

Introduction Système Exploitation: Processus

BUT 2

Abdelkader Gouaïch
gouaich@lirmm.fr

INTRODUCTION

DÉFINITIONS: PROCESSUS, PROGRAMME ET THREAD

Programme et Processus

Le concept de processus : une abstraction fondamentale pour Unix

Distinguer entre un *programme* et un *processus*.

Programme: un fichier exécutable résultat de la phase de compilation.

Processus = une activité

Programme et Processus

Pour son exécution un processus à besoin:

- du tableau d'instructions à exécuter
- mémoire de travail modifiée par les instructions
- de ressources annexes offertes par l'OS
- de fichiers ouverts
- des droits pour déterminer les actions permises
- d'un ou plusieurs thread d'exécution

Processus et thread

Processus = activité

Le *thread* est le moteur de cette activité.

Il permet d'interpréter les instructions sur une CPU pour changer l'état de la mémoire.

Processus et thread

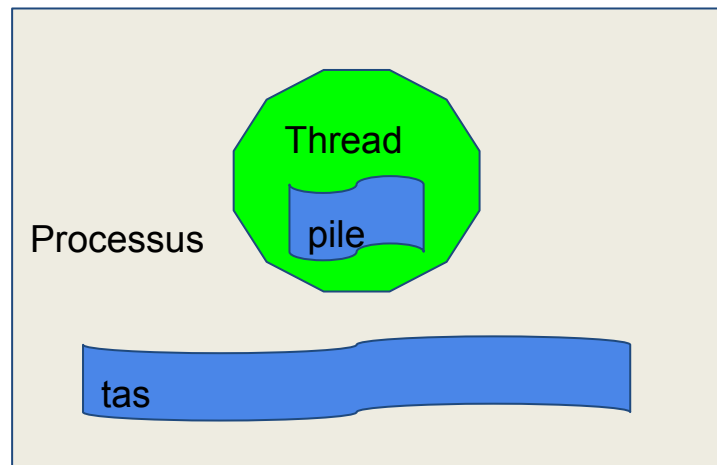
Chaque thread dispose:

- d'un espace mémoire **propre** pour les variables des fonctions (appelé la *pile*)
- de l'état du processeur: des *registres* du processeur
- d'un pointeur (adresse) vers la prochaine instruction à exécuter

Processus et thread

La programmation **monothread**: 1 thread

Le processus ~ son thread principal.



Processus et thread

La programmation **multi thread**: n threads

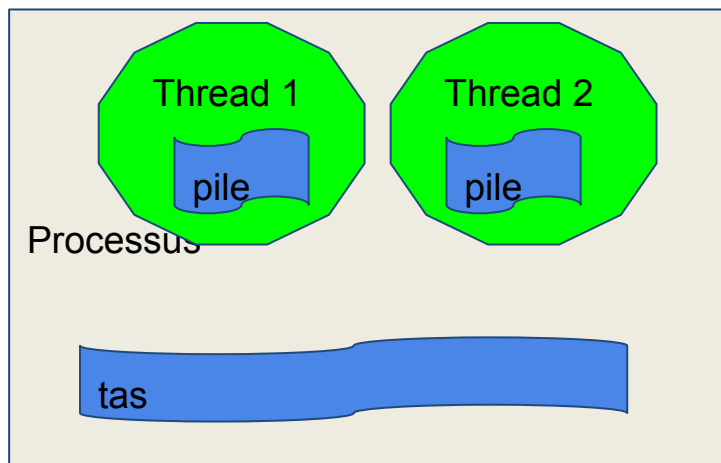
Distinction entre:

- le processus,
- son thread principal
- les autres threads lancés au *runtime*.

Processus et thread

La programmation **multi thread**:

Pour que l'exécution des différentes activités soit cohérente => chaque thread va disposer de son espace privé pour la gestion des appels des fonctions et les registres du processeur.

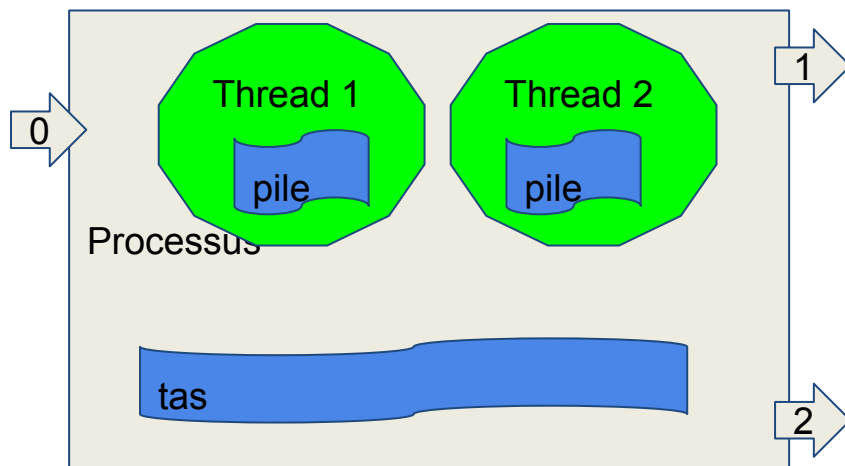


Processus et thread

La programmation **multi thread**:

Les threads vont partager un espace de travail commun appelé le *tas*.

Les ressources (fichiers ouverts) sont partagées



Le process ID

OS associe à chaque nouveau processus un identifiant unique.

Process ID ou PID: un entier unique à un moment donné.



le PID d'un processus terminé peut être réutilisé

Le process ID

Des processus particuliers:

- le process *idle*, de PID 0 => aucune activité n'est en cours d'exécution
- le process *init*, de PID 1 => est le premier processus après le *boot*.
- le process *init* crée les autres processus.

Le programme de init:

- `/sbin/init`
- `/etc/init`

Le PID

Le PID est un entier de type `pid_t`.

Renommage du type `int`.

Avec cette technique: le compilateur sait que ce sont des PIDs et pas des entiers ordinaires

Le PID

Signature de la fonction C avoir le PID:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid (void);
```

Le PID

Pour récupérer le PID du processus parent:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid (void);
```


Le PID

Pour afficher ces valeurs:

```
printf ("Mon PID=%jd\n", (intmax_t) getpid ());
```

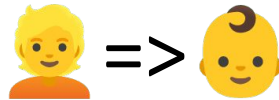
```
printf ("PID du Parent =%jd\n", (intmax_t)  
getppid ());
```

LA HIÉRARCHIE DES PROCESSUS

Relation Père Fils

Un processus peut engendrer un nouveau processus.

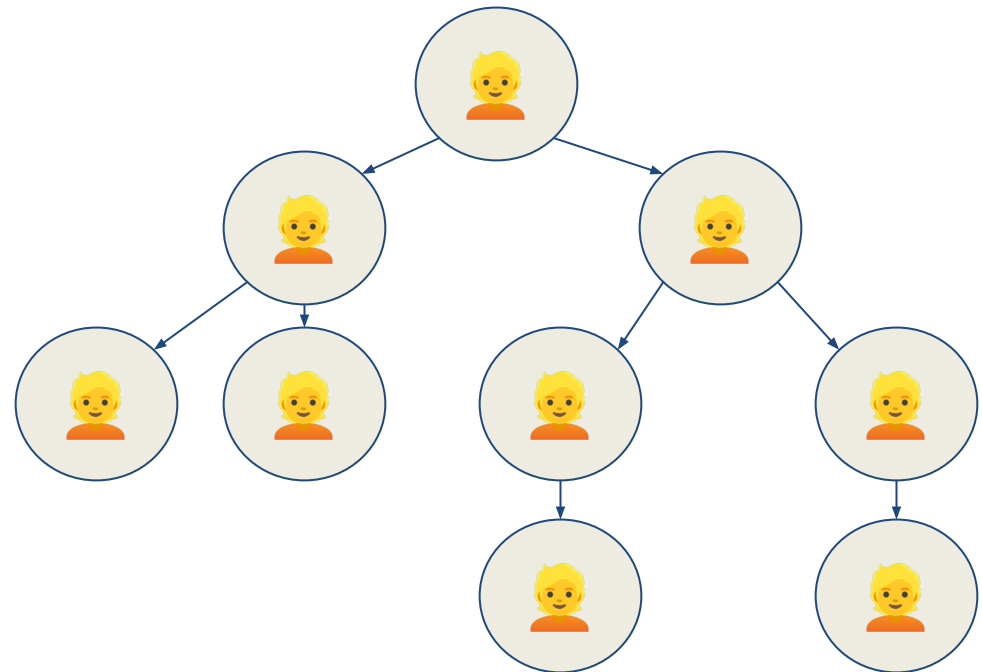
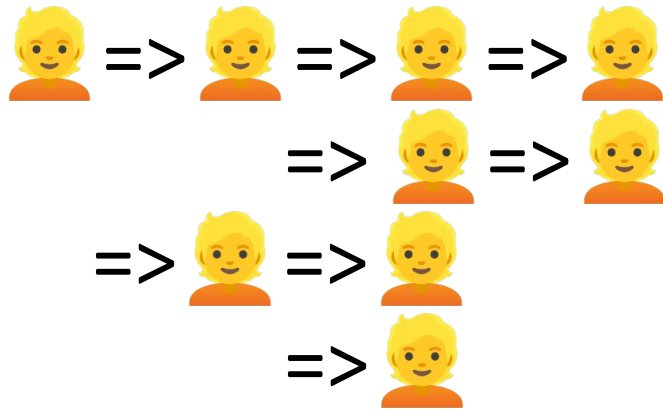
Le processus à l'origine de la création est appelé le *père* et le processus engendré est appelé le *fils*



Le process Init

Chaque processus dans le système est généré par un autre.

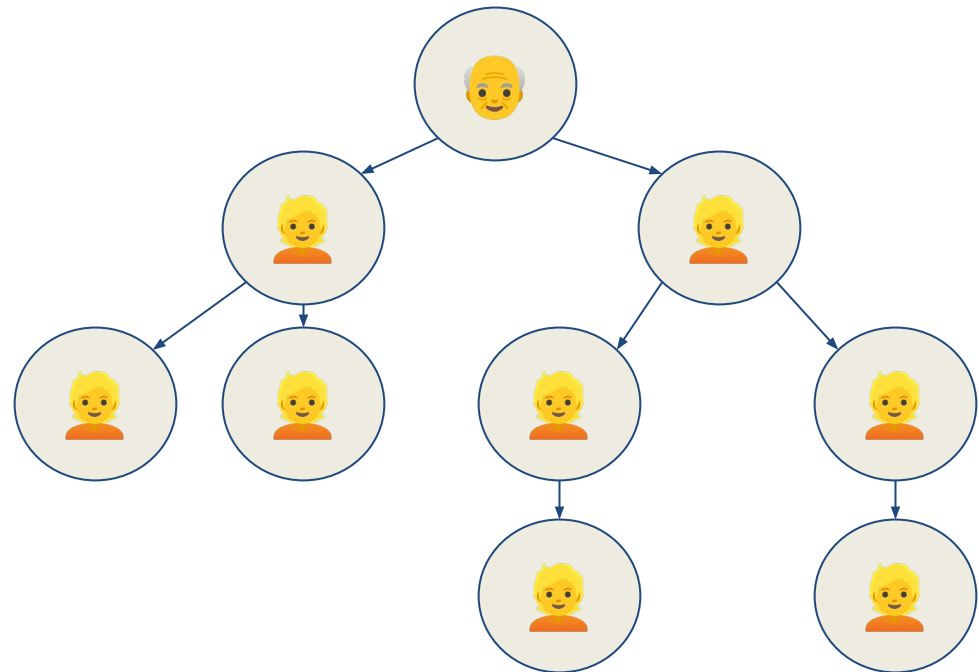
La seule exception concerne le processus init (PID=1)



Représentation arborescente

La relation père/fils peut se représenter comme un arbre hiérarchique :

- la racine de l'arbre est le processus init
- chaque processus ne possède qu'un seul père connu par son PPID



Gestion des droits

Un processus est associé à un utilisateur UID et à un groupe d'utilisateurs GID.

Un processus fils hérite de l'UID et du GID de son père à la création

UID et GID sont utilisés pour les gérés les droits d'accès aux ressources

CRÉATION DE NOUVEAUX PROCESSUS

Introduction

Unix distingue :

- le chargement d'un programme en mémoire pour son exécution
- la création d'un nouveau processus par duplication.

Chargement image

L'image du nouveau programme est chargée et *remplace* l'ancienne image.

L'exécution du processus continue ainsi avec la nouvelle image.

Nous remarquons que le PID du processus n'est pas modifié => même processus

Réalisation par les fonctions `exec`.

Duplication

Conservation du programme exécutable mais un *nouveau* processus fils est créé avec un PID différent.

Pour traiter les cas de duplication des processus, nous utilisons la famille des fonctions `fork`.

Appel système exec

La famille exec

plusieurs fonctions qui jouent le même rôle:
remplacer l'image du programme avec une
nouvelle.

Appel système exec

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

`execl()` remplace le programme par le programme indiqué par `path`.

Les nouveaux arguments du processus sont indiqués à partir de `arg`.

Appel système exec

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

Le paramètre `arg` : le nom que nous souhaitons donner au processus.

Les autres arguments sont indiqués à sa suite comme des valeurs séparées par des virgules.

La constante `NULL` termine la liste des paramètres.

Appel système exec

```
int ret;  
ret = execl ("/bin/ls", "monls", NULL);  
if (ret == -1) perror ("execl");
```

Remplace le processus courant qui exécute *ce* programme C par le programme se trouvant à “/bin/ls”

Appel système exec

Héritage après exec

Les valeurs suivantes sont conservées après un `exec`:

- le pid du processus
- le ppid du processus
- le UID et le GID associés au processus
- l'ensemble des fichiers ouverts

Appel système fork()

😞 fork: une bifurcation

La signature:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```


Appel système fork()

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

fork() va créer un nouveau processus:

- la même image du programme
- le ppid est identique au pid du père
- les signaux en attente sont réinitialisés
- les fichiers ouverts sont hérités

Appel système fork()

Exemple

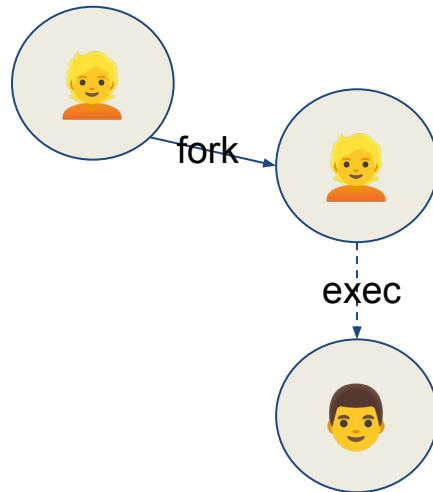
```
pid_t pid;
pid = fork ();
if (pid > 0)
{
    printf ("parent pid=%d!\n", pid);
}
else {
    if (!pid) {
        printf ("enfant!\n");
        else {
            if (pid == -1) {perror ("fork");}
        }
    }
}
```

Combinaison fork/exec

Combinaison des deux fonctions : fork/exec.

`fork()` va prendre en charge la création du nouveau processus.

`exec()` va charger une nouvelle image du programme.



Example

```
pid_t pid;
pid = fork ();
if (pid == -1) perror ("fork"); /* fils */
if (!pid)
{
    const char *args[] = { "ls", NULL };
    int ret;
    ret = execv ("/bin/ls", args);
    if (ret == -1) {
        perror ("execv");
        exit (EXIT_FAILURE);
    }
}
```

TERMINER UN PROCESSUS

exit()

Mettre fin à un processus `exit()` :

```
#include <stdlib.h>  
void exit (int status);
```

exit()

L'appel à exit va permettre au kernel de réaliser les opérations suivantes:

- appel des triggers
- écriture des données I/O
- destruction des fichiers temporaires
- libération des ressources allouées par le noyau: mémoire, fichiers ouverts, sémaphores.
- notification au parent de la fin de son processus fils

Enregistrement d'un trigger sur terminaison

trigger: une fonction déclenchée par un événement particulier

La fonction d'enregistrement du trigger:

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

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


Enregistrement d'un trigger sur terminaison

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

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

Le type `void (*function) (void)`

- un type de fonction `(void)=>(void)`

Le trigger est appelé:

- `exit()` est appelée
- `main()` fait un `return`.

Enregistrement d'un trigger sur terminaison

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

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

En cas de `exec` : tous les triggers enregistrés sont effacés.

Si le processus termine à cause d'un signal, alors les triggers ne seront pas appelés.

Enregistrement d'un trigger sur terminaison

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

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

Enregistrer plusieurs triggers avec plusieurs appels de atexit.

L'ordre d'appel des fonctions est LIFO (Last In First Out)

Enregistrement d'un trigger sur terminaison

```
#include <stdio.h>
#include <stdlib.h>
void out (void)
{
    printf ("atexit() fonctionne bien!\n");
}
int main (void)
{
    if (atexit (out)) fprintf(stderr, "atexit()
erreur!\n");

    return 0;
}
```

Attendre la fin des processus fils

Un processus parent peut être intéressé par l'état de terminaison de son processus fils

- le fils va terminer son traitement
- nous devons garder certaines informations pour le père.

Cet état post terminaison du processus fils est appelé l'état *zombie*.

Attendre la fin des processus fils

Pour lire consulter l'état de terminaison du processus fils, nous allons utiliser la fonction `wait()`:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
```

Attendre la fin des processus fils

```
pid_t wait (int *status);
```

L'appel à cette fonction est bloquant.

Le processus appelant va attendre qu'un de ces fils termine pour lire le code de terminaison.

Nous allons obtenir le PID du fils concerné comme valeur retour de la fonction `wait`.

Attendre la fin des processus fils

```
pid_t wait (int *status);
```

Interpretation du status

Si le pointeur `int* status != NULL` alors nous pouvons extraire des information du fils.

Attendre la fin des processus fils

```
pid_t wait (int *status);
```

Les fonctions suivantes
pour interpréter le contenu
de `int* status`:

```
#include <sys/wait.h>
int WIFEXITED (status);
int WIFSIGNALED
(status);
int WIFSTOPPED (status);
int WIFCONTINUED
(status);
int WEXITSTATUS
(status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

Attendre la fin des processus fils

```
pid_t wait (int *status);
```

WIFEXITED

WIFEXITED (status) retourne vrai => processus fils a terminé normalement avec un appel à exit().

Dans ce cas, la macro `int WEXITSTATUS (status)` ; permet de lire la valeur donnée comme paramètre à exit(); l'information, juste un octet, passe donc du fils au père.

Attendre la fin des processus fils

```
pid_t wait (int *status);
```

WIFSIGNALED

WIFSIGNALED (status) est vrai
=> la fin du processus fils a été causée par la
réception d'un signal

Dans ce cas, la macro WTERMSIG retourne le
numéro du signal reçu par le fils.

Attendre la fin des processus fils

Exemple

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (void)
{
    int status;
    pid_t pid;
    if (!fork ()) return 1; //fils termine de suite
    pid = wait (&status);
    if (pid == -1) perror ("wait");
    printf ("pid=%d\n", pid);

    if (WIFEXITED (status)) printf ("exit status=%d\n", WEXITSTATUS (status));

    if (WIFSIGNALED (status)) {
        printf ("Terminaison par un signal=%d%s\n", WTERMSIG (status), WCOREDUMP
(status) ? " (dumped core)" : "");
    }

    if (WIFSTOPPED (status)) printf ("Stope par un signal=%d\n", WSTOPSIG (status));

    if (WIFCONTINUED (status)) printf ("Continue\n");

    return 0;
}
```

Attendre la fin des processus fils

Pour attendre un fils particulier, connaissant son PID nous pouvons utiliser la fonction suivante:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options);
```

LA GESTION DES DROITS

Utilisateurs et Groupes

Unix est un système d'exploitation multi utilisateurs.

Chaque utilisateur est identifié par un entier unique *UID*.

Il existe également des groupes et chaque groupe est identifié par un entier unique le *GID*.

Utilisateurs et Groupes

La correspondance entre les identifiants, les entiers, et les noms symboliques:

- `/etc/passwd`
- `/etc/group`

Les différents User IDs

real user id:

UID de l'utilisateur qui a *lancé* le processus.
Cette valeur est copiée lors d'un `fork` et ne change pas avec `exec`.

Seul le root peut modifier cette valeur et les processus utilisateurs ne peuvent pas la modifier.

Les différents User IDs

effective user id

Le UID utilisé par le noyau

Au départ, cette valeur = real user id

• ➡ Conservée par `fork` et `exec`.

Les différents User IDs

effective user id

`setuid` va permettre de la mettre à la valeur de `effective user id` à celle du propriétaire du programme

Après `exec` si le programme binaire chargé exécute `setuid()` alors le processus qui était à l'origine de l'`exec` va voir son `effective user id` modifié = le propriétaire du fichier exécutable

Les différents User IDs

effective user id

Exemple:

Si nous lançons le programme

`/usr/bin/passwd` avec `exec` alors notre processus va avoir comme effective user id le root.

Ceci va permettre à un processus lancé par un utilisateur simple d'accéder et de modifier un fichier protégé qui contient les mots de passe.

Les différents User IDs

saved user id

Cette valeur est une sauvegarde de l'effective user id avant sa modification.

Cette valeur est héritée par `fork`.

Dans le cas d'un `exec`, cette valeur récupère automatiquement l'effective user id courant.

Seul `root` peut modifier cette valeur.

LES SESSIONS ET LES GROUPES DE PROCESSUS

Introduction

Chaque processus est membre d'un groupe appelé 'process group'.

process group = agrégat de processus pour une tâche complexe.

Nous appelons cette tâche un 'job'.

Job

Un avantage de regrouper les processus dans un groupe est de pouvoir communiquer plus simplement avec l'ensemble des processus impliqués dans le job.

Un signal au groupe => signal transmis à l'ensemble des processus du groupe.

PGID

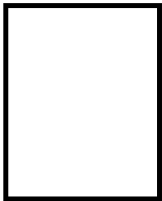
Un groupe de processus possède un identifiant (pgid) et un processus leader. Le pgid va correspondre au PID du leader.

Le groupe de processus existera tant qu'il y a un moins un processus en cours d'exécution.

Cela veut dire que le groupe existera même si son leader a terminé.

Les sessions

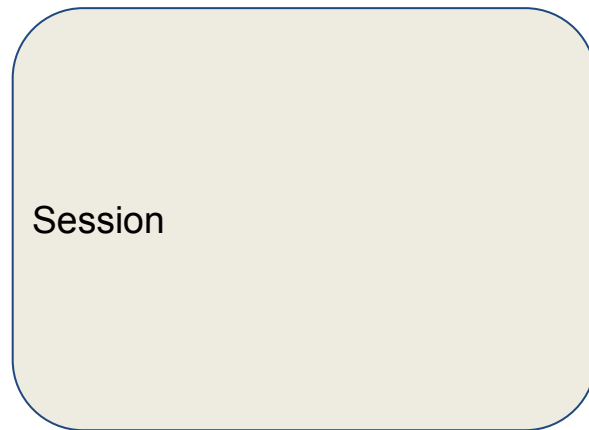
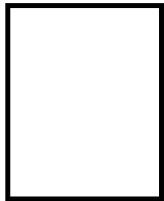
Un *terminal* est l'interface par laquelle l'utilisateur interagit avec le système d'exploitation en saisissant des commandes et en consultant les résultats des commandes en texte.



Les sessions

Une *session* est une collection de groupes de processus.

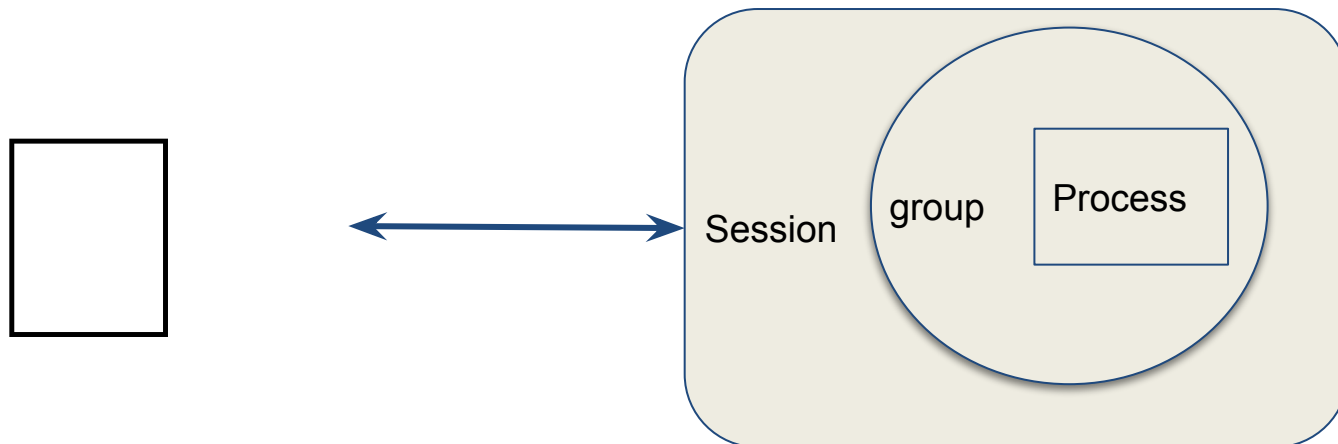
Le but d'une session est de faciliter l'interaction entre le terminal et les groupes de processus de l'utilisateur.



Les sessions

Utilisateur se connecte, le processus de login va créer une nouvelle *session*.

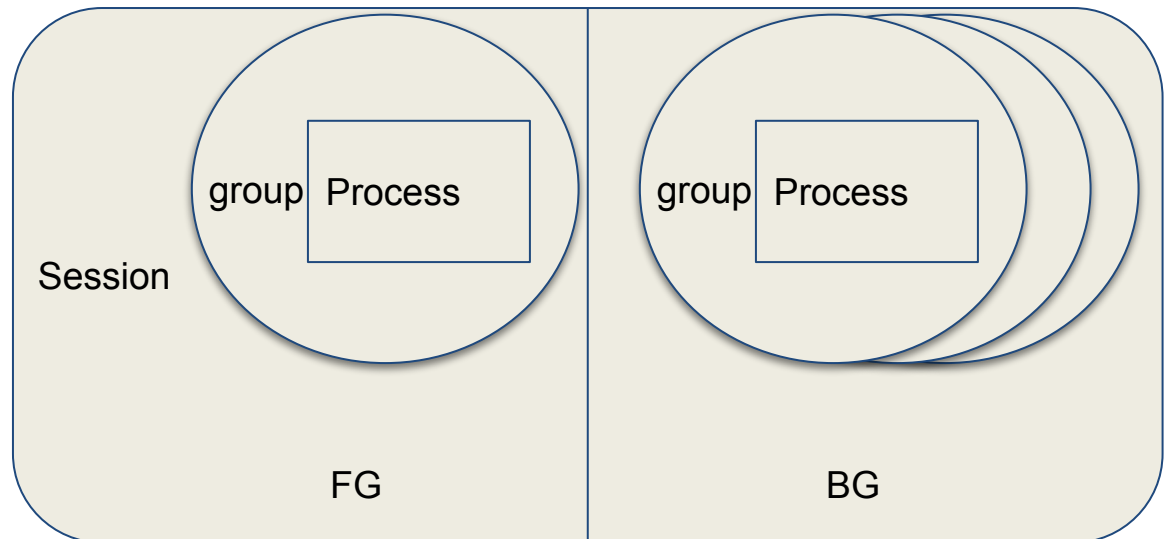
Cette session va contenir uniquement le processus de login qui sera son leader.



Les sessions

Foreground et Background

Dans une session, nous avons un seul groupe de processus en premier plan (foreground) et 0-n groupes de processus en arrière plan (background).



Les sessions

Foreground et Background

A la fin d'une session, un signal de fin (`SIGQUIT`) est envoyé uniquement au groupe foreground.

Les autres processus en background peuvent donc continuer leurs activités.

Les sessions

Foreground et Background


Si le terminal détecte une combinaison de touches indiquant une interruption (généralement un Ctrl^C) alors un signal d'interruption `SIGINT` est envoyé à tous les processus foreground.

Les processus background sont encore une fois épargnés.

Les sessions

Nous pouvons aussi créer de nouvelles sessions qui ne sont pas liées au processus de login.

Les groupes de processus dans ces sessions sont donc indépendants des actions de connexion et déconnexion des utilisateurs.

Cette fonctionnalité sera très utile pour les daemons 

Exemple: daemon de gestion du réseau