Processus

1 NOTION DE PROCESSUS

Un processus est un **programme** en **cours d'exécution**. Il est caractérisé par :

- un ensemble d'instructions à exécuter - souvent stockées dans un fichier sur lequel on clique pour lancer un programme (par exemple *firefox.exe*)
- un espace mémoire dédié à ce processus pour lui permettre de travailler sur des données qui lui sont propres : si vous lancez deux instances de *firefox*, chacune travaillera indépendamment de l'autre avec ses propres données.
- des ressources matérielles : processeur, entrées-sorties (accès à internet en utilisant la connexion Wifi).

2 ETATS ET ORDONNANCEMENT D'UN PROCESSUS

Dans un système multitâche plusieurs processus sont actifs simultanément, mais un processeur (simple cœur) ne peut exécuter qu'une instruction à la fois.

Le système d'exploitation, avec son **ordonnanceur** (ou **scheduler** en anglais) va partager le temps de processeur disponible entre tous les processus en **sélectionnant** le **processus** à **exécuter** parmi ceux qui sont **prêts**.

Le **système d'exploitation alloue** à chacun des processus les **ressources** dont il a besoin en termes de mémoire, entrées-sorties ou temps processeur.

Un processus peut se trouver dans différents états :

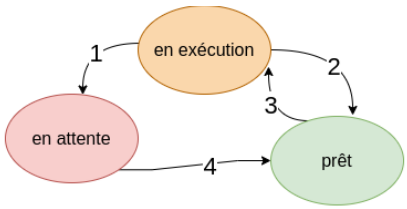
- **prêt** (*ready*): le processus attend son tour,
- **élu** ou en **exécution** (*running*): le processus a accès au processeur pour exécuter ses instructions,
- **bloqué** ou en **attente** (*sleeping*) : le processus attend qu'un événement ou l'accès à une ressource,
- **arrêté** (*stopped*) : le processus a fini son travail ou a reçu un signal de terminaison (SIGTERM, SIGKILL, ...). Il libère les ressources qu'il occupe.

Si une **ressource** demandée est **inaccessible**, un processus en cours **d'exécution** peut passer en **attente (1)**.

Lorsque la **ressource** est **disponible**, le processus devient **prêt (4)**.

Le système d'exploitation (**ordonnanceur**) lui autorise son **exécution (3)**.

Il le passera en état **prêt** afin de pouvoir exécuter un autre **processus (2).**



1 : Le processus se met en attente d'un événement
2 : L'ordonnanceur passe la main à un autre processus
3 : L'ordonnanceur choisit ce processus
4 : L'événement attendu se produit

3 CRÉATION D'UN PROCESSUS, PID, PPID

La création d'un processus peut intervenir :

- par une **action** d'un **utilisateur** (lancement de l'application),
- par un **appel** d'un autre **processus** « **père** »,
- au **démarrage** du système (le processus « init » est alors le père).

Un processus est caractérisé par un identifiant unique : son **PID** (Process Identifier).

Lorsqu'un processus engendre un fils, l'OS génère un nouveau numéro de processus pour le fils. Le fils connaît aussi le numéro de son père : le **PPID** (Parent Process Identifier).

Ci-dessous : le processus du terminal bash (885) a été lancé par sshd (884).

```
UID        PID  PPID  C STIME TTY      TIME       CMD
root       814     1  0 10:55 ?        00:00:00 /usr/sbin/sshd -D
root       829     2  0 10:55 ?        00:00:00 [kworker/0:6-events]
root       872   814  0 10:55 ?        00:00:00 sshd: nsi [priv]
nsi        875     1  0 10:55 ?        00:00:00 /lib/systemd/systemd --user
nsi        876   875  0 10:55 ?        00:00:00 (sd-pam)
nsi        884   872  0 10:55 ?        00:00:00 sshd: nsi@pts/0
nsi        885   884  0 10:55 pts/0    00:00:00 -bash
```

4 GÉRER LES PROCESSUS DEPUIS LE SHELL

Création d'un processus qui va exécuter le programme `commande`

```
% commande
```

`ps` : Liste les processus

- Pour lister les processus actifs :

```
% ps ax
```

- Pour lister les processus actifs avec leurs utilisateurs :

```
% ps axu
```

- Pour lister les processus actifs avec les informations sur les utilisateurs et sur l'occupation mémoire :

```
% ps axum
```

`top` Affichage en continu des informations relatives aux processus

- **`M`** : trie la liste par ordre décroissant d'occupation mémoire. Pratique pour repérer les processus trop gourmands
- **`P`** : trie la liste par ordre décroissant d'occupation processeur
- **`i`** : filtre les processus inactifs.
- **`V`** : permet d'avoir la vue arborescente sur les processus.
- **`q`** : permet de quitter top

`kill` « tue » le processus désigné avec un *signal* de terminaison :

- SIGTERM (15) : demande la terminaison d'un processus. Cela permet au processus de se terminer proprement en libérant les ressources allouées.
- SIGKILL (9) : demande la terminaison immédiate et inconditionnelle d'un processus. C'est une terminaison violente à n'appliquer que sur les processus récalcitrants qui ne répondent pas au signal SIGTERM.

```
% kill -9 pid
```

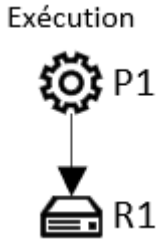
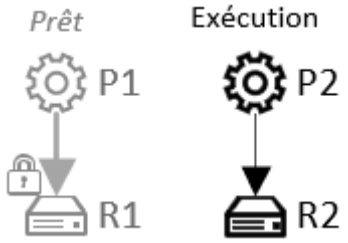
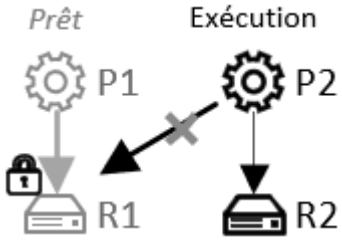
`killall` « tue » le processus désigné a« tue » les processus désignés par leur nom

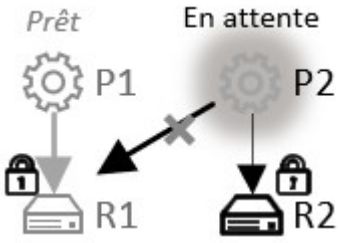
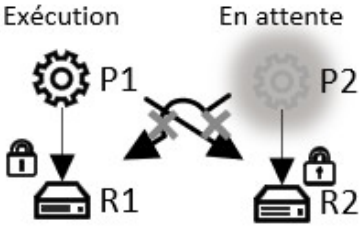
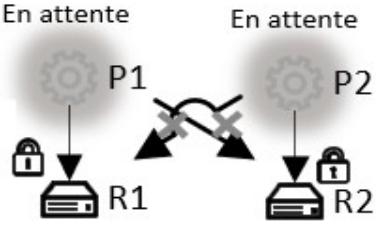
```
% killall nom
```

`pstree` permet de visualiser l'arbre de processus.

5 INTERBLOCAGE (OU DEADLOCK)

L'interblocage peut se produire lorsque des processus concurrents s'attendent mutuellement. Les processus sont alors bloqués définitivement.

		
❶Le processus P1 est exécuté. Il accède à la ressource R1.	❷L'ordonnanceur passe la main au processus P2. Celui ci accède à la ressource R2.	❸Puis P2 essaye d'accéder à la ressource R1 qui est toujours utilisée par P1.

		
❹L'ordonnanceur place P2 en attente tant que R1 est utilisée.	❺L'ordonnanceur passe la main au processus P1. Celui ci essaye d'accéder à la ressource R2 qui est toujours utilisée par P2...	❻L'ordonnanceur place P1 en attente tant que R2 est utilisée... Les 2 processus se sont inter-bloqués...

5.1 PRIORITES

Sous Linux, on peut passer des consignes à l'ordonnanceur en fixant des priorités aux processus dont on est propriétaire : Cette priorité est un nombre entre -20 (plus prioritaire) et +20 (moins prioritaire). Seul root peut modifier la priorité de -20 à 20. Les autres utilisateurs ne peuvent que diminuer la priorité de leurs processus.

On peut agir à 2 niveaux :

- fixer une priorité à une nouvelle tache **dès son démarrage** avec la commande `nice`
- modifier la priorité d'un processus **en cours d'exécution** grâce à la commande `renice`

Les colonne `PR` et `NI` de la commande `top` montrent le niveau de priorité de chaque processus
Le lien entre `PR` et `NI` est simple : **PR = NI + 20** ce qui fait qu'une priorité `PR` de 0 équivaut à un niveau de priorité maximal.
Exemple : Pour baisser la priorité du process `terminator` dont le `PID` est `21523`, il suffit de taper

```
renice +10 21523
```

6 ACTIVITES

6.1 VIDEOS

Visionner les vidéos :
https://www.youtube.com/watch?v=8O4VQVrhoW8

https://www.youtube.com/watch?v=tVceqy6vVqQ

6.2 AFFICHAGE DES PROCESSUS

Les activités peuvent être réalisées sur l'émulateur en ligne :

https://bellard.org/jslinux/vm.html?url=buildroot-x86.cfg

1. Se connecter sur un serveur debian et afficher les processus avec :
```
ps -ef
```
2. Quel est le PID du processus `init` ?
3. Quel est le PPID de `init` ?
4. `init` possède t-il un frère ?
5. Citer quelques descendants directs de `init`

6.3 PLUSIEURS SCRIPTS PYTHON

6. Créer un script python en utilisant l'éditeur `nano` :
```
nano test.py
```
7. Compléter le script avec le code ci-dessous :
```
while True:
    a=2
```
8. Lancer le script en arrière plan (ne bloque pas le terminal avec `&`) avec la commande :
```
python3 test.py &
```
9. Afficher les processus. Trouver le PID du script `test.py`.
10. Arrêter le script avec la commande `kill.`
11. Lancer plusieurs fois le script. Afficher les processus. Trouver les PID des processus d'exécution du script `test.py`.
12. Quel est le parent des scripts python?
13. Déterminer l'arborescence des processus python en remontant par parent jusqu'au processus `init.`
14. Tuer le processus `sshd.`

6.4 PRIORITES

15. Lancer plusieurs fois le script python précédent.
16. Afficher avec `top` les processus et observer les niveaux de priorité et les ressources utilisées par les scripts.
17. Identifier l'id d'un script et augmenter sa priorité au maximum avec `renice.`
18. Relancer `top` et observer les modifications.

6.5 INTERBLOCAGE

19. Créer un script Python `ecrit_1.py` contenant le code suivant :

```
import time
f = open("fichier.txt", "a")
f.write("\n Ajouté par le 1er script!\n")
while True:
      f.write("a")
      time.sleep(1)
f.close()
```

20. Analyser le code et déterminer ce que fait le script.
21. Lancer le script et l'arrêter après quelques secondes avec CTRL-C
22. Vérifier que le `fichier.txt` a été écrit.

23. Créer un deuxième script `ecrit_2.py` qui ajoute des « Z » dans le même `fichier.txt`.
24. Lancer le script `ecrit_2.py` et valider le fonctionnement.
25. Lancer les 2 en parallèle avec la commande ci-dessous :

```
python3 ecrit_1.py ||python3 ecrit_2.py
```

26. Arrêter (CTRL-C) le script `ecrit_1.py` au bout de 10s puis `ecrit_2.py` au bout de 5s.
27. Vérifier qu'il y a une dizaine de « a » et 5 « Z » dans le fichier. Que peut on en conclure ?
28. Quelle instruction bloque l'accès au fichier ?
29. Quelle instruction libère l'accès au fichier ?
30. Faire un interblocage entre les 2 scripts avec 2 fichiers en utilisant pour un algorithme comme :

- `ouvrir fichier_1`
- `attendre 5s`
- `ouvrir fichier_2`

- `écrire dans fichier_1`
- `écrire dans fichier_2`

- `fermer fichier_1`
- `fermer fichier_2`