05 - Processus

Gaétan Richard
gaetan.richard@info.unicaen.fr

L2 – S4 2011/2012

1. Présentation

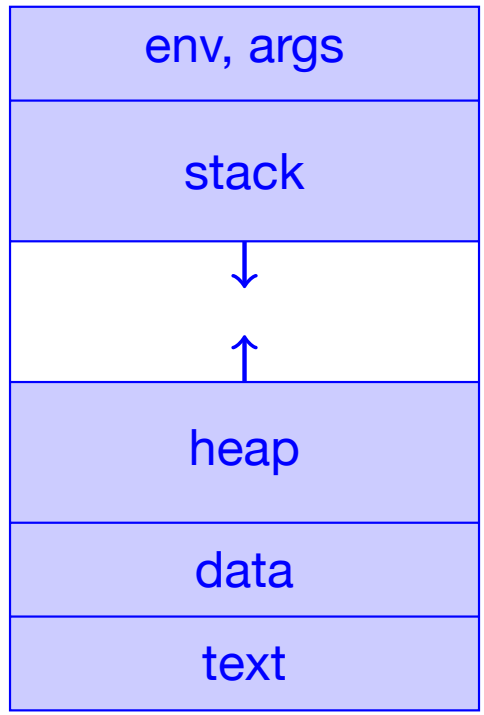
Lancement

Rappel : Sur le disque, on dispose de fichiers contenant du code exécutable. Pour pouvoir exécuter le code correspondant, il est nécessaire de disposer du droit d'exécution (bit x).

Nous verrons le mécanisme permettant le lancement de nouveau processus en détail plus loin.

Mise en mémoire

L'organisation en mémoire d'un processus est représentée dans la figure ci-dessous.



Méta-informations

- un propriétaire, un groupe ;
- un identifiant (pid) ;
- l'identifiant de son père (ppid) ;
- une table des descripteurs de fichiers ;
- des informations pour l'ordonnancement (priorité, …) ;
- des informations pour les ressources (limites) ;
- une table des signaux reçus (*cf cours sur les signaux*) ;
- …

bit setuid / setgid

Pour déterminer les droits d'un processus, on regarde le propriétaire et le groupe **effectif**. En général, ceux-ci sont les mêmes que le propriétaire et groupe sauf en cas de bit **setuid** ou **setgid**.

Méta-informations du shell

Il est possible d'accéder à ces informations aux travers des variables :

- **UID** et **GID** contiennent l'identifiant de l'utilisateur et du groupe ;
- **$** contient le PID ;
- **PPID** contient le PID du père.

Il est également possible d'avoir accès à l'identifiant de l'utilisateur par les commandes **id** et **whoami**.

Il est possible de changer le groupe d'un processus avec les commandes **su** et **sudo**.

Pour les autres processus

Il est possible d'obtenir les méta-informations des processus par l'intermédiaire de la commande **ps**.

Il existe également la commande **pstree** qui permet d'afficher le résultat sous forme d'un arbre.

Procfs : Sous Linux, ces informations sont également accessibles par l'intermédiaire du système de fichiers monté dans /proc. Chaque processus disposant d'un répertoire au nom de son *pid*.

Getpid and co

En python, il existe les fonctions pour récupérer ces informations :

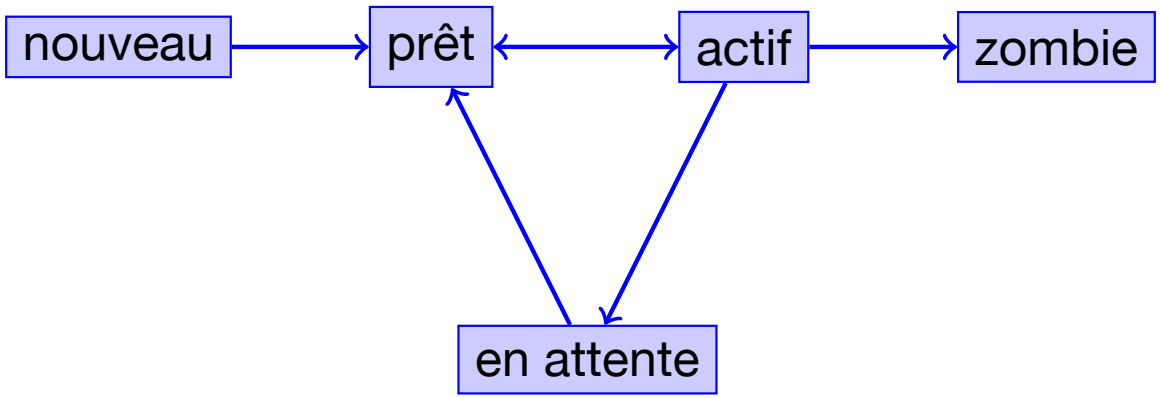
- **os.getpid()**,
- **os.getppid()**,
- **os.getuid()**,**os.geteuid()**,
- **os.getgid()**,**os.getegid()**,**os.getpgid(*pid*)**

Il est également possible de modifier ces valeurs (sous réserve d'en avoir les droits) à l'aide des versions **setuid**,**seteuid**, **setegid**, …

2. États

États

En simplifiant, un processus peut être dans les états suivants :



Un processus peut également être envoyé dans le swap.

Observer les processus

La commande **top** permet d'afficher les informations sur les processus en cours d'exécution sur la machine. Il existe également parfois des variantes plus évoluées de cette commande (**htop**).

Limites

Il est possible et courant d'imposer des limites sur les processus. Ces limites peuvent prendre la forme :

- du temps CPU utilisé ;
- du nombre de fichiers ouverts ;
- de la taille des fichiers ouverts ;
- de la taille du tas et de la pile ;
- du nombre de processus fils ;
- …

Il existe deux types de limites :

- les soft limits qui sont celles par défaut d'un processus crée ;
- les hard limits qui sont celles maximales que l'utilisateur peut fixer.

Manipulation des limites

Les limites en shell se manipulent à l'aide de la commande **ulimit**.

En python, ces fonctionnalités sont disponibles par l'intermédiaire du module **resource** et notamment des fonctions :

- **resource.getrlimit(*resource*)**
- **resource.setrlimit(*resource*, *limits*)**

3. Ordonnancement

Principes

Un ordinateur possède un (ou quelques) processeur(s).

Chaque processeur peut être composé d'une ou plusieurs unité(s) de calcul : les cœurs

Le système d'exploitation permet d'avoir évidement plus de processus que juste le nombre de cœurs.

Fonctionnement : Le système d'exploitation donne accèss aux processus à une unité de calcul : l'**ordonnancement**.

Découpage

Une fois le processus sur une unité de calcul, il existe plusieurs cas dans lesquels le processus s'en sépare :

- Le processus s'arrête ;
- le processus le demande explicitement ;
- le processus est en attente (cas d'un appel système) ;
- le système choisit d'arrêter le processus (système préemptif).

Dans ce cas, le système d'exploitation détermine un nouveaux processus à affecter à l'unité de calcul.

Modèles simples

Il existe plusieurs techniques pour déterminer comment affecter les tâches aux unités de calculs.

- **FIFO** : on affecte dans l'ordre d'apparition ;
- **Round Robin** : on effectue un bloc de chaque processus à tour de rôle ;
- **Priorité** : chaque processus dispose d'une valeur de priorité et on choisit le processus de plus forte priorité à chaque fois.

Modèle évolués

Le modèle actuel dans la plupart des systèmes d'exploitation est basé sur les principes suivants :

- chaque processus possède une priorité de base ;
- cette priorité augmente quand le processus est inactif et diminue quand il est actif (le taux de changement dépend de la priorité de base) ;
- le système choisit parmi les processus de plus forte priorité.

Note : Ce système est également couplé à un autre système pour assurer des capacités temps-réel.

Nice

Sous UNIX, il est possible de changer la priorité de base d'un processus. Cette valeur est comprise entre -20 et 20 (plus le nombre est bas, plus la priorité est grande).

Il est toujours possible de diminuer la priorité d'un processus, l'augmentation demande des droits particuliers (`root`).

Avec le shell, il est possible d'effectuer ces opérations à l'aide des commandes **nice** au lancement et **renice** durant l'exécution.

Sous python, cette fonctionnalité est accessible par l'appel système : **os.nice(*increment*)**.

Sleep / yield

Un processus peut demander à dormir pendant un certain temps à l'aide de la commande **sleep**.

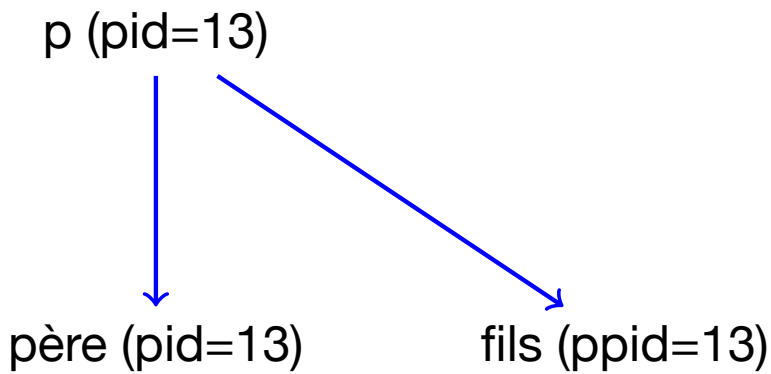
En python, cette fonctionnalité est accessible par la commande **time.sleep(*secs*)**.

En python, il existe aussi le mot-clé **yield** qui permet d'interrompre une fonction. Il est alors possible de la reprendre à l'aide de la méthode **.next()**.

4. Manipulation

Duplication de processus

Le seul moyen de créer un nouveau processus est de dupliquer un processus existant. Cette opération est réalisé à l'aide de l'appel système **fork**.



En python

La fonction **os.fork()** permet de dupliquer des processus.

La fonction renvoie :

- 0 pour le fils ;
- le pid du fils pour le père.

Exemple :

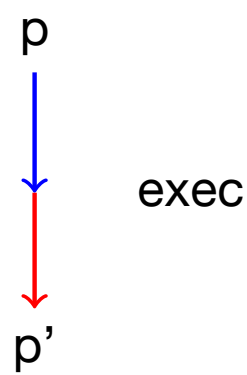
```
from os import *

swith fork() :
    case 0 :
        ... fils ...
        exit 0
    else :
        ... père ...
```

Recouvrement

Pour lancer un autre programme, la méthode consiste à remplacer le programme courant par un autre : le **recouvrement**.

Ceci s'effectue à l'aide de l'appel système **exec**.



En python

Il existe différentes versions suivant que l'on souhaite donner les arguments un à un (**l**) ou sous forme de tableau (**v**) ; que l'on souhaite avoir l'environnement en argument (**e**) ; ou que l'on souhaite chercher dans le path (**p**).

os.execl(*path, arg0, arg1, ...*), os.execle(*path, arg0, arg1, ..., env*),
os.execlp(*file, arg0, arg1, ...*), os.execlpe(*file, arg0, arg1, ..., env*),
os.execv(*path, args*), os.execve(*path, args, env*),
os.execvp(*file, args*), os.execvpe(*file, args, env*)

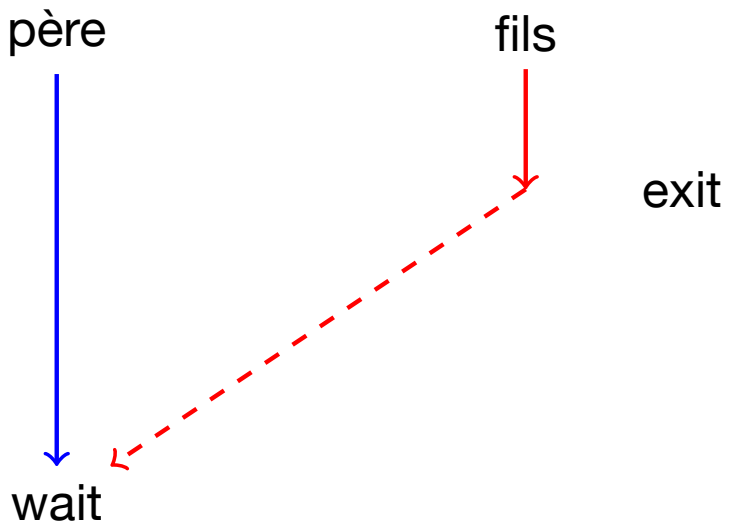
Exemple :

```
...
os.execlp("monscript.sh", "monscript.sh", "arg1")
... jamais atteint ...
```

Fin d'un processus

Pour terminer un processus, il est possible de sortir à l'aide de la fonction **exit**. Cette fonction prend un entier en argument. Le processus devient alors un zombie.

Cette valeur peut-être récupérée par le père à l'aide de la commande **wait** qui attend au besoin la fin d'un processus. Le processus zombie correspondant est alors définitivement détruit.



En python

Il existe plusieurs versions de **wait** selon les options que l'on souhaite :

- **os.wait()** ;
- **os.waitpid(*pid, options*)** ;
- **os.wait3(*[options]*)** ;
- **os.wait4(*pid, options*)** ;

Orphelins

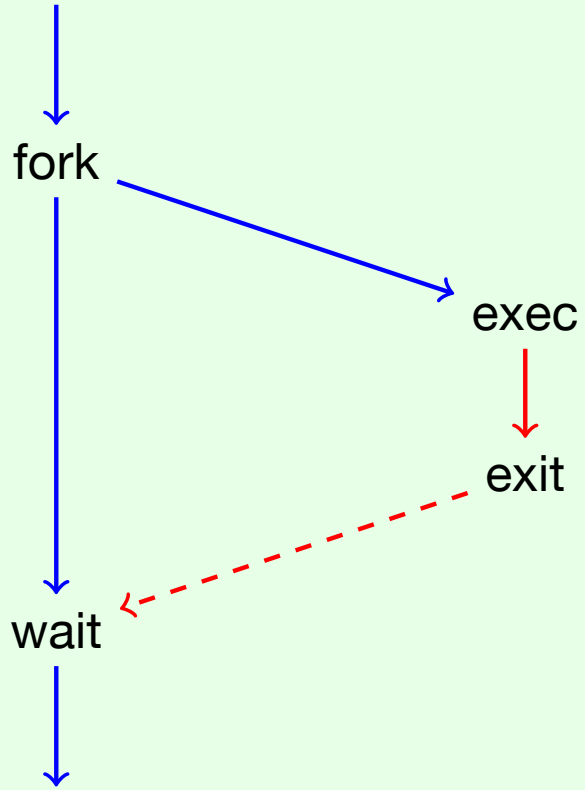
Principe : Tout processus possède un père.

Exception : le premier processus (**init**) n'a pas de père.

Orphelin : quand le père d'un processus meurt, soit celui-ci est détruit ; soit il reste en vie et est alors récupéré par **init**.

Fonctionnement en shell

En shell, chaque commande lancée effectue un fork et un exec de la commande. Il attend ensuite la fin de l'exécution de la commande.



Sous-shell et arrière plan

Il est possible de lancer un **sous-shell** en utilisant les parenthèses. Les commandes sont alors réalisées dans un processus fils du shell.

Il est aussi possible de lancer une commande en **arrière-plan** en ajoutant le symbole **&** à la fin d'un commande. Dans ce cas, le shell n'attend pas la fin de l'exécution (pas de wait).

5. Threads

Remarques sur les processus

Lors de l'utilisation d'un **fork**, on doit dupliquer physiquement les méta-données du processus ainsi que sont contenu.

Les deux processus obtenu sont donc complètement indépendants.

Note : Maintenant, de nombreux systèmes d'exploitations utilisent du copy on write qui permet d'éviter le copie physique de la mémoire tant que cela n'est pas nécessaire.

Principe

Dans certains cas, on souhaiterait disposer du parallélisme sans avoir recours à la lourdeur des processus.
Pour ce cas, il existe des **threads** (processus légers) qui permettent d'effectuer du parallélisme à l'intérieur d'un même processus.

Dans ce cas, on a deux unités en train de s'exécuter qui partagent la même mémoire.

En python

En python, il existe le module **threading** permet de gérer les threads.

La création se fait à l'aide du constructeur :
threading.Thread(*group=None, target=None, name=None, args=(), kwargs={}*).

Il est ensuite possible de manipuler ces éléments de façon similaire au processus.

Chercher les problèmes

Comme la mémoire est partagée, les threads posent de nombreux problèmes de concurrence. Ceci fera l'objet d'un cours par le suite.

La combinaison usuelle est de d'abord faire des fork et ensuite des threads. Utiliser les deux en même temps demande une bonne réflexion pour comprendre les comportements obtenus (lorsque ceux-ci ne sont pas buggés).