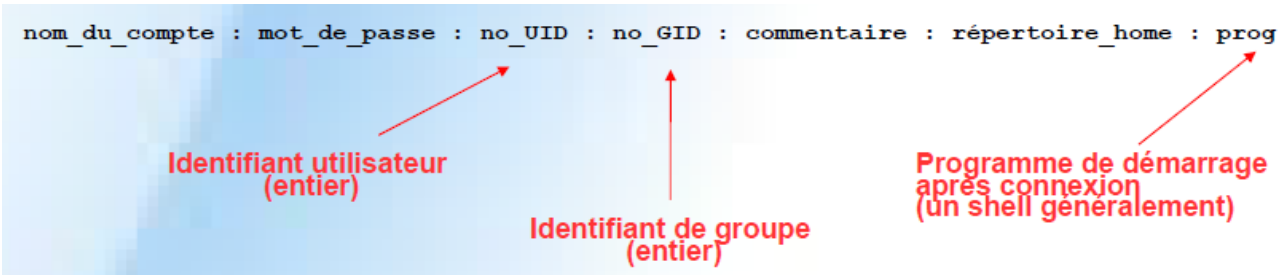
Matière : Système d'exploitation 1

TP : Gestion des processus sous UNIX

1. Utilisateurs et groupes

Le fichier **/etc/passwd** contient toutes les informations relatives aux utilisateurs (logins, mots de passe, ...).

Chacune de ses lignes possède le format spécial suivant :



Exemples :

root:12dGe12ge35qF:0:0:root:/users/root:/bin/bash

pascal:12dGeg5AqFdhr2:500:100:Pascal:/users/pascal:/bin/tcsh

2. Les droits d'accès :

2.1 catégories d'utilisateurs :

***Propriétaire** (user)*	u
***Groupe** (group)*	g
***Autres** (others)*	o

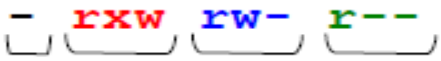
2.2 types de droits :

***Lecture** (read)*	r
***Ecriture** (write)*	w
***Exécution** (execute)*	x

2.3 types de fichier :

Ordinaire	-
***Répertoire** (directory)*	d
***Lien symbolique** (link)*	l
Spécial	c ou b

Les droits d'un fichier sont représentés par une chaine de 10 caractères :



Propriétaire groupe autres

La commande **ls** permet de visualiser le contenu du répertoire. Avec l'option **l**, la commande **ls -l** permet de visualiser les droits et avec l'option **a**, la commande **ls -a** permet de visualiser tous les fichiers contenus dans le répertoire courant.

3. **Caractéristiques d'un processus UNIX :** Un processus possède de nombreuses caractéristiques :

- le programme qu'il exécute (**CMD**)
- un numéro d'identification que lui affecte le système (**PID**)
- un créateur (**PPID**)
- l'utilisateur pour lequel il fonctionne (**UID**)
- le terminal ou la fenêtre du processus (**TTY**)
- une consommation CPU (**CPU**)
- une consommation mémoire (**MEM**)
- une durée de traitement (**TIME**)
- une heure de lancement (**STIME**)
- un facteur de priorité (**C**) ou **PR**...

4. **Types de processus UNIX :** Processus *utilisateur* vs. Processus *système (démons*)

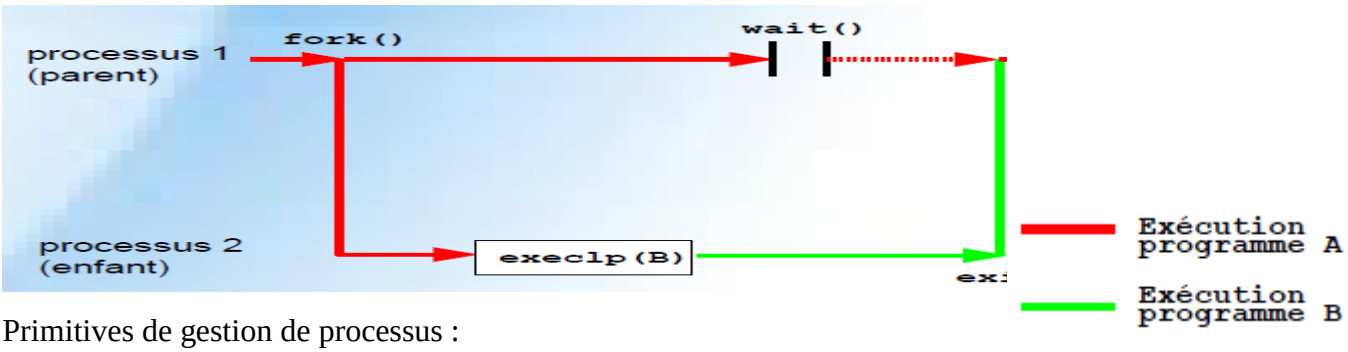
Un processus est créé par un autre processus Exemple d'arborescence de processus :

Commandes UNIX de gestion des processus :

- Liste des processus satisfaisant un critère donné (spécifié en option) : **ps** *[-options]*
- **ps -aux** : Visualisation de programmes en cours d'exécution avec leur identifiant (PID)
- **kill** PID : Terminaison immédiate du processus de pid PID
- Changement de la priorité d'un processus **: nice** ***-valeur commande***

  Exp : nice -5 gcc programme.c
- Destruction d'un processus : **kill -9** *no_processus* **Exp : kill -9 521**

Manipulation de processus en langage C



Primitives de gestion de processus :

- int **fork** (); création de processus
- int **execlp** (char *comm, char *arg0, ..., NULL);
- void **exit** (int status); terminaison d'un processus
- int **wait** (int *ptr_status); attente de la terminaison d'une processus fils
- pid_t **waitpid**(pid_t pid, int *status, int opts)
- int **sleep** (int seconds);
- int **getpid** (); identification d'un processus
- int **getppid** ();identification le père d'un processus

Identité d'un processus

Un processus est identifié par son pid. Il peut connaître le pid de son père. Les deux primitives retournent respectivement le pid du processus et de son père.

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

pid_t getpid(void);
pid_t getppid(void);
```

Terminaison du processus courant

Le processus courant se termine automatiquement lorsqu'il cesse d'exécuter la fonction main(). Les primitives suivantes lui permettent d'arrêter explicitement son exécution :

```
#include <unistd.h>

void _exit (int status) ;
void exit (int status) ;
```

Ces deux primitives provoquent la terminaison du processus courant. Le paramètre ***status*** spécifie un code de retour, compris entre 0 et 255, à communiquer au processus père.

Par convention, en cas de terminaison normale, un processus doit retourner la valeur 0.

Avant de terminer l'exécution du processus, exit() exécute les fonctions de « nettoyage » des librairies standard.

Attente de terminaison d'un processus fils

La primitive : wait() suspend l'exécution du processus appelant jusqu'à ce qu'un de ses processus fils se termine. Si un processus fils s'est déjà terminé, `wait()` retourne le résultat immédiatement.

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

pid_t wait (int *status) ;
```

`wait()` retourne le pid du processus fils si le retour est dû à la terminaison d'un processus fils ; -1 en cas d'erreur.

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`.

Exercice (à partir du shell)

- Ouvrez une fenêtre shell et taper les commandes ps, ps -e, ps -f puis ps -ef.
- Que font ces commandes. Expliquez les résultats obtenus.

Solution

La commande **ps** permet de lister les processus en cours d'exécution. Sans arguments, ne sont listés que les processus exécutés à partir du shell en cours d'exécution, lui compris. Le programme shell utilisé s'appelle bash. On doit donc noter la présence de deux processus en cours d'exécution : ***l'interpréter bash*** et la ***commande ps*** elle-même.
L'argument **-e** permet d'afficher tous les processus et l'argument **-f** ajoute des informations concernant l'environnement d'exécution des processus. Les deux arguments peuvent être cumulés par simple concaténation **(-ef)**

Comment sont créés les processus

C'est l'appel système **fork()** qui crée un processus **fils**. Le père et le fils ont le même code et s'exécutent en même temps à partir du retour du fork(). Dans le père, la valeur de retour de fork() est le numéro (**pid**) du fils (**jamais nul**). Dans le fils, c'est **0**. D'où la forme standard pour utiliser fork() :

```
int pid ;
pid = fork();
if (pid != 0) {/* code du père */}
else { /* code du fils */}
```

Exemple :

```
main() {
int pid ;
pid = fork();
if (pid != 0) { /* père */    printf("Je suis le processus pere");}
else { /* fils */       printf("Je suis le processus fils");     }
```

Exercice (fork)

Ecrire un programme qui utilise une fourche pour créer un programme fils (fonction fork(). Le processus père affichera à l'écran le message "je suis le père de PID : XXXX", tandis que le processus fils affichera "je suis le fils de PID : YYYY" (utiliser la fonction int getpid()).

Corrigé :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
int pid;
pid = fork();
if ( pid==-1 ) {
        perror("Exercice 9.\n");
        exit(1);
                }
else if ( pid==0) {
        // processus fils
        printf("Je suis le fils de PID %d\n", getpid() );
                }
```

```
else {
        printf("Je suis le Pere de PID %d\n", getpid() );
        }
}
```

fonction execl()

La fonction execl() est un mécanisme complémentaire à la création de processus implanté via la fonction fork() dont dispose le système Unix.

Ecrire un programme appelé bonjour dont le code afficher à l'écran la chaine de caractères suivante : "bonjour". Ecrire puis exécuter le deuxième programme suivant : programme bonjour.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
printf("preambule du bonjour. \n");
execl("./bonjour", (char*) 0);
printf("postambule du bonjour. \n");
}
```

Que peut-on conclure ?

Corrigé (programme bonjour.c) :

```
#include <stdio.h>
int main() {
printf("Bonjour\n");
}
```

Seuls les affichages des textes "**preambule du bonjour**." et "**Bonjour**" apparaissent.

Explication : la fonction execl() lance l'exécution du programme bonjour qui est chargé en mémoire a la place du programme appelant. Ce dernier ne reprend donc pas son exécution lorsque le programme appelé a terminé la sienne, et le dernier affichage "postambule du bonjour" ne peut donc avoir lieu.

Exercice : Qu'affiche l'exécution du programme suivant :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
 int main ()
{
int pid; int x = 1;
pid = fork();
if (pid == 0) {
        printf("x=%d\n", ++x);
        exit(0);
            }
printf(" x=%d\n", --x);
}
```

Que serai l'affichage si on enlève l'instruction exit(0) ?