou plusieurs groupes. Un groupe est identifié par un **GID** (*group identity*) et possède un nom. Leur correspondance est gérée par **/etc/grou**p.

→ La commande **uid** affiche l'**UID**, le login name, le **GID** et le nom du groupe de l'utilisateur.

Exemples:

1. connexion des utilisateurs (getty) : les processus **getty** , lancés par init, réalisent les 2 opérations suivantes :

- préparation du terminal en fonction des éléments de /etc/gettydefs, initialisation du groupe avec **setpgrp** et ouverture du fichier spécial correspondant au terminal défini
- bouclage sur les 3 opérations suivantes :
  - affichage d'un prompt contenu dans /etc/issue et d'un message d'invitation à la connexion,
  - lecture du nom de connexion
  - exécution du processus login qui se substitue à getty (même pid, chef de groupe)

2. administration de l'utilisateur (login) : login, qui reçoit le nom d'utilisateur lu par getty, lit le mot de passe, compare le login et le mot de passe crypté aux données de **/etc/passwd** . Si une concordance est bonne, login :

- met à jour les fichiers de comptabilité,
- modifie les droits de propriété du terminal pour concordance avec ceux de l'usager,
- détermine le répertoire d'accueil d'après /etc/passwd, initialise les variables HOME, PATH, LOGNAME,...
- exécute (par recouvrement) le programme nommé à côté de ce mot de passe dans /etc/passwd. Lorsque la session est achevée (fin de sh par exemple), il y a retour à **init** et suppression de tous les processus attachés au défunt shell
- exécute /etc/profile et aussi **/etc/.profile** (commun à tous les utilisateurs), **.profile** (s'il existe).

3. Lorsqu'on demande une **redirection de sortie** (> ou >>), le shell ferme le fichier de descripteur 1 (stdout en général), ouvre le fichier de redirection qui prend le descripteur 1 venant d'être libéré, et exécute le fork évoqué ci-dessus. Il en est de même pour une redirection d'entrée (cf. chapitre 11).

4. La fonction C **system** crée un processus shell qui interrompt le programme en cours (processus père) jusqu'à la fin du processus fils (argument de system).

5. La fonction **ioctl** prototypée dans termio.h permet de reconfigurer les pilotes de terminaux. Très bon exemple dans Braquelaire p. 358-359, 361-368.

1.4 Ordonnancement d'un processus

L'**ordonnanceur** ou **scheduler** (gestionnaire des ressources et d'enchaînement des processus) gère plusieurs processus concurremment et autorise le multitâche par partage de temps. Il attribue des **quanta** de temps. A l'expiration de son **quantum** de temps, le processus actif, s'il n'est pas terminé, sera obligé de relâcher le processeur. Un autre processus est alors élu et attribué au processeur. Le processus suspendu reprend son exécution quand son tour arrive à nouveau pour disposer du processeur.

Sous UNIX, l'horloge de l'ordonnanceur délivre 100 **tops**, ou cycles mineurs, par seconde et un **quantum**, ou cycle majeur, correspond à plusieurs tops (par exemple 100, soit 1 seconde).

Un processus peut prendre l'un des 9 états décrits dans **/usr/include/sys/proc.h** (codés dans le champ **S** dans la table des processus), parmi lesquels :

- actif : en mode utilisateur (pas de droit d'accès aux ressources du système) ou en mode système (ou noyau)

- activable ou prêt à exécuter : en mémoire ou en zone de swap. Il est éligible par l'ordonnanceur. Ces deux états sont codés **R**

- bloqué : en attente d'un événement (horloge, montage d'un disque, résultat d'un autre processus, ...), en mémoire ou en zone de swap. Il ne consomme pas de temps CPU. Il repasse à l'état activable dès que les conditions le permettent. Désigné par **S** (sleep) pendant les 20 premières secondes), puis par **I**

- non prêt : en création ou zombie (désigné par **Z**, le processus est achevé, mais son père n'en a pas encore connaissance). Il ne consomme pas de temps CPU.

L'ordonnanceur d'UNIX SystemV met en œuvre un algorithme de calcul de la priorité des processus candidats au processeur à partir de quatre paramètres figurant dans la table des processus :

C ou p_pcpu: rend compte de l'utilisation du processus. Si le processus utilise le processeur, C est incrémenté à chaque top d'horloge et divisé par 2 à chaque quantum.

NI ou p_nice: (nice) paramètre fixé par l'utilisateur (fonction UNIX **nice**) entre 0 (priorité maximale) et 39 (priorité minimale) ou à défaut par le système à la valeur 20.

Exemple : nice - 12 tp.e
abaisse de 12 la priorité du processus associé à l'exécution de tp.e
nice admet des arguments entre 1 et 19 (10 par défaut)

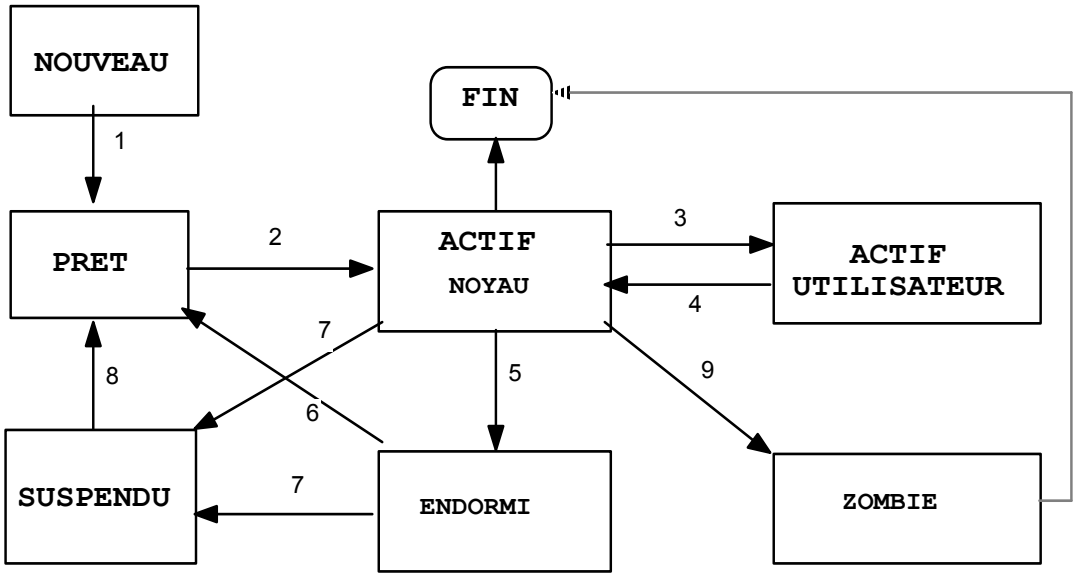
PRI ou p_user: (priority) est fixé à 60.

NZERO: est fixé à 20.

La priorité globale d'un processus utilisateur à un quantum donné vaut :

$$(C/2) + PRI - NZERO + NI$$

Elle ne peut donc être inférieure à 40. Les valeurs 0 à 39 sont réservées aux processus système (PRI = 0). Plus sa valeur est élevée, moins un processus a de chances d'accéder au processeur.



1 : le processus créé par fork a acquis les ressources nécessaires à son exécution
2 : le processus vient d'être élu par l'ordonnanceur
3 : le processus revient d'un appel système ou d'une interruption
4 : le processus a réalisé un appel système ou une interruption est survenue
5 : le processus se met en attente d'un événement (libération de ressource, terminaison de processus par wait). Il ne consomme pas de temps UC
6 : l'événement attendu par le processus s'est produit
7 : conséquence d'un signal particulier
8 : réveil du processus par le signal de continuation SIGCONT
9 : le processus s'achève par exit, mais son père n'a pas pris connaissance de sa terminaison. Il ne consomme pas de temps UC et ne mobilise que la ressource table des processus

La commande **ps** propose 4 options : **- a** (tous les processus attachés au terminal), **- e** (tous les processus), **- f** (full : plus de renseignements), **- l** (long : presque tous les renseignements). Les champs de la tables des processus qui sont édités sont les suivants en fonction des options - f ou - l :

F (- l) localisation d'un processus (0 = hors MC, 1 = en MC, 2 processus système, 4 = verrouillé car en attente de fin d'E/S, 10 = en vidage)

S (-l) state (O = non existant, S = sleeping, W = waiting, R = running, Z = zombie, X = en évolution, P = pause, I = attente d'entrée au terminal ou input, L = attente d'un fichier verrouillé ou locked)

UID	nom du propriétaire
PID et PPID	
PRI et NI (- l)	voir ci-dessus
SZ	taille en nombre de blocs
WCHAN (- l)	dans l'état W, l'événement attendu
STIME (- l)	heure de démarrage
TTY	ligne associée au terminal

TIME	temps d'exécution cumulé (mm:ss)
CMD	commande (et ses arguments si - f) exécutée par le processus

1.5 Gestion du va-et-vient (overlay)

L'échangeur (processus système de pid 0, créé au moment du démarrage du système) assure la gestion du va-et-vient de la mémoire. Il assure trois fonctions :

- *allocation d'un espace de swap :* la gestion de l'espace disque d'échange est différente de celle du reste du disque. On gère un ensemble contigu de blocs de taille fixe, à raison d'une map par périphérique d'échange, en utilisant les fonctions **swapalloc** (int taille) et **swapfree** (int adresse, int taille) de demande et de libération d'unités d'échange. Il peut y avoir plusieurs périphériques d'échange, créés et détruits dynamiquement.

- *transfert en mémoire :* l'échangeur examine tous les processus qui sont "prêt et transféré sur disque" et sélectionne celui qui a été transféré sur disque depuis le plus longtemps (allocation de MC, lecture du processus sur disque et libération de l'espace de swap). Il recommence jusqu'à ce qu'il n'y ait plus de processus "prêt et transféré sur disque" (il s'endort) ou bien jusqu'à ce qu'il n'y ait plus de MC libre (il exécute le point suivant).

- *transfert hors de la mémoire :* l'échangeur cherche à transférer d'abord les processus "bloqué" avant les "prêt", mais jamais les "zombie", ni les processus "verrouillé par le SE". Un processus "prêt" ne peut être transféré que s'il a séjourné au moins 2 s. en MC (même sans avoir été élu). S'il n' y a aucun processus transférable, l'échangeur s'endort et est réveillé toutes les secondes jusqu'à la fin de la nécessité de transfert.

Il peut y avoir "étreinte fatale" lorsque tous les processus en MC sont "endormi", tous les processus "prêt" sont transférés et il n' y a plus de place en MC ni en zone d'échange sur disque.

En plus du va-et-vient normal, il existe deux possibilités de transfert :

- à l'exécution d'un **fork**, lorsque la création du fils ne peut être faite directement en MC : on crée le fils sur disque par recopie du père

- lors de l'accroissement de la taille d'un processus : le SE transfère le processus sur disque en réservant une zone de swap plus grande; au retour en MC, le processus sera "plus grand".

2. PROGRAMMATION DE PROCESSUS AVEC LE SHELL

Exécution en parallèle (background)

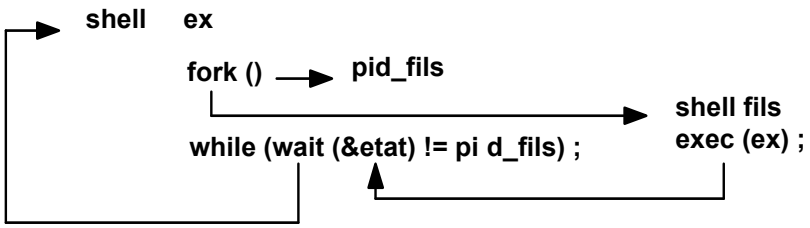
Soit **ex** un fichier de commandes exécutable.

1er cas : on tape **$ ex** Voici ce qui se passe :

1. le shell lit la commande ex
2. il se duplique au moyen de la fonction fork ; il existe alors un shell père et un shell fils
3. grâce à la fonction exec, le shell fils ***recouvre*** son segment de texte par celui de ex qui s'exécute.
4. le shell père récupère le pid du fils retourné par fork ().

5. à la fin de l'exécution de ex, le shell père reprend son exécution.

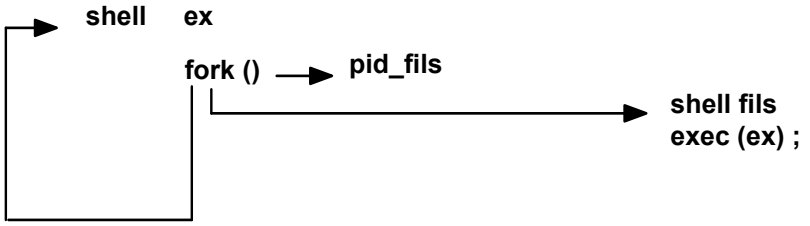
La fonction wait lui permet de connaître son achèvement : wait (&etat) retourne le pid du processus fils à son achèvement (etat : entier).



2ème cas : on tape **$ ex&** pour que ex s'exécute en arrière-plan (en parallèle au shell, mais n'accède pas à l'entrée standard réservée au shell). Voici ce qui se passe :

1. à 4. comme ci-dessus

5. le shell père reçoit le pid du fils, émet un prompt, puis reprend son activité sans attendre la fin du fils.



Attention, *un processus lancé en arrière-plan ne réagit pas aux interruptions émises au clavier.*

Les variables d'un processus shell ne sont pas transmises à son fils, à l'exception des variables d'environnement. En particulier, login transmet les variables d'environnement à ses fils. Si l'on veut que le fils d'un processus shell reconnaisse la variable x de son père, on exécutera dans le père la commande

export x

Si l'on veut retirer cette variable de l'environnement accessible au fils, on exécutera :

unset x

Il n'existe aucun moyen d'exporter une variable d'un sous-shell vers son père.

Exemple :

```
$ x=bonjour
$ export x
$ ksh                 # création d'un nouveau sous-shell
$ echo $x
bonjour
$ x='au revoir'
$ <CTRL-D>            # retour au shell de départ
$ echo $x
bonjour
```

La commande **nohup** (*no hangup*, pas d'arrêt imprévu) empêche l'arrêt d'un processus, même si l'utilisateur se déconnecte.

stdout et stderr sont redirigés par défaut sur le fichier **nohup.out**

Exemple :

```
$ nohup sort oldfic > newfic &
550
```

$

3. PROGRAMMATION DE PROCESSUS AVEC C

Les fonctions UNIX citées ci-dessus sont toutes accessibles depuis C.

3.1 fork ()

fork engendre un fils **partageant** le segment de textes du père, s'il est partageable, et disposant d'une **copie** de son segment de données. Il retourne 0 au fils et le pid du fils au père si la création a réussi (-1 sinon). Chaque processus a accès à la description du *terminal de contrôle* (terminal où l'on s'est connecté) contenue dans **/dev/tty**. Un processus fils hérite du terminal de contrôle du père. Le fils hérite d'une **copie** des descripteurs de fichiers ouverts. Les pointeurs sur fichiers sont partagés. Père et fils ont leurs propres tampons d'accès aux fichiers, dupliqués initialement par fork (). A l'exécution d'un fork (), le fils ferme les E/S standards si elles sont redirigées, puis il ouvre les fichiers de redirection. Il hérite donc des E/S redirigées.

Un processus fils n'hérite pas du temps d'exécution du père (initialisé à 0 pour le fils). Il n'hérite pas de la priorité du père, sauf de son paramètre nice.Il n'hérite pas des verrous sur les fichiers détenus par le père. Il n'hérite pas des signaux en suspens (signaux envoyés au père, mais pas encore pris en compte).

Tout processus, sauf le processus de pid 0, est créé par fork.

3.2 exit ()

void exit (int etat)

La fonction provoque la terminaison du processus avec le code de retour **etat** (0 = bonne fin). Si le père est un shell, il récupère **etat** dans la variable **$?**

A l'exécution de **exit**, tous les fils du processus sont rattachés au processus de pid 1.**exit** réalise la libération des ressources allouées au processus et notamment ferme tous les fichiers ouverts. Si le père est en attente sur **wait**, il est réveillé et reçoit le code de retour du fils.

3.3 wait ()

int wait (int *p_etat)

- suspend le processus jusqu'à ce qu'un de ses fils prenne fin. Si le fils se termine normalement, et que le père ne fait pas **wait** pour l'attendre, on dit que le fils est devenu un **processus zombie**. Il n'est plus pris en compte par l'ordonnanceur, son compteur d'alarme est annulé, mais son entrée dans la table des processus n'est pas enlevée. Elle est signalée par la mention **<defunct>**

- retourne le pid du fils à son achèvement, ou -1 s'il n'y a plus de fils actif. Si un ou plusieurs fils sont zombis, c'est le pid de l'un de ces fils qui est retourné. L'algorithme de wait privilégie donc la suppression des zombies.

Si le père s'achève sans attendre la fin d'un fils par **wait**, le fils sera pris en charge par **init**, le processus système de pid 1.
Si **p_etat** est non nul, ***p_etat** fournit des informations sur le type de terminaison du processus fils :

- si le processus fils s'est achevé normalement par un appel à **exit,** l'avant-dernier octet de ***p_etat** contient la valeur de l'argument de **exit** (l'octet de faible poids est à zéro)

- sinon, le processus fils s'est terminé par suite de la réception d'un **signal** (voir paragraphe 6). Alors, les bits 0 à 6 donnent le numéro du signal (0 à 127) et les octets de fort poids sont à zéro. Si le bit 7 est à 1 (valeur de l'octet de faible poids : 128 à 255), il y a eu un fichier **core** créé. On peut récupérer le numéro du signal en enlevant 128 au contenu de l'octet de faible poids.

3.4 La famille des fonctions de recouvrement (exec)

Il s'agit d'une famille de 6 fonctions permettant le lancement de l'exécution d'un nouveau programme par un processus. Le segment de texte est remplacé par celui du nouveau programme. Il y a recouvrement en mémoire, **mais pas de création d'un nouveau processus**. Le retour est -1 en cas d'erreur.

Les 6 fonctions sont **execl, execv, execle, execve, execlp, execvp**. Par exemple :

int execv (char *nom, char * argv [])

nom : pointeur sur le nom du programme à exécuter,
argv : liste de pointeurs (terminée par NULL) sur les arguments transmis au nouveau programme avec argv[0] : "nom" du programme pointé par nom

Un excellent exemple est donné dans Rifflet p. 251. Attention aux fichiers : seuls les descripteurs des fichiers ouverts (situés en zône système) sont conservés par un appel à exec. Les modes d'ouverture, les structures FILE associées sont recouverts, ainsi que les tampons d'accès. Il est donc recommandé d'appeler **fflush** avant exec pour vider les tampons dans les fichiers, y compris pour **stdout**. Un très bon exemple en est donné dans Rifflet p. 253.

Autre exemple intéressant : Braquelaire p. 448-449

3.5 Exemples

Exemple 1 :

```
main ()
{
   int pid_fils;
   bloc1
   if ((pid_fils = fork()) == 0)
       bloc2
   else
       bloc3
}
```

Le processus père exécutera le bloc 1. Puis il créera un fils identique par **fork** et exécutera le bloc 3 puisque **fork** lui retournera une valeur non nulle.

Le processus fils n'exécutera pas le bloc 1 car à sa création, son compteur ordinal pointera sur la ligne contenant **fork**. Comme **fork** lui retourne 0, il exécutera le seul bloc2. L'affichage des résultats

des blocs 2 et 3 peut être entrelacé, et pas nécessairement de façon identique d'une exécution à une autre.

Exemple 2 :

```
main ()
 {
   int m,n;
   printf ("processus père.Fils non encore créé\n");
   if (fork () == 0)
   {
        printf ("pid du fils %d\n", getpid());
             exit (1);
   }
   else {
        printf ("pid du père %d\n", getpid());
             m = wait (&n);
             printf ("fin du processus de pid %d avec valeur de retour de wait
%d\n",m,n);
    }
}
```

3.6 Les événements ou signaux

Le fichier **/usr/include/signal.h** contient notamment la définition de NSIG = 32 constantes caractérisant des événements, dont plusieurs génèrent un fichier **core** (copie de l'image mémoire sur disque; ils sont suivis d'un * dans le tableau ci-dessous) :

- interruptions matérielles (frappe d'un caractère,...)
- interruptions logicielles externes (terminaison d'un autre processus,...) ou internes(erreur arithmétique, violation mémoire,...).

NOM	NUMERO	ROLE
SIGHUP	1	émis à tous les processus associés à un terminal ou un modem quand il déconnecte
SIGINT	2	émis à tous les processus associés à un terminal lorsque <INTR> (par défaut <DEL> ou CTRL-C) est frappé au clavier
SIGQUIT	3*	émis à tous les processus associés à un terminal lorsque <QUIT> (par défaut CTRL- \ ou CTRL-Z) est frappé au clavier
SIGILL	4*	instruction illégale
SIGTRAP	5*	émis après chaque instruction en cas d'exécution en mode trace, par un débogueur
SIGABR	6*	abort
SIGEMT	7	piège d'instruction débogueur
SIGFPE	8*	erreur dans une instruction en virgule flottante
SIGKILL	9	arrêt obligatoire du processus (ne peut être ni ignoré, ni capturé)
SIGBUS	10*	erreur d'adressage sur le bus
SIGSEGV	11*	violation des limites de l'espace mémoire
SIGSYS	12*	mauvais argument dans un appel système
SIGPIPE	13	écriture dans un tube sans lecteur
SIGALRM	14	signal associé à une horloge (cf. ci-dessous fonction système alarm).
SIGTERM	15	signal de terminaison normale d'un processus
SIGUSR1	16	à la disposition des utilisateurs pour la communication entre processus
SIGUSR2	17	idem au précédent
SIGCHLD	18	mort d'un fils provoquée par exit
SIGPWR	19	panne d'alimentation électrique
SIGWINCH	20	changement de taille de fenêtre
SIGURG	21	message socket urgent

SIGPOLL	22	événement stream pollable (cf. sockets)
SIGSTOP	23	signal stop envoyé
SIGSTP	24	stop par l'utilisateur
SIGCONT	25	continuation
SIGTTIN	26	stop sur l'entrée du terminal
SIGTOU	27	stop sur la sortie du terminal
SIGVTALRM	28	expiration d'un temporisateur virtuel
SIGPROF	29	temporisateur de profile expiré
SIGXCPU	30*	temps maximal CPU expiré
SIGXFSZ	31*	taille maximum du fichier dépassée

Les signaux 18 à 21, ainsi que le signal 25, sont ignorés par défaut. Les autres provoquent l'arrêt du processus, avec éventuellement la création d'un fichier core.
Les signaux 4, 6, 7, 10 et 11 sont bien décrits dans BRAQUELAIRE, p. 434.

La fonction **signal**, prototypée par :

void (* signal (int sig, void (*fonc)(int))) (int)

permet d"intercepter le signal **sig** et d'exécuter la fonction (ou **handler**) pointée par **fonc**.
Le processus reprend son exécution à la fin de l'exécution de **fonc**.

fonc peut prendre des valeurs prédéfinies (macros) :

SIG_IGN : signal ignoré, sauf SIGKILL
SIG_DFL : fin du processus, avec ou sans fichier core selon **sig**

Exemple :

```
#include <signal.h>
        main ()
        {       signal (SIGINT,SIG_IGN);
        signal (SIGQUIT,SIG_IGN);
 /* à partir d'ici, les interruptions SIGINT (par défaut CTRL-C) et SIGQUIT (par défaut
CTRL- \) seront désarmées   */
                .....
                signal (SIGINT, SIG_DFL);
              signal (SIGQUIT,SIG_DFL);
  /* à partir d'ici, SIGINT ou SIGQUIT mettront fin au processus */
                .....
```

Lorsque **fonc** est un pointeur sur une fonction utilisateur, 3 actions se produisent à l'arrivée de **sig** :

- le programme courant du processus est interrompu
- le processus est dérouté vers la fonction pointée par **fonc**, *avec en argument le n° du signal*
- après exécution de cette fonction, le programme courant du processus reprend son exécution, sauf si la fonction a réalisé un exit

Lorsqu'un processus est lancé en arrière-plan, les commandes signal (SIGINT,SIG_IGN) et signal (SIGQUIT,SIG_IGN) sont lancées et empêchent son interruption par les moyens classiques (CTRL-C ou CTRL- \). Il faut recourir à kill -9 pour l'interrompre.

Un processus fils hérite du comportement de son père vis à vis des signaux.

La fonction prototypée par : **unsigned alarm (unsigned s)**, et décrite dans <unistd.h>, envoie SIGALRM au processus après s secondes (s <= MAX_ALARM définie dans <sys/param.h>). Une telle demande annule une demande antérieure du même type. Si s = 0, on annule simplement la demande antérieure. Après un *fork*, l'horloge du fils est mise à 0, quelque soit celle du père. Ce n'est pas le cas après un *exec*.

alarm retourne le temps restant avant l'envoi du signal correspondant à cette demande antérieure.

Un processus peut envoyer un signal **s** à un processus de pid **p**, à condition qu'ils aient tous deux le même propriétaire, grâce à la fonction **kill** prototypée par :

int kill (int p, int s)

kill retourne 0 si l'opération s'est bien déroulée, -1 sinon.
La valeur **s** = 0 permet de tester si le processus de pid **p** existe.

Il existe évidemment la commande système **kill** bénéficiant des mêmes propriétés.
Exemple : kill -9 12356 envoie le signal 9 au processus de pid 12356

La commande shell **trap** permet d'exécuter un processus de déroutement à la réception d'un signal donné.

Exemples : trap 'rm toto.c;exit' 3 4
a pour effet de demander l'exécution de la séquence rm toto;exit à la réception des signaux 3 ou 4 (exit est bien sûr nécessaire pour assurer la fin du processus).

trap " 3 4 permettrait d'ignorer les signaux 3 et 4

La fonction **int pause (void)** décrite dans <unistd.h> met le processus en attente de l'arrivée du premier événement non masqué; pause retourne -1 si l'événement provoque le déroutement sur une autre fonction.

De bons exemples faisant intervenir signal, alarm, wait et pause sont donnés dans BRAQUELAIRE, p. 435-440, 461-464 et surtout 440-443.

A noter aussi les fonctions **int sigsetjmp et int siglongjmp**, décrites dans <setjmp.h>, qui respectivement sauve le contexte d'un processus et restaure ce contexte.

BIBLIOGRAPHIE

J.-P.BRAQUELAIRE, Méthodologie de la programmation en Langage C, Masson,1993

A.B. FONTAINE et Ph. HAMMES, UNIX Système V, Masson, 1993

J.L. NEBUT, UNIX pour l'utilisateur, Technip, 1990

J.M. RIFFLET, La programmation sous UNIX, Mc Graw-Hill, 1992