LAB4 - La gestion des proces s

Table des matières

By R

Objectifs

Ce « LAB » présente les notions de base de gestions de processus. Il est basé sur la mise en parallèle de notions relatives aux commandes Shell (et Scripts Shell) avec leur équivalent en langage C.

- Documents : «Linux : la gestion des processus ».
- Notions abordées : le « scheduler », les signaux, la gestion synchrone et asynchrone, /proc
- Commandes et fichiers exploités : ps, pstree, top, trap, kill, fork(), exec().
- **Travail à rendre :** Vous devrez répondre directement à plusieurs questions au sein de ce document, vous le déposerez sur Moodle sous le nom : ***LAB4_noms.pdf***. Il y a un code C à écrire dans la question 2, vous le déposerez sur Moodle sous le nom ***part-2.c***.

Gestion des processus

.1 Shell

Lisez le document « Linux – gestion des processus »
Répondez aux questions suivantes dans les zones **(____)** en utilisant **une couleur distincte**.

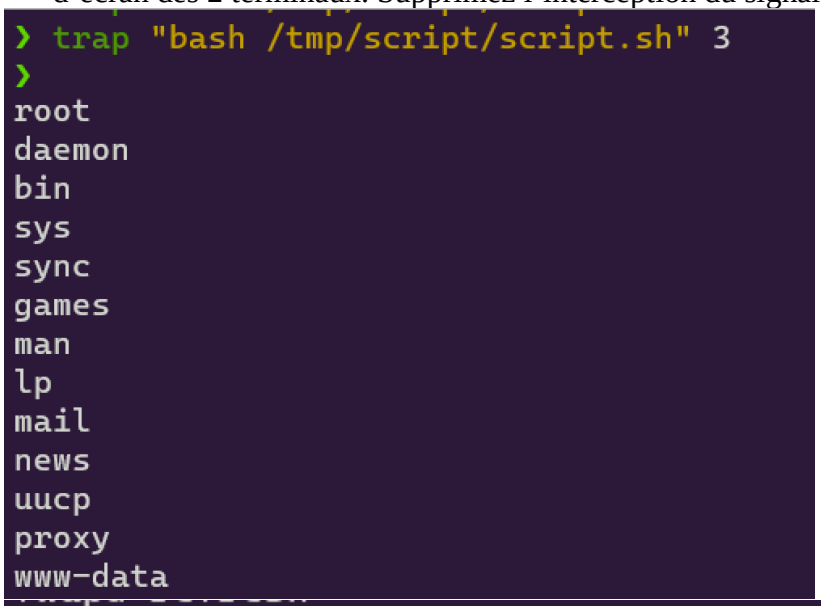
- Dans un terminal : lancez un programme (*firefox*, *gedit*, etc.). En se lançant, votre éditeur affiche (ou pas...) des informations sur le canal des erreurs. Comment feriez-vous pour qu'il n'affiche plus ces erreurs ? **(commande : 2> /dev/null)**. Testez. Info : il est toujours intéressant de lancer les applications graphiques à partir d'un terminal, car cela permet de voir des dysfonctionnements souvent cachés par les interfaces graphiques (idem sous Windows).
- Pourquoi n'avez-vous plus la main sur votre terminal ? **(Car l'instance du terminal est entrain d'executer un processus en premier plan)**. Lancez en aveugle la commande « id ». Fermez votre éditeur via ses menus graphiques. Cela est-il normal ? **(Oui, pendant de le terminal est occupé a éxécuté le commande id est mise en queue, elle est donc exécuter a la suite de la fermeture du processus)**. Relancez-le de manière asynchrone et sans affichage des messages d'erreur **(commande : firefox 2> /dev/null &)**.
- Définissez les 13 champs affichés par la commande « ps -faxl » **(**
- **« F » :** identificateur sur la localisation du processus [1= en mémoire vive; 0= terminé],
- **« UID » : User ID de l'user qui a exécuter la comande,**
- **« PID » : ID du processus éxécuté,**
- **« PRI » :** priorité (plus le nombre est grand, plus la priorité est petite),
- « PPID » : parent process ID (ID du processus parent,
- **« VSZ » : taille memoire virtuel taille alloué ,**
- **« RSS » : taille utilisé par le process,**
- **« WCHAN » :** événement attendu par le processus**,**
- **« STAT » : Etat du processus (status) ,**
- **« TTY » : Interface attaché au processus lors de son execution ,**
- **« TIME » : It is the total accumulated CPU utilization time for a particular process,**
- **« COMMAND » : detail de la commande exécuter avec sa hiérarchie d'execution,**
- **« NI » :** valeur du paramètre nice (gentillesse ou amabilité) pour le calcul de la priorité **).**

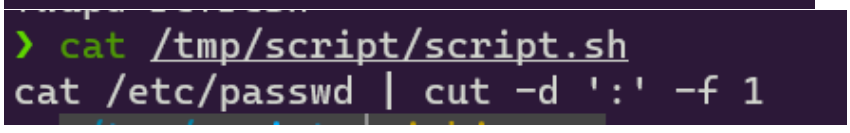
Lancez un éditeur (*gedit*, kate, etc.), quels sont le PID et le PPID de votre éditeur ? **(PID : 1701)**. Quel est le PPID du père de votre éditeur ? **(PPID : 10)**

- Affichez la liste des signaux disponibles sur votre système **((kill -l) HUP INT QUIT ILL TRAP ABRT**

BUS FPE KILL USR1 SEGV USR2 PIPE ALRM TERM STKFLT CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH POLL PWR SYS). En tant que père de votre éditeur, votre shell peut lui envoyer des signaux. Fermer votre éditeur en lui envoyant le signal d'interruption « SIGINT » **(commande : kill -s INT 1701)**.

- Le PPID d'un shell correspond au processus qui gère le terminal et les entrées clavier. Quel est-il sur votre système ? **(process : init)**. Quelle commande permet de connaître les caractéristiques du terminal : **(command : stty -a)**. En analysant cette commande, vous pourrez connaître les séquences de touches associées à certains envois de signaux. Quelle séquence pour le signal SIGINT ? **(Ctrl+Z)**. Testez.
- Lancez votre éditeur de manière synchrone (vous n'avez donc plus la main). Lancez-lui le signal de suspension (SIGTSTP) de deux manières différentes **(1 :Ctrl+Z) (2 : kill -s STOP <PID>)**. Vous reprenez ainsi la main. Votre éditeur est bien toujours présent, mais comme le processus ne passe plus dans la boucle du microprocesseur, il est devenu inerte (sans activité). Vérifiez. Vous pouvez le réactiver en lui envoyant le signal (SIGCONT) de deux manières différentes **(1 : fg) (2 : kill -s CONT <PID>)**. Testez. Vous pouvez réactiver tous les processus stoppés via la commande « bg » (background). INFO : vous connaissez maintenant un bon moyen de reprendre la main sur votre terminal quand vous lancez un processus graphique.
- Qu'arrivera-t-il à votre éditeur si son père meurt ? **(Il meurt aussi)**. Testez en fermant le terminal. Rouvrez un terminal et lancez-le de manière asynchrone afin qu'il ne dépende plus de son père en cas de mort subite de ce dernier **(commande : xterm &)**. Vérifiez en fermant le terminal. Qui est devenu PPID de votre éditeur ? **(10)**.
- Les signaux sont traités par les processus via les standards de programmation. Le « shell » est un processus comme un autre. Ses programmeurs ont prévu que l'utilisateur puisse intercepter les signaux afin de modifier l'action prévue normalement. La commande « trap » qui est interne au « shell » permet ainsi d'intercepter et de reprogrammer l'action des signaux. Ainsi, dans un « shell », la commande « trap 'ls' *nro_signal* » permet d'intercepter le signal numéro '*nro_signal*' quand il est reçu afin de lancer la commande « ls » au lieu du traitement normalement prévu. À quelle séquence de touche correspond le signal « SIGINT » ? **(Ctrl+C)**. Reprogrammer l'action liée à ce signal afin de lancer la commande « ps ». Testez. Comment pouvez-vous réinitialiser cette programmation à sa valeur par défaut ? **(il suffit de relancer un terminal)**.
- Créez un « script shell » qui affiche la liste des usagers de votre système. Cette liste est dans le fichier « /etc/passwd » (vous pouvez judicieusement exploiter la commande « cut »). Dans un terminal (et donc un shell), modifiez le comportement du signal numéro 3 afin de lancer votre script. Ouvrez un deuxième terminal. Déterminez le PID du premier terminal et envoyez-lui le signal numéro 3. Faites une capture d'écran des 2 terminaux. Supprimez l'interception du signal numéro 3.





- Quand un processus reçoit le signal numéro 1 (SIGHUP), il se ferme normalement (HUP = Hang UP = raccrocher). À partir du deuxième terminal, lancez le signal numéro 1 sur le premier terminal. Rouvrez un nouveau terminal et interceptez le signal 1 afin qu'il ne se ferme plus lors de la réception de ce signal. Testez. Quel signal faudrait-il envoyer pour fermer un processus qui ne veut plus rien savoir ? **(_____)**. Qui traite en fait ce signal ? **(le kernel)**. INFO : C'est parfois le seul moyen de supprimer un processus planté (boucle infinie par exemple).

Gestion interactive : Sous KDE, GNOME : <CTRL> + <ECHAP>. Sous Windows : <CTRL> + <ALT> + <Suppr>. Sous l'environnement minimaliste LXDE : commande « lxtask ». Dans un terminal texte : commande « top » (puis « h » pour connaître les interactions possibles).

.2 Création de processus en C

Dans les LAB précédents, vous avez exploité la fonction « system (commande) » de la « librairie C standard (stdlib) » qui lance un « shell » à l'intérieur duquel votre commande est exécutée. Votre programme appelant attend la fin de cet appel pour poursuivre son exécution. L'enchaînement est donc synchrone.
Vous allez maintenant exploiter les possibilités du système multiprocessus qui permet de créer un nouveau processus géré de manière asynchrone et donc traité en parallèle (fonction « fork » et « exec »).
Appuyez-vous sur le guide suivant :
http://mtodorovic.developpez.com/linux/programmation-avancee/?page=page_3#L3

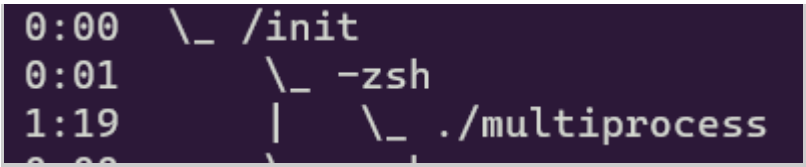
Réalisez un programme en C qui crée un processus fils. Le père affiche son PID et le PID de son fils et attend que vous tapiez la touche « p ». Le processus fils affiche son PID et son PPID. Il ne fait qu'attendre la réception d'un signal de son père (que vous définirez) pour mourir. Le père envoie ce signal quand vous tapez la touche « p ». Quand il s'est assuré que son fils est bien mort, il quitte aussi cette dure vie.
Réalisez une capture d'écran de son exécution.

```
❯ gcc multiprocess.c -o multiprocess -w
❯ ./multiprocess
PID pere : 7932 | PID fils : 7933
        PPID : 7932 | PID : 7933
```

```
 0:00  \_ /init
 0:01      \_ -zsh
 0:00      |   \_ ./multiprocess
 0:00      |       \_ ./multiprocess
```

Puis j'appuie sur la touche p (j'ai ajouté une boucle while infini dans le processus père pour voir la disparaissions du process fils)

```
❯ gcc multiprocess.c -o multiprocess
❯ ./multiprocess
PID pere : 8419 | PID fils : 8420
        PPID : 8419 | PID : 8420
p
```



Une fois le fils créé, le père et le fils veulent écrire « je suis le (père/fils) et mon PID est ... » dans un même fichier. Ils ne peuvent cependant pas le faire en même temps... Ajoutez une méthode de résolution de ce problème d'accès concurrent à une ressource (sémaphore, mutex, etc.).

Rendez ce code C sur Moodle dans un fichier part-2.c.

.3 Schéma « gestion des processus »

Réalisez **ci-dessous** le **schéma** de gestion des processus sous Linux. Vous tenterez de faire apparaître les termes suivants : « microprocesseur », « mémoire vive », « zone de swap », « scheduler » « inerte » « en attente », « en sommeil », « actif », « mort », « zombie », « priorité ». Dans un court **texte**, vous expliquerez la vie trépidante d'un processus.

Le programme est lancé sur l'ordinateur, les instructions sont exécuté par le microprocesseur, le Schedule est en charge d'attitrer le niveau de priorité du programme. Si le programme effectue un fork, l'exécution des instructions est maintenant exécuté par 2 processus (père et fils), avec l'API execvp il est possible de changer le cours des instruction du process vers un autre programme.
A la fin de la tâche du processus père, il rentre en état endormi et attend (wait()) la fin de l'exécution du processus fils, ainsi les deux processus peuvent se terminer respectivement.