Programmation Système

AMEWUHO Kofi Amenyo Jean

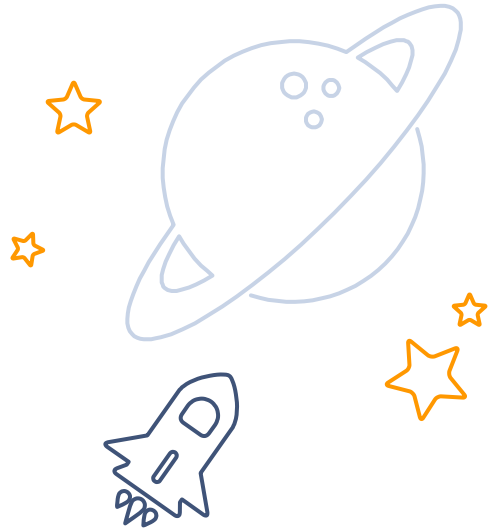
Ingénieur de conception informatique

Email : avenakaj@gmail.com

Tel : 90-50-94-17

2022 - 2023

Avant de commencer, réfléchissez sur quelques questions et réponses fondamentales. Par exemple, pourquoi avez-vous besoin de ce type de programmation ? Comment ces concepts peuvent-ils vous faciliter la vie ? Si vous connaissez les réponses à ces questions, votre parcours d'apprentissage sera facile et vous pourrez vous rapporter à ces concepts de différentes manières.



Ne perdez pas votre motivation si vous ne pouvez pas tout comprendre après le premier passage. Dans de nombreux cas, cela peut sembler compliqué, mais petit à petit, cela vous sera plus facile.

TP

Écrire un processus chien de garde (processus zombie), exécuté en arrière-plan, qui affiche le nombre de processus actifs dans le système toutes les 5 secondes, ainsi que les processus en question. Cet état du système devra être capturé et sauvegardé dans un fichier de journalisation (log.txt).

Trouvez la meilleure manière de mettre fin à l'exécution de ce processus sans interagir directement avec elle.

NB : Utiliser la commande Kill pour terminer un processus est une mauvaise manière de terminer un processus car ses ressources utilisées restent ouvertes.

1

Rappels

Système d’exploitation

Qu'est-ce qu’un système d’exploitation ?

Interfaces d’un système d’exploitation

Protection des ressources

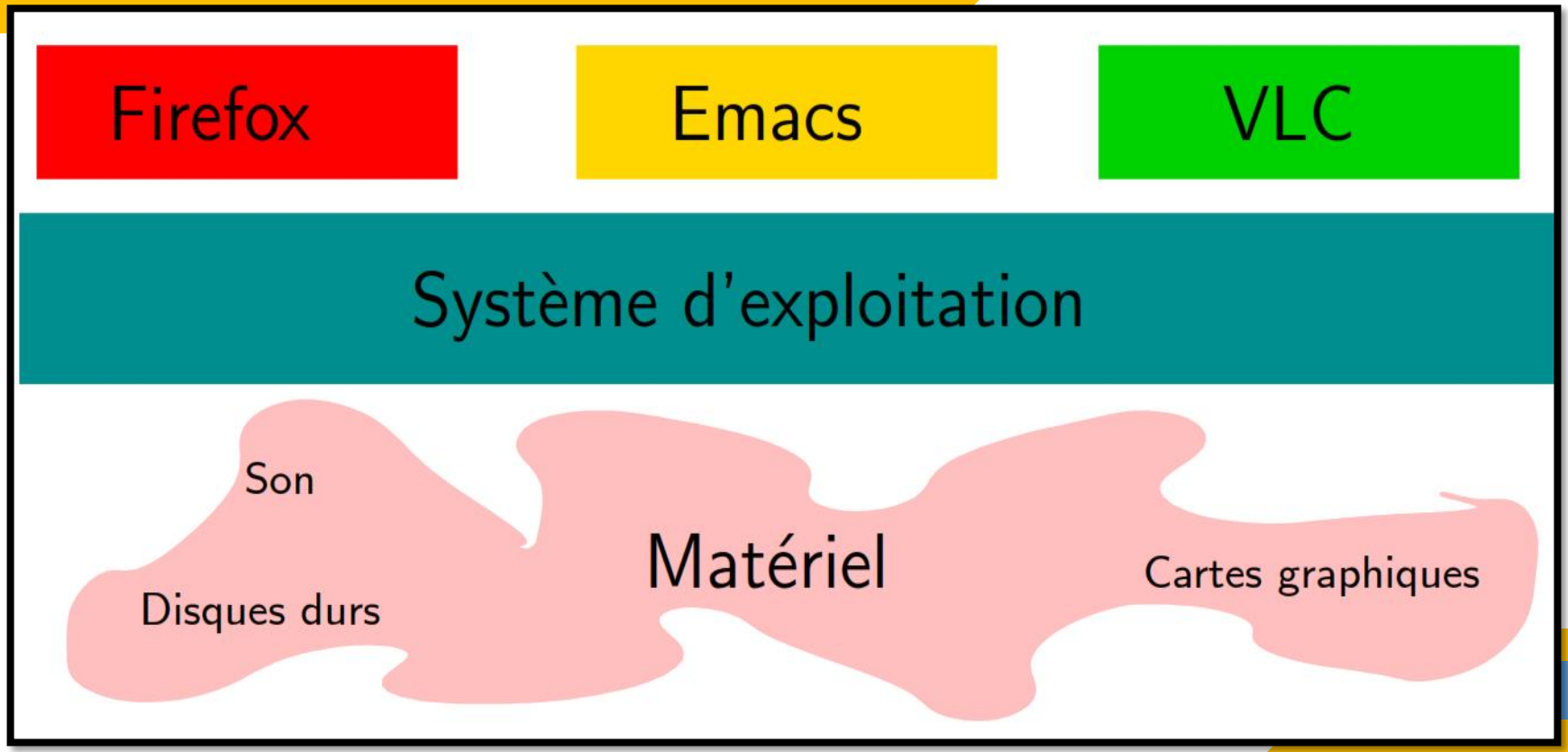
Qu’est-ce qu’un système d’exploitation ?

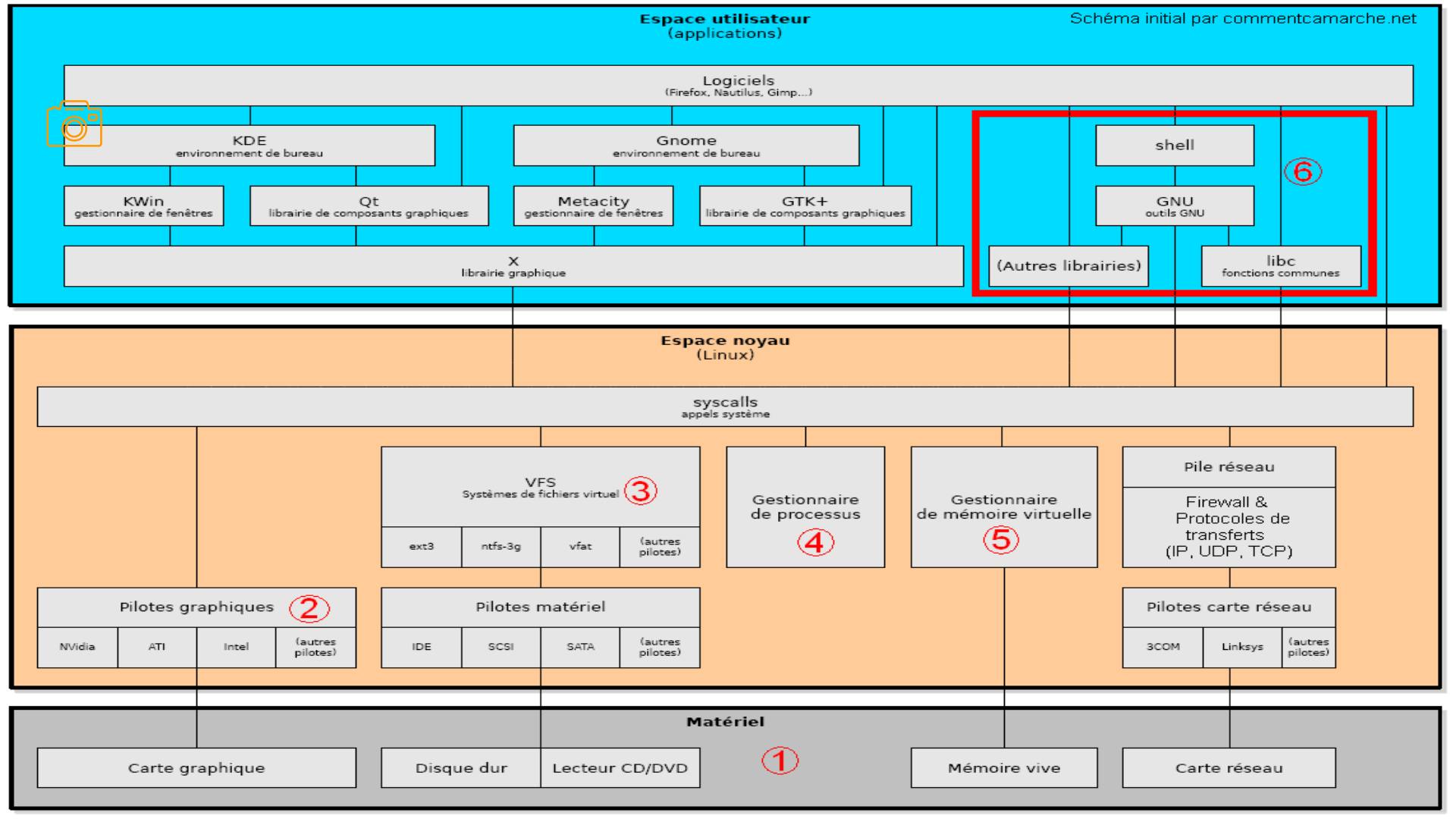
Un système d'exploitation, ou logiciel système, ou Operating System (OS), est un logiciel qui, dans un appareil électronique, pilote les dispositifs matériels et reçoit des instructions de l'utilisateur ou d'autres logiciels (ou applications).

En informatique, un système d'exploitation est un groupe de programmes qui facilitent l'utilisation d'un ordinateur. Il s'agit d'un logiciel qui reçoit des sollicitations pour employer les ressources de la machine comme le disque dur pour stocker de la mémoire, ou des périphériques pour établir une communication visuelle ou auditive.

- Offre une interface unifiée du matériel aux applications
- Protège les ressources (et les applis entre elles)
- Virtualise les ressources

Qu'est-ce qu'un système d'exploitation ?




Espace utilisateur
(applications)
Schéma initial par commentcamarche.net
Logiciels
(Firefox, Nautilus, Gimp...)
KDE
environnement de bureau
Gnome
environnement de bureau
shell
6
KWin
gestionnaire de fenêtres
Qt
librairie de composants graphiques
Metacity
gestionnaire de fenêtres
GTK+
librairie de composants graphiques
GNU
outils GNU
X
librairie graphique
(Autres librairies)
libc
fonctions communes
Espace noyau
(Linux)
syscalls
appels système
VFS
Systèmes de fichiers virtuel
3
ext3
ntfs-3g
vfat
(autres pilotes)
Gestionnaire de processus
4
Gestionnaire de mémoire virtuelle
5
Pile réseau
Firewall & Protocoles de transferts (IP, UDP, TCP)
Pilotes graphiques
2
NVidia
ATI
Intel
(autres pilotes)
Pilotes matériel
IDE
SCSI
SATA
(autres pilotes)
Pilotes carte réseau
3COM
Linksys
(autres pilotes)
Matériel
Carte graphique
Disque dur
Lecteur CD/DVD
1
Mémoire vive
Carte réseau

Rôle d'un système d'exploitation

Deux fonctions complémentaires :

- Adaptation d'interface : offre une interface plus pratique que le matériel
  - Dissimule les détails de mise en œuvre (abstraction)
  - Dissimule les limitations physiques (taille mémoire) et le partage des ressources
- Gestion des ressources (mémoire, processeur, disque, réseau, affichage, …)
  - Alloue les ressources aux applications que le demandent
  - Partage les ressources entre les applications
  - Protège les applications les unes des autres ; empêche l'usage abusif des ressources

Interfaces d'un système d'exploitation

Deux interfaces :

- Interface de programmation (Application Programming Interface)
  - Utilisable à partir des programmes s'exécutant sous le système
  - Composée d'un ensemble d'appels systèmes (procédures spéciales)
- Interface de commande ou interface utilisateur
  - Utilisable par un utilisateur humain, sous forme textuelle ou graphique
  - Composée d'un ensemble de commandes
    - Textuelles (ex : rm *.o)
    - Graphique (ex : déplacer un fichier vers la corbeille)

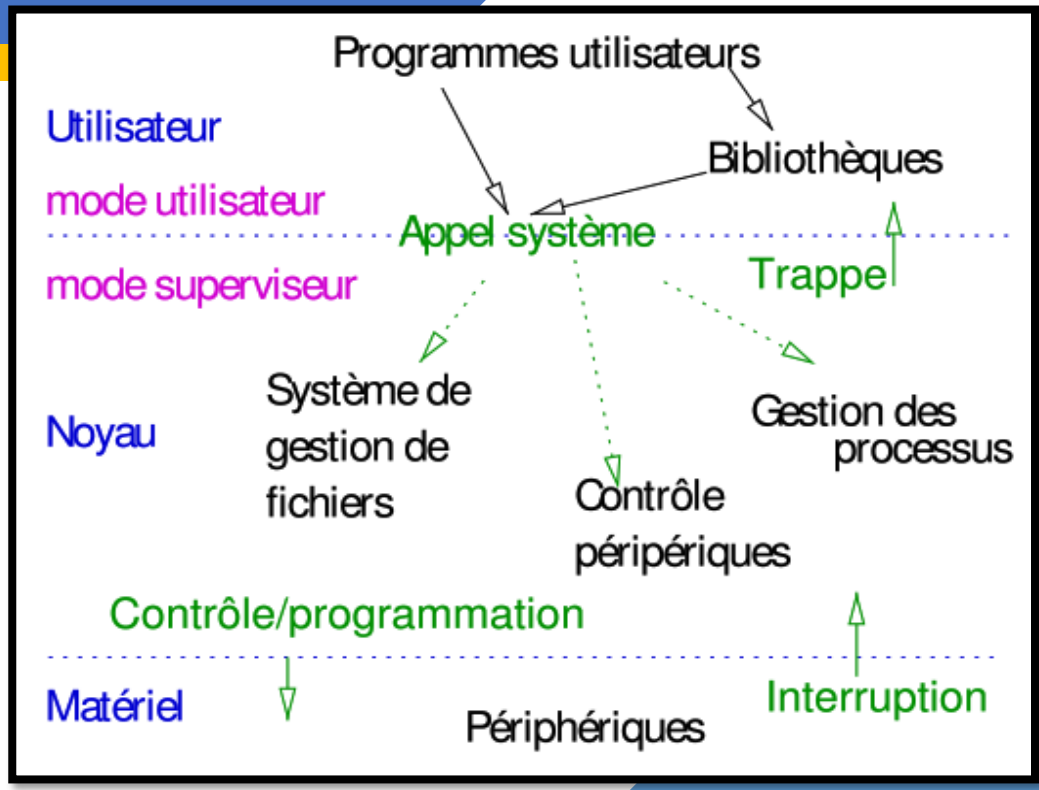
Exemples d'usage des interfaces d'UNIX

- Interface de programmation
  - Programme copiant un fichier *fich1* dans un autre *fich2* grâce à l'utilisation des appels système read et write
- Interface de commande
  - *$ cp fich1 fich2*

```
while (nb_lus = read(fich1, buf, BUFFSIZE )) {
  if ((nb_lus == -1) && (errno != EINTR)) {
    break; /* erreur */
  } else if (nb_lus > 0) {
    pos_buff = buf;
    a_ecrire = nb_lus;
    while (nb_ecrits =
            write(fich2,pos_buff,a_ecrire)) {
      if ((nb_ecrits == -1) && (errno != EINTR))
        break;  /* erreur */
      else if (a_ecrire == 0)
        break;  /* fini */
      else if (nb_ecrits > 0) {
        pos_buff += nb_ecrits;
        a_ecrire -= nb_ecrits;
} } } }
```

Découpage en couches du système

- Niveau utilisateur
  - Applications utilisateur
  - Logiciels de base
  - Bibliothèque système
  - Appels de fonction
- Niveau noyau
  - Gestion des processus
  - Système de fichier
  - Gestion de la mémoire
- Niveau matériel
  - Firmware, contrôleur



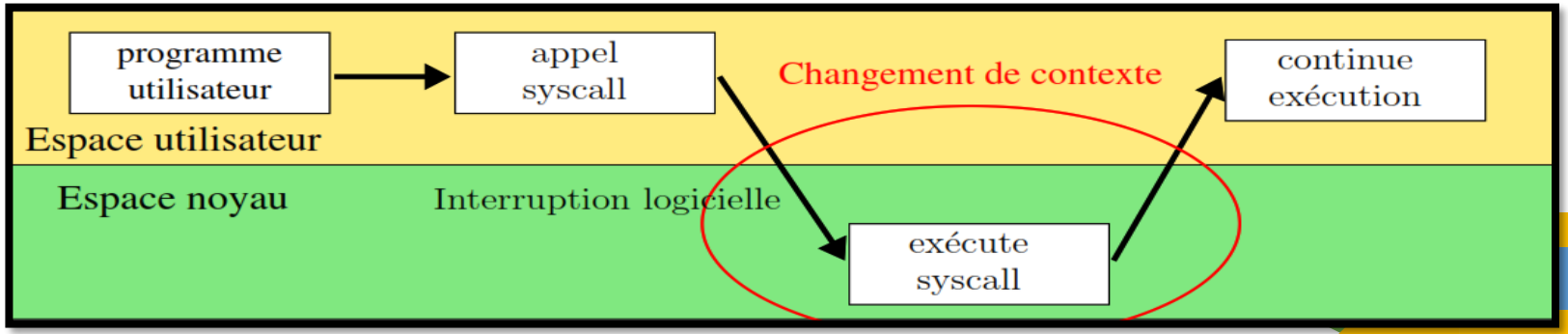
Protection des ressources

Méthodes d'isolation des programmes (et utilisateurs) dangereux

- **Préemption :** ne donner aux applications que ce qu'on peut leur reprendre
- **Interposition :** pas d'accès direct, vérification de la validité de chaque accès
- **Mode privilégié :** certaines instructions machines sont réservées à l'OS

Exemples de ressources protégées

- **Processeur :** préemption → Interruption (matérielles) à intervalles réguliers pour redonner le contrôle à l'OS (l'OS choisit le programme devant s'exécuter ensuite)
- **Mémoire :** interposition (validité de tout accès à la mémoire vérifiée au préalable)
- **Exécution :** mode privilégié (espace noyau) ou normal (espace utilisateur)
  - Le CPU bloque certaines instructions assembleur (E/S) en mode utilisateur



Limites de la protection

Les systèmes réels ont des failles. Ils ne protègent pas de tout.

- While true ; do mkdir toto ; cd toto ; done **(en shell)**
- While(1) { fork(); } **(en C)**
- While(1) { char *a=malloc(512); *a='1'; } **(en C)**

Réponse classique de l'OS : gel (voire pire)

On suppose que les utilisateurs ne sont pas si mal intentionnés

Unix was not designed to stop people from doing stupid things, because that would also stop them from doing clever things. – Doug Gwyn

Limites de la protection

- Qu’est-ce qu’un système d’exploitation ?
- Donnez les rôles et fonctions d’un système d’exploitation
- Quelles sont les différentes interfaces d’un système d’exploitation ?
- Quelles sont les techniques classiques de protection des ressources ?

Langage C

Pourquoi le langage C ?

Différents langages permettent au programmeur de construire des programmes qui seront exécutés par le processeur. En réalité, le processeur ne comprend qu'un langage : le langage machine. Ce langage est un langage binaire dans lequel toutes les commandes et toutes les données sont représentés sous la forme de séquences de bits.

Le langage machine est peu adapté aux humains et il est extrêmement rare qu'un informaticien doive manipuler des programmes directement en langage machine. Par contre, pour certaines tâches bien spécifiques, comme par exemple le développement de routines spéciales qui doivent être les plus rapides possibles ou qui doivent interagir directement avec le matériel, il est important de pouvoir efficacement générer du langage machine. Cela peut se faire en utilisant un langage d'assemblage. Chaque famille de processeurs a un langage d'assemblage qui lui est propre. Le langage d'assemblage permet d'exprimer de façon symbolique les différentes instructions qu'un processeur doit exécuter. Le langage d'assemblage est converti en langage machine grâce à un assembleur.

Pourquoi le langage C ?

Le langage d'assemblage est le plus proche du processeur. Il permet d'écrire des programmes compacts et efficaces. C'est aussi souvent la seule façon d'utiliser des instructions spéciales du processeur qui permettent d'interagir directement avec le matériel pour par exemple commander les dispositifs d'entrée/sortie. C'est essentiellement dans les systèmes embarqués qui disposent de peu de mémoire et pour quelques fonctions spécifiques des systèmes d'exploitation que le langage d'assemblage est utilisé de nos jours. La plupart des programmes applicatifs et la grande majorité des systèmes d'exploitation sont écrits dans des langages de plus haut niveau.

Le langage C, développé dans les années 70 pour écrire les premières versions du système d'exploitation Unix, est aujourd'hui l'un des langages de programmation les plus utilisés pour développer des programmes qui doivent être rapides ou doivent interagir avec le matériel. La plupart des systèmes d'exploitation sont écrits en langage C.

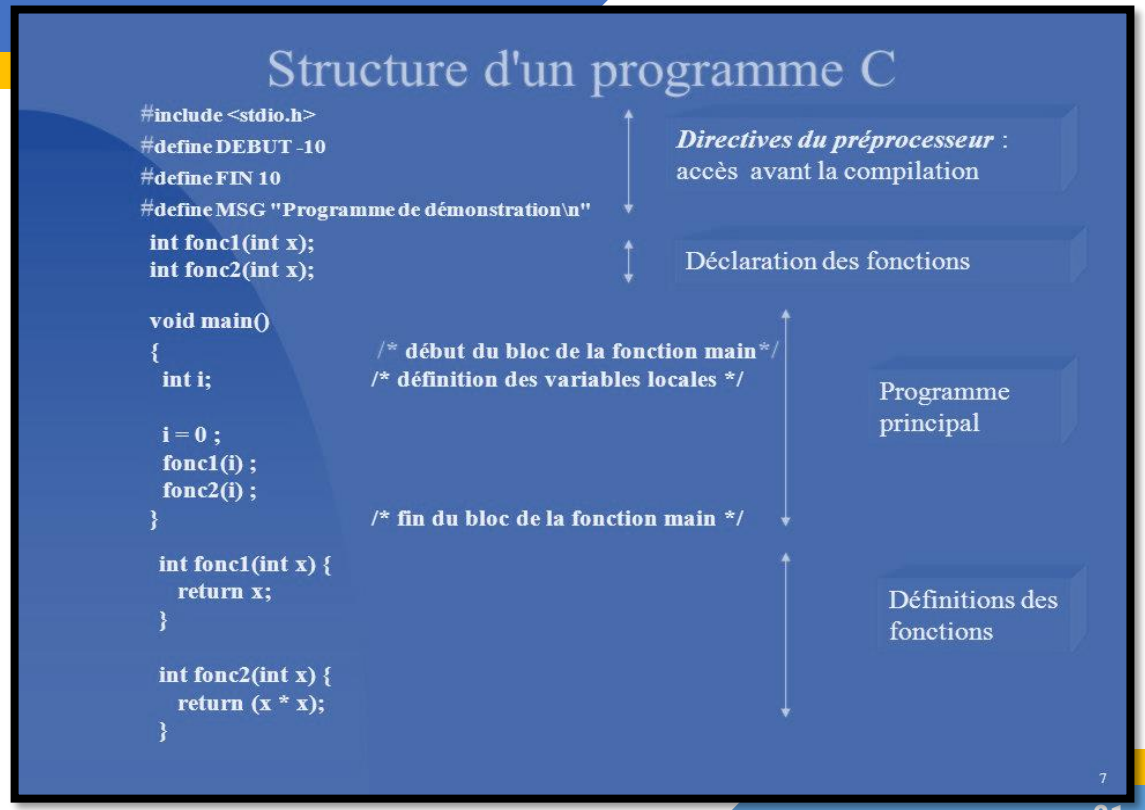
Le langage C a été conçu à l'origine comme un langage proche du processeur qui peut être facilement compilé, c'est-à-dire traduit en langage machine, tout en conservant de bonnes performances.

Structure générale d'un code en C

Un programme C se présente de la façon suivante :

[directives au préprocesseur]
[déclarations de variables externes]
[fonctions secondaires]
main(arguments formels)
{
  déclarations de variables internes
  instructions
  instruction de retour
}

type fonction_secondaire (arguments)
{
  déclarations de variables internes
  instructions
}



La fonction « main » et ses arguments formels

Tout programme C doit contenir une fonction nommée main dont la signature est :

```
int main(int argc, char *argv[ ])
```

Lorsque le système d'exploitation exécute un programme C compilé, il démarre son exécution par la fonction main et passe à cette fonction les arguments fournis en ligne de commande.

Le nombre de paramètres est passé dans la variable entière **argc** et le tableau de chaînes de caractères char ***argv[ ]** contient tous les arguments.

Par convention, en C le premier argument (se trouvant à l'indice 0 du tableau argv) est le nom du programme qui a été exécuté par l'utilisateur.

En pratique, le système d'exploitation passe également les variables d'environnement à la fonction main. Sous certaines variantes de Unix, et notamment Darwin/MacOS ainsi que sous certaines versions de Windows, le prototype de la fonction main inclut explicitement ces variables d'environnement.

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

Type de retour de la fonction « main »

En C, la fonction main retourne un entier. Cette valeur de retour est passée par le système d'exploitation au programme (typiquement un shell ou interpréteur de commandes) qui a demandé l'exécution du programme. Grâce à cette valeur de retour il est possible à un programme d'indiquer s'il s'est exécuté correctement ou non.

Par convention, un programme qui s'exécute sous Unix doit retourner EXIT_SUCCESS lorsqu'il se termine correctement et EXIT_FAILURE en cas d'échec. La plupart des programmes fournis avec un Unix standard respectent cette convention.

Dans certains cas, d'autres valeurs de retour non nulles sont utilisées pour fournir plus d'informations sur la raison de l'échec. En pratique, l'échec d'un programme peut être dû aux arguments incorrects fournis par l'utilisateur ou à des fichiers qui sont inaccessibles.

Type de retour de la fonction « main »

La librairie **<stdlib.h>** contient la définition de différentes fonctions et constantes de la librairie standard et notamment **EXIT_SUCCESS** et **EXIT_FAILURE**. Ces constantes sont définies en utilisant la macro **#define** du préprocesseur :

```
#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0
```

Exemples de codes en C

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

int main(int argc, char *argv[])
{

   printf("Hello, world!\n");

   return EXIT_SUCCESS;
}
```

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

int main(int argc, char *argv[])
{
  int i;

  printf("Ce programme a %d argument(s)\n", argc);

  for (i = 0; i < argc; i++)
    printf("argument[%d] : %s\n", i, argv[i]);

  return EXIT_SUCCESS;
}
```

Compilation d'un code C sous Unix

Pour être exécuté, un programme doit être compilé. Il existe de nombreux compilateurs permettant de transformer le langage C en langage machine. Dans le cadre de ce cours, nous utiliserons gcc.

gcc supporte de très nombreuses options dont l'option -Wall qui le force à afficher tous les messages de type warning et l'option -o suivie du nom de fichier qui indique le nom du fichier dans lequel le programme exécutable doit être sauvegardé par le compilateur.

Exemple : *gcc -Wall -o hello hello.c*

Le programme exécutable peut alors être exécuté comme un script en précédant son nom des caractères ./ depuis l'interpréteur de commandes. Cela peut être suivi par les arguments à envoyer à la fonction **main**.

Exemple : *./hello*

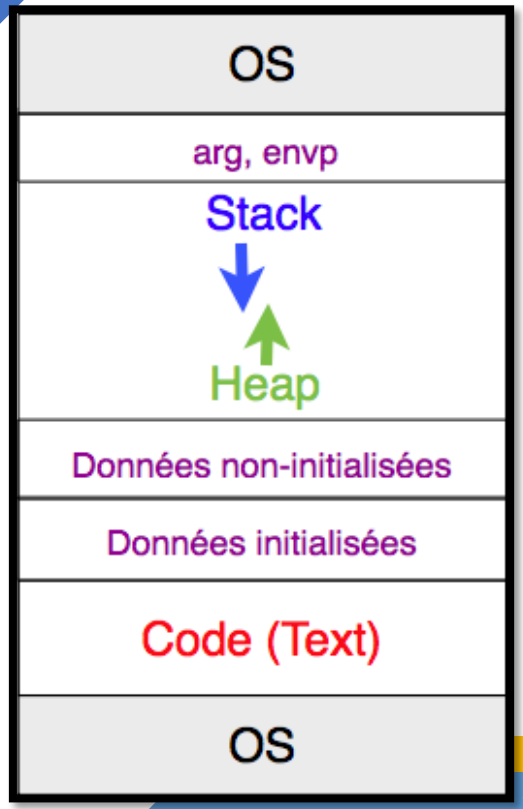
./hello Arg1 Arg2 Arg3

Organisation de la mémoire

Organisation d'un programme Linux en mémoire

Lors de l'exécution d'un programme en mémoire, le système d'exploitation charge depuis le système de fichier le programme en langage machine et le place à un endroit convenu en mémoire. Lorsqu'un programme s'exécute sur un système Unix, la mémoire peut être vue comme étant divisée en **six** zones principales. Ces zones sont représentées schématiquement dans la figure ci-contre.

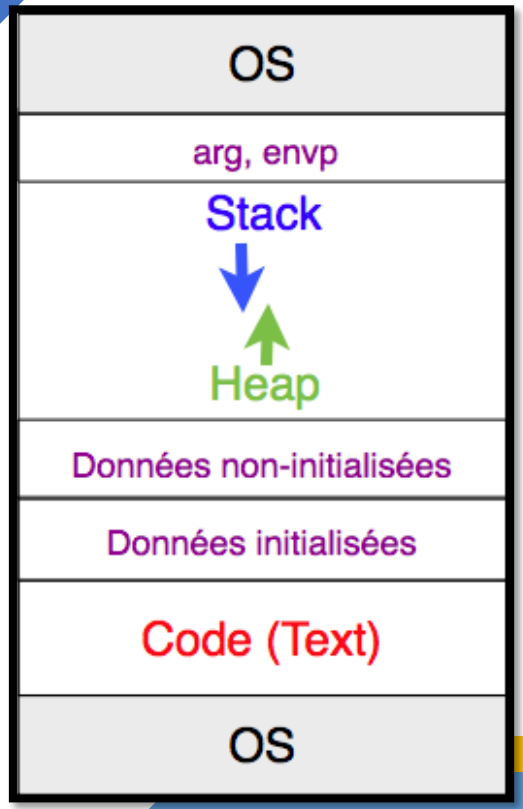
La figure présente une vision schématique de la façon dont un processus Linux est organisé en mémoire centrale. Il y a d'abord une partie de la mémoire qui est réservée au système d'exploitation (OS dans la figure). Cette zone est représentée en grisé dans la figure.



Organisation d'un programme Linux en mémoire

Le segment text

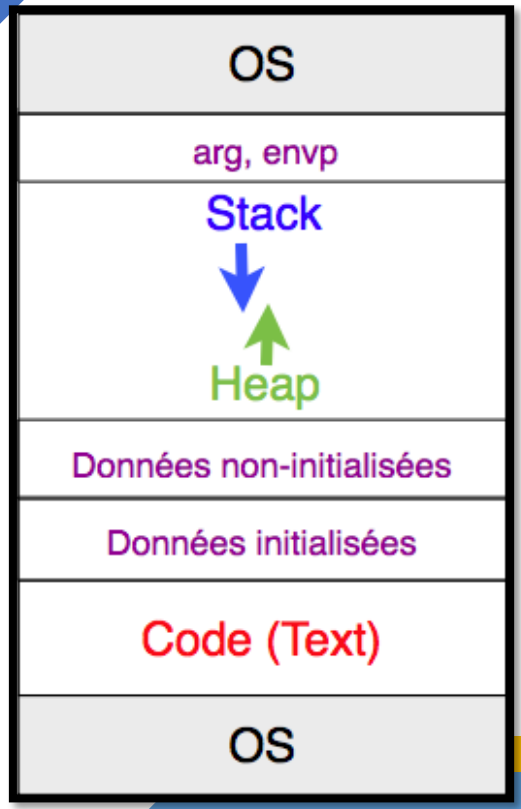
La première zone est appelée par convention le **segment text**. Cette zone se situe dans la partie basse de la mémoire. C'est dans cette zone que sont stockées toutes les instructions qui sont exécutées par le micro-processeur. Elle est généralement considérée par le système d'exploitation comme étant uniquement accessible en lecture. Si un programme tente de modifier son segment text, il sera immédiatement interrompu par le système d'exploitation. C'est dans le segment text que l'on retrouvera les instructions de langage machine correspondant aux fonctions de calcul et d'affichage du programme. Nous en reparlerons lorsque nous présenterons le fonctionnement du langage d'assemblage.



Organisation d'un programme Linux en mémoire

Le segment des données initialisées

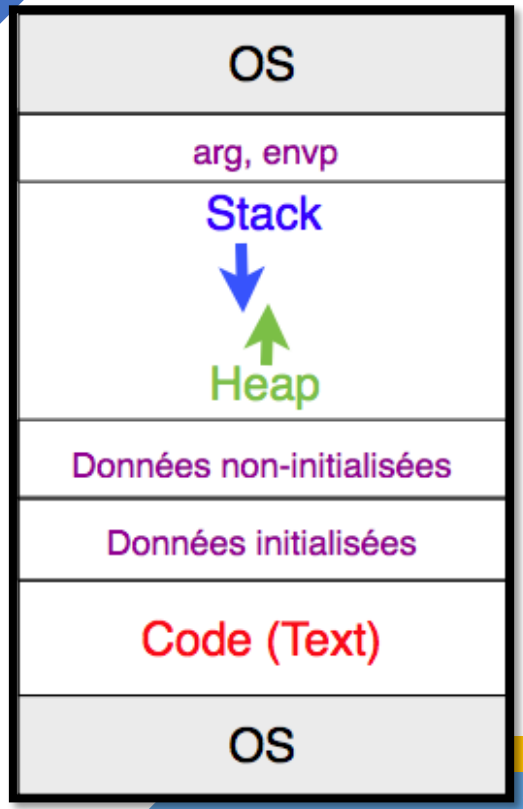
La deuxième zone, baptisée segment des données initialisées, contient l'ensemble des données et chaînes de caractères qui sont utilisées dans le programme. Ce segment contient deux types de données. Tout d'abord, il comprend l'ensemble des variables globales explicitement initialisées par le programme (dans le cas contraire, elles sont initialisées à zéro par le compilateur et appartiennent alors au segment des données non-initialisées). Ensuite, les constantes et les chaînes de caractères utilisées par le programme.



Organisation d'un programme Linux en mémoire

Le segment des données non-initialisées

La troisième zone est le segment des données non-initialisées, réservée aux variables non-initialisées. Cette zone mémoire est initialisée à zéro par le système d'exploitation lors du démarrage du programme.

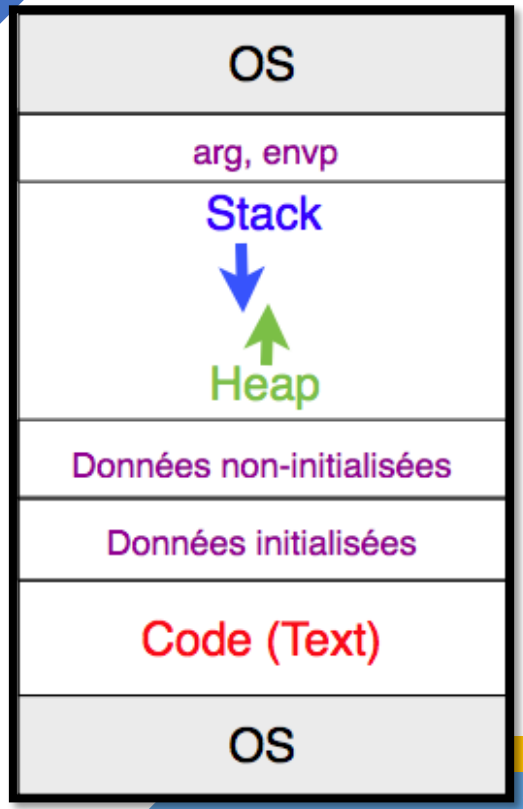


Organisation d'un programme Linux en mémoire

Le tas (ou heap)

La quatrième zone de la mémoire est le tas (ou heap en anglais). C'est une des deux zones dans laquelle un programme peut obtenir de la mémoire supplémentaire pour stocker de l'information. Un programme peut y réserver une zone permettant de stocker des données et y associer un pointeur.

En C, la plupart des processus allouent et libèrent de la mémoire en utilisant les fonctions **malloc** et **free** qui font partie de la librairie standard.

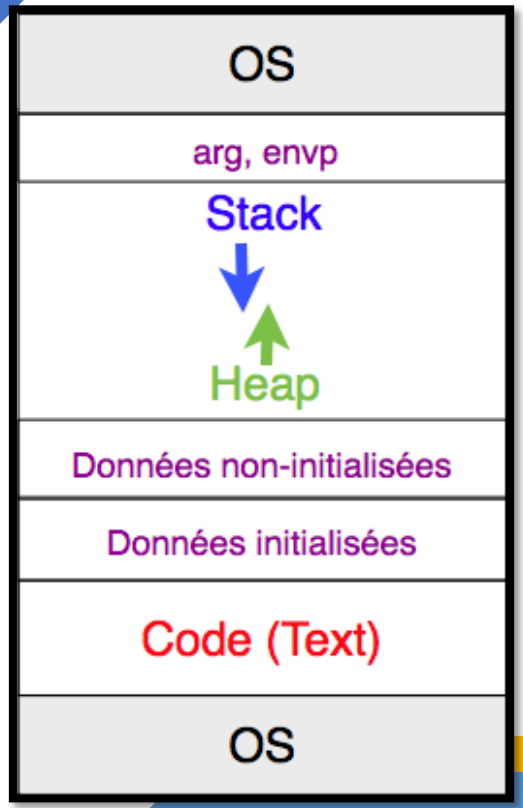


Organisation d'un programme Linux en mémoire

Les arguments et variables d'environnement

Lorsque le système d'exploitation charge un programme Unix en mémoire, il initialise dans le haut de la mémoire une zone qui contient deux types de variables. Cette zone contient tout d'abord les arguments qui ont été passés via la ligne de commande. Le système d'exploitation met dans **argc** le nombre d'arguments et place dans **char *argv[]** tous les arguments passés avec dans **argv[0]** le nom du programme qui est exécuté.

Cette zone contient également les variables d'environnement. Ces variables sont généralement relatives à la configuration du système. Leurs valeurs sont définies par l'administrateur système ou l'utilisateur. De nombreuses variables d'environnement sont utilisées dans les systèmes Unix. Elles servent à modifier le comportement de certains programmes.



Organisation d'un programme Linux en mémoire

Les arguments et variables d'environnement

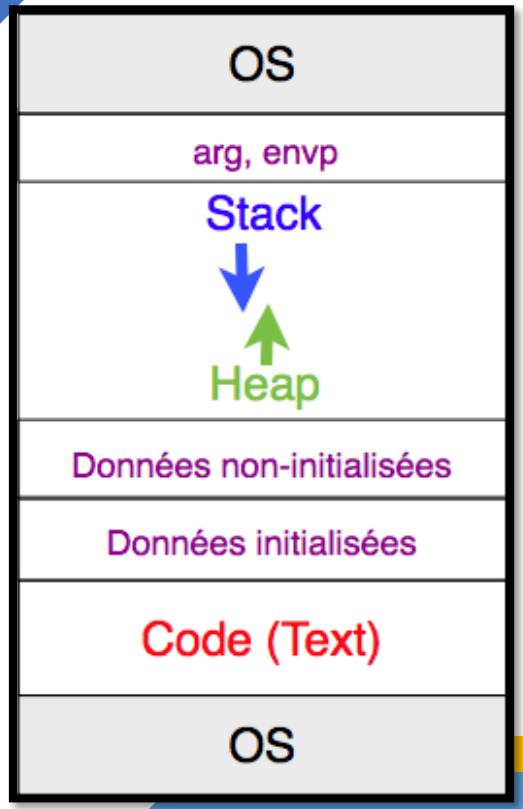
Une liste exhaustive de toutes les variables d'environnement est impossible, mais en voici quelques unes qui sont utiles en pratique :

HOSTNAME : le nom de la machine sur laquelle le programme s'exécute. Ce nom est fixé par l'administrateur système via la commande **hostname**

SHELL : l'interpréteur de commande utilisé par défaut pour l'utilisateur courant. Cet interpréteur est lancé par le système au démarrage d'une session de l'utilisateur. Il est stocké dans le fichier des mots de passe et peut être modifié par l'utilisateur via la commande **passwd**

USER : le nom de l'utilisateur courant. Sous Unix, chaque utilisateur est identifié par un numéro d'utilisateur et un nom uniques. Ces identifiants sont fixés par l'administrateur système via la commande **passwd**

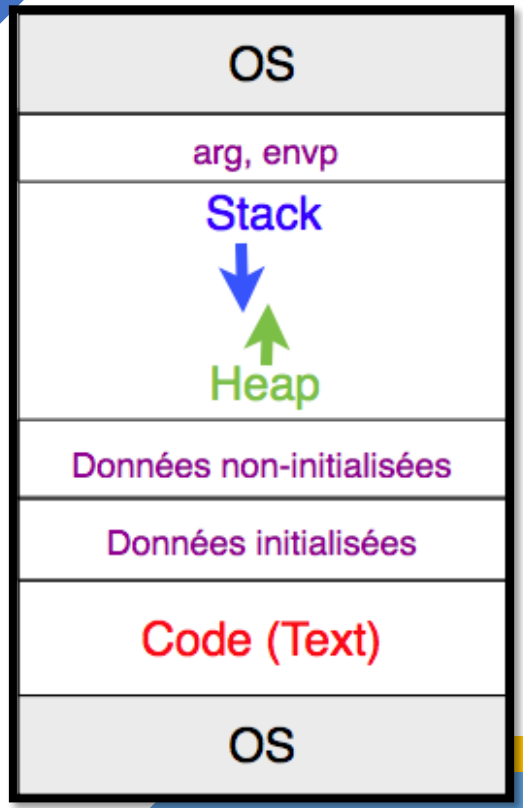
HOME: le répertoire d'accueil de l'utilisateur courant. Ce répertoire d'accueil appartient à l'utilisateur. C'est dans ce répertoire qu'il peut stocker tous ses fichiers.



Organisation d'un programme Linux en mémoire

Les arguments et variables d'environnement

PRINTER : le nom de l'imprimante par défaut qui est utilisée
PATH: cette variable d'environnement contient la liste ordonnée des répertoires que le système parcourt pour trouver un programme à exécuter. Cette liste contient généralement **les répertoires dans lesquels le système stocke les exécutables standards**, comme /usr/local/bin:/bin: /usr/bin: /usr/local/sbin: /usr/sbin:/sbin: ainsi que des répertoires relatifs à des programmes spécialisés comme /usr/lib/mozart/bin:/opt/python3/bin. L'utilisateur peut ajouter des répertoires à son PATH avec bash en incluant par exemple la commande PATH=$PATH:$HOME/local/bin:. dans son fichier .profile.
La librairie standard contient plusieurs fonctions qui permettent de manipuler les variables d'environnement d'un processus.

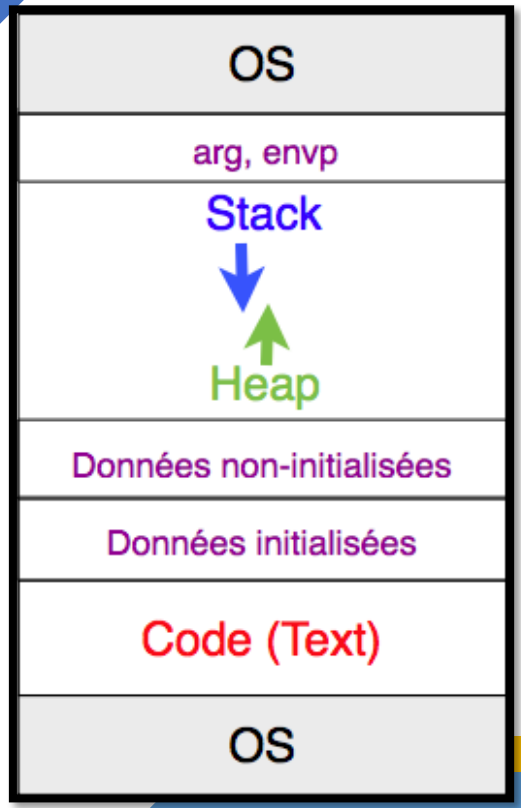


Organisation d'un programme Linux en mémoire

La pile (stack)

La pile ou stack en anglais est la dernière zone de mémoire utilisée par un processus. C'est une zone très importante car c'est dans cette zone que le processus va stocker l'ensemble des variables locales mais également les valeurs de retour de toutes les fonctions qui sont appelées. Cette zone est gérée comme une pile, d'où son nom.

Le langage C utilise le passage par valeur, les valeurs des arguments d'une fonction sont copiés sur la pile avant de démarrer l'exécution de cette fonction. Lorsque la fonction prend comme argument un entier, cette copie prend un temps très faible. Par contre, lorsque la fonction prend comme argument une ou plusieurs structures de grand taille, celles-ci doivent être entièrement copiées sur la pile.



2

Processus

Qu'est-ce qu’un processus ?

Comment gère-t-on un processus ?

Comment les processus communiquent-ils entre eux ?

Qu’est-ce qu’un processus ?

Il est très important de différencier la notion de programme et la notion de processus. Un programme, c'est une suite d'instructions (notion **statique**), tandis qu'un processus, c'est l'image du contenu des registres et de la mémoire centrale (notion **dynamique**).

On peut imaginer un processus comme un programme en cours d'exécution auquel est associé un environnement processeur et un environnement mémoire. En effet, au cours de son exécution, les instructions d'un programme modifient les valeurs des registres (le compteur ordinal, le registre d'état...) ainsi que le contenu de la pile. Cette représentation est très imparfaite car une application peut non seulement utiliser plusieurs processus concurrents, mais un unique processus peut également lancer l'exécution d'un nouveau programme, en remplaçant entièrement le code et les données du programme précédent.

Sous Unix, toute tâche qui est en cours d'exécution est représentée par un processus. Un processus est une entité comportant à la fois des données et du code. **On peut considérer un processus comme une unité élémentaire en ce qui concerne l'activité sur le système**.

Qu'est-ce qu'un processus ?

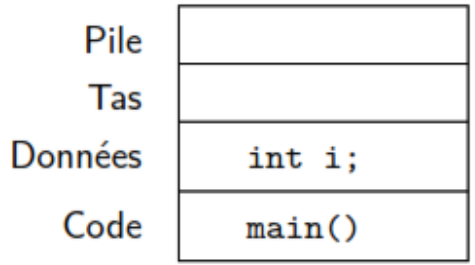
Programme :

Code + données (**passif**)

```
int i;
int main() {
  printf("Salut\n");
}
```

▶ Processus :

Programme **en cours d'exécution**



Question : Si un même programme est lancé deux fois, obtiendra-t-on un même processus pour les deux exécutions ?

Réponse : Non, les processus seront différents. Le principe est le même que celle de la différence entre deux objets d'une même classe.

Lister les processus sous Linux

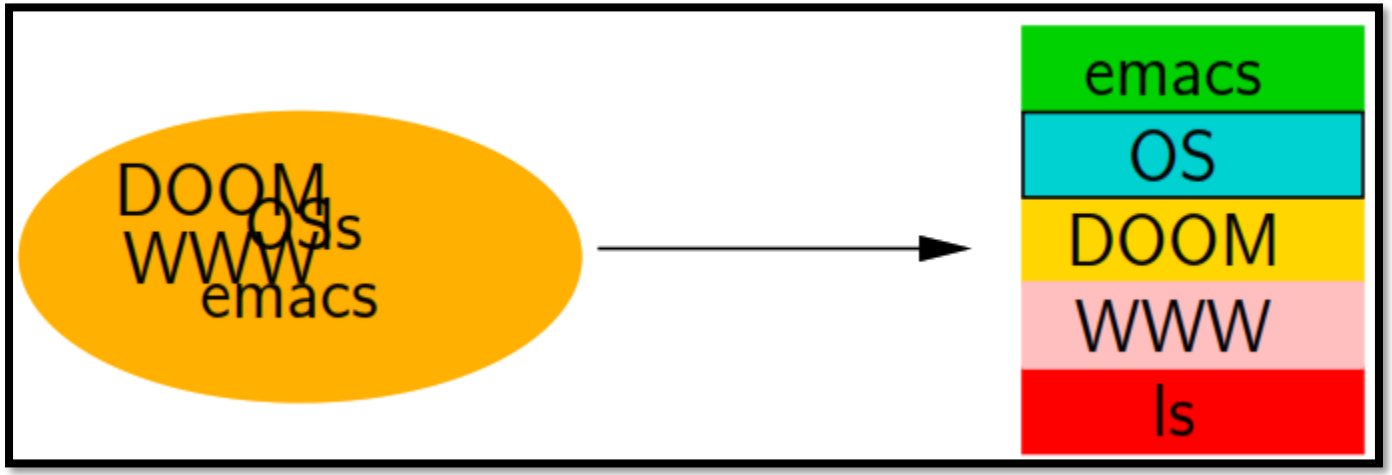
On peut examiner la liste des processus présents sur le système à l'aide de la commande ps, et plus particulièrement avec ses options, qui nous permettent de voir les processus endormis, et ceux qui appartiennent aux autres utilisateurs.

Exemple : ps aux

La commande ps affiche plusieurs colonnes dont la signification nous donne différents types d'informations sur les processus listés.

Utilité des processus : Simplicité

- Simplifier en isolant chaque activité de l'ordinateur en processus séparés

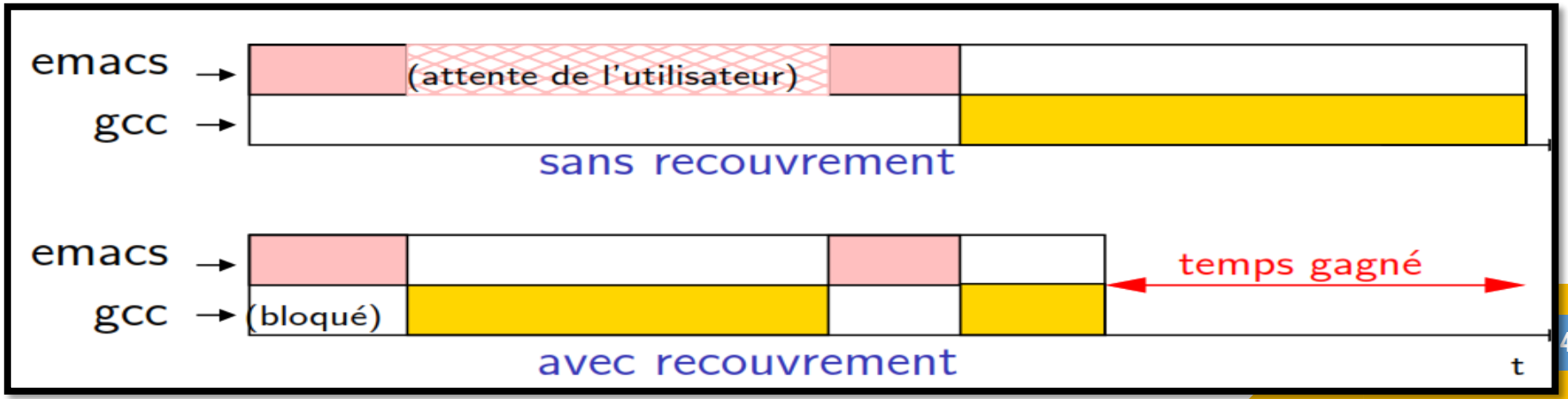


- L'OS s'occupe de chaque processus de la même façon et chaque processus ne s'occupe de l'OS
- La décomposition est une réponse classique à la complexité

Utilité des processus : Efficacité

- Mieux gérer les communications qui bloquent les processus. Les communications dans ce cas sont à prendre au sens large : réseau, disque, utilisateur, autres programmes…

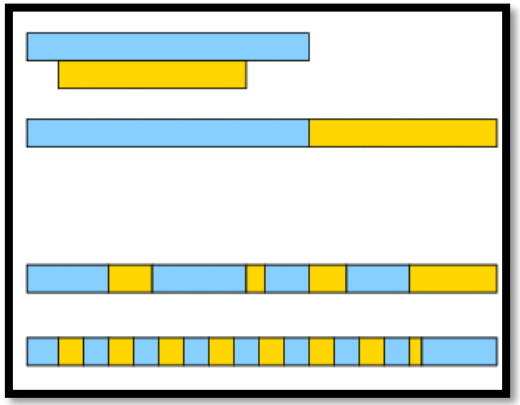
L'image ci-dessous illustre le cas de recouvrement des temps de calculs et communications :



Parallélisme et pseudo-parallélisme

Que faire quand deux processus sont prêts à s'exécuter ?

- Si deux processeurs physiques sont disponibles, tout va bien.
- Sinon, FCFS (Premier venu, premier servi) ? Mauvaise interactivité !
- Autre possibilité : le pseudo-parallélisme (ils s'exécutent successivement pendant des laps de temps très courts)



Le pseudo-parallélisme fonctionne grâce aux interruptions matérielles régulières rendant le contrôle du processeur à l'OS. Il permet également de recouvrir le temps des calculs et communications.

Relations entre les processus

Compétition

Plusieurs processus peuvent vouloir accéder en même temps à une ressource exclusive (ne pouvant être utilisée que par un seul à la fois) comme le processeur, l'imprimante ou la carte son.

Une solution parmi tant d'autre est le FCFS ; les autres attendent leur tour.

Coopération

Plusieurs processus peuvent collaborer pour une tâche commune et souvent, ils doivent se synchroniser.

Par exemple P1 produit un fichier et P2 imprime le fichier ou P1 met à jour un fichier et P2 consulte le fichier. La synchronisation se ramène au fait que P2 doit attendre que P1 ait franchi un certain point de son exécution.

Faire attendre un processus

- Attente active

Processus 1

```
while (ressource occupée)
   { };
ressource occupée = true;
```

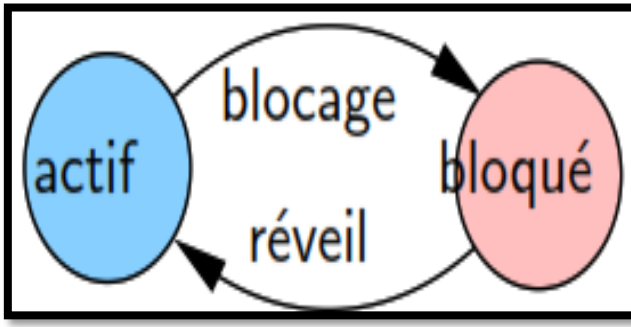
Processus 2

```
ressource occupée = true;
utiliser ressource;
ressource occupée = false;
```

Cette attente active constitue un gaspillage de ressources surtout si les processus sont exécutés en pseudo-parallélisme.

Faire attendre un processus

- Blocage du processus




```
...
sleep(5); /* se bloquer pour 5 secondes */
...
```

Définition d'un nouvel état de processus : bloqué. L'exécution est suspendue et le réveil est explicitement lancé par un autre processus ou par le système.

Gestion des processus

État d'un processus

Un processus peut avoir plusieurs états :

- **exécution** (R pour running) : le processus est en cours d'exécution ;
- **sommeil** (S pour sleeping) : quand il est interrompu au bout d'un quantum de temps ;
- **arrêt** (T pour stopped) : le processus a été temporairement arrêté par un signal. Il ne s'exécute plus et ne réagira qu'à un signal de redémarrage ;
- **zombie** (Z pour zombie) : le processus s'est terminé, mais son père n'a pas encore lu son code de retour.

De plus, sous Unix, un processus peut évoluer dans deux modes différents : le mode noyau et le mode utilisateur. Généralement, un processus utilisateur entre dans le mode noyau quand il effectue un appel système.

Organisation des processus

Les processus sont organisés en **hiérarchie**. Chaque processus doit être lancé par un autre. La racine de cette hiérarchie est le programme initial.

Le processus inactif du système (System idle process : le processus que le noyau exécute tant qu'il n'y a pas d'autres processus en cours d'exécution) a le PID 0. C'est celui-ci qui lance le premier processus que le noyau exécute, le programme initial. Généralement, sous les systèmes basés sous Unix, le programme initial se nomme **init**, et il a le PID 1.

Après son chargement, le programme initial 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.

Identification des processus

Un processus peut être identifié par 3 valeurs, à savoir le PID, le PPID, le UID et le GID.

- PID (Process IDentifier) : Chaque processus peut être identifié par son numéro de processus, ou PID. Un numéro de PID est unique dans le système : il est impossible que deux processus aient un même PID au même moment.
- PPID (Parent PID) : PID du procéssus créateur.
- UID (User IDentifier) : Les systèmes basés sur Unix sont particulièrement axés sur le côté multi-utilisateur. Chaque utilisateur possède un identifiant, sous forme numérique, nommé UID. En conséquence, nous pouvons également distinguer les processus entre eux par l'UID de l'utilisateur qui les a lancés.
- GID (Group IDentifier) : Chaque utilisateur du système appartient à un ou plusieurs groupes. Un processus fait donc également partie des groupes de l'utilisateur qui l'a lancé.

Vie et mort des processus

Tout processus a un début et une fin.

- Début : création par un autre processus
- Fin
  - Autodestruction (à la fin du programme)
  - Destruction par un autre processus
  - Destruction par l'OS
  - Certains processus ne se terminent pas avant l'arrêt de la machine. Ils sont appelés « daemon » (Disk And Execution Monitor). Ils réalisent des fonctions du système comme le login des utilisateurs, les impressions, les serveurs web…

Création des processus

Création des processus dans UNIX

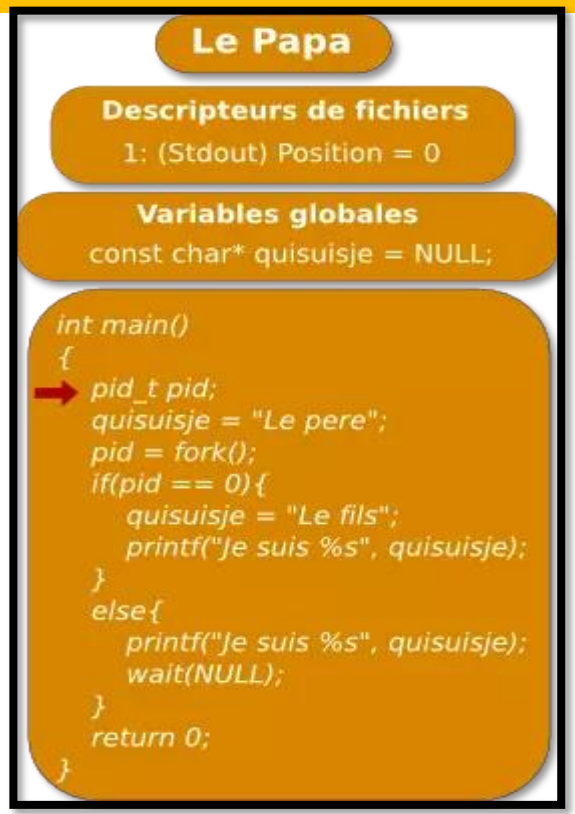
- Par l'interface de commande
  - Chaque commande est exécutée dans un processus séparé
  - Il est possible de créer des processus en pseudo-parallélisme :
    - $ **prog1 & prog2** crée deux processus pour exécuter prog1 et prog2
    - $ **prog1 & prog1** lance deux instances de prog1
- **Par l'interface de programmation**, l'API : clonage avec l'appel système fork.

Appel système pid_t fork()

- Effet : clone le processus appelant
- Le processus créé (fils) est une copie conforme du processus créateur (père) ; copies conformes comme une cellule qui se divise en deux
- Ils se reconnaissent par la valeur de retour de fork() :
  - Pour le père : le pid du fils (ou -1 si erreur)
  - Pour le fils : 0

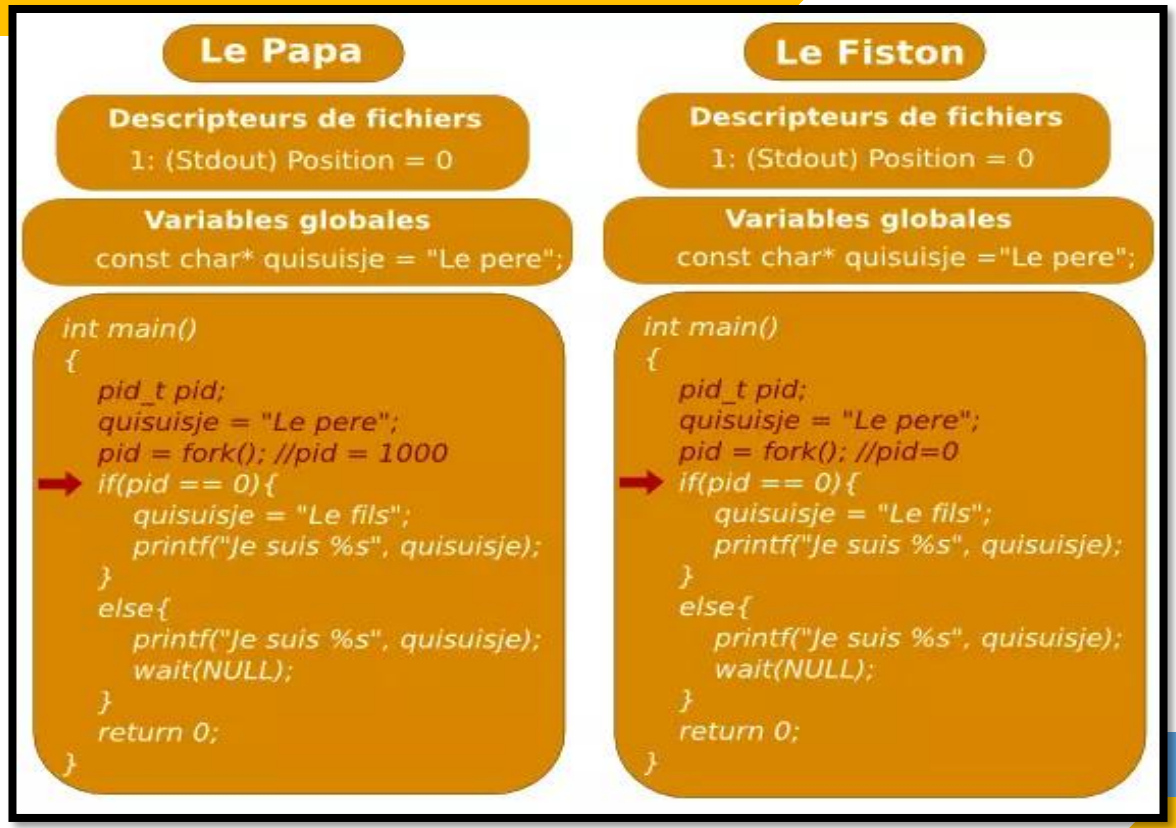
```
if (fork() != 0) {
   printf("je suis le père, mon PID est %d\n", getpid());
} else {
   printf("je suis le fils, mon PID est %d; mon père est %d\n",
          getpid(), getppid());
   /* en général exec(), (exécution d'un nouveau programme) */
}
```

Appel système pid_t fork()

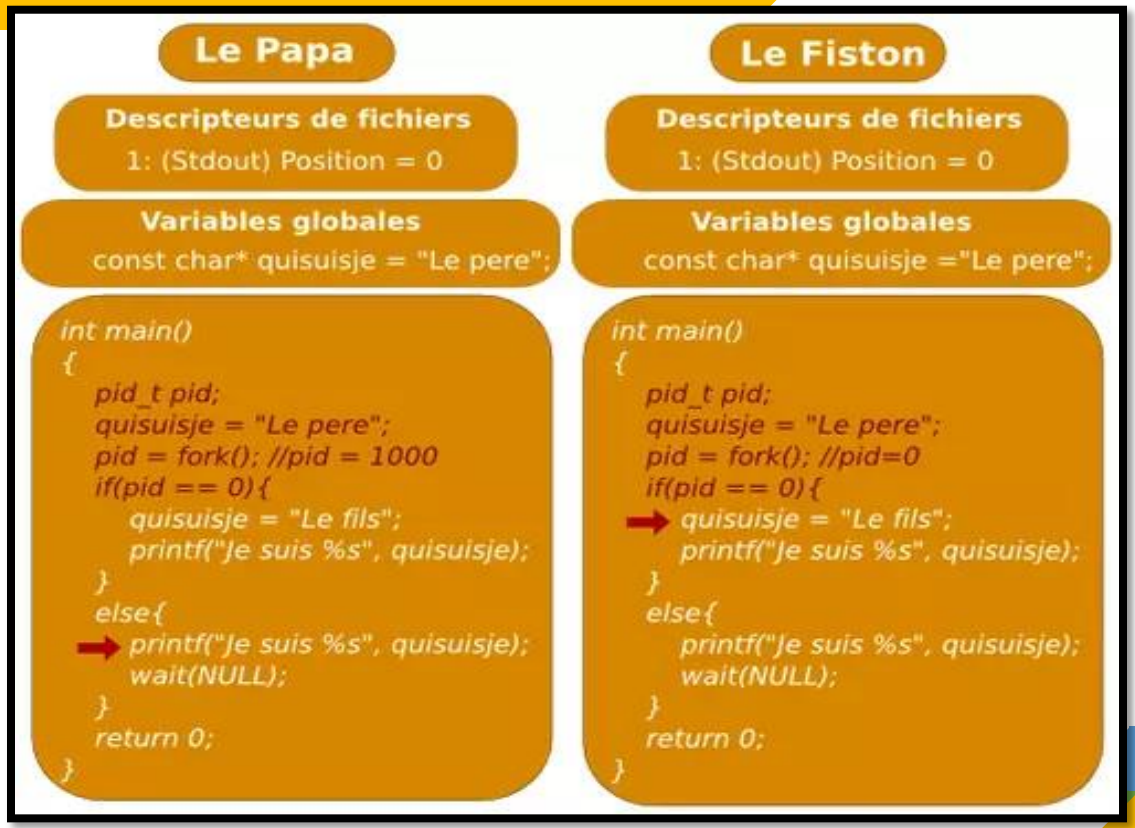


La flèche rouge pointe vers la prochaine instruction à exécuter.

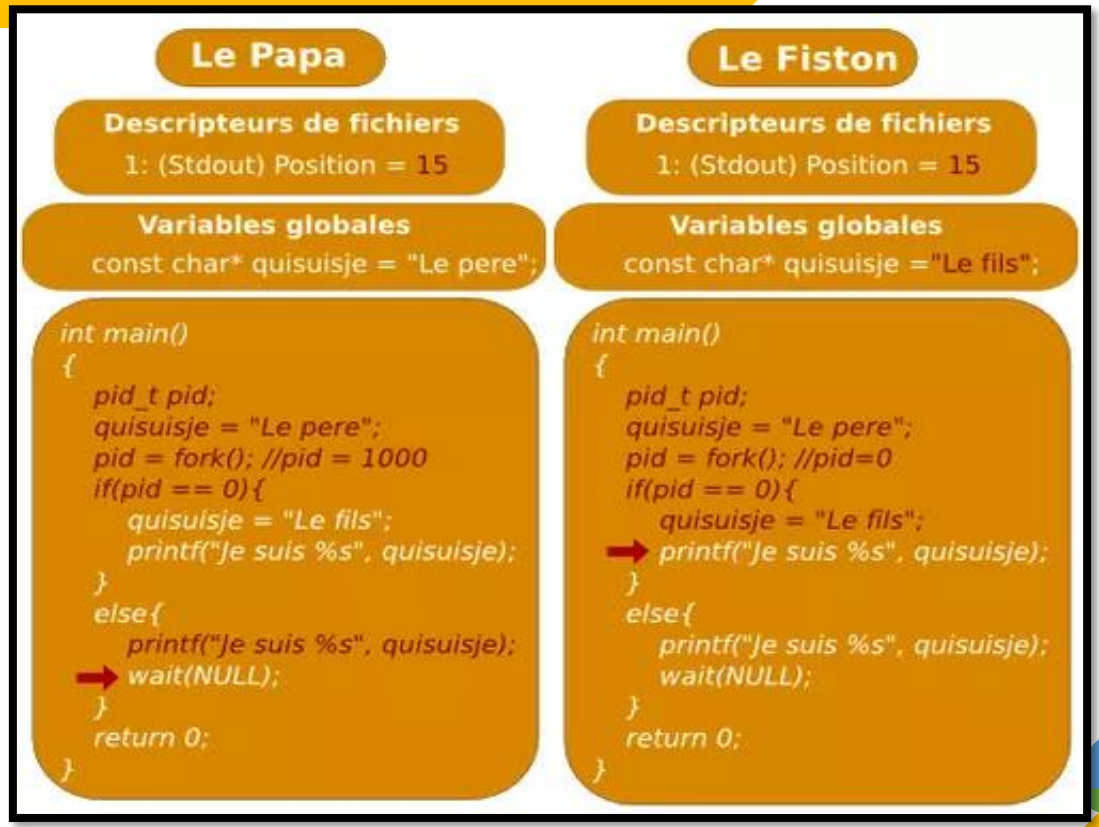
Appel système pid_t fork()



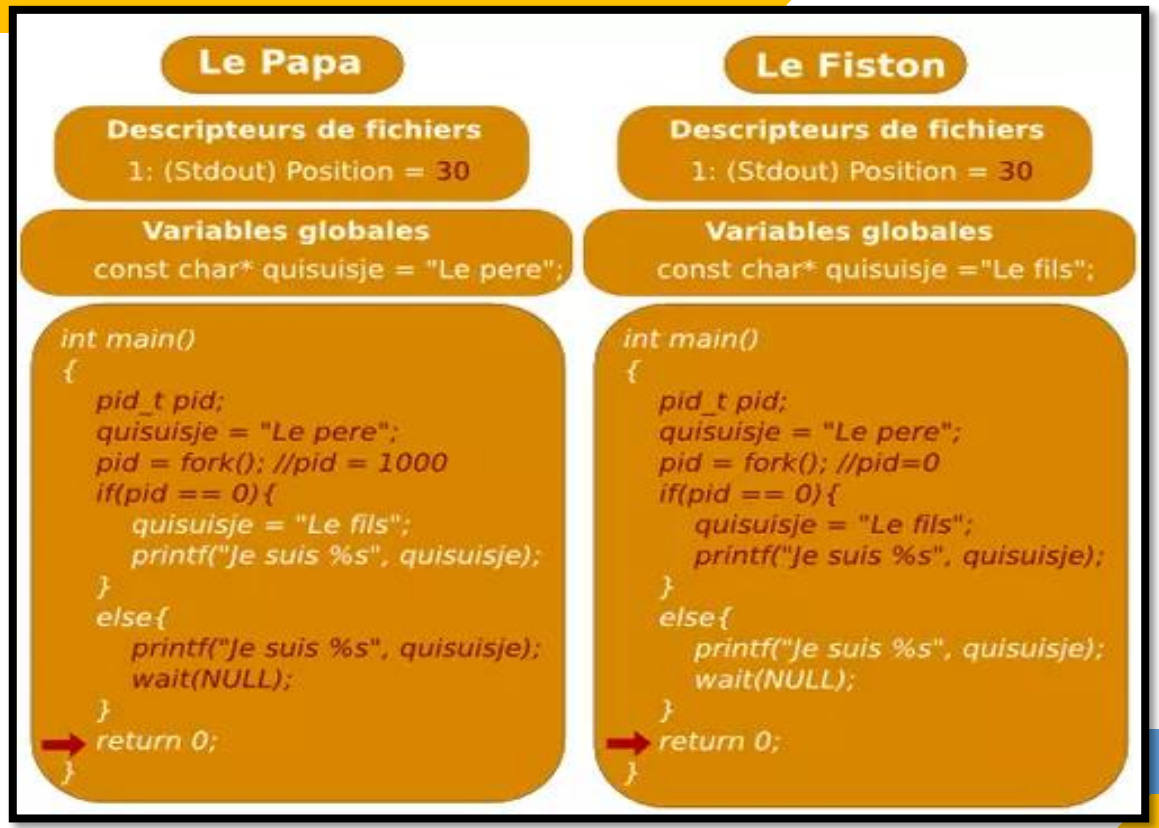
Appel système pid_t fork()



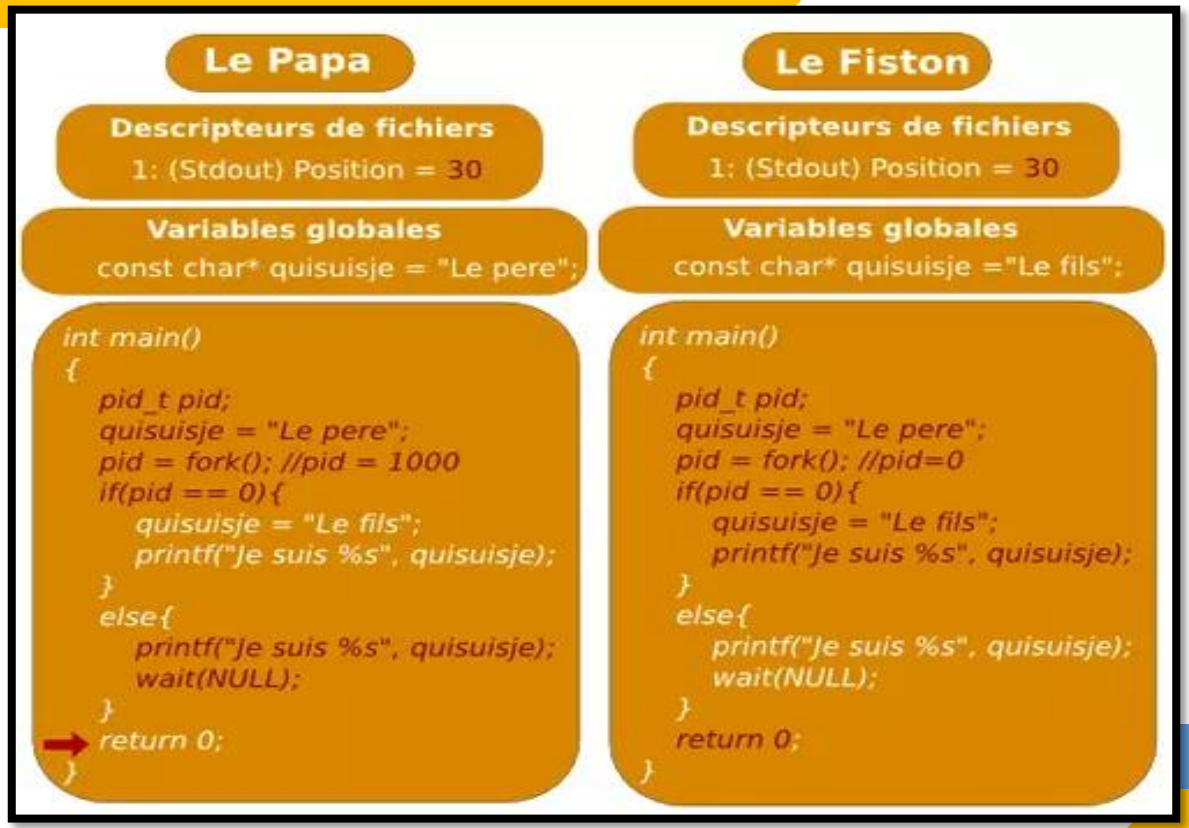
Appel système pid_t fork()



Appel système pid_t fork()



Appel système pid_t fork()



Appel système pid_t fork()

Bon nombre de fonctions utilisées avec la programmation système (notamment les appels-système comme fork) nécessiterons l'inclusion de la bibliothèque <unistd.h>.

La fonction fork retourne une valeur de type pid_t. Il s'agit généralement d'un int ; il et déclaré dans <sys/types.h>.

Dans le cas où la fonction a renvoyé -1 et donc qu'il y a eu une erreur, le code de l'erreur est contenue dans la variable globale **errno**, déclarée dans le fichier errno.h. Ce code peut correspondre à deux constantes :

- ENOMEM : le noyau n'a plus assez de mémoire disponible pour créer un nouveau processus ;
- EAGAIN : ce code d'erreur peut être dû à deux raisons : soit il n'y a pas suffisamment de ressources systèmes pour créer le processus, soit l'utilisateur a déjà trop de processus en cours d'exécution. Ainsi, que ce soit pour l'une ou pour l'autre raison, vous pouvez rééditer votre demande tant que fork renvoie EAGAIN.

Voici le prototype de la fonction fork :

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

pid_t fork(void);
```

Autres fonctions

- La fonction getpid retourne le PID du processus appelant.
- La fonction getppid retourne le PPID du processus appelant.
- La fonction getuid retourne l'UID du processus appelant.
- La fonction getgid retourne le GID du processus appelant.

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

pid_t getpid(void);
```

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

pid_t getppid(void);
```

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

uid_t getuid(void);
```

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

gid_t getgid(void);
```

Exemple de code

Complétez, compilez et exécutez le code suivant puis indiquer les différents résultats :

```
int i=3;
if (fork() != 0) {
     printf("je suis le père, mon PID est %d\n", getpid());
     i += 4;
} else {
     printf("je suis le fils, mon PID est %d; mon père est %d\n",
             getpid(), getppid());
     i += 9;
}
printf("pour %d, i = %d\n", getpid(), i);
```

Exemple de code (test de clonage)

- Complétez le code suivant.
- Compilez le code et exécutez avec la commande suivante : ./testClonage & ps
- Que remarquez-vous ?

testClonage.c

```
int main() {
  if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    sleep(10) /* blocage pendant 10  secondes */
    exit(0);
 } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(10) /* blocage pendant 10  secondes */
    exit(0);
  }
}
```

Exercices

▷ **Question 1:** Qu'affiche l'exécution du programme suivant :

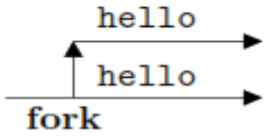
```
1  int main() {
2      pid_t pid;
3      int x = 1;
4
5      pid = fork();
6      if (pid == 0) {
7          printf("Dans fils : x=%d\n", ++x);
8          exit(0);
9      }
10
11     printf("Dans père : x=%d\n", --x);
12     exit(0);
13 }
```

Exercices

▷ **Question 2:** On considère les deux programmes suivants et leur schéma d'exécution.

- *Exemple :* un clonage

```
1 int main() {
2     fork();
3     printf("hello!\n");
4     exit(0);
5 }
```



- *Programme 1 :* Illustrer l'exécution du programme suivant.

```
1 int main() {
2     fork();
3     fork();
4     printf("hello!\n");
5     exit(0);
6 }
```

- *Programme 2 :* Même question pour le programme suivant.

```
1 int main() {
2     fork();
3     fork();
4     fork();
5     printf("hello!\n");
6     exit(0);
7 }
```

Exercices

Question 3: Combien de lignes «hello!» imprime chacun des programmes suivants?

Programme 1 :

```
1  int main() {
2    int i;
3
4    for (i=0; i<2; i++)
5      fork();
6    printf("hello!\n");
7    exit(0);
8  }
```

Programme 2 :

```
1  void doit() {
2    fork();
3    fork();
4    printf("hello!\n");
5  }
6  int main() {
7    doit();
8    printf("hello!\n");
9    exit(0);
10 }
```

Programme 3 :

```
1  int main() {
2    if (fork())
3      fork();
4    printf("hello!\n");
5    exit(0);
6  }
```

Programme 4 :

```
1  int main() {
2    if (fork()==0) {
3      if (fork()) {
4        printf("hello!\n");
5      }
6    }
7  }
```

Terminaison d'un programme

Terminaison normale d'un processus

Un programme peut se terminer de deux façons différentes. La plus simple consiste à laisser le processus finir le main avec l'instruction **return** suivie du code de retour du programme. Une autre est de terminer le programme grâce à la fonction :

```c
#include <stdlib.h>

void exit(status);
```

Celle-ci a pour avantage de quitter le programme quel que soit la fonction dans laquelle on se trouve.

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

void quit(void)
{
    printf(" Nous sommes dans la fonction quit().\n");
    exit(EXIT_SUCCESS);
}

int main(void)
{
    quit();
    printf(" Nous sommes dans le main.\n");
    return EXIT_SUCCESS;
}
```

Terminaison anormale d'un processus

Pour quitter, de manière propre, un programme, lorsqu'un bug a été détecté, on utilise la fonction :

```c
#include <stdlib.h>

void abort(void);
```

Un prototype simple pour une fonction qui possède un défaut majeur : il est difficile de savoir à quel endroit du programme le bug a eu lieu.

Pour y remédier, il est préférable d'utiliser la macro assert, déclarée dans <assert.h> qui fonctionne comme suit :

- Elle prend en argument une condition.
- Si cette condition est vraie, assert ne fait rien.
- Si elle est fausse, la macro écrit un message contenant la condition concernée, puis quitte le programme. Cela permet d'obtenir une bonne gestion des erreurs.

Terminaison anormale d'un processus

N'utilisez assert que dans les cas critiques, dans lesquels votre programme ne peut pas continuer si la condition est fausse.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <assert.h>
#include <sys/types.h>

int main(void)
{
    pid_t pid_fils;

    do {
    pid_fils = fork();
    } while (pid_fils == -1 && (errno == EAGAIN));

    assert(pid_fils != -1);

    if (pid_fils == 0) {
    printf("\n Je suis le fils !\n");
    } else {
    printf("\n Je suis le père !\n");
    }

    return EXIT_SUCCESS;
}
```

Exécution de routines de terminaison

Routines de terminaison

Grâce à la programmation système, il est possible d'exécuter automatiquement telle ou telle fonction au moment où le programme se termine normalement, c'est-à-dire à l'aide des instructions exit et return.

Pour cela, deux fonctions sont disponibles : **atexit** et **on_exit**.

Il est à noter qu'il est préférable d'utiliser atexit plutôt que on_exit, la première étant conforme C89, ce qui n'est pas le cas de la seconde.

on_exit

```c
#include <stdlib.h>

int on_exit(void (*function) (int, void *), void *arg);
```

La fonction prend donc en paramètre deux arguments :

- un pointeur sur la fonction à exécuter, qui sera de la forme **void fonction(int codeRetour, void* argument)** . Le premier paramètre de cette routine est un entier correspondant au code transmis avec l'utilisation de return ou de exit.
- L'argument à passer à la fonction.

Elle renvoie 0 en cas de réussite ou -1 sinon.

Votre fonction de routine de terminaison recevra alors deux arguments : le premier est un int correspondant au code transmis à return ou à exit et le second est un pointeur générique correspondant à l'argument que l'on souhaitait faire parvenir à la routine grâce à **on_exit**.

atexit

```c
#include <stdlib.h>

int atexit(void (*function) (void));
```

Le paramètre est un pointeur de fonction vers la fonction à exécuter lors de la terminaison. Elle renvoie 0 en cas de réussite ou -1 sinon.

Vous pouvez également enregistrer plusieurs fonctions à la terminaison. Si c'est le cas, lors de la fin du programme, les fonctions mémorisées sont invoquées dans l'ordre inverse de l'enregistrement.

Atexit : Exemple 01

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

void routine(void)
{
    printf(" Execution de la routine de terminaison !\n");
    printf(" Au revoir !\n");
}

int main(void)
{
    if (atexit(routine) == -1) {
        perror("atexit :");
        return EXIT_FAILURE;
    }

    printf(" Fin du programme !\n");

    return EXIT_SUCCESS;
}
```

Atexit : Exemple 02

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

void routine1(void)
{
    printf(" Routine de terminaison 01\n");
}

void routine2(void)
{
    printf(" Routine de terminaison 02\n");
}

void routine3(void)
{
    printf(" Routine de terminaison 03\n");
}

int main(void)
{
    if (atexit(routine3) == -1) {
        perror("atexit :");
        return EXIT_FAILURE;
    }

    if (atexit(routine2) == -1) {
        perror("atexit :");
        return EXIT_FAILURE;
    }

    if (atexit(routine1) == -1) {
        perror("atexit :");
        return EXIT_FAILURE;
    }

    printf(" Fin du programme !\n");

    return EXIT_SUCCESS;
}
```

Communication par signaux

Signaux

Définition : évènement asynchrone

- Émis par l'OS ou un autre processus
- Destiné à un ou plusieurs processus

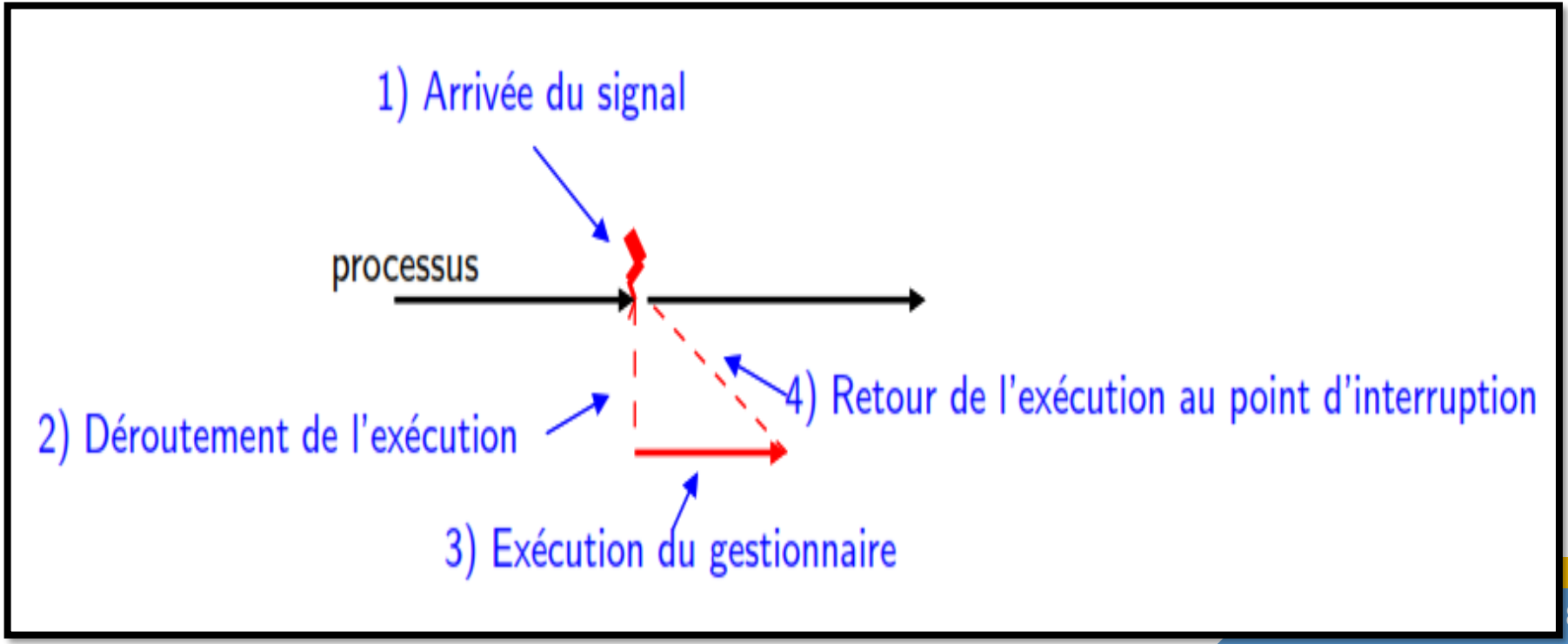
Intérêts et limites

- Simplifient le contrôle d'un ensemble de processus
- Pratiques pour traiter des évènements liés au temps
- Mécanisme de bas niveau à manipuler avec précaution

Comparaison avec les interruptions matérielles

- Analogie : la réception déclenche l'exécution d'un gestionnaire
- Différences : interruption reçue par processeur ; signal reçu par processus

Fonctionnement des signaux



Remarques sur les signaux

- On ne peut signaler que ses propres processus (même uid)
- Il existe différents signaux, identifiés chacun par un nom symbolique et une valeur entière
- Le système dispose d'un gestionnaire par défaut pour chacun
- Si le gestionnaire est vide, le signal est ignoré
- Il est possible de changer le gestionnaire (sauf exceptions)
- Il est possible de bloquer un signal : mise en attente, délivré après blocage
- Les traitements possibles dans le gestionnaire sont limités

Nom symbolique	Cause/signification	Par défaut
SIGINT	frappe du caractère <CTRL-C>	terminaison
SIGTSTP	frappe du caractère <CTRL-Z>	suspension
SIGSTOP	blocage d'un processus (*)	suspension
SIGCONT	continuation d'un processus stoppé	reprise
SIGTERM	demande de terminaison	terminaison
SIGKILL	terminaison immédiate (*)	terminaison
SIGSEGV	erreur de segmentation (violation de protection mémoire)	terminaison +*core dump*
SIGALRM	top d'horloge (réglée avec `alarm`)	teminaison
SIGCHLD	terminaison d'un fils	ignoré
SIGUSR1	pas utilisés par le système	terminaison
SIGUSR2	(disponibles pour l'utilisateur)	terminaison

Quelques exemples de signaux

- Les signaux SIGKILL et SIGSTOP ne peuvent pas être interceptés, bloqués ou ignorés. Leur gestionnaires est non modifiable.
- Consulter le résultat de man 7 signal pour plus d'informations sur les signaux

État d'un signal

- Signal pendant (pending)
  - Arrivé au destinataire, mais pas encore traité
- Signal traité
  - Le gestionnaire du signal a démarré (et peut-être même fini)
- Bloqué
  - Il est bloqué, ou retardé : il sera délivré lorsque débloqué. Lors de l'exécution du gestionnaire d'un signal, ce signal est bloqué.
- Attention : au plus un signal pendant de chaque type. L'information est codée sur un seul bit. S'il arrive un autre signal du même type, le second est perdu.

Envoyer un signal à un autre processus

- Interfaces
  - Commande
  - Programmation
- À qui envoyer le signal ?

```
kill -NOM victime
```

```
#include <signal.h>

int kill(pid_t victime, int sig);
```

  - Si victime > 0, kill() envoie le signal sig au processus dont le PID est égal à victime.
  - Si victime = 0, kill() envoie le signal sig à tous les processus dont le GID est égal à celui de l'expéditeur, à l'exception de ceux auxquels l'expéditeur n'a pas l'autorité appropriée pour envoyer un signal.
  - Si victime = -1 :
    - Si super utilisateur, kill() envoie le signal sig à tous les processus sauf système et émetteur
    - Si non, kill() envoie le signal sig à tous les processus dont l'utilisateur est propriétaire
  - Si victime < -1, kill() envoie le signal sig à tous les processus dont le GID est égal à la valeur absolue de victime, à l'exception de ceux auxquels l'expéditeur n'a pas l'autorité appropriée pour envoyer un signal.

Synchronisation entre processus père et fils

Processus Zombie

À la fin ou suspension d'un processus, il envoie automatiquement un signal SIGCHLD à son processus père.

Par défaut, ce signal est ignoré au niveau du processus père.

Le processus fils devient alors un processus **zombie** ou **orphelin**.

Processus Zombie

Au moment de la terminaison d'un processus, le système désalloue les ressources que possède encore celui-ci mais ne détruit pas son bloc de contrôle. Le système passe ensuite l'état du processus à la valeur TASK_ZOMBIE (représenté généralement par un Z dans la colonne " statut " lors du listage des processus par la commande ps). Le signal SIGCHLD est alors envoyé au processus père du processus qui s'est terminé, afin de l'informer de ce changement. Dès que le processus père a obtenu le code de fin du processus achevé au moyen de l'appel système wait, le processus terminé est définitivement supprimé de la table des processus.

Étant donné que les processus zombies ne peuvent pas être supprimés par les méthodes classiques (y compris pour les utilisateurs privilégiés), le système se retrouve alors encombré de processus achevés (" morts ") mais encore visibles. Ceux-ci ne consomment, à proprement parler, pas plus de ressources systèmes que les quelques octets de mémoire occupés par le bloc de contrôle dans la table des processus ; toutefois, le nombre de processus étant limité par le nombre possible de PID, un trop grand nombre de zombies peut empêcher le système de créer de nouveaux processus. Cette métaphore de horde de processus défunts, impossibles à tuer car déjà morts, est à l'origine du terme de " zombie ".

Processus Zombie

La seule manière d'éliminer ces processus zombies est de causer la mort du processus père, par exemple au moyen du signal SIGKILL.

Les processus fils sont alors automatiquement rattachés au processus n°1, généralement init, qui se charge à la place du père original d'appeler wait sur ces derniers. Si ce n'est pas le cas, cela signifie que init est défaillant (ou que le processus n°1 n'est pas init, mais un autre programme n'ayant pas été prévu pour ça).

Le seul moyen de se débarrasser des zombies, dans ce cas, est le redémarrage du système.

Éviter les processus zombie

Le processus doit attendre de lire les codes de retours de tous ses processus fils avant sa terminaison. Cela est possible grâce à l'utilisation de la fonction wait disponible dans le bibliothèque <sys/wait.h> déclarée comme :

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

pid_t wait(int *status);
```

Lorsque l'on appelle cette fonction, cette dernière **bloque le processus à partir duquel elle a été appelée jusqu'à ce qu'un de ses fils se termine**. Elle renvoie alors le PID de ce dernier.

En cas d'erreur, la fonction renvoie la valeur -1.

Attention : il faut mettre autant de wait qu'il y a de fils.

Éviter les processus zombie

Si **status** n'est pas un pointeur nul, le **status** du processus fils (valeur retournée par exit()) est mémorisé à l'emplacement pointé par **status**. De manière plus précise :

- l'octet de poids faible est un code indiquant pourquoi le processus fils s'est arrêté ;
- si le processus fils a effectué un appel à exit(), l'octet précédent contient le code de retour.

Ces informations peuvent être accédées facilement à l'aide des macros suivantes définies dans sys/wait.h :

- WIFEXITED (**status**) : renvoie vrai si le statut provient d'un processus fils qui s'est terminé normalement ;
- WEXITSTATUS (**status**) :(si WIFEXITED (**status**) renvoie vrai) renvoie le code de retour du processus fils passé à exit() ou la valeur retournée par la fonction main() ;
- WIFSIGNALED (**status**) :renvoie vrai si le statut provient d'un processus fils qui s'est terminé à cause de la réception d'un signal ;
- WTERMSIG (**status**) : (si WIFSIGNALED (status) renvoie vrai) renvoie la valeur du signal qui a provoqué la terminaison du processus fils.

Éviter les processus zombie

```c
/* La fonction father_process effectue les actions du processus père */
void father_process(int child_pid)
{
    printf(" Nous sommes dans le père !\n"
       " Le PID du fils est %d.\n"
       " Le PID du père est %d.\n", (int) child_pid, (int) getpid());

    if (wait(&status) == -1) {
        perror("wait :");
        exit(EXIT_FAILURE);
    }

    if (WIFEXITED(status)) {
    printf(" Terminaison normale du processus fils.\n"
           " Code de retour : %d.\n", WEXITSTATUS(status));
    }

    if (WIFSIGNALED(status)) {
    printf(" Terminaison anormale du processus fils.\n"
           " Tué par le signal : %d.\n", WTERMSIG(status));
    }

}
```

Attendre la fin de n'importe quel processus

Il existe également une fonction qui permet de suspendre l'exécution d'un processus père jusqu'à ce qu'un de ses fils, dont on doit passer le PID en paramètre, se termine. Il s'agit de la fonction **waitpid** :

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

pid_t waitpid(pid_t pid, int *status, int options);
```

Plus précisément, la valeur de pid est interprétée comme suit :

- si pid > 0, le processus père est suspendu jusqu'à la fin d'un processus fils dont le PID est égal à la valeur pid ;
- si pid = 0, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils appartenant à son groupe ;
- si pid = -1, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils ;
- si pid < -1, le processus père est suspendu jusqu'à la mort de n'importe lequel de ses fils dont le GID est égal.

Attendre la fin de n'importe quel processus

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

pid_t waitpid(pid_t pid, int *status, int options);
```

Le second argument, **status**, a le même rôle qu'avec wait.

Le troisième argument permet de préciser le comportement de **waitpid**. On peut utiliser deux constantes :

- WNOHANG : ne pas bloquer si aucun fils ne s'est terminé.
- WUNTRACED : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Dans le cas où cela ne nous intéresse pas, il suffit de mettre le paramètre 0.

Notez que waitpid(-1, status, 0) correspond à la fonction wait.

3

Exécution de programmes

Les fonctions de la famille exec

La famille exec

En réalité, il existe six fonctions appartenant à cette famille : **execl**, **execle**, **execlp**, **execv**, **execve** et **execvp** toutes disponibles dans la bibliothèque <unistd.h>.

Parmi eux, seule la fonction **execve** est un **appel système**, les autres sont implémentées à partir de celui-ci.

Ces fonctions permettent de remplacer un programme en cours par un autre programme sans en changer le PID. Autrement dit, on peut remplacer le code source d'un programme par celui d'un autre programme en faisant appel à une fonction **exec**.

La famille exec

- Suffixe en **-v** : les arguments sont passés sous forme de tableau ;
- Suffixe en **-l** : les arguments sont passés sous forme de liste ;
- Suffixe en **-p** : le fichier à exécuter est recherché à l'aide de la variable d'environnement **PATH** ;
- Pas de suffixe en **-p** : le fichier à exécuter est recherché relativement au répertoire de travail du processus père ;
- Suffixe en **-e** : un nouvel environnement est transmis au processus fils ;
- Pas de suffixe en **-e** : l'environnement du nouveau processus est déterminé à partir de la variable d'environnement externe environ du processus père.

La famille exec

```c
int execve(const char *filename, char *const argv[], char *const envp[]);
```

```c
int execl(const char *path, const char *arg,  ...);
```

```c
int execlp(const char *file, const char *arg, ...);
```

```c
int execle(const char *file, const char *arg, ..., char *const envp[]);
```

```c
int execv(const char *path, char *const argv[]);
```

```c
int execvp(const char *file, char *const argv[]);
```

La famille exec

Le premier argument correspond au chemin complet d'un fichier objet exécutable (si path) ou le nom de ce fichier (si file).

Le second argument correspond aux paramètres envoyés au fichier à exécuter : soit sous forme de liste de pointeurs sur des chaînes de caractères, soit sous forme de tableau. Le premier élément de la liste ou du tableau est le nom du fichier à exécuter, le dernier est un pointeur NULL.

Le troisième argument éventuel est une liste ou un tableau de pointeurs d'environnement.

De plus, toutes ces fonctions renvoient -1. errno peut correspondre à plusieurs constantes, dont EACCESS (vous n'avez pas les permissions nécessaires pour exécuter le programme), E2BIG (la liste d'argument est trop grande), ENOENT (le programme n'existe pas), ETXTBSY (le programme a été ouvert en écriture par d'autres processus), ENOMEM (pas assez de mémoire), ENOEXEC (le fichier exécutable n'a pas le bon format) ou encore ENOTDIR (le chemin d'accès contient un nom de répertoire incorrect).

La famille exec

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* On recupere les arguments */
int main(int argc, char *argv[])
{
    /* On cree un tableau contenant le nom du programme, l'argument et le
       dernier "NULL" obligatoire  */
    char *arguments[] = { "ping", "-c", argv[1], argv[2], NULL };

    /* On lance le programme  */
    if (execv("/usr/bin/ping", arguments) == -1) {
    perror("execv");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

La fonction system

La fonction system

La fonction system est semblable aux exec, mais elle est beaucoup plus simple d'utilisation.

En revanche, on ne peut pas y passer d'arguments.

Son prototype est :

```c
#include <stdlib.h>

int system(const char *command);
```

La fonction system comporte des failles de sécurité très importantes.

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

int main(void)
{
    system("clear");
    return EXIT_SUCCESS;
}
```

THANKS!

Any questions?
You can find me at
90 50 94 17
avenakaj@gmail.com