1 Commande Unix

1.1 Commande `ps` et `top`

Nous allons utiliser les deux commandes `ps` et `top` qui permettent de lister les processus dans le terminal.

1.2 Dans un terminal

1. Dans un terminal : *à taper*

```
# terminal
nsi@lin$ gnote &
nsi@lin$ ps
```

2. résultat : *à compléter*

```
# terminal
PID TTY          TIME CMD
.
.
.
```

3. Dans le terminale : *à taper*

```
# terminal
nsi@lin$ ps -u
```

4. Que représente chacune des colonnes ? *à compléter*
   - `USER` indique ....
   - `PID` donne l'identifiant numérique du processus.
   - `%CPU` et `%MEM` indiquent respectivement ....
   - `STAT` indique l'état du processus, `S` pour *sleeping*, le processus est en attente et `R` pour *running*, le processus est dans l'état prêt ou élu.
   - `COMMAND` indique la commande utilisé pour lancer le programme.
   - `START` et `DATE` indiquent respectivement ....
5. Fermer l'application `Gnote` et utiliser à nouveau la commande `ps` dans le terminale. Que constatez-vous ?

   ....
6. Dans le terminale : *à taper*

```
# terminal
nsi@lin$ top
```

   Quelles sont les principales différences avec la commande `ps` ?

   — ....
   — ....

1.3 La commande `kill`

La commande `cat` est utilisée qu'à titre d'exemple, sa fonctionnalité n'est pas importante ici.

1. Dans un premier terminal : *à taper*

```
# terminal
nsi@lin$ cat
```

2. Dans un deuxième terminal : *à taper*
   Nous allons envoyer un signal de terminaison au processus cat par le biais de la commande **killall** qui envoie un signal aux processus dont le nom est indiqué avec le numéro du signal.


```
# terminal
nsi@lin$ killall cat
```

3. Que constatez-vous ?

   ---

4. Dans le premier terminale taper à nouveau `cat`, puis, dans le ceuxième terminale, à l'aide de la commande `ps`, déterminer le PID du processus ainsi créé : *à compléter*

```
# terminal
USER     PID    %CPU %MEM  VSZ    RSS TTY      STAT START   TIME COMMAND
                 0.0  0.0                      S+           0:00 cat
```

   Remarque : On peut également utiliser la commande `pgrep` en tapant dans le terminal `pgrep cat`.

5. dans le deuxième terminale, **en utilisant le PID de cat que vous avez obtenu** : *à taper*

```
# terminal
nsi@lin$ kill 17369
```

6. Que constatez-vous ?

1.4 Les commandes

Commande

- **ps :** liste les processus actifs attachés au terminal, les processus sont identifiés par leur PID.
- **top :** fournit une vue dynamique temps réel du système en cours d'exécution.
- **kill :** interrompt le processus dont le PID est donné en paramètre.
- **killall :** interrompt les processus dont le nom st donné en paramètre.

Exercice 1 Tester les différentes options `-u`, `-a`, `-e` et `-f` de la commande `ps`. Déterminer leur fonctionnement à l'aide de la commande `man ps`.

2 Les processus

2.1 Définition d'un processus

Définition 1 *Un **processus** est une instance d'exécution d'un programme.*

- *Un processus est décrit par :*
  - *la mémoire allouée par le système pour l'exécution du programme ;*
  - *les ressources utilisées par le programme ;*
  - *les valeurs stockées dans les registres du processeur.*
- *Un processus possède un numéro unique le PID (Process ID).*

Les notions de programmes et de processus sont différentes : le même programme exécuté plusieurs fois générera plusieurs processus.

2.2 Un exemple

1. Vous disposez dans votre dossier d'un fichier **exemple.py** (son contenu n'a pas d'importance.) Dans un terminal vous exécuter la commande suivante :

```
# terminal
nsi@lin$ python3 exemple.py
```

2. Dans un deuxième terminal : *à taper*


```
# terminal
nsi@lin$ pgrep python3
```

pour obtenir le PID du processus.

```
# terminal
  PID  USER    PR  NI    VIRT    RES    SHR S  %CPU %MEM   TIME+   COMMAND
 5963 pascal  20   0  183088  20040   9396 S   0,7  0,1   0:38.73 python3.8
```

3. Dans le deuxième terminale toujours : *à taper* **en utilisant le PID de cat que vous avez obtenu**

```
# terminal
nsi@lin$ top -p 5963
```

4. Que constatez-vous ?

2.3 Les différents états d'un processus

Définition 2 *Les principaux états d'un processus sont les suivants :*

- ***Prêt** : le processus peut être le prochain a s'exécuter. Il est dans la file des processus qui attendent leur tour.*
- ***Elu (actif ou exécution)** : les processus est entrain de s'exécuter.*
- ***Bloqué** : le processus est interrompu et en attente d'un événement externe (entrée/sortie, allocation mémoire, etc.)*

*On peut rajouter à ces trois états deux états éphémères **nouveaux** et **terminé** qui correspondent respectivement à un processus en cours de création et au système d'exploitation qui désallou les ressources attribués a un processus qui vient de se finir.*

2.4 Le cycle de vie d'un processus

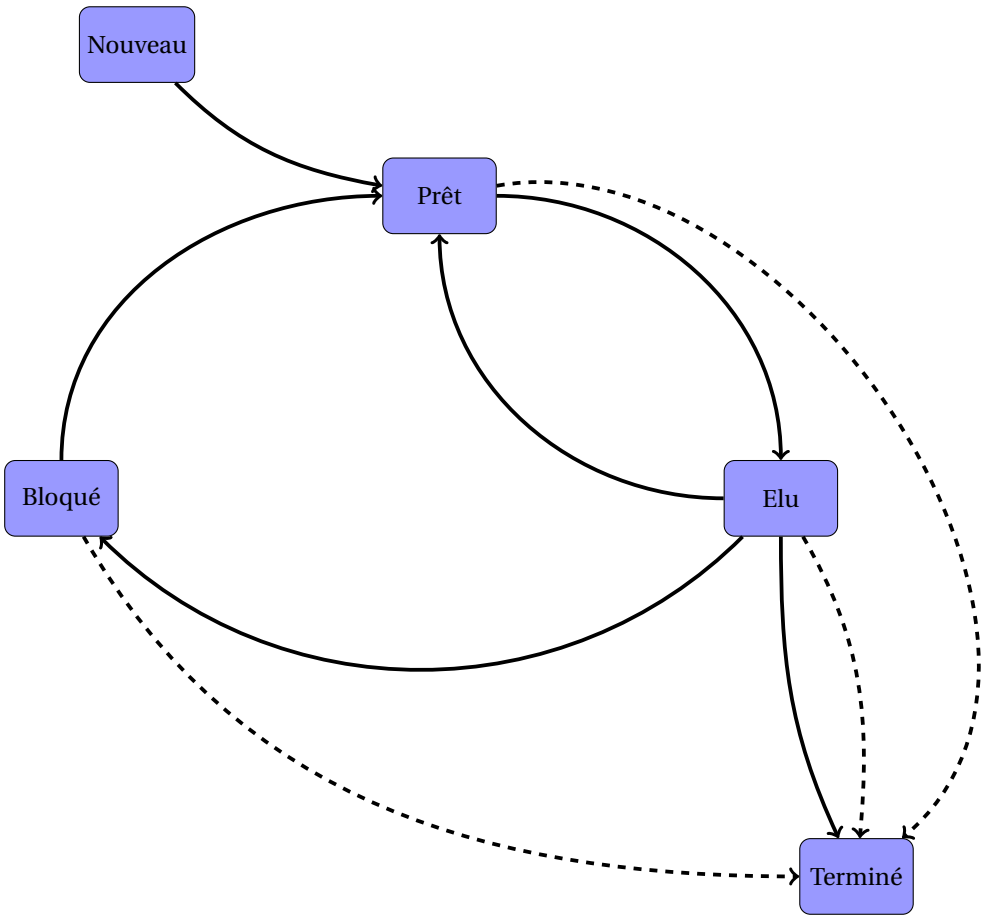
Après sa création un processus est mis dans l'état prêt. En temps normal un processus variera de entre *prêt, élu* et *bloqué*.

Plusieurs processus peuvent être dans l'état *prêt* mais un seul sera placé dans l'état *élu* ; c'est **l'ordonnanceur** (*scheduler*) du système d'exploitation qui est chargé de classer les processus dans une file. C'est lui qui décide quel processus est actif et, de manière permanente, désactive un processus pour en activer un autre.

Alors qu'il est élu, le processus peut avoir besoin d'attendre une ressource quelconque comme, par exemple, une ressource en mémoire. Il doit alors quitter momentanément le processeur pour que celui-ci puisse être utilisé à d'autres tâches). Le processus passe donc dans l'état bloqué.

Une fois le processus terminé, il est placé dans l'état *terminé*, le système d'exploitation libère alors les ressources qui lui sont alloués. Notons que quelque soit l'état du processus, il peut se terminer de façon anormal (erreur dans un programme, problème matériel, intéruption de l'utilisateur, etc.).

Exercice 2 Le shéma ci-dessous résume le cycle de vie d'un processus. Compléter les flèches du shéma en y ajoutant l'un des termes suivants :
mise en exécution par l'ordonnanceur, interruption par l'ordonnanceur, terminaison normale, terminaison anormale, attente d'une ressource ou *obtention de la resource.*



3 Programmation concurrente

3.1 Processus concurent

Vous disposez dans votre dossier d'un fichier **ecritfichier.py** dont le contenu est le suivant :

ecrirefichier.py

```
from os import getpid
pid = str(getpid())
with open ("test.txt", "w") as fichier:
        for i in range(1000):
                fichier.write(pid + " : " + str(i) + "\n")
                fichier.flush()
```

Ce programme importe la fonction utilitaire `getpid` du module `os`. Celle-ci ne prend pas d'argument et renvoie simplement l'identifiant du processus dans lequel on se trouve.

1. Dans un terminale exécuter ce programme : *à taper*

terminal

```
nsi@lin$ python3 ecritfichier.py &
```

2. Le terminale affiche le PID du processus : *à compléter*

terminal

```
nsi@lin$ [1]
```


3. Que devriez-vous trouver dans le fichier `test.txt` ?

   ```
   -------
   -------
   ---
   ---------
   ```

4. Vérifier votre réponse en ouvrant le fichier et expliquer en une phrase ce que fait ce programme.

   ```
   ------------------------------------------------------------------------------
   ```

5. On veut maintenant lancer trois copies de ce programme en même temps : *à taper*

   ```
   # terminal
   nsi@lin$ python3 ecritfichier.py & python3 ecritfichier.py & \
   python3 ecritfichier.py &
   ```

6. Quelle type de lignes peut-on observer dans le fichier `text.txt` ?

   ```
   ---------
   ---------
   ---------
   ---------
   ---------
   ---------
   ```

7. Quelle impression peut-on avoir à la lecture de ce fichier ?

   ```
   ------------------------------------------------------------------------
   ```

8. La réalité est plus complexe. Chaque processus a dans sa mémoire un *curseur* contenant la position à laquelle il s'est arrêté. Lorsqu'un processus va reprendre son exécution, il va continuer à écrire là où il c'était arrêté et peut alors écraser ce qui a été écrit par un autre processus.
   L'enchaînement d'exécutions de processus de PID 82208, 82209 et 82210 représenté ci-dessous est un exemple de ce qui peut arriver.

```
      82208                    82209                82210
curseur:9970          |                     |
write("82208 : 840\n")|                     |
write("82208 : 841\n")|                     |
curseur:9994          |                     |
                      |                     |curseur:9970
                      |                     |write("82210 : 840\n")
                      |                     |write("82210 : 841\n")
                      |                     |curseur:9994
                      |curseur:9970         |
                      |write("82209 : 840\n")|
                      |write("82209 : 841\n")|
                      |write("82209 : 842\n")|
                      |curseur:10006        |
                      |                     |curseur:9994
                      |                     |write("82210 : 842\n")
                      |                     |curseur:10006
curseur:9994          |                     |
write("82208 : 842\n")|                     |
write("82208 : 843\n")|                     |
curseur:10018         |                     |
                      |curseur:10006        |
                      |write("82209 : 843\n")|
                      |curseur:10018        |
                      |                     |curseur:10006
                      |                     |write("82210 : 843\n")
                      |                     |curseur:10018
```

   Quelles lignes sont écrites dans le fichier `test.txt` à la fin de cet enchaînement ?

   ```
   ---------
   ---------
   ---------
   ---------
   ```


3.2 Programmation concurrente en Python

Définition 3 *Un thread ou fil (d'exécution) représente l'exécution d'un ensemble d'instructions démarré par un processus. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'exécution.*

Le module `threading` de la bibliothèque standard de Python permet de démarrer des *threads*.

1. Vous disposez dans votre dossier d'un fichier **exemple_threads.py** dont le contenu est le suivant :

exemple_threads.py
```
import threading

def hello(n):
    for i in range(5):
        print("Je suis le thread",n,"et ma valeur est",i)
    print("-------Fin du thread ",n)

for n in range(4):
    t = threading.Thread(target=hello, args=[n])
    t.start()
```

L'expression `threading.Thread(target=hello, args=[n])` permet de créer un objet de type `Thread`, l'argument `target` est une fonction et l'argument `args` est un tableau des arguments qui seront passés à la fonction.
Dans un terminale, exécuter la commande suivante : *à taper*

terminal
```
nsi@lin$ python3 comptage_parallele.py
```

Que constatez-vous ?

---

2. Vous disposez dans votre dossier d'un fichier **pb_threads.py** dont le contenu est le suivant :

pb_threads.py
```
import threading
COMPTEUR = 0

def incr():
    global COMPTEUR
    for c in range(100000):
        v = COMPTEUR
        COMPTEUR = v + 1

th = []
for n in range(4):
    t = threading.Thread(target=incr, args=[])
    t.start()
    th.append(t)

for t in th:
    t.join()
print("valeur finale", COMPTEUR)
```

La méthode `.join()` permet d'attendre que le *thread* auquel on l'applique soit terminé.
Quelle est la valeur de `COMPTEUR` après une exécution simple du programme `incr()` ? Que fait cette fonction ?

Quelle valeur doit-on alors s'attendre à obtenir pour la variable globale COMPTEUR à la fin de l'exécution de ce programme ?

---

3. Dans un terminale : *à taper*

terminal
```
nsi@lin$ python3 pb_threads.py
```

Que constatez-vous ?

---

4. Pour comprendre ce qui c'est passé, voici un shéma similaire à celui du chapitre précédent, avec trois threads : *à compléter*

```
      thread 0                thread 1                thread 2
v = 42                |                       |
COMPTEUR = 43         |                       |
v =  43               |                       |
COMPTEUR =            |                       |
                      |                       |v = 44
                      |                       |COMPTEUR = 45
                      |                       |v = 45
                      |v = 45                 |
                      |COMPTEUR = 46          |
                      |                       |COMPTEUR = 46
                      |                       |v = 46
                      |                       |COMPTEUR = 47
v = 47                |                       |
COMPTEUR = 48         |                       |
v =                   |                       |
COMPTEUR =            |                       |
v =                   |                       |
                      |                       |v =
                      |v = 48                 |
                      |COMPTEUR = 49          |
                      |v = 49                 |
                      |COMPTEUR =             |
COMPTEUR =            |                       |
                      |                       |COMPTEUR =
```

5. Le problème vient du partage de la variable `COMPTEUR` entre tous les threads. Il faut garantir l'accès exclusif à cette variable entre sa lecture et son écriture. Pour cela, on modifie le programme de la manière suivante : *à taper*

pb_threads.py
```
import threading
verrou = threading.Lock()
...
def incr():
    global COMPTEUR
    for c in range(100000):
        verrou.acquire()
        v = COMPTEUR
        COMPTEUR = v + 1
        verrou.release()
...
```

Le contructeur `Lock()` permet de créer un verrou, la méthode `.acquire()` d'acquerir le verrou et la méthode `.release()` permet de le rendre. Un seul thread à la fois peut acquérir un verrouce qui garantit un accès exclusif.

6. Relancer le programme une fois les modifications faites. Que constatez-vous ?

Exercice 3 On considère la programme `pb_threads.py` avant modification. Répondre aux questions et justifier.

1. Est ce que le programme peut afficher un résultat plus grand que 400 000 ?
2. Si on ajoute `t.join()` juste après `t.start()`, est ce que le programme affiche toujours 400 000 ?
3. Si on transforme le corps de la boucle de la fonction `incr()` par

pb_threads.py
```
verrou.acquire()
v = COMPTEUR
verrou.release()
verrou.acquire()
COMPTEUR = v + 1
verrou.release()
```

en ayant définie une variable globale `verrou = threading.Lock()`, est ce qu'alors le programme affiche 400 000 ?

Exercice 4 Dans le programme `exemple_threads.py`, il y a des *verrous cachés* que l'on peut supprimer à l'aide de l'option -u de python.

terminal
```
nsi@lin$ python3 -u comptage_parallele.py
```

1. Tester cette commande et observer le résultat.
2. Modifier le programme à l'aide d'un verrou.

3.3 Interblocage

1. Dans votre dossier, vous disposez d'un ficher **interblocage.py** dont le contenu est le suivant :

interblocage.py
```
import threading

verrou1 = threading.Lock()
verrou2 = threading.Lock()

def f1():
    verrou1.acquire()
    print("Section critique 1.1")
    verrou2.acquire()
    print("Section critique 1.2")
    verrou2.release()
    verrou1.release()

def f2():
    verrou2.acquire()
    print("Section critique 2.1")
    verrou1.acquire()
    print("Section critique 2.2")
    verrou1.release()
    verrou2.release()

t1 = threading.Thread(target=f1, args=[])
t2 = threading.Thread(target=f2, args=[])
t1.start()
t2.start()
```

Que constatez-vous ? (relancer plusieurs fois le programme si nécessaire)

---

2. Le schéma ci-dessous représente une exécution possible :

```
      thread t1                thread t2
verrou1.acquire()      |
"Section critique 1.1" |
                       |verrou2.acquire()
                       |"Section critique 2.1"
```


Quelle est la situation de chacun des threads ?

On dit que l'on se trouve dans une situation **d'interblocage**.

3. Un interblocage (*deadlock*) est caractérisé par quatre conditions, appelées *conditions de Coffman*, du nom d'Edward Grady Cofman Jr. (1934-), informaticien américain qui les a décrites en 1971.
   (i) *Exclusion mutuelle* : au moins une ressource du système doit être en accès exclusif.
   (ii) *Rétention et attente* : un processus détient une ressource et demande une autre ressource détenue par un autre processus.
   (iii) *Non préemption* : une ressource ne peut être libérée que par le processus qui la détient (et ne peut être « préempté » ou acquise de force par un autre processus).
   (iv) *Attente circulaire* : Les processus bloqués P_1, P_2, ... , P_n sont tels que P_1 attend une ressource détenue par P_2, P_2 attend une ressource détenue par P_3 et ainsi de suite jusqu'à P_n qui attend une ressource détenue par P_1.

Exercice 5 On suppose que l'on dispose des deux programmes suivants :

- **enregistrer_micro** : acquiert la carte son en accès exclusif (pour accéder au micro) et écrit les sons enregistrés en `mp3` sur sa sortie standard et s'arrête lorsqu'il a écrit l'équivalent de 10 secondes de son.
- **jouer_son** : acquiert la carte son en accès exclusif (pour accéder au haut-parleur) et joue le contenu qu'il reçoit sur son entrée standard.

Les cartes son sont typiquement des périphériques ne pouvant être utilisés que par au plus un processus.
Ces deux programmes peuvent être exécutés séquentiellement de la manière suivante :

```
# terminal
nsi@lin$ enregistrer_micro > message.mp3
nsi@lin$ cat message.mp3 | jouer_son
```

La première ligne enregistre les sons émis dans un fichier `message.mp3` et la deuxième redirige le contenu du fichier sur l'entrée de la commande `jouer_son`.

1. Expliquer pourquoi on se trouve dans une situation d'interblocage lorsque l'on enchîne directement les deux commandes :

   ```
   # terminal
   nsi@lin$ enregistrer_micro | jouer_son
   ```

2. Montrer que les quatre conditions de Coffman sont remplies. *Pour la condition (iii), on fait l'hypothèse que `jouer_son` ne peut pas forcer l'acquisition de la carte son.*

4 Exercices

Exercice 6 On suppose qu'Alice exécute dans son terminal la commande `ps -a -u -x`. Le procesus correspondant à cette commande fera partie des processus affichés dans la sortie. Dire quel sera l'état de ce processus et justifier.

Exercice 7
1. Ouvrir un navigateur, par exemple firefox, et déterminer combien de processus sont crées.
2. Quels sont pour chaque processus le PID ?
3. Que se passe-t-il si, depuis la fenêtre firefox, on ouvre un nouvel onglet, ou si on ouvre une nouvelle fenêtre ?

Exercice 8 Considérons un petit système embarqué : un petit ordinateur relié à trois LED A, B et C. Une LED peut être éteinte ou allumée et on peut configurer sa couleur. On dispose de trois programmes qui affichent des signaux lumineux en faisant clignoter les LED. Chaque programme possède une LED primaire et une LED secondaire. Le programme P_1 affiche ses signaux sur A (primaire) et B (secondaire) en vert. Le programme P_2 affiche ses signaux sur B (primaire) et C (secondaire) en orange. Le programme P_3 affiche ses signaux sur C (primaire) et A (secondaire) en rouge.
Comme les LED ne supportent pas d'être configurées dans deux couleurs en même temps, le système propose deux primitives `acquerirLED(nom)` et `rendreLED(nom)` qui permettent respectivement d'acquérir et de relâcher une LED. Si une LED est déjà acquise, alors `acquerirLED()` bloque.
On suppose que chacun des trois programme P_1, P_2 et P_3 effectue les actions suivantes en boucle :

1. acquérir sa LED primaire
2. acquérir sa LES secondaire

3. configurer les couleurs
4. émettre des signaux
5. rendre la LED secondaire
6. rendre la LED primaire
7. recommencer en 1

Montrer qu'il existe un entrelacement des exécutions qui place P_1, P_2 et P_3 en interblocage.

Exercice 9 Ecrire un programme Python s'inspirant de `interblocage.py` simulant l'exercice précédent. Constater qu'en exécutant suffisamment le code votre programme bloque.

Exercice 10 Dans votre dossier, vous avez un fichier `thread_sum.py` qui calcul la somme des entiers de 0 à `N`, la valeur de `N` étant spécifié dans le programme.

1. Quelle contrainte doit respecter `N_threads` ?
2. Le programme donne également son temps d'exécution. Varier la valeur de `N_threads` et observer le temps d'exécution. Que remarquez vous ?
3. En vous inspirant de `thread_sum.py`, écrire un programme`thread_prod.py` qui calcul le produit des entiers de 1 à `N` (prendre pour valeur 1000) et observer à nouveau le temps d'exécution.

Exercice 11 Dans votre dossier, vous avez un fichier `telecharge_liste_seq.py` dont le code est le suivant.

telecharge_liste_seq.py

```
import requests
import time

def telecharge_adresse(adresse):
    '''Télécharge le code html de la page situé à l'adresse donnée en paramètre'''
    print(adresse)
    #télécharge le contenu de l'adresse donnée
    resp = requests.get(adresse)
    #créé le nom du fichier
    nom = "".join(x for x in adresse if x.isalpha()) + "html"
    #copie le contenu dans le fichier
    with open(nom, "wb") as fichier:
        fichier.write(resp.content)


def telecharge_liste(liste_adresses):
    '''Télécharge le code source des adresses données en paramètre.'''
    for adresse in liste_adresses:
        telecharge_adresse(adresse)

liste_adresses = ["https://www.google.com",\
              "http://accromath.uqam.ca/",\
              "https://www.laquadrature.net",\
              "http://images.math.cnrs.fr"]
t0 = time.time()
telecharge_liste(liste_adresses)
t1 = time.time()
print(t1-t0, "secondes pour télécharger", len(liste_adresses), "adresses.")
```

Il permet de téléchargé le contenu d'une liste de pages web.

1. Exécuter le programme et observer le temps d'exécution.
2. Modifier le programme `telecharge_liste` de sorte que chaque page soit téléchargé par un thread diffirent. Observer alors à nouveau le temps d'exécution.

