UNIVERSITÉ CAEN NORMANDIE

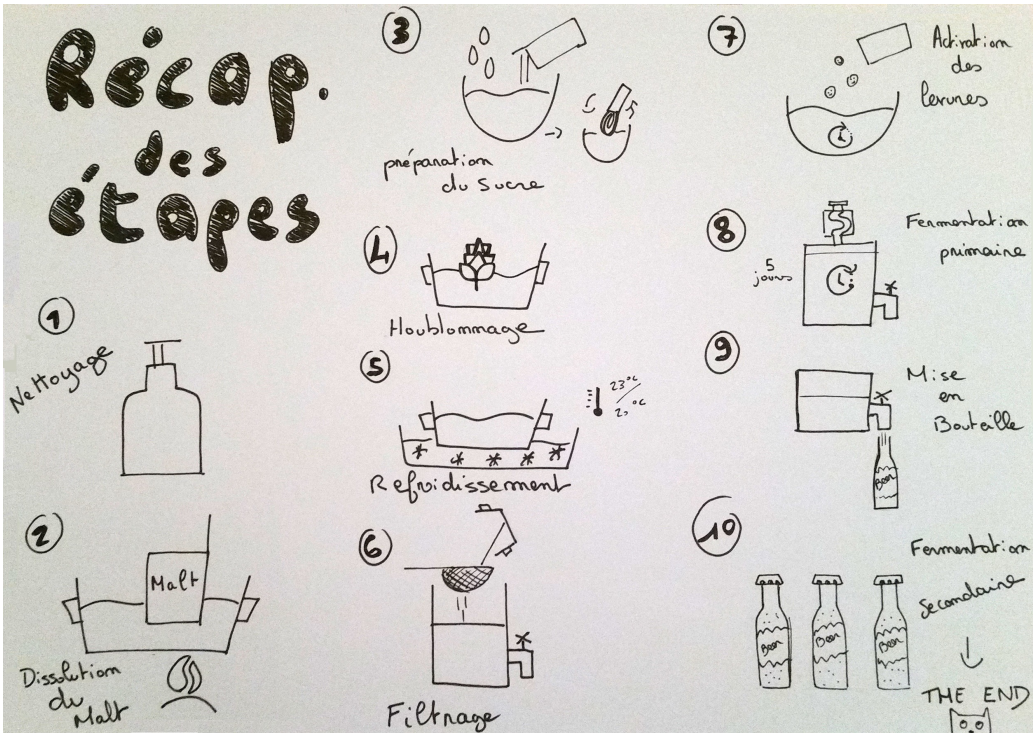
Les processus

22.10.2018

Le programme c'est la recette de la fabrication de la bière

le processus c'est fabriquer ce bon nectar et la boire entre ami(e)s, hum...☺

Auteur : Pascal Fougeray



1 Introduction

Dans de nombreuses applications, surtout celles faisant appel aux principes Client-Serveur[1], il est nécessaire de dupliquer le processus serveur, à chaque fois qu'un client désire se connecter[2]. On peut faire rapidement la manipulation en montant l'un des serveurs suivant Web, Telnet ou FTP... sur une machine et à chaque fois qu'un client se connecte, on lance la commande ps et l'on constate qu'un nouveau processus de même nom est lancé et occupe une place en mémoire.

Pour certains serveurs, tel le serveur Web *apache*, même si personne n'est connecté il y a dès le début plusieurs processus *httpd* en attente de connexion. Nous aurons l'occasion de développer tout cela lors du cours sur la Communication par *Socket*.

2 Rappels

— ***User Mode*** vs ***Kernel Mode*** et les espaces mémoire
Un système d'exploitation est découpé en 2 parties : le **noyau** et le reste donc nous les **utilisateurs** ! ! !
Il y a donc 2 modes de fonctionnement le mode noyau et le mode et bien pas noyau le mode des utilisateurs

1. Voir le cours sur la communication par socket
2. Nous aurons de voir cela lors du cours sur La communication par *Socket*.

Le noyau fait l'interface entre l'utilisateur et le matériel
Si on parle d'espaces mémoire, cela veut dire qu'il y a 2 espaces mémoire celui du noyau et celui des utilisateurs qui font
- 1Go et 3Go sur un vieux système 32 bits soit 25% pour le kernel
- 512Go (549 755 813 888) et beaucoup beaucoup sur un système 64 bits 2^64 - 2^39 = 18 446 744 073 709 551 616 - 549 755 813 888 je vous laisse le faire ?
  18 446 743 523 953 737 728 soit 0,000000000000056% pour le kernel

— **Définition** : Un processus fournit à un instant t l'image de l'état d'avancement de l'exécution d'un programme.
Le programme est le produit d'une compilation donnant un objet inerte correspondant au contenu d'un fichier.

— **Constitution** : Un processus est constitué du programme qu'il est en train d'exécuter, de l'ensemble des données que ce même programme manipule et d'un ensemble d'informations dont le SE à besoin pour prendre en compte ce processus.
Tout cela constitue le contexte d'exécution :
- les valeurs des registres du processeur correspondant à l'état d'avancement de l'exécution,
- pile d'exécution,
- liens avec l'utilisateur,
- le système E/S etc...

— **pid** *processus identificator* : un processus possède un identificateur qui lui est propre et qui est une valeur comprise entre 1 et 32767, le processus père de tous les autres est le processus 1, ***Init***. Ce n'est plus toujours vrai depuis **SystemD** ! ! ! voir plus loin ! ! !
Pour connaitre la valeur max de processus, donc pid : ***sysctl kernel.pid_max*** (attention avant c'était sysctl kern.pid_max manque le el au kernel ! ! !)
Elle doit vous renvoyer la valeur : **kernel.pid_max = 32768**
Cette valeur est trouvée dans le répertoire ***/proc/sys/kernel*** le fichier ***pid_max***

— **Affichage des processus** : La commande ***ps*** affiche la liste des processus en cours d'exécution, ceux exécutés par l'utilisateur lui même, ceux exécutés par les autres utilisateurs et tous ceux du système. Exemple : ***ps -auf, ps jf, ps -edfl***

```
man ps
[...]
PROCESS STATE CODES
Here are the different values that the s, stat and state output specifiers
(header "STAT" or "S") will display to describe the state of a process.
D    Uninterruptible sleep (usually IO)
R    Running or runnable (on run queue)
S    Interruptible sleep (waiting for an event to complete)
T    Stopped, either by a job control signal or because it is being traced.
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    Defunct ("zombie") process, terminated but not reaped by its parent.
```

— **État d'un processus** : Un processus peut être dans différents états,

D En sommeil non interruptible (À un processus résident quelque part dans le système qui n'arrive pas à effectuer une tâche ou une autre. Typiquement un ***tar*** très long. Au passage, c'est un truc bien sympathique parce que ça bloque à la fois le processus mais aussi le périphérique. Le processus est typiquement en train d'effectuer une tâche non interruptible qui ne peut pas se faire en raison d'une erreur ou d'une autre)

R En cours d'exécution (le ***process*** est actif et consomme des ressources, en ***user mode*** ou ***kernel mode***)

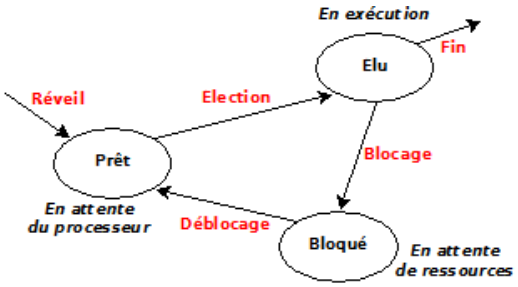
S En sommeil (le process n'est pas actif mais susceptible d'être réveillé par un appel système

T Stoppé ou stracé (Le processus a reçu un signal d'arrêt temporaire et attend un ***SIGCONT***)

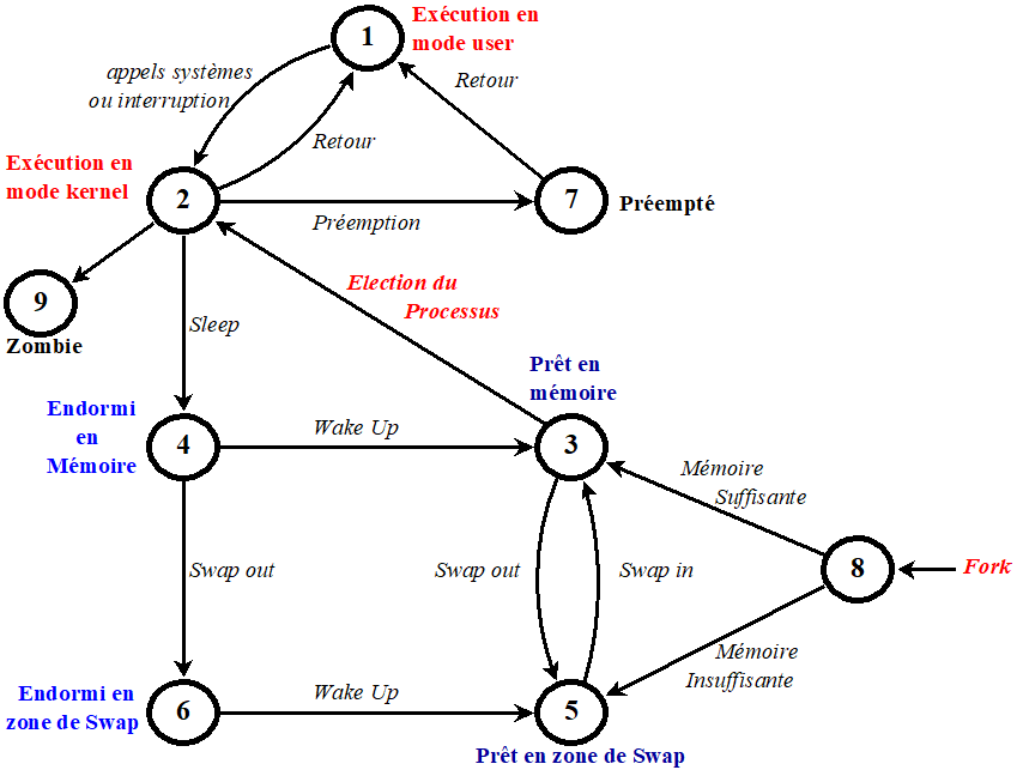
Z Zombie (l'état 'Z' correspond à un fils dont le père n'est pas encore allé à l'enterrement. En d'autres termes, la valeur de retour du processus n'a pas été lu par le père (ou par ***init*** dans le cas où le père n'existe plus il est oprhelin). Ce processus ne consomme plus que la place de la structure de description du processus à l'exclusion de toute autre ressource)

X **mort** pas besoin de l'expliquer... et de toute manière vous ne pouvez pas le voir ☺

Pour rire un peu comme dans cet Amphi, un seul Processus est R (Le prof) tous les autres sont S (Vous !)



— L'**ordonnanceur** ou *schbeduler* un **composant du noyau** du SE choisissant l'ordre d'exécution des processus sur les processeurs d'un ordinateur. Il permet à tous les processus de s'exécuter à un moment ou un autre et d'utiliser au mieux le processeur pour l'utilisateur. Pour que chaque tâche s'exécute sans se préoccuper des autres et/ou aussi pour exécuter les tâches selon les contraintes imposées au système, l'ordonnanceur du noyau du SE effectue des **commutations de contexte** de celui-ci.
Les priorités des processus créent un phénomène appelé **"ordonnancement"** lié à la gestion de la mémoire.



— le **quantum**. Le temps de processeur total disponible divisé en petites plages.
Un processus ne peut monopoliser le processeur que le temps d'un quantum.
Définir la durée d'un quantum de temps est critique pour les performances du système.
Un quantum de temps trop faible risque de monopoliser les ressources pour l'ordonnancement.
Un quantum trop long dégrade la maniabilité du système.
La durée d'un quantum est donc un choix difficile. Linux utilise des tranches de temps d'environ **50ms**, mais la "formule" généralement utilisée est ***"Choose a duration as long as possible, while keeping good system response time*** "
— **préemptivitée** : le ***scheduler*** peut interrompre l'exécution d'un processus non-noyau pour donner le processeur à un autre processus, et ce même au milieu d'un quantum.
— **priorité** : chaque processus possède une priorité comprise entre -20 et +19. Plus la priorité est basse plus le ***scheduler*** privilégiera ce processus.

— **Le commutation de contexte** : Sur un ordinateur, vous n'avez qu'un seul µP qui est constitué de registres. Ces registres contiennent des données représentant le processus en cours d'exécution (la taille mémoire utilisée, le compteur ordinal...).
Quand le système décide de passer un processus P1 à un processus P2, il faut faire un changement

de contexte, donc sauvegarder le contexte du processus P1 et récupérer le contexte du processus P2. Ceci à un cout au niveau des temps d'exécution. Oui oui un système multitâches est plus lent qu'un système monotâche !

— **Le contexte d'un processus** : l'ensemble des données qui permettent de **reprendre l'exécution** d'un processus interrompu.
Le contexte d'un processus est l'ensemble de

1. son état
2. son mot d'état : en particulier
   (a) La valeur des registres actifs
   (b) Le compteur ordinal
3. les valeurs des variables globales statiques ou dynamiques
4. son entrée dans la table des processus
5. sa zone u (u pour user)
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.

— **La table des Processus et zone u** : Tous les processus sont associés à une entrée dans la table des processus appartenant 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 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.

— **Espace d'adressage virtuel d'un processus**
Le fait qui consiste à attribuer une zone mémoire pour chaque donnée du programme source s'appelle le **processus d'allocation mémoire**.
Le **compilateur** doit être capable de **substituer chaque référence à une variable par son adresse**. Cela s'appelle le processus de substitution. Par exemple la référence à un élément de tableau, champs de structure, fonction virtuelle, etc.
L'espace d'adressage virtuel d'un processus est subdivisé en régions qui sont Code, Data static, Datas dynamiques, les arguments, Pile et le Tas.

Zone u	Code ou texte	static	globale	Pile	→ *dynamiques* ←	Tas	les arguments

— Zone u : Informations sur le processus
— Code ou Texte : Instructions
— Les zones contenant le code, les données statiques et globales ont des tailles connues
— Le **tas** et la **pile** s'accroissent et rétrécissent durant l'exécution du programme. 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. ESP-EBP sur x86.

3 Les processus sous Linux /PROC

Le système de fichiers ***proc*** est un ***pseudosystème*** de fichiers fournissant une interface avec **les structures de données du noyau**. Il est généralement monté (je préfère lié ! ! !) sur le répertoire ***/proc***. La plupart des fichiers sont en lecture seule, normal ils appartiennent au noyau et même le root n'y a pas accès ! ! !

Mais quelques uns (fichiers) permettent la modification de variables du noyau. Comme nous avons fait un peu de réseau et que je vous ai parlé d'IPv6... si vous voulez que votre noyau ne parle pas l'IPv6 et bien il suffit de lancer les 4 commandes suivantes :

1. Désactiver ipv6 pour toutes les interfaces : ***sysctl -w net.ipv6.conf.all.disable_ipv6=1***
2. Désactiver l'auto configuration pour toutes les interfaces : ***sysctl -w net.ipv6.conf.all.autoconf=0***
3. Désactiver ipv6 de la configuration par défaut : ***sysctl -w net.ipv6.conf.default.disable_ipv6=1***
4. Désactiver l'auto configuration par défaut : ***sysctl -w net.ipv6.conf.default.autoconf=0***
   Ceci à chaud et si on veut que cela soit définitif et bien il suffit d'écrire ces instructions en dur dans le fichier ***/etc/sysctl.conf*** et de lancer la commande ***sysctl -p***

Rien ne vous empêche de le faire pour Ipv4 et de désactiver la réponse au **ping** vous savez la couche 3 du modèle OSI qui soit disant ne marche plus alors que les couches 7 telles http sont accessibles ☺

sysctl -w net.ipv4.icmp_echo_ignore_all=1 renvoie net.ipv4.icmp_echo_ignore_all = 1

Puis un ping localhost renvoie rien ☺

Bon pourquoi cette partie, tout cela pour vous persuader que le répertoire /PROC n'est pas que le répertoire de Processus mais un image de tout ce que connait le noyau ! ! !

Et comme une image ça ne prend pas de place et bien la **taille** occupée dans le SGF par ce répertoire /proc est **NULLE** ! ! ! !

La preuve un du -h renvoie que des 0 ! ! !

Autre preuve et TP à faire ! ! ! Éteindre et aller voir ce qu'il y a dans dans ce répertoire et bien il n'y a rien. Comment faire cela et bien c'est comme pour voir si la lumière est éteinte dans le réfrigérateur quand la porte est fermé. Je vous laisse trouver la solution qui existe ! ! !

et il est lié à

 # mount | grep proc

proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)

Bon revenons au répertoire /proc et ce qui nous intéresse des processus. Faisons un ***ls*** de son contenu, nous obtenons en "*gros*"

```
root@debian95-Rx-Sys-Fougeray:/proc# ls
1     107  116  16    2     2487  2671  2989  334  347  384  466  540  843  875  971    buddyinfo  devices      fs          keys         locks       pagetypeinfo  stat          tty
10    108  12   17    21    2488  2677  2992  341  349  385  468  7    844  9    979    bus        diskstats    interrupts  key-users    meminfo     partitions    swaps         uptime
1000  109  13   18    2120  25    27    3     342  362  41   472  75   848  902  982    cgroups    dma          iomem       kmsg         misc        sched_debug   sys           version
103   11   14   180   213   2555  28    317   343  380  419  476  784  850  926  985    cmdline    driver       ioports     kpagecgroup  modules     schedstat     sysrq-trigger vmallocinfo
104   111  15   187   22    2558  2869  319   344  381  42   481  8    855  927  994    consoles   execdomains  irq         kpagecount   mounts      self          sysvipc       vmstat
105   112  154  19    23    2590  29    322   345  382  423  5    83   857  968  acpi   cpuinfo    fb           kallsyms    kpageflags   mtrr        slabinfo      thread-self   zoneinfo
106   113  155  1951  24    26    2902  333   346  383  434  500  840  871  970  asound crypto     filesystems  kcore       loadavg      net         softirqs      timer_list
root@debian95-Rx-Sys-Fougeray:/proc#
```

Tous les beaux chiffres et nombres en bleu sont des répertoires et la valeur représente les PID des processus en cours.

Ces répertoires sont appelés **répertoires de processus** car ils font référence à un **ID** de processus et contiennent des informations se rapportant à ce dernier. Le propriétaire et le groupe de chaque répertoire de processus prennent la valeur de l'utilisateur qui exécute le processus.

Lorsque le processus est terminé, son répertoire de processus /proc/ disparaît.

Chaque répertoire de processus contient (liste non exhaustive ! !) les lignes suivantes :

cmdline Contient la **commande** émise au début du processus.

cwd Représente un lien symbolique vers le répertoire de travail courant pour ce processus.

environ Fournit une liste des **variables d'environnement** du processus. La variable d'environnement est indiquée en majuscule et la valeur en minuscule.

exe Représente un lien symbolique vers le fichier exécutable de ce processus.

fd Représente un répertoire contenant tous les **descripteurs de fichiers** pour un processus donné. Ces derniers sont fournis sous forme de liens numérotés :

maps Contient une liste des topologies de mémoire vers les divers fichiers exécutables et les fichiers bibliothèques associés à ce processus.

Selon la complexité du processus, ce fichier peut être relativement long.

mem Représente la mémoire occupée par le processus. Ce fichier ne peut pas être lu par l'utilisateur ! ! ! Tiens pourquoi ?

cat : mem : Erreur d'entrée/sortie

root Représente un lien vers le répertoire ***root*** du processus.

stat Montre l'état du processus.

statm Montre l'état de la mémoire utilisée par le processus.

status Montre l'état du processus sous une forme plus lisible que ***stat*** ou ***statm***.

self est un lien symbolique vers le répertoire de /proc correspondant au processus courant. La destination du lien /proc/self dépend du processus qui l'utilise : chaque processus voit son propre répertoire comme destination du lien.

4 Démarrage d'un Linux Moderne !

Vous avez surement dû apprendre que le premier processus lancé par sa majesté le ***kernel*** lui même est le fameux processus ***Init*** de PID 1, c'est vrai, mais depuis quelques années cela a beaucoup beaucoup changé. Les microprocesseurs étant multi-coeurs etc... Linux a évolué et le noyau ne lance plus un seul processus mais 2, toujours init et en plus le processus ***kthreadd*** qui est un processus démon ou daemon comme vous voulez

Lancez la commande ***ps -alx*** et vous obtenez :

```
root@debian95-Rx-Sys-Fougeray:~# ps alx
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0     1     0  20   0 138892  6716 SyS_ep Ss   ?          0:00 /sbin/init
1     0     2     0  20   0      0     0 -      S    ?          0:00 [kthreadd]
1     0     3     2  20   0      0     0 -      S    ?          0:00 [ksoftirqd/0]
1     0     5     2   0 -20      0     0 -      S<   ?          0:00 [kworker/0:0H]
1     0     7     2  20   0      0     0 rcu_gp S    ?          0:00 [rcu_sched]
1     0     8     2  20   0      0     0 -      S    ?          0:00 [rcu_bh]
1     0     9     2 -100  -      0     0 -      S    ?          0:00 [migration/0]
1     0    10     2   0 -20      0     0 -      S<   ?          0:00 [lru-add-drain]

1     0   116     2  20   0      0     0 -      S    ?          0:00 [kworker/0:3]
1     0   147     2  20   0      0     0 -      S    ?          0:00 [jbd2/sda1-8]
1     0   148     2   0 -20      0     0 -      S<   ?          0:00 [ext4-rsv-conver]
4     0   174     1  20   0  40468  4584 SyS_ep Ss   ?          0:00 /lib/systemd/systemd-journa
1     0   180     2  20   0      0     0 -      S    ?          0:00 [kauditd]
4     0   198     1  20   0  45776  3484 SyS_ep Ss   ?          0:00 /lib/systemd/systemd-udevd
4     0   306     1  20   0  29960  2816 -      Ss   ?          0:00 /usr/sbin/cron -f
4   108   307     1  20   0  47016  3212 SyS_po Ss   ?          0:00 avahi-daemon: running [debi
4     0   308     1  20   0  46496  4800 SyS_ep Ss   ?          0:00 /lib/systemd/systemd-logind
```

Lancés par le kernel !!!

Tout plein de daemons !!!

A partir d'ici c'est le processus Init qui gère !!!

Lancez la commande ***ps -axjf*** et vous obtenez quelque chose un peu identique mais sous une forme. Vous pouvez constater une arborescence avec 2 racines, celle des processus ayant pour père ***kthreadd*** et la seconde

ayant pour père ***init***

```
root@debian95-Rx-Sys-Fougeray:~# ps axjf
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     2     0     0 ?           -1 S        0   0:00 [kthreadd]
    2     3     0     0 ?           -1 S        0   0:00  \_ [ksoftirqd/0]
    2     4     0     0 ?           -1 S        0   0:00  \_ [kworker/0:0]
    2     5     0     0 ?           -1 S<       0   0:00  \_ [kworker/0:0H]
    2     7     0     0 ?           -1 S        0   0:00  \_ [rcu_sched]
    2     8     0     0 ?           -1 S        0   0:00  \_ [rcu_bh]
    2     9     0     0 ?           -1 S        0   0:00  \_ [migration/0]
    2    10     0     0 ?           -1 S<       0   0:00  \_ [lru-add-drain]
    2    11     0     0 ?           -1 S        0   0:00  \_ [watchdog/0]
    2   184     0     0 ?           -1 S        0   0:00  \_ [kworker/0:4]
    0     1     1     1 ?           -1 Ss       0   0:00 /sbin/init
    1   174   174   174 ?           -1 Ss       0   0:00 /lib/systemd/systemd-journald
    1   198   198   198 ?           -1 Ss       0   0:00 /lib/systemd/systemd-udevd
    1   306   306   306 ?           -1 Ss       0   0:00 /usr/sbin/cron -f
    1   307   307   307 ?           -1 Ss     108   0:00 avahi-daemon: running [debian95-Rx-Sys-
  307   332   307   307 ?           -1 S      108   0:00  \_ avahi-daemon: chroot helper
    1   308   308   308 ?           -1 Ss       0   0:00 /lib/systemd/systemd-logind
    1   309   309   309 ?           -1 Ss     107   0:00 /usr/bin/dbus-daemon --system --address
    1   323   323   323 ?           -1 Ssl      0   0:00 /usr/sbin/rsyslogd -n
    1   324   324   324 ?           -1 Ss       0   0:00 /usr/sbin/anacron -dsq
    1   367   367   367 tty1       510 Ss       0   0:00 /bin/login --
  367   510   510   367 tty1       510 S+       0   0:00  \_ -bash
    1   373   373   373 ?           -1 Ss       0   0:00 /usr/sbin/sshd -D
  373   541   541   541 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0,pts/1
  541   547   547   547 pts/0      735 Ss       0   0:00      \_ -bash
  547   735   735   547 pts/0      735 S+       0   0:00      |   \_ man ps
  735   748   735   547 pts/0      735 S+       0   0:00      |       \_ pager
  541   755   755   755 pts/1      763 Ss       0   0:00      \_ -bash
  755   763   763   755 pts/1      763 R+       0   0:00          \_ ps axjf
```

Une autre commande qui appartient au paquet ***psmisc*** contenant les utilitaires qui utilisent le système de fichiers virtuels ***/proc***

```
root@debian95-Rx-Sys-Fougeray:~# pstree
systemd─┬─anacron
        ├─apache2───2*[apache2───26*[{apache2}]]
        ├─avahi-daemon───avahi-daemon
        ├─cron
        ├─cups-browsed─┬─{gdbus}
        │              └─{gmain}
        ├─cupsd───2*[dbus]
        ├─dbus-daemon
        ├─dhclient
        ├─login───bash
        ├─rsyslogd─┬─{in:imklog}
        │          ├─{in:imuxsock}
        │          └─{rs:main Q:Reg}
        ├─sshd───sshd─┬─bash───pstree
        │             └─bash
        ├─systemd───(sd-pam)
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-timesyn───{sd-resolve}
        └─systemd-udevd
root@debian95-Rx-Sys-Fougeray:~#
```

— ***fuser*** : identifie les processus utilisant des fichiers ou sockets.
— ***killall*** : tue les processus par leur nom (ex ń killall -HUP named ż).
— ***peekfd*** : affiche les données passant par un descripteur de fichier.
— ***pstree*** : affiche un arbre des processus actifs.
— ***prtstat*** : imprime le contenu de /proc/<pid>/stat
Voilà voilà pour un peu de MAJ de vos connaissances ☺

5 Lancer un nouveau processus

Il est possible de générer l'exécution d'un programme à partir d'un autre et de créer ainsi un nouveau processus, en utilisant la fonction de bibliothèque **system**().

#include <stdlib.h>
int system (const char *chaîne) ;

La primitive **system** exécute la commande lui ayant été transmise sous la forme d'un chaîne et attend qu'elle s'achève. Comme la fonction **system** utilise un shell pour lancer le programme, on peut la lancer en arrière plan en ajoutant l'esperluette à la fin de la commande. Le fonctionnement est le suivant :

Le programme systeme est lancé, il appelle **system** et doit attendre la fin du processus lancé via l'appel à **system**.

```
1  #include <stdlib.h>
   #include <stdio.h>

   int main() {
     int valeur_de_retour;
6    printf("Exécution de ps avec system\n");
     valeur_de_retour = system ("ps auf");
     printf("valeur de retour %d\n",
         valeur_de_retour);
     return valeur_de_retour;
   }
```

6 Créer un processus

Un *fork* ou une fourchette en anglais. En informatique on parle souvent de *fork*, par exemple libre office est un *fork d'Openoffice*, cela veut dire qu'il en est une copie ?

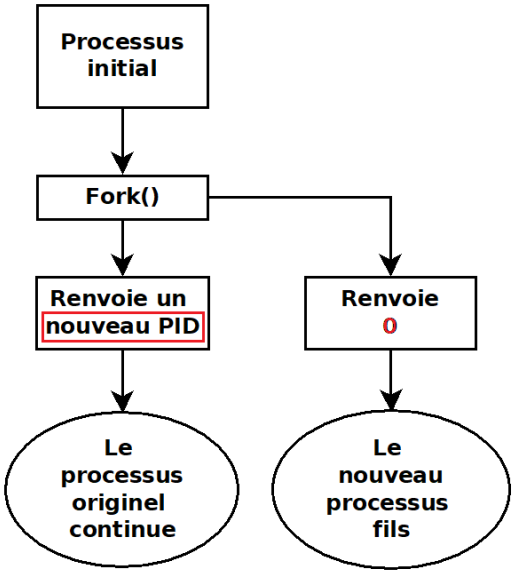
Autre exemple : ***MariaDB*** est un fork de ***MySQL*** d'Oracle
https://fr.wikipedia.org/wiki/Fork_(d%C3%A9veloppement_logiciel)
...

Donc : Un nouveau processus peut-être créé en appelant la primitive ***fork***.

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

Cet appel système copie le processus actif en insérant dans la table des processus une nouvelle entrée dotée des mêmes attributs, ce nouveau processus est quasiment identique à l'original, il exécute le même code mais avec ses propres espaces de données, environnement et descripteurs de fichiers.
L'appel *fork* dans le père renvoie le PID du processus fils. Celui-ci continue de s'exécuter comme l'original, à la différence que *fork* renvoie 0 dans le processus fils.

Cela permet de savoir qui est qui.



L'utilisation typique de l'appel **fork** est la suivante :

1. ***pid_t pid = fork();***
2. ***if pid (pid==0) {***
3. ***/* Instructions du fils */ }***
4. ***else if (pid >0) { // pid du fils <>0***
5. ***/* Instructions du père */ }***
6. ***else {***
7. ***/* Erreur (Instruction du père) */ }***

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

int main(){
  pid_t pid;
  char *message;
  int n;
  printf("Lancement du fork\n");
  pid=fork();
  switch(pid) {
    case -1:
      perror("echec de fork");
      return 1;
    case 0:
      message = "le fils";
      n=10;
      break;
    default:
      message = "le papa";
      n=3;
      break;
    }
  for(; n>0; n--){
    printf("%s, pid : %i et papa : %i \n
        ",\
       message,getpid(),getppid());
    sleep(1);
  }
  return 0;
}
```

Le primitive **fork** renvoie -1 en cas d'échec souvent en raison du nombre limité de processus fils qu'un père peut posséder (**CHILD_MAX**), un manque d'espace empêchant la création d'une entrée dans la table des processus ou une mémoire virtuelle insuffisante.

Dans l'exemple précédent, le programme s'exécute comme 2 processus, le père affiche 3 messages et le fils 8. Ce programme ne fonctionne pas correctement, pourquoi ?

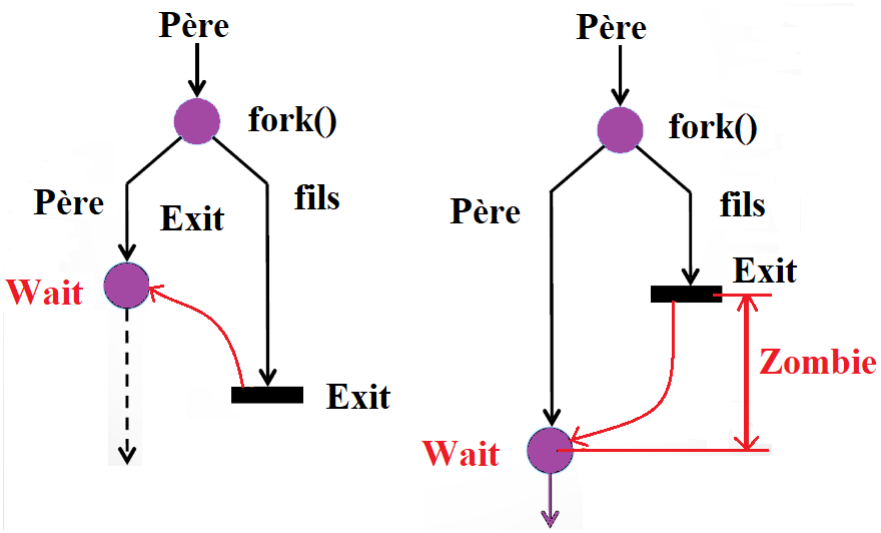
Réponse :

L'ordre d 'exécution entre le père et le fils est quelconque

7 Le zombie

Après nous verrons les processus démons qui n'ont rien à voir avec les processus zombies ☺.

Tout processus se terminant passe dans l'état zombie et y reste tant que son père n'a pas pris connaissance de sa terminaison.



Ceci est dû au faite que tout processus se terminant possède une valeur de retour à laquelle son père peut accéder.

Mais contrairement à la fonction appelante qui est bloquée par la fonction appelée, dans le cas des processus, le père et le fils peuvent se dérouler en parallèle. Le système fourni donc au père un moyen lui permettant d'accéder au code de retour du fils terminé. Il y a aussi l'envoie au père du signal ***SIGCHLD***[3]. Les seules informations maintenues dans le bloc de contrôle d'un processus zombie sont :

- Son code de retour ;
- Ses temps d'exécution dans les 2 modes (***kernel*** et ***user***) ; La commande ***time*** les donne !
- Son identité et celle de son père.

```
// Mon nom est : make-zombie !!!
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv){

  if (fork() == (pid_t)0){
    printf("fin du fils de %d\n",getpid());
    exit(0);}
  // On est dans le père qui dort 1h comme un étudiant en amphi ?
  sleep(3600);
  return 0;
}
```

Après avoir compilé ce programme et l'avoir lancé, listez les processus en cours d'exécution à l'aide de la commande ***ps -e -o pid, ppid,stat,cmd | grep zombie***

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus# ./make-zombie &
[1] 11149
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus# fin du fils de 11150

root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus# ps -e -o pid,ppid,stat,cmd | grep zombie
11149 11070 S    ./make-zombie
11150 11149 Z    [make-zombie] <defunct>
11152 11070 S+   grep zombie
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus#
```

Elle dresse la liste des identifiants de processus, de processus père, de leur statut et de la ligne de commande du processus.

Observez qu'en plus du processus père *make-zombie*, un autre processus *make-zombie* est affiché. Il s'agit du processus ***fils*** ; notez que identifiant de son père est celui du processus *make-zombie* principal. Le processus *fils* est marqué comme ***<defunct>*** et son code de statut est Z pour zombie.

Que se passe-t-il lorsque le programme principal de ***make-zombie*** se termine sans appeler ***wait*** ?

Le processus zombie est-il toujours présent ?

Non - relancez **ps** et notez que les 2 processus ***make-zombie*** ont disparu.

3. Voir le cours sur les signaux.

Lorsqu'un programme se termine, un processus spécial hérite de ses *fils*, le programme **init**[4], qui s'exécute toujours avec un identifiant de processus valant 1 (il s'agit du premier processus lancé lorsque Linux démarre).

Le processus *init* libère automatiquement les ressources de tout processus zombie dont il hérite

8 L'orphelin

Si on reprend le code précédent générant un processus Zombie mais que cette fois-ci c'est le père qui se termine et quitte sans attendre son fils, alors le fils devient un orphelin et il est "adopté" par le Processus INIT

```
// Mon nom est : make-orphelin !!!
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv){

  if (fork() == (pid_t)0){
    printf("le fils dort %d et son père est %d\n",getpid(), getppid());
    // On est dans le père qui dort 1h comme un étudiant en amphi ?
    sleep(15);
    printf("le fils se réveille %d et son père est %d\n",getpid(), getppid());
    sleep(5); // on le fait dormir 5s de plus pour voir !
    exit(0);}
  // le père ne fait rien et quitte sans attendre son fils !
  sleep(5); // pour voir que le père est bien le père au début !!!
  printf("fin du père de %d\n",getpid());
  return 0;
}
```

Après avoir compilé ce programme et l'avoir lancé, listez les processus en cours d'exécution à l'aide de la commande ***ps -e -o pid, ppid,stat,cmd | grep orphelin***

On obtient ceci

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus# ./make-orphelin &
[1] 11234
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus# le fils dort 11235 et son père est 11234

root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus#  ps -e -o pid,ppid,stat,cmd | grep orphelin
11234 11070 S    ./make-orphelin
11235 11234 S    ./make-orphelin
11237 11070 S+   grep orphelin
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus# fin du père de 11234

[1]+  Fini                    ./make-orphelin
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus#  ps -e -o pid,ppid,stat,cmd | grep orphelin
11235     1 S    ./make-orphelin
11239 11070 S+   grep orphelin
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus# le fils se réveille 11235 et son père est 1

root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/processus#
```

On peut constater qu'au début le père est bien le père enuite le père est le Processus de PID 1 soit Init !

9 Les Daemons

Après avoir étudié les processus zombies qui n'ont rien à voir avec les processus démons, étudions ces derniers☺.

Après son chargement, le ***kernel*** lance le programme initial qui gère le reste du démarrage : initialisation du système, lancement d'un programme de connexion...

Il va également se charger de lancer les démons. **Un démon (du terme anglais daemon) est un processus qui est constamment en activité et fournit des services au système.**

4. Si vous êtes sur un système... avec INIT

Remarque : vous avez surement appris que le premier processus est le processus ***init*** dont le PID est 1... [5]

Si oui, et bien il va falloir utiliser votre ***systemd*** (D comme... ***Daemon*** et non l'autre... !)

Depuis quelques années déjà, ***Init*** n'est plus utilisé et est obsolète pour lancer les daemons qui appartiennent au noyau (**mode *kernel* vs mode *user***) presque tout comme on en doit plus utiliser ***ifconfig*** mais ***ip addr***

Si vous voulez plus d'informations à ce sujet, je vous invite à lire un de mes supports de cours (mis sur le site et que je ne vais pas avoir le temps de vous expliquer...), quelques pages et/ou aller sur cette page du site **Linuxfr**

https://linuxfr.org/news/systemd-l-init-martyrise-l-init-bafoue-mais-l-init-libere

Bon revenons au processus **Daemon**

Nous n'aurons pas le temps en TP de développer un processus Daemon, mais lire le code et le comprendre est intéressant.

Je vous invite à lire cette page : http://www.enderunix.org/documents/eng/daemon.php

Je m'en suis inspiré et j'ai modifié le code de manière à ne plus avoir de warning.

```
/*
 2  UNIX Daemon Server Programming Sample Program
    Levent Karakas <levent at mektup dot at> May 2001

    To compile: cc -o exampled examped.c
    To run:     ./exampled
 7  To test daemon: ps -ef|grep exampled (or ps -aux on BSD systems)
    To test log:    tail -f /tmp/exampled.log
    To test signal: kill -HUP `cat /tmp/exampled.lock`
    To terminate:   kill `cat /tmp/exampled.lock`
    */
12
    #include <stdio.h>
    #include <stdlib.h> // pour exit
    #include <string.h> // pour strlen
    #include <sys/stat.h> // pour umask
17  #include <fcntl.h>
    #include <signal.h>
    #include <unistd.h>

    #define RUNNING_DIR "."
22  #define LOCK_FILE   "exampled.lock"
    #define LOG_FILE    "exampled.log"

    void log_message(filename,message)
    char *filename;
27  char *message;
    {
    FILE *logfile;
        logfile=fopen(filename,"a");
        if(!logfile) return;
32      fprintf(logfile,"%s\n",message);
        fclose(logfile);
    }

    void signal_handler(sig)
37  int sig;
    {
        switch(sig) {
        case SIGHUP:
            log_message(LOG_FILE,"hangup signal catched");
```

5. Il radote le prof... non non il met une seconde couche !

```
42              break;
        case SIGTERM:
            log_message(LOG_FILE,"terminate signal catched");
            exit(0);
            break;
47      }
    }

    void daemonize()
    {
52  int i,lfp;
    char str[10];
        if(getppid()==1) return; /* already a daemon */
        i=fork();
        if (i<0) exit(1); /* fork error */
57      if (i>0) exit(0); /* parent exits */
        /* child (daemon) continues */
        setsid(); /* obtain a new process group */
        for (i=getdtablesize();i>=0;--i) close(i); /* close all descriptors */
        i=open("/dev/null",O_RDWR); dup(i); dup(i); /* handle standart I/O */
62      umask(027); /* set newly created file permissions */
        chdir(RUNNING_DIR); /* change running directory */
        lfp=open(LOCK_FILE,O_RDWR|O_CREAT,0640);
        if (lfp<0) exit(1); /* can not open */
        if (lockf(lfp,F_TLOCK,0)<0) exit(0); /* can not lock */
67      /* first instance continues */
        sprintf(str,"%d\n",getpid());
        write(lfp,str,strlen(str)); /* record pid to lockfile */
        signal(SIGCHLD,SIG_IGN); /* ignore child */
        signal(SIGTSTP,SIG_IGN); /* ignore tty signals */
72      signal(SIGTTOU,SIG_IGN);
        signal(SIGTTIN,SIG_IGN);
        signal(SIGHUP,signal_handler); /* catch hangup signal */
        signal(SIGTERM,signal_handler); /* catch kill signal */
    }
77
    int main(void)
    {
        daemonize();
        while(1) sleep(1); /* run */
82      return 0;
    }
```

10 Le père attend son fils

Dans l'exemple précédent, nous avons constaté que le processus père se terminait avant le processus fils et donc que ce dernier devenait un processus orphelin dont le père devient le processus de PID 1, INIT.

Il faut "synchroniser" le père avec le fils. Pour cela nous allons demander au père d'attendre le fils.

Nous allons utiliser la primitive **wait**().

#include <sys/wait.h>

pid_t wait(int *status);

La fonction retourne le **pid** du fils terminé et le code de retour de celui-ci dans la variable status.

On ajoute un morceau de code **dans la partie du père** lui ordonnant de se terminer simplement lorsque son fils a terminé.

Il y a aussi la primitive **waitpid**(), même principe que **wait**() et en plus elle permet de spécifier quel fils il faut attendre.

#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
Exemple : ***Waitpid(fils, NULL, 0);***

```
#include <sys/wait.h>

int code_de_retour; // en variable locale
//Donc appartient au papa et au fils
code_de_retour = 1664; // dans le code du fils
code_de_retour = 0; // dans le code du papa
// cette partie après la boucle for
if (pid != 0) {
  int val_de_sortie;
  pid_t pid_fils;
  pid_fils = wait (&val_de_sortie);
  printf("Terminaison du fils : PID = %d \n",
     pid_fils);
  if(WIFEXITED(val_de_sortie))
    printf("Le fils quitte avec le code %d\n",
      WEXITSTATUS(val_de_sortie));
  else
    printf("fin du fils non réglementaire \n");
 }
return (code_de_retour);
}
```

11 Héritage

Un processus hérite d'un grand nombre d'attributs de son père. Il n'hérite pas des attributs suivants :
- Un identification unique PID ;
- L'identification de son processus père ;
- Les signaux[6] pendants ;
- La priorité du processus fils est initialisée à une valeur standard, et la valeur du paramètre ***nice*** est héritée ;
- Les verrous sur les fichiers détenus par le processus père ne sont pas hérités.

Le code suivant permet de constater que le fils hérite d'une copie des descripteurs de fichier du père.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
  char ch[16];
  int desc;
  desc=open("fichier", O_RDWR, 0);
  if(fork()==(pid_t)0) { // dans le fils
    write(desc, "fiston", 6);
    sleep(2);
    read(desc, ch, 4);
    printf("Chaine lue fils : %s \n", ch);
    }
  else { // dans le père
    sleep(1);
    read(desc, ch, 2);
```

6. Voir le mécanisme des signaux dans un prochain cours

```
    printf("Chaine lue papa : %s \n", ch);
    write(desc, "papa", 4);
  }
  return 0;
}
```

Le code suivant permet de constater que le fils hérite des attributs : les propriétaires, le répertoire et la valeur de ***nice***

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/times.h>

int main() {
  char buf[1024]; // Pour le répertoire de travail */
  struct tms temps; /* Pour les nombres de clics */
  int i;
  nice(10);
  for(i=0; i<10000000; i++);    // Une boucle consommatrice de CPU
  if(fork()==(pid_t)0){ //fils
    printf("\nCaractéristiques du fils\n");
    printf("uid=%d euid=%d gid=%d egid=%d\n", getuid(), geteuid(), getgid(), getegid());
    printf("Répertoire de travail : %s\n", getcwd(buf,1024));
    printf("nice : %d\n", nice(0)+20);
    times( &temps );
    printf("Clics en mode utilisateur %d\n", temps.tms_utime);
    printf("Clics en mode systeme %d\n", temps.tms_stime);}
  else{ //papa
    sleep(5); /* Le père s'exécutera après le fils */
    printf("\nCaractéristiques du père\n");
    printf("uid=%d euid=%d gid=%d egid=%d\n", getuid(), geteuid(), getgid(), getegid());
    printf("Répertoire de travail : %s\n", getcwd(buf,1024));
    printf("nice : %d\n", nice(0)+20);
    times( &temps );
    printf("Clics en mode utilisateur %d\n", temps.tms_utime);
    printf("Clics en mode systeme %d\n", temps.tms_stime);}
}
```

12 Un peu plus

12.1 Vfork

Il existe une variante à l'appel système fork(), c'est vfork(), cet appel système permet lui aussi de créer un fils, mais le père est bloqué pendant l'exécution du fils. Vous me direz à quoi cela peut servir ?, je vais y répondre...

Cela permet d'aller beaucoup plus vite, car vfork() sert à créer un nouveau processus sans effectuer de copie de la table des pages mémoire du processus père et le code est partagé avec plusieurs applications.

Ce principe est surtout utilisé dans les systèmes embarqués où l'empreinte mémoire est beaucoup plus faible. on remplace la glibc par une ***uClibc*** qui est 5 fois plus petite, ce qui remplace Linux par ***uClinux*** qui fonctionne sur la majeur partie des systèmes embarqués.

Le code qui suit montre une utilsation de cette primitive vfork(). Dans ce code si vous remplacez simplement la primitive ***vfork***() de la ligne 6 par un simple fork, le programme ne se comporte plus de la même façon.

```
int n; // une variable globale
int main(){
  int pid;
```

```
  printf("L'@ virtuelle de n dans le père %p \n", &n);
  n=0;
  switch(pid=vfork()){
  case -1:
    perror("echec de vfork");
    exit(2);
  case 0: // le fils
    printf("L'@ virtuelle de n dans le fils %p \n", &n);
    n=1;
    printf("fils s'endort 10s et le père est bloqué \n");
    sleep(10);
    printf("Terminaison du fils \n");
    _exit(0);
  default: // le père
    sleep(1); // afin que le père soit élu après le fils
    printf("Reprise d'execution du père \n");
    n++; // n vaut alors 2 car le fils la mis à 1...
    printf("Valeur de n dans le père %d \n",n);
    exit(0); }}
```

12.2 Thread

Il existe une extension de ce modèle de processus, ce sont les **processus léger** (***thread***) en opposition aux processus lourd. Un processus classique est constitué d'un espace d'adressage avec un seul fil d'exécution constitué du compteur ordinal et une pile d'exécution.

dans le domaine des processus léger, on ne dispose que d'un seul espace d'adressage admettant plusieurs fils (pas le fiston, le fil comme celui à coudre...) d'exécution et chacun de ces fils d'exécution est appelé ***thread***, ***processus léger*** ou ***activités*** possédant chacun un compteur ordinal et une pile d'exécution privée. L'entité contenant les différents fils d'exécution est appelé processus ou acteur.

Les avantages sont :

- la commutation de contexte est très allégées. Le processeur n'a besoin que e changer de pile et de compteur ordinal, le contexte mémoire reste inchangé ;
- l'opération de création d'un nouveau fil d'exécution est elle aussi plus légère puisqu'il n'est plus nécessaire de dupliquer complètement l'espace d'adressage du processus père.

Les 2 sources ci-dessous permettent de comparer les 2 mécanismes, processus et *thread*

12.3 Comparaison Processus vs Thread

Pour les programmes tirant partie du parallélisme, le choix entre processus et threads peut être difficile. Voici quelques pistes pour vous aider à déterminer le modèle de parallélisme qui convient le mieux à votre programme :

- Tous les *threads* d'un programme doivent exécuter le même code. Un processus fils, au contraire, peut exécuter un programme différent en utilisant une fonction **exec**.
- Un *thread* peut endommager les données d'autres *threads* du même processus car les threads partagent le même espace mémoire et leurs ressources. Par exemple, une écriture sauvage en mémoire via un pointeur non initialisé au sein d'un *thread* peut corrompre la mémoire d'un autre *thread*. Un processus corrompu par contre, ne peut pas agir de cette façon car chaque processus dispose de sa propre copie de l'espace mémoire du programme.
- Copier le contenu de la mémoire pour un nouveau processus a un cout en performances par rapport à la création d'un nouveau *thread*. Cependant, la copie n'est effectuée que lorsque la mémoire est modifiée, donc ce cout est minime si le processus fils ne fait que lire la mémoire.
- Les *threads* sont utilisés pour les programmes qui ont besoin d'un parallélisme finement contrôlé. Par exemple, si un problème peut être décomposé en plusieurs tâches presque identiques, les *threads* peuvent être un bon choix. Les processus sont utilisés pour des programmes ayant besoin d'un parallélisme plus grossier.
- Le partage de données entre des *threads* est trivial car ceux-ci partagent le même espace mémoire. Le partage de données entre des processus nécessite l'utilisation de mécanismes IPC que nous allons voir plus tard.

```c
// processus_thread_exemple.c
#include <stdio.h>
#include <unistd.h>

int i; // Variable globale
int main(){
  pid_t pid;
  i=0;
  pid=fork();

  if(pid == 0){ //dans le fils
    i = i + 1000;
    printf("hello, fils valeur de i:%d\n",i);
    i = i + 2000;
    printf("hello, fils valeur de i:%d\n",i);
  }
  else { //dans le père
    i = i + 1000;
    printf("hello, père valeur de i:%d\n",i);
    i = i + 2000;
    printf("hello, père valeur de i:%d\n",i);
  }
  return 0;}
```

```c
// thread_exemple.c
#include <stdio.h>
#include <pthread.h>

int i; // Variable globale
void addition() {
  i = i + 10;
  printf("hello, thread fils valeur de i:%d\n",i);
  i = i + 20;
  printf("hello, thread fils valeur de i:%d\n",i);
}
int main(){
  pthread_t num_thread;
  i=0;
  if(pthread_create(&num_thread, NULL,
            (void*(*))addition, NULL) == -1)
    perror("problème pthread_create\n");
  i = i + 1000;
  printf("hello, thread principal valeur de i:%d\n",i);
  i = i + 2000;
  printf("hello, thread principal valeur de i:%d\n",i);
  pthread_join(num_thread, NULL);
  return 0;}
// à compiler avec gcc -Wall -lpthread thread_exemple.c
```

13 Conclusion

Écrire une application multiprocessus, ce n'est pas si "difficile" que cela, il suffit de bien connaitre les principes et de les appliquer avec rigueur. Il n'existe pratiquement plus d'application digne de ce nom n'utilisant pas ces mécanismes, surtout si vous utilisez un système d'exploitation multitâches.

Nous allons voir dans les prochains cours comment faire communiquer ces processus, leur envoyer des signaux et les utiliser dans la programmation Client/Serveur et les faire communiquer à l'aide de *socket*.