1 Processus

1.1 Différence entre programme et processus

Exercice 1

1. La commande geany permet de lancer depuis une ligne de commande l'exécution du programme d'édition de textes geany.

   Vérifier le chemin d'accès au programme et ses permissions en exécutant les commandes suivantes :

   ```
   fjunier@fjunier:~$ which geany
   /usr/bin/geany
   fjunier@fjunier:~$ ls -l /usr/bin/geany
   -rwxr-xr-x 1 root root 14488 mars 22 2020 /usr/bin/geany
   ```

2. Ouvrir un `shell` et saisir la commande `geany &`. Le symbole `&` lance la commande `geany` en arrière-plan, ainsi la console ne se bloque pas en attente de fin d'exécution.

   Écrire `Bonjour` et dans le fichier ouvert par défaut dans l'application geany.

   Noter le nombre qui s'affiche après la commande.

   ```
   fjunier@fjunier:~$ geany &
   [1] 14266
   ```

3. Reprendre la question précédente avec une nouvelle commande `geany &`. Écrire `Hello` dans le fichier ouvert par défaut dans la nouvelle fenêtre geany.

   On vient de créer une nouvelle instance d'exécution du programme geany. On parle de processus.

   Noter le nombre qui s'affiche après la commande . Est-il le même que le précédent ?

   On a créé deux instances d'exécution du programme geany, appelées aussi processus. Chaque processus est identifié par un numéro.

   ```
   administrateur@smob-ubuntu-p01:~$ geany &
   [1] 15071
   administrateur@smob-ubuntu-p01:~$ geany &
   [2] 15092
   ```

4. Exécuter la commande `ps -eo pid,stat,command` qui affiche trois informations sur les processus en cours :

   - `pid` est leur identifiant de processus ;
   - `stat` est leur statut ;
   - `command` qui a lancé le processus.

   On pourra filtrer les résultats avec la commande grep.

   ```
   fjunier@fjunier:~$ ps -eo pid,stat,command
       PID STAT COMMAND
   ```

```
     1 Ss   /sbin/init splash
     2 S    [kthreadd]
     3 I<   [rcu_gp]
     ..................
 14266 Sl   geany
 14272 Ss+  /bin/bash
 14409 I    [kworker/3:0-events]
 14453 Sl   geany
     ..................

fjunier@fjunier:~$ ps -eo pid,stat,command | grep geany # sé
    lectionner uniquement les lignes avec geany
 14266 Sl   geany
 14453 Sl   geany
 16471 S+   grep --color=auto geany
```

Quelle information retrouve-t-on dans la colonne PID? et dans la colonne SAT?

Dans la colonne PID, on retrouve l'identifiant de chaque processus :

```
administrateur@smob-ubuntu-p01:~$ ps -eo pid,stat,command |grep geany
  15071 Sl   geany
  15092 Sl   geany
  15117 S+   grep --color=auto geany
```

Dans la colonne STAT est affiché l'état du processus : Sl dénote un processus bloqué en attente d'un événement et multithreadé (plusieurs sous-processus car il s'agit d'un programme avec interface graphique).

La documentation de la commande ps est accessible avec man ps.

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output
       specifiers (header "STAT" or "S") will display to describe the
           state of
       a process:

               D    uninterruptible sleep (usually IO)
               I    Idle kernel thread
               R    running or runnable (on run queue)
               S    interruptible sleep (waiting for an event to
                    complete)
               T    stopped by job control signal
               t    stopped by debugger during the tracing
               W    paging (not valid since the 2.6.xx kernel)
               X    dead (should never be seen)
               Z    defunct ("zombie") process, terminated but not
                    reaped by
                    its parent

       For BSD formats and when the stat keyword is used, additional
```

```
       characters may be displayed:

              <    high-priority (not nice to other users)
              N    low-priority (nice to other users)
              L    has pages locked into memory (for real-time and
                   custom IO)
              s    is a session leader
              l    is multi-threaded (using CLONE_THREAD, like NPTL
                   pthreads
                   do)
              +    is in the foreground process group
```

5. Exécuter la commande ci-dessous en remplaçant 14266 par le PID du premier processus lancé avec la commande geany.

```
fjunier@fjunier:~$ kill -SIGSTOP 14266
```

Essayer d'écrire dans le fichier ouvert. Que se passe-t-il?

Il n'est plus possible d'écrire dans le fichier ouvert par le processus geany de pid 15071 qui a reçu le signal SIGSTOP. Le processus est passé dans l'état T qui signifie (voir la page de manuel de ps) *arrêté par signal de contrôle*. Il s'agit de l'état **bloqué** dans le cycle de vie d'un processus (voir schéma page 17). On peut toujours écrire dans le fichier ouvert par l'autre processus.

```
administrateur@smob-ubuntu-p01:~$ kill -SIGSTOP 15071
administrateur@smob-ubuntu-p01:~$ ps -eo pid,stat,command |grep geany
  15071 Tl   geany
  15092 Sl   geany
  15420 S+   grep --color=auto geany

[1]+  Arrêté                 geany
```

6. Exécuter la commande ci-dessous en remplaçant 14266 par le PID du premier processus lancé avec la commande geany.

```
fjunier@fjunier:~$ kill -SIGCONT 14266
```

Essayer d'écrire dans le fichier ouvert. Que se passe-t-il?

Le signal SIGCONT change l'état du processus de pid 1071 qui est de nouveau actif. On peut de nouveau écrire dans le fichier ouvert par ce processus lancé avec la commande geany.

```
administrateur@smob-ubuntu-p01:~$ kill -SIGCONT 15071
administrateur@smob-ubuntu-p01:~$ ps -eo pid,stat,command |grep geany
  15071 Sl   geany
  15092 Sl   geany
  15912 S+   grep --color=auto geany
```

7. Exécuter la commande ci-dessous en remplaçant 14266 par le PID du premier processus lancé avec la commande geany.

```
fjunier@fjunier:~$ kill -SIGTERM 14266
```

Que se passe-t-il?

L'envoi du signal SIGTERM n'arrête pas le processus de pic 15071. En effet SIGTERM demande au processus de s'arrêter proprement et d'en informer ses éventuels fils.

```
administrateur@smob-ubuntu-p01:~$ kill -SIGTERM 15071
administrateur@smob-ubuntu-p01:~$ ps -eo pid,stat,command |grep geany
  15071 Sl   geany
  15092 Sl   geany
  16222 S+   grep --color=auto geany
```

8. Exécuter la commande ci-dessous en remplaçant `14266` par le `PID` du premier processus lancé avec la commande `geany`.

```
fjunier@fjunier:~$ kill -SIGKILL 14266
```

Que se passe-t-il?

Contrairement à SIGCONT et SIGTERM qui sont envoyés au processus, les signaux SIGSTOP et SIGKILL sont envoyés au système d'exploitation qui prend la main et stoppe ou met fin brutalement au processus sans lui laisser le temps de libérer les ressources ouvertes ou d'en informer ses processus fils. La fenêtre graphique ouverte par le processus est fermée sans avertissement.

```
administrateur@smob-ubuntu-p01:~$ kill -SIGKILL 15071
administrateur@smob-ubuntu-p01:~$ ps -eo pid,stat,command |grep geany
  15092 Sl   geany
  17635 S+   grep --color=auto geany
[1]-  Processus arrêté    geany
```

9. La commande `kill` permet à l'utilisateur d'envoyer un **signal** à un **processus** pour le stopper, le reprendre, le terminer . . . . Avec la commande `man 7 signal`, afficher les pages du manuel de la ligne de commandes et retrouver les significations des signaux `SIGSTP`, `SIGCONT`, `SIGTERM` et `SIGKILLL`.

Notons que `kill -SIGKILL 15071` équivaut à `kill -9 15071` et que `kill -SIGTERM 15071` équivaut à `kill -15 15071`.

On peut afficher la signification des différents signaux disponibles avec la commande `man 7 signal` :

```
Standard signals
       Linux supports the standard signals listed below. The second
            column of
       the  table  indicates  which  standard  (if  any) specified the
            signal:
       "P1990" indicates that the signal is described in the
            original
       POSIX.1-1990 standard; "P2001" indicates that the signal was
            added in
       SUSv2 and POSIX.1-2001.

       Signal     Standard  Action  Comment
```

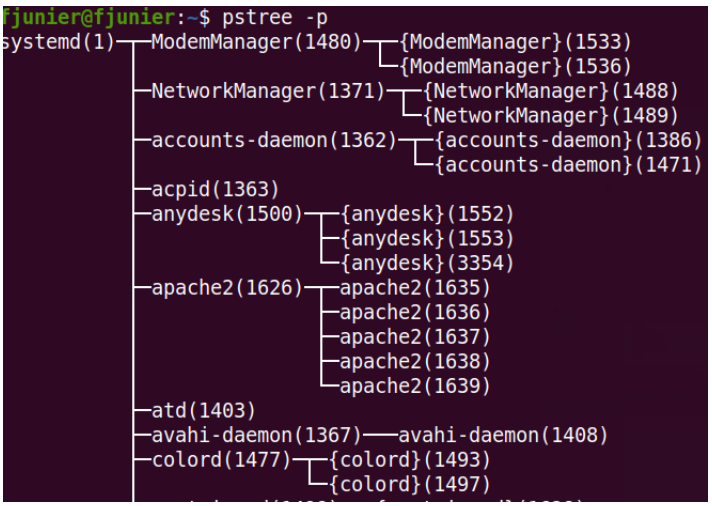
```
SIGABRT     P1990     Core   Abort signal from abort(3)
SIGALRM     P1990     Term   Timer signal from alarm(2)
SIGBUS      P2001     Core   Bus error (bad memory access)
SIGCHLD     P1990     Ign    Child stopped or terminated
SIGCLD        -       Ign    A synonym for SIGCHLD
SIGCONT     P1990     Cont   Continue if stopped
SIGEMT        -       Term   Emulator trap
SIGFPE      P1990     Core   Floating-point exception
SIGHUP      P1990     Term   Hangup detected on controlling
    terminal
                             or death of controlling process
SIGILL      P1990     Core   Illegal Instruction
SIGINFO       -             A synonym for SIGPWR
SIGINT      P1990     Term   Interrupt from keyboard
SIGIO         -       Term   I/O now possible (4.2BSD)
SIGIOT        -       Core   IOT trap. A synonym for SIGABRT
SIGKILL     P1990     Term   Kill signal
SIGLOST       -       Term   File lock lost (unused)
SIGPIPE     P1990     Term   Broken pipe: write to pipe with no
                             readers; see pipe(7)
SIGPOLL     P2001     Term   Pollable event (Sys V).
                             Synonym for SIGIO
SIGPROF     P2001     Term   Profiling timer expired
SIGPWR        -       Term   Power failure (System V)
SIGQUIT     P1990     Core   Quit from keyboard
SIGSEGV     P1990     Core   Invalid memory reference
SIGSTKFLT     -       Term   Stack fault on coprocessor (unused
    )
SIGSTOP     P1990     Stop   Stop process
SIGTSTP     P1990     Stop   Stop typed at terminal
SIGSYS      P2001     Core   Bad system call (SVr4);
                             see also seccomp(2)
SIGTERM     P1990     Term   Termination signal
SIGTRAP     P2001     Core   Trace/breakpoint trap
SIGTTIN     P1990     Stop   Terminal input for background
    process
SIGTTOU     P1990     Stop   Terminal output for background
    process
SIGUNUSED     -       Core   Synonymous with SIGSYS
SIGURG      P2001     Ign    Urgent condition on socket (4.2BSD
    )
SIGUSR1     P1990     Term   User-defined signal 1
SIGUSR2     P1990     Term   User-defined signal 2
SIGVTALRM   P2001     Term   Virtual alarm clock (4.2BSD)
SIGXCPU     P2001     Core   CPU time limit exceeded (4.2BSD);
                             see setrlimit(2)
SIGXFSZ     P2001     Core   File size limit exceeded (4.2BSD);
                             see setrlimit(2)
```

```
       SIGWINCH      -       Ign     Window resize signal (4.3BSD, Sun)

       The signals SIGKILL and SIGSTOP cannot be caught, blocked, or
           ignored.
```

10. On a vu que l'utilisateur peut créer, suspendre ou interrompre un processus. Mais comment apparaissent les processus du système d'exploitation qui s'exécutent en arrière plan ?

Saisir la commande `pstree -p`.



Quelle structure reconnaît-on dans cet affichage ?

On reconnaît une structure arborescente : chaque processus a un unique processus père et peut avoir un ou plusieurs processus fils. Le processus racine de l'arbre est `systemd`. Le nom de ce programme vient de « system daemon » : le daemon du système. La page de manuel `man systemd` confirme qu'il s'agit du premier processus de pid 1 exécuté automatiquement au démarrage du système avec la commande `/sbin/init`

```
DESCRIPTION
       systemd is a system and service manager for Linux operating
           systems.
       When run as first process on boot (as PID 1), it acts as init
           system
       that brings up and maintains userspace services. Separate
           instances are
       started for logged-in users to start their services.

       systemd is usually not invoked directly by the user, but is
           installed
       as the /sbin/init symlink and started during early boot.
```

On peut observer une hiérarchie des processus : chaque processus est le fils d'un processus parent (colonne `PPID` dans l'affichage avec `ps -elf` des informations complètes sur les processus).

Que représente le processus `systemd` (anciennement `init`) de `PID 1` ? Reconstruire le chemin de ce processus vers le premier processus lancé avec la commande `geany`.

On peut relancer un processus avec la commande geany et remonter jusqu'au processus de pid 1.

- processus de pid 19558 lancé avec la commande geany a pour processus père de pid 15055 lancé avec la commande bash
- processus de pid 15055 lancé avec la commande bash a pour processus père de pid 6851 lancé avec la commande usr/libexec/gnome-terminal-server
- processus de pid 6851 lancé avec la commande usr/libexec/gnome-terminal-server a pour processus père de pid 3253 lancé avec la commande /lib/systemd/systemd –user
- processus de pid 3253 lancé avec la commande /lib/systemd/systemd –usera pour processus père de pid 1 lancé avec la commande /sbin/init

```
administrateur@smob-ubuntu-p01:~$ geany &
[3] 19558
administrateur@smob-ubuntu-p01:~$ ps -elf |grep geany
0 S adminis+  15092  15055 0  80   0 - 191252 do_sys 10:26 pts/1
    00:00:02 geany
0 S adminis+  19558  15055 8  80   0 - 191252 do_sys 11:03 pts/1
    00:00:00 geany
0 S adminis+  19586  15055 0  80   0 -  2902 pipe_r 11:04 pts/1
    00:00:00 grep --color=auto geany
administrateur@smob-ubuntu-p01:~$ ps -elf |grep 15055
0 S adminis+  15055   6851 0  80   0 -  3569 do_wai 10:26 pts/1
    00:00:00 bash
0 S adminis+  15092  15055 0  80   0 - 191252 do_sys 10:26 pts/1
    00:00:02 geany
0 S adminis+  19558  15055 2  80   0 - 191252 do_sys 11:03 pts/1
    00:00:00 geany
4 R adminis+  19617  15055 0  80   0 -  3534 -     11:04 pts/1
    00:00:00 ps -elf
0 S adminis+  19618  15055 0  80   0 -  2902 pipe_r 11:04 pts/1
    00:00:00 grep --color=auto 15055
administrateur@smob-ubuntu-p01:~$ ps -elf |grep 6851
0 S adminis+   6851   3253 0  80   0 - 209368 do_sys 09:46 ?
    00:00:25 /usr/libexec/gnome-terminal-server
0 S adminis+   6859   6851 0  80   0 -  3357 do_sel 09:46 pts/0
    00:00:00 bash
0 S adminis+   8855   6851 0  80   0 -  3357 do_wai 09:47 pts/2
    00:00:00 bash
0 S adminis+  11612   6851 0  80   0 -  3357 do_wai 09:58 pts/5
    00:00:00 bash
0 S adminis+  15055   6851 0  80   0 -  3569 do_wai 10:26 pts/1
    00:00:00 bash
0 S adminis+  19647  15055 0  80   0 -  2902 pipe_r 11:04 pts/1
    00:00:00 grep --color=auto 6851
administrateur@smob-ubuntu-p01:~$ ps -elf |grep 3253
4 S adminis+   3253      1 0  80   0 -  4981 ep_pol 09:32 ?
    00:00:00 /lib/systemd/systemd --user
```

Définition 1 *Processus*

Un **processus** est une **instance d'exécution** d'un programme.
Un processus est caractérisé par un **contexte d'exécution** dynamique dont les deux composants principaux sont :

- un *fil d'exécution* avec les valeurs du compteur ordinal et des registres ;
- un *ensemble de ressources* (programme, données, variables, fichiers, entrée/sortie sur des périphériques …) référencées dans un espace d'adressage …

Deux instances d'exécution du même programme constitueront deux processus distincts.
Un processus est une activité :

- il est créé : par l'utilisateur, par un autre processus …
- il vit et peut prendre alors différents états : prêt, en cours d'exécution, bloqué ;
- il se termine : librement ou contraint par un autre processus.

1.2 Multiprogrammation et pseudo-parallélisme

Exercice 2

Ouvrir une ligne de commande dans son espace personnel sur le réseau.

1. Créer un répertoire `processus`, se déplacer dans ce répertoire puis créer quatre fichiers `pA`, `pB`, `pC`, `output` avec la succession de commandes :

```
fjunier@fjunier:~$ mkdir processus
fjunier@fjunier:~$ cd processus
fjunier@fjunier:~$ touch pA pB pC output
fjunier@fjunier:~$ ls
output pA pB  pC
```

   `processus` est désormais notre *répertoire de travail*, dont on peut afficher le chemin avec la commande `pwd`.

2. Saisir une commande qui fixe le droit d'exécution sur les trois fichiers `pA`, `pB` et `pC` pour l'utilisateur.

   Par défaut les fichiers créés n'ont le droit d'exécution pour aucun des trois profils utilisateur/groupe/autres :

```
administrateur@smob-ubuntu-p01:~/processus$ ls -l
total 0
-rw-rw-r-- 1 administrateur administrateur 0 déc. 30 11:31 output
-rw-rw-r-- 1 administrateur administrateur 0 déc. 30 11:31 pA
-rw-rw-r-- 1 administrateur administrateur 0 déc. 30 11:31 pB
-rw-rw-r-- 1 administrateur administrateur 0 déc. 30 11:31 pC
```

On positionne le droit d'exécution pour le profil utilisateur sur les trois fichiers pA, pB et pC avec la commande `chmod` :

```
administrateur@smob-ubuntu-p01:~/processus$ chmod u+x pA pB pC
administrateur@smob-ubuntu-p01:~/processus$ ls -l
total 0
-rw-rw-r-- 1 administrateur administrateur 0 déc. 30 11:31 output
-rwxrw-r-- 1 administrateur administrateur 0 déc. 30 11:31 pA
-rwxrw-r-- 1 administrateur administrateur 0 déc. 30 11:31 pB
-rwxrw-r-- 1 administrateur administrateur 0 déc. 30 11:31 pC
```

3. Saisir les programmes suivants en langage BASH dans les fichiers `pA`, `pB` et `pC`.

pA

```
#!/bin/bash
while true
    do
        echo "A"
    done
```

pB

```
#!/bin/bash
for ((i=1; i <= 100; i++))
    do
        echo "B" >> output
    done
```

pC

```
#!/bin/bash
for ((i=1; i <= 100; i++))
    do
        echo "C" >> output
    done
```

Que font ces programmes lorsqu'on les exécute ?

Le programme `pA` va afficher en boucle infinie le caractère "A" dans le terminal.

Le programme `pB` va écrire cent fois le caractère "B" à la fin du fichier `output`.

Le programme `pC` va écrire cent fois le caractère "C" à la fin du fichier `output`.

4. Ouvrir deux terminaux de ligne de commande depuis le répertoire de travail `processus` :

- dans l'un, exécuter la commande `top` pour afficher dynamiquement les processus en cours d'exécution ;
- dans l'autre, exécuter le programme `pA` (à condition qu'ils soit exécutable cf question 2.) avec la commande `./pA` .

```
top - 10:01:38 up  2:01,  1 user,  load average: 1,46, 1,00, 0,65
Tâches: 351 total,   4 en cours, 347 en veille,   0 arrêté,   0 zombie
%Cpu(s): 38,4 ut, 25,6 sy,  0,0 ni, 36,0 id,  0,1 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :  28037,1 total,  12463,8 libr,   4406,5 util,  11166,8 tamp/cache
MiB Éch:   2048,0 total,   2048,0 libr,      0,0 util.  23126,7 dispo Mem

    PID UTIL.      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TEMPS+ COM.
  17758 fjunier    20   0   12044   3232   2988 R 100,0   0,0   0:07.62 pA
  11692 fjunier    20   0  825992  58780  42112 R  88,3   0,2   1:04.22 gnome-t+
  13081 root       20   0       0      0      0 D  18,0   0,0   0:06.10 kworker+
  17585 fjunier    20   0 1092676 161300  37776 S  14,3   0,6   0:03.94 shutter
   3019 fjunier    20   0 4077996 331716 113148 R  11,3   1,2   3:04.33 gnome-s+
   2867 fjunier    20   0  501712 124220  74920 S   9,7   0,4   2:49.73 Xorg
  14292 root       20   0       0      0      0 I   7,7   0,0   0:03.43 kworker+
```

Repérer l'identifiant PID du processus créé par l'exécution de cette commande ? Quel est son statut (ou état) (colonne S) ?

Le statut du processus lancé avec la commande `./pA` est `R` pour *en cours d'exécution.*

```
administrateur@smob-ubuntu-p01:~$ top

top - 11:43:28 up 2:13, 2 users, load average: 0,97, 0,66, 0,51
Tâches: 322 total, 4 en cours, 318 en veille, 0 arrêté, 0 zombie
%Cpu(s): 31,0 ut, 26,7 sy, 0,0 ni, 42,2 id, 0,1 wa, 0,0 hi, 0,0 si,
    0,0 st
MiB Mem :  7821,6 total,  2060,4 libr,  2391,1 util,  3370,2 tamp/cache
MiB Éch:  2048,0 total,  2048,0 libr,    0,0 util.  4611,7 dispo Mem

   PID UTIL.    PR  NI    VIRT    RES    SHR S %CPU %MEM    TEMPS+ COM.
 23458 adminis+ 20   0   12048   1076    932 R 100,0  0,0   0:15.53 pA
 20323 root     20   0       0      0      0 I  12,3  0,0   0:02.45
     kworker+
 22930 root     20   0       0      0      0 I  11,3  0,0   0:01.88
     kworker+
 15904 root     20   0       0      0      0 R  10,7  0,0   0:01.80
     kworker+
  3352 adminis+ 20   0  606408 114364 67776 S  3,0   1,4   4:29.97 Xorg
  3493 adminis+ 20   0 4413568 282608 114088 S 2,7  3,5   5:18.62
      gnome-s+
   199 root    -51   0       0      0      0 S  0,3  0,0   0:38.89 irq
       /126+
 23366 root     20   0       0      0      0 I  0,3  0,0   0:00.08
     kworker+
 23444 adminis+ 20   0   14780   4192   3268 R  0,3  0,1   0:00.08 top
```

Quel est son taux d'occupation du processeur ? de la mémoire ?

Rapidement il mobilise la majeure partie des ressources du processeur (mais pas de la mémoire).

Est-ce le seul processus en train de s'exécuter ? Que signifie le statut/état S ?

Il existe d'autres processus en cours d'exécution avec la statut `R`, dont celui lancé par la commande `top` !

Appuyer sur la touche `k`, saisir le `PID` du processus et le code `9` du signal pour forcer la terminaison du processus. Quelle commande est équivalente ? Il est aussi possible d'arrêter le processus avec la combinaison de touches `CTRL + C` dans son terminal.

Dans l'exercice 1, on a vu qu'on peut forcer l'interruption de ce processus de pid 23458 avec la commande `kill -SIGKILL 23458` ou `kill -9 23458`.

5. **a.** Depuis la ligne de commande, exécuter la commande `nice -n 20 ./pC & nice -n 0 ./pB & ps`.

Elle lance en même temps l'exécution des deux programmes `pB` et `pC` en arrière plan (la console n'est pas bloquée) et affiche les processus en cours avec la commande `ps`. La commande `nice` positionne la priorité d'exécution d'un programme : `pC` est exécuté avec une priorité faible et le programme `pB` avec une priorité haute, les priorités sont classées sur une échelle décroissante de 0 à 20.

```
administrateur@smob-ubuntu-p01:~/processus$ nice -n 20 ./pC &
    nice -n 0 ./pB & ps
[1] 24296
[2] 24297
   PID TTY          TIME CMD
 23426 pts/2    00:00:00 bash
 24298 pts/2    00:00:00 ps
[1]-  Fini                  nice -n 20 ./pC
[2]+  Fini                  nice -n 0 ./pB
```

Que signifient ces priorités ? nous allons le découvrir.

Afficher le contenu de `output` dans la console avec `less output` et observer la répartition des caractères `"B"` et `"C"`.

On observe que le fichier `output` contient bien 100 caractères B et 100 caractères C (un par ligne) mais que la répartition n'est pas uniforme : les caractères B sont très majoritaires dans les 10 premières lignes et absents des 10 dernières lignes.

```
administrateur@smob-ubuntu-p01:~/processus$ cat output |grep "B"|
     wc -l
100
administrateur@smob-ubuntu-p01:~/processus$ cat output |grep "C"|
    wc -l
100
administrateur@smob-ubuntu-p01:~/processus$ head -10 output
C
B
B
B
B
B
B
C
B
B
administrateur@smob-ubuntu-p01:~/processus$ tail -10 output
C
C
C
C
C
C
C
C
C
C
```

Hypothèse importante : On suppose que les deux processus sont exécutés sur un seul processeur.

Les deux processus ont-ils été exécutés l'un après l'autre ?

Non un processus n'a pas été exécuté complètement avant l'autre car on observe une alternance de "B" et de "C" dans le fichier `output`.

Les deux processus peuvent-ils avoir été exécutés de façon strictement parallèle ?

Non les processus n'ont pas été exécutés de façon strictement parallèle car la répartition des "B" et des "C" n'est pas uniforme dans le fichier `output` : plus de "B" au début et plus de "C" à la fin. Il semblerait que le processus lancé avec la commande `./pB` se soit exécuté de façon prioritaire.

b. Supprimer le fichier `output` avec une commande appropriée puis le recréer.

```
administrateur@smob-ubuntu-p01:~/processus$ rm output
administrateur@smob-ubuntu-p01:~/processus$ touch output
```

c. Reprendre la question **a.** avec la commande :

```
nice -n 0 ./pC & nice -n 20 ./pB & ps
```

Que peut-on observer ?

Cette fois il semblerait que le processus lancé avec la commande `./pC` se soit exécuté de façon prioritaire sur l'unique processeur.

```
administrateur@smob-ubuntu-p01:~/processus$ nice -n 0 ./pC & nice
     -n 20 ./pB & ps
[1] 36388
[2] 36389
   PID TTY          TIME CMD
  23426 pts/2    00:00:00 bash
  36390 pts/2    00:00:00 ps
[1]-  Fini                  nice -n 0 ./pC
[2]+  Fini                  nice -n 20 ./pB
administrateur@smob-ubuntu-p01:~/processus$ head -10 output
C
C
C
C
C
C
C
C
B
C
administrateur@smob-ubuntu-p01:~/processus$ tail -10 output
B
B
B
B
B
B
B
B
B
B
```

```
B
B
```

d. Reprendre la question **a.** avec la commande `nice -n 0 ./pC & nice -n 0 ./pB & ps`. Que peut-on observer ?

Cette fois on observe une répartition plus uniforme des caractères "B" et "C" dans le fichier `output`. On peut conjecturer que les deux processus ont été exécutés sur l'unique processeur de façon alternée avec un équilibre des priorités.

```
administrateur@smob-ubuntu-p01:~/processus$ nice -n 0 ./pC & nice
      -n 0 ./pB & ps
[1] 37339
[2] 37340
    PID TTY          TIME CMD
  23426 pts/2    00:00:00 bash
  37341 pts/2    00:00:00 ps
[1]-  Fini                  nice -n 0 ./pC
[2]+  Fini                  nice -n 0 ./pB
administrateur@smob-ubuntu-p01:~/processus$ head -10 output
C
B
C
C
B
C
B
B
C
C
administrateur@smob-ubuntu-p01:~/processus$ tail -10 output
C
C
B
C
B
C
B
B
C
B
```

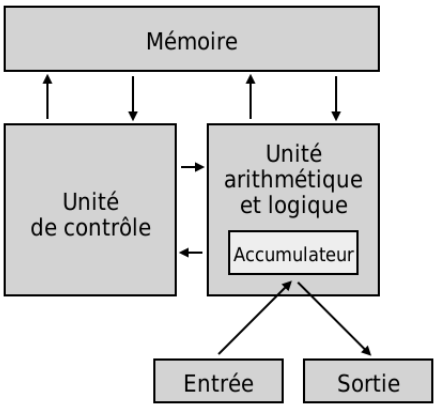
e. Proposer un modèle d'exécution des processus en cours qui serait compatible avec les observations précédentes.

On peut conjecturer que le système d'exploitation découpe le temps en unités et qu'il donne l'accès au processeur à la fin de chaque unité de temps à un certain processus selon un algorithme qui tient compte de la priorité du processus.

Le processeur étant très rapide, l'utilisateur peut avoir l'impression d'une exécution en parallèle même si chaque processus progresse dans son exécution à tour de rôle sur le processeur mais avec des changements de contextes très rapides. On peut parler de *pseudo-parallélisme*.

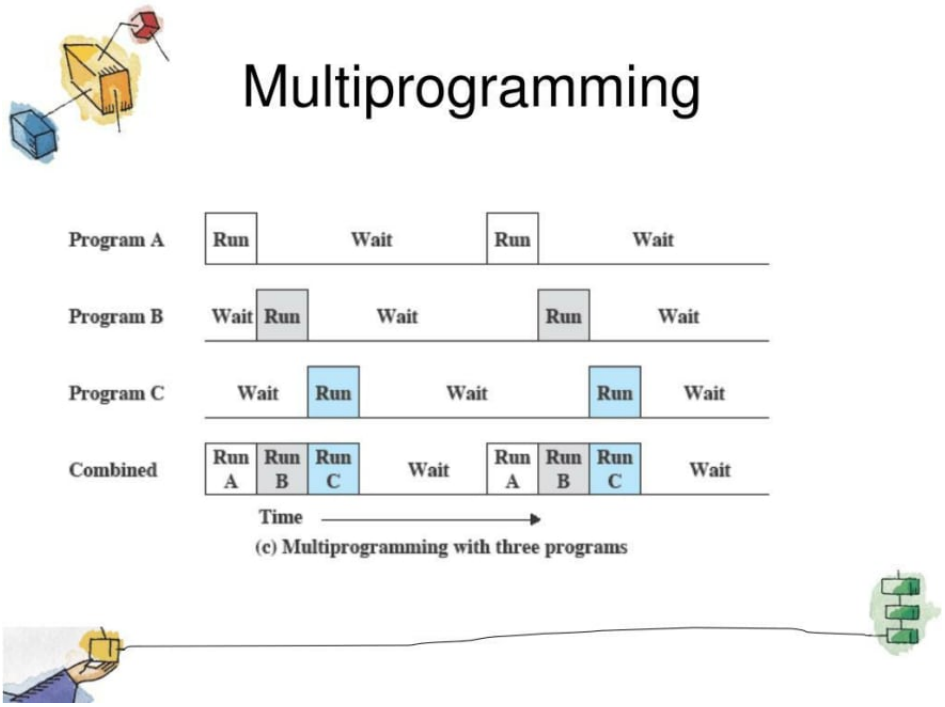
Définition 2 *Multiprogrammation*

Hypothèse importante : On considère un ordinateur d'architecture Von Neumann avec un unique processeur.



Les systèmes d'exploitations modernes permettent la **multiprogrammation**, c'est-à-dire qu'on peut lancer en même temps l'exécution de plusieurs programmes.
On parle de systèmes à **temps partagé**.
Un programme en cours d'exécution est un **processus**.
Un ordinateur monoprocesseur ne peut exécuter qu'un seul processus à la fois. Les processus progressent en parallèle, mais ils sont exécutés séquentiellement, à tour de rôle, sur le processeur. Chaque unité de temps d'exécution, ou **quantum**, est attribuée à un **processus élu** par un programme du système d'exploitation appelé **ordonnanceur**. Ces choix de **commutation de contexte** sont guidés par des algorithmes **d'ordonnancement**.
Le **contexte d'exécution d'un processus** doit donc être sauvegardé dans une **table des processus** qui contient toutes les informations permettant de reprendre son exécution :

- son identifiant `PID`
- son état
- sa priorité
- compteur ordinal, variables, adressage mémoire
- ressources ouvertes (fichiers)
- la liste des périphériques en attente . . .



Méthode

Dans une ligne de commandes `Linux` (de la famille `UNIX`), plusieurs commandes permettent d'observer les processus en cours :

- La commande `ps` fournit un instantané figé à un instant t des processus en cours, sous la forme d'un tableau dont les colonnes sont des attributs et les lignes les processus.

Commande	Sémantique
`ps -elf`	Affiche des informations détaillées, sur tous les processus en cours
`ps -eo pid,user,stat,time,command`	Affiche le pid, le propriétaire, l'état, le temps, la commande de lancement

- La commande `top` fournit un tableau similaire mais dynamique.
- La commande `pstree` affiche la hiérarchie des processus sous la forme d'un arbre. La racine est le premier processus, tous les autres sont ses descendants.
- La commande `kill` permet d'interrompre un processus en lui envoyant un **signal d'interruption**.

Commande	Sémantique	Combinaison de touches équivalente
`kill -SIGTERM`	Terminaison propre	
`kill -SIGSTOP`	Suspension	CTRL + Z
`kill -SIGCONT`	Reprise	
`kill -SIGKILL`	Terminaison immédiate	CTRL + C

2 Cycle de vie d'un processus et ordonnancement

2.1 Cycle de vie d'un processus

Définition 3 *États d'un processus*

Un **processus** peut traverser plusieurs états au cours de son cycle de vie :

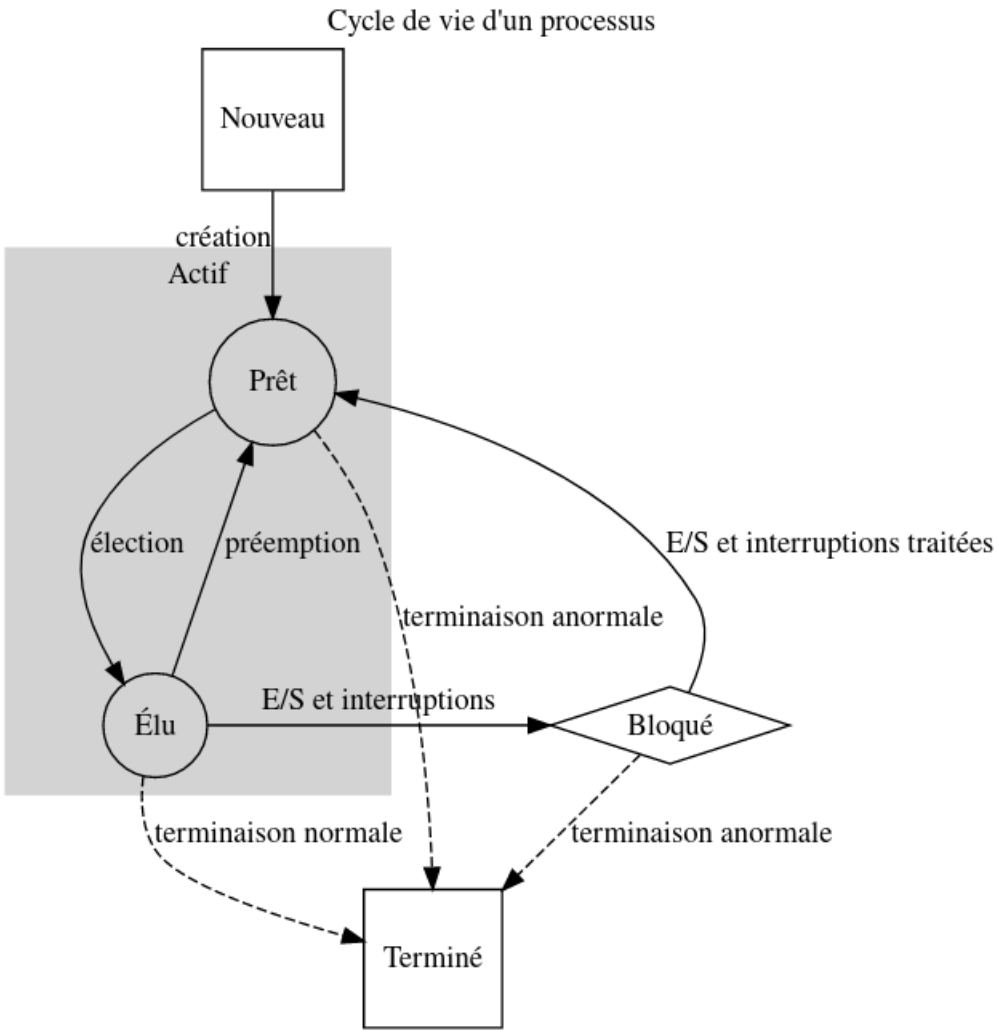
- ☞ Un processus est créé comme fils d'un processus père (on parle de *fork*) et reçoit alors un **identifiant unique** ou `PID`. Sous Linux, le processus père de numéro 1 est `systemd` (le processus de démarrage numéroté 0, s'arrête à la fin du *boot*). Même s'il est créé par une action de l'utilisateur, un processus a un processus père (repéré par son `PPID`), par exemple la ligne de commande `bash` si on lance une commande dans un terminal.
- ☞ Dès qu'il est créé, un processus devient **prêt** à l'exécution. S'il est **élu** par **l'ordonnanceur**, alors il s'exécute sur le processeur. Lorsque l'ordonnanceur bascule de l'état **élu** à **prêt** un processus qui n'est pas terminé, on parle de **préemption**.

  Les processus **prêts** sont mis en attente dans une structure de **file d'attente** qui peut être gérée selon différents **algorithmes d'ordonnancement** :

  - choix du plus récent (FIFO) ;
  - choix de celui avec le temps d'exécution le plus court ;
  - choix par priorité.

  Un processus **prêt** ou **élu** est appelé processus **actif**, mais attention, le seul processus s'exécutant sur le processeur est celui qui est élu !
- ☞ L'exécution d'un processus peut être bloquée dans l'attente de l'accès à une ressource (mémoire bloquée par un verrou, attente de lecture/écriture sur un périphérique). Le périphérique passe alors en l'état **bloqué**, il n'est plus actif. Il pourra être réveillé par un signal d'interruption si l'accès à la ressource est libéré et revenir à l'état prêt.
- ☞ Enfin un processus peut se terminer, de façon normale ou anormale (erreur d'exécution, accès impossible à une ressource . . ....)

Les transitions entre ces états sont précisées sur les arcs du graphe orienté ci-dessous.



Exercice 3

Les questions sont indépendants.

1. *Bac Candidats libres 2021, sujet 2.*

   Les états possibles d'un processus sont : prêt, élu, terminé et bloqué.

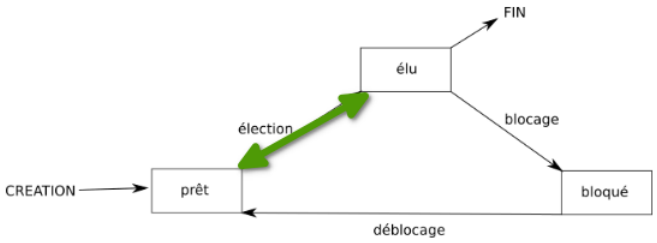
   a. Expliquer à quoi correspond l'état élu.

   Quand un processus est dans l'état élu, cela signifie que ce processus est en cours d'exécution.

   b. Proposer un schéma illustrant les passages entre les différents états.

   Source : `https://pixees.fr/informatiquelycee/term/suj_bac/2021/Correction_sujet_05.pdf`

   Notez que la flèche reliant les états *prêt* et *élu* doit être bidirectionnelle.



2. Faire l'exercice 9 p. 258 du manuel.

> **9** **a.** Quelles informations affiche la commande `ps` ?
>
> **b.** Lancer cette commande sur votre système.
>
> **c.** Essayer l'option `ps -l`.
>
> **d.** À quoi correspond la colonne `PPID` ? Qu'ont en commun les processus qui partagent une même valeur de PPID ?

La commande `ps` affiche des informations sur les processus actifs : pid, temps d'exécution, commande qui a lancé le processus. Par défaut uniquement les processus lancés depuis le terminal actif sont affichés.

```
administrateur@smob-ubuntu-p01:~/processus$ ps
    PID TTY          TIME CMD
  23426 pts/2    00:00:00 bash
  43069 pts/2    00:00:00 ps
```

La commande `ps -l` affiche des informations plus détaillées comme le ppid qui est l'identifiant du processus père. Tous les processus qui ont le même ppid sont fils du même processus : c'est le cas de tous les processus lancés par une commande depuis le même terminal.

```
administrateur@smob-ubuntu-p01:~/processus$ ps -l
F S   UID    PID    PPID  C PRI  NI ADDR SZ WCHAN TTY        TIME CMD
0 S  1000   23426   21933  0  80   0 -  3564 do_wai pts/2  00:00:00 bash
4 R  1000   43077   23426  0  80   0 -  3514 -      pts/2    00:00:00 ps
```

3. Faire l'exercice 10 p. 258 du manuel.

> **10** **a.** Trouver dans le manuel la fonction Python qui permet d'afficher le PID du processus parent.
>
> **b.** Proposer un programme qui affiche son PID, le PID de son parent et termine.
>
> **c.** Exécuter plusieurs fois ce programme depuis un terminal.
>
> **d.** Que remarque-t-on ? Comment expliquer cette situation ?

Le module `os` de la bibliothèque standard rassemble quelques fonctions d'appel système comme celles permettant d'obtenir le pid du processus ou celui du processus parent.

```
import os
print("pid du processus : ", os.getpid())
print("pid du processus parent : ", os.getppid())
```

En exécutant plusieurs fois le programme ci-dessus on observe que le pid du processus change mais pas celui du processus parent qui est celui du terminal ou la commande `python3 python_pid.py` est lancée.

```
administrateur@smob-ubuntu-p01:~$ python3 python_pid.py
pid du processus : 5369
pid du processus parent : 4651
administrateur@smob-ubuntu-p01:~$ python3 python_pid.py
pid du processus : 5373
pid du processus parent : 4651
administrateur@smob-ubuntu-p01:~$ python3 python_pid.py
pid du processus : 5380
pid du processus parent : 4651
administrateur@smob-ubuntu-p01:~$ ps
    PID TTY          TIME CMD
   4651 pts/1    00:00:00 bash
   5715 pts/1    00:00:00 ps
```

2.2 Ordonnancement

Définition 4 *Ordonnancement*

☞ **Qu'est-ce que l'ordonnancement ?**

Sur un ordinateur fonctionnant en temps partagé, à un instant donné, le nombre de processeurs est le plus souvent très inférieur au nombre de processus en cours. On dit que les processus sont en **concurrence**.

Pour simplifier, considérons une machine avec un seul processeur.

L'ordonnanceur est le programme du système d'exploitation qui choisit l'ordre d'exécution des processus sur le processeur.

L'ordonnancement désigne l'activité de **l'ordonnanceur**.

Le processus choisi est dit **élu**, les processus **prêts** pour l'exécution sont stockés dans une file d'attente.

☞ **Quels sont les objectifs de l'ordonnancement ?**

Les objectifs communs de l'ordonnancement sont :

- l'attribution équitable de temps de processeur à chaque processus ;
- l'équilibre et l'optimisation dans l'utilisation des ressources du système.

Certains objectifs peuvent être spécifiques selon les systèmes :

- sur les *systèmes interactifs* (ordinateurs personnels, poste de travail), le temps de réponse est privilégié ;
- sur les *systèmes de traitement par lot* (fiches de paye, comptabilité, sauvegardes . . ....) la capacité de traitement (en nombre de lots par unité de temps) et le délai de rotation ((durée entre le moment où un processus est prêt et celui où il est terminé) sont privilégiés ;
- sur les *systèmes en temps réel* (multimédia, médecine, transport, industrie …) le respect des délais, la préservation de la qualité sont des priorités .

☞ **Quand ordonnancer ?**

L'ordonnanceur peut élire un nouveau processus extrait de la file d'attente des processus prêts dans différents cas :

- le processus élu se termine ;
- le processus élu est bloqué dans l'accès à une ressource ;
- une interruption matérielle d'entrée/sortie a débloqué un processus qui devient prêt à remplacer le processus élu ;
- une interruption matérielle de signal d'horloge indique la fin de l'unité de temps d'exécution, ou quantum, allouée au processus élu.

Si **l'ordonnanceur** prend en compte les interruptions par signal d'horloge pour faire un nouveau choix de processus élu à la fin de chaque **quantum** de temps, on parle d'**ordonnancement préemptif**.

Si **l'ordonnanceur** laisse le processus élu s'exécuter jusqu'à ce qu'il se termine ou se bloque, on parle d'**ordonnancement non préemptif**.

Il existe de nombreux algorithmes d'ordonnancement qui dépendent des buts visés :

- les algorithmes non préemptifs sont utilisés pour les systèmes à traitement par lot ou certains systèmes en temps réel :
  - *premier arrivé, premier servi* : on utilise une file d'attente FIFO et on élit le premier de la file ;
  - *exécution du processus le plus court d'abord* : optimise le délai de rotation mais il faut connaître les temps d'exécution à l'avance.
- les algorithmes préemptifs sont utilisés pour les systèmes interactifs :
  - *tourniquet, round robin* : on utilise une file d'attente FIFO et on élit le premier de la file, chaque processus accède au processeur pendant un quantum de temps puis retourne à la fin de la file ;
  - *par priorité* : chaque processus reçoit une priorité et le processus avec la plus grande priorité est élu. Pour éviter qu'un processus accapare le processeur, sa priorité diminue avec son temps processeur.
  - *par tirage au sort, par choix du plus rapide à terminer* etc …

Exercice 4 *Découvrir des algorithmes d'ordonnancement*

Exercice 13 page 258 du manuel faire sur feuille.

13 **Ordonnancements FIFO et préemptif**

L'ordonnancement FIFO (premier arrivé, premier servi) est un ordonnancement *non préemptif* dans lequel les processus sont élus dans l'ordre de leur arrivée. Un processus est caractérisé par sa date de création t_0 et sa durée d.

Soient les processus **PA**, **PB** et **PC** de dates de création respectives 0, 4 et 6, et de durées respectives 6, 12 et 6.

1. Déterminer pour chacun des temps t de 0 à 24 le processus actif. Il est possible de présenter le résultat sous forme d'un chronogramme.

Le *temps de latence* est le délai entre la création d'un processus et le début de son exécution. Le *temps CPU* est le temps total durant lequel le processeur exécute le processus. Le *temps réel écoulé* est le délai entre la création d'un processus et sa terminaison.

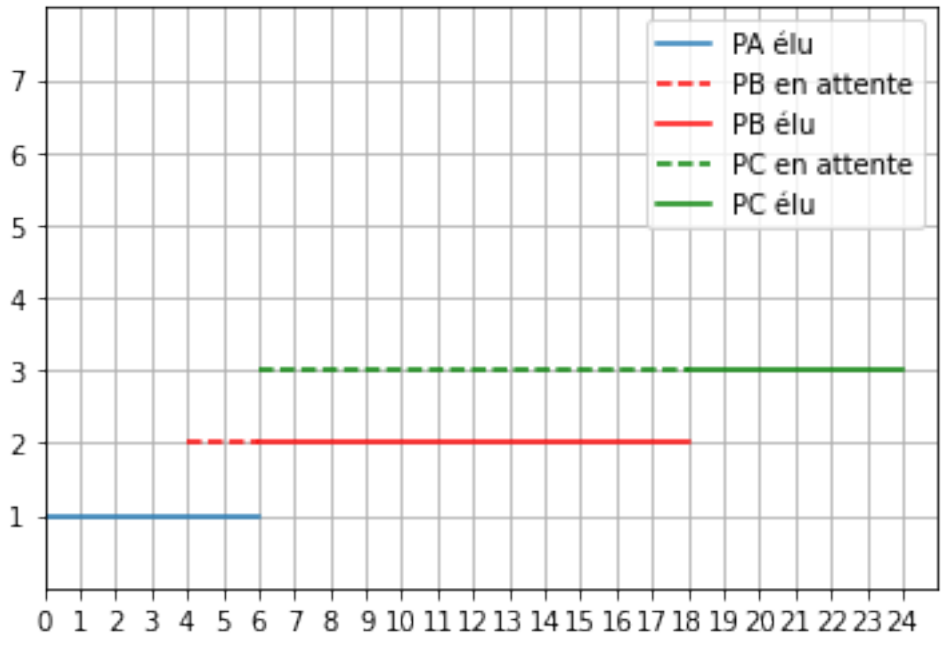
2. a. Quel est le temps de latence de chacun des processus ?

b. Quel est le temps CPU de chacun des processus ?

c. Quel est le temps réel écoulé de chacun des processus ?

On suppose maintenant un ordonnancement *préemptif*. La durée q du quantum est fixée à 2 unités. Parmi l'ensemble des processus prêts, le système choisit celui qui n'a pas été exécuté depuis le plus longtemps.

Question 1 Algorithme non préemptif (premier arrivé = premier servi), chronogramme d'état des processus



Question 2 Le **temps CPU** (ou temps de traitement) est la durée pendant laquelle une unité centrale de traitement (CPU) est utilisée pour traiter les instructions d'un programme informatique ou d'un système d'exploitation, par opposition au **temps réel** écoulé entre le début et la fin de l'exécution du programme.

	PA	PB	PC
Temps de latence	0	2	12
Temps CPU	6	12	6
Temps réel	6	14	18

3. a. Quel est le temps de latence de chacun des processus ?

b. Quel est le temps CPU de chacun des processus ?

c. Quel est le temps réel écoulé de chacun des processus ?

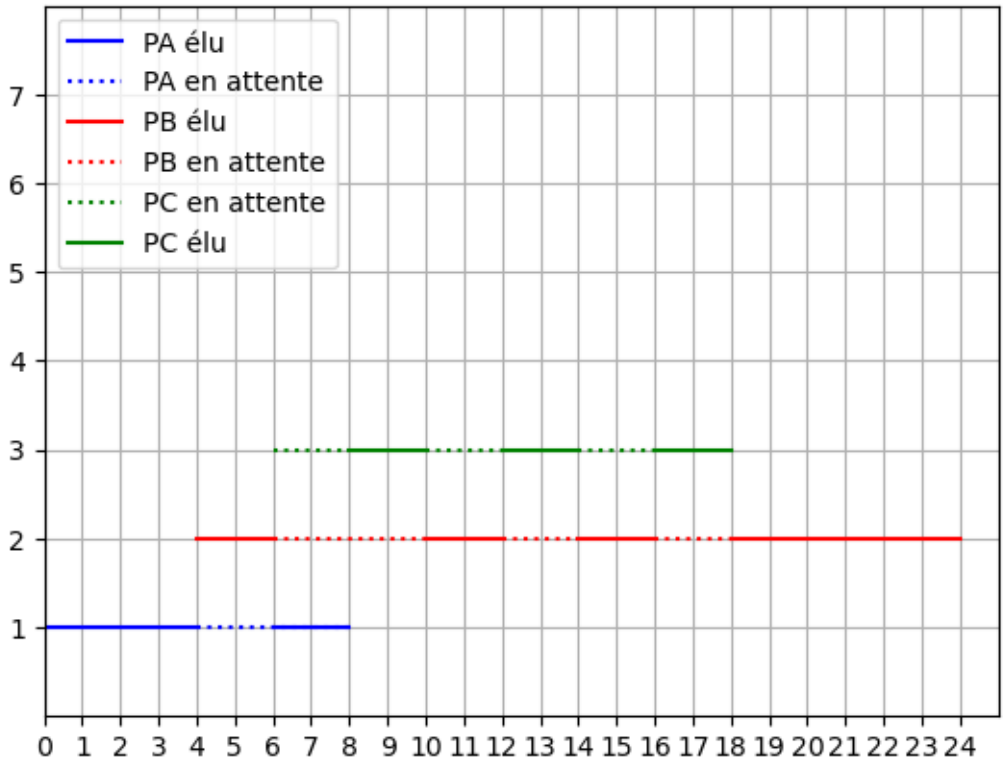
Soit un autre ensemble de processus à ordonnancer : **PW**, **PX**, **PY**, **PZ** de t_0 respectif 0, 1, 2, 3 et de même durée $d = 6$.

4. a. Pour chacun des ordonnancements FIFO et préemptif, donner les temps de latence, temps CPU et temps réels écoulés des quatre processus.

b. Sur la base de ces deux exemples, identifier l'avantage d'un ordonnancement préemptif sur un ordonnancement FIFO.

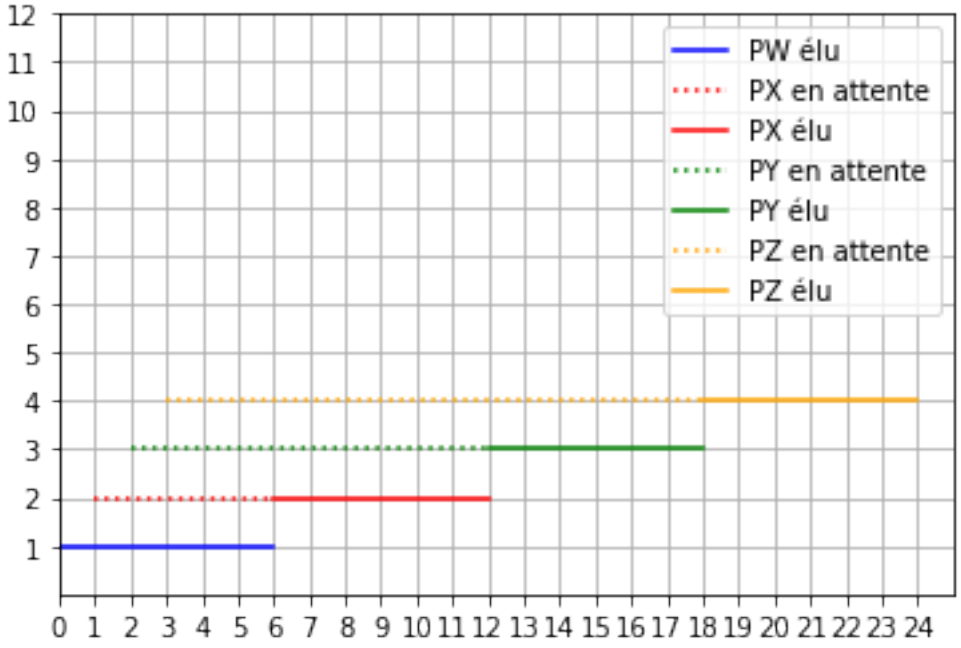
Question 3 Ordonnancement préemptif avec un quantum de 2 unités de temps de type tourniquet (l'ordonnanceur élit le processus qui ne s'est pas exécuté depuis le plus longtemps, les processus en attente sont dans une liste de type FIFO).

	PA	PB	PC
Temps de latence	0	0	2
Temps CPU	6	12	6
Temps réel	8	20	12



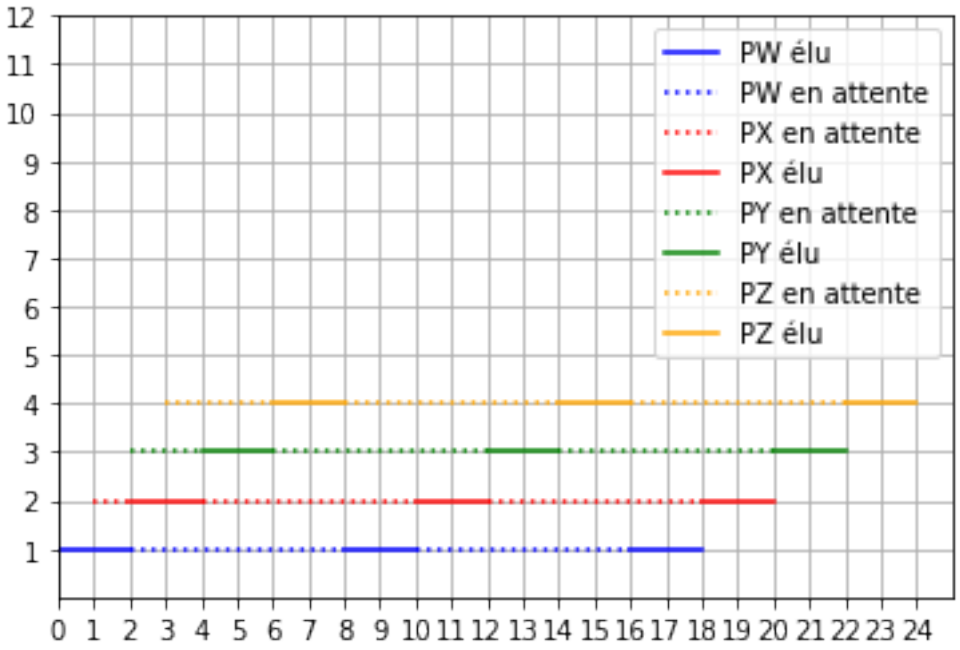
Question 4

On donne d'abord un chronogramme pour un ordonnancement non préemptif :



	PW	PX	PY	PZ
Temps de latence	0	5	10	15
Temps CPU	6	6	6	6
Temps réel	6	11	16	21

On donne ensuite un chronogramme pour un ordonnancement préemptif de type tourniquet avec file d'attente FIFO :



	PW	PX	PY	PZ
Temps de latence	0	1	2	3
Temps CPU	6	6	6	6
Temps réel	18	19	20	21

Comparaison des deux types d'ordonnancement

- L'ordonnancement non préemptif permet aux processus arrivés en premier de bénéficier d'un meilleur temps réel d'exécution mais au prix d'un temps de latence plus important pour les processus qui sont placés en fin de file d'attente.
- L'ordonnancement préemptif permet de mieux équilibrer à la fois le temps réel d'exécution (en particulier si tous les processus ont le même temps CPU) et le temps de latence qui est lissé et globalement plus réduit.

Exercice 5 *Bac candidats libres 2021 sujet 2*

On suppose que quatre processus C_1, C_2, C_3 et C_4 sont créés sur un ordinateur, et qu'aucun autre processus n'est lancé sur celui-ci, ni préalablement ni pendant l'exécution des quatre processus.
L'ordonnanceur, pour exécuter les différents processus prêts, les place dans une structure de données de type file FIFO. Un processus prêt est enfilé et un processus élu est défilé.

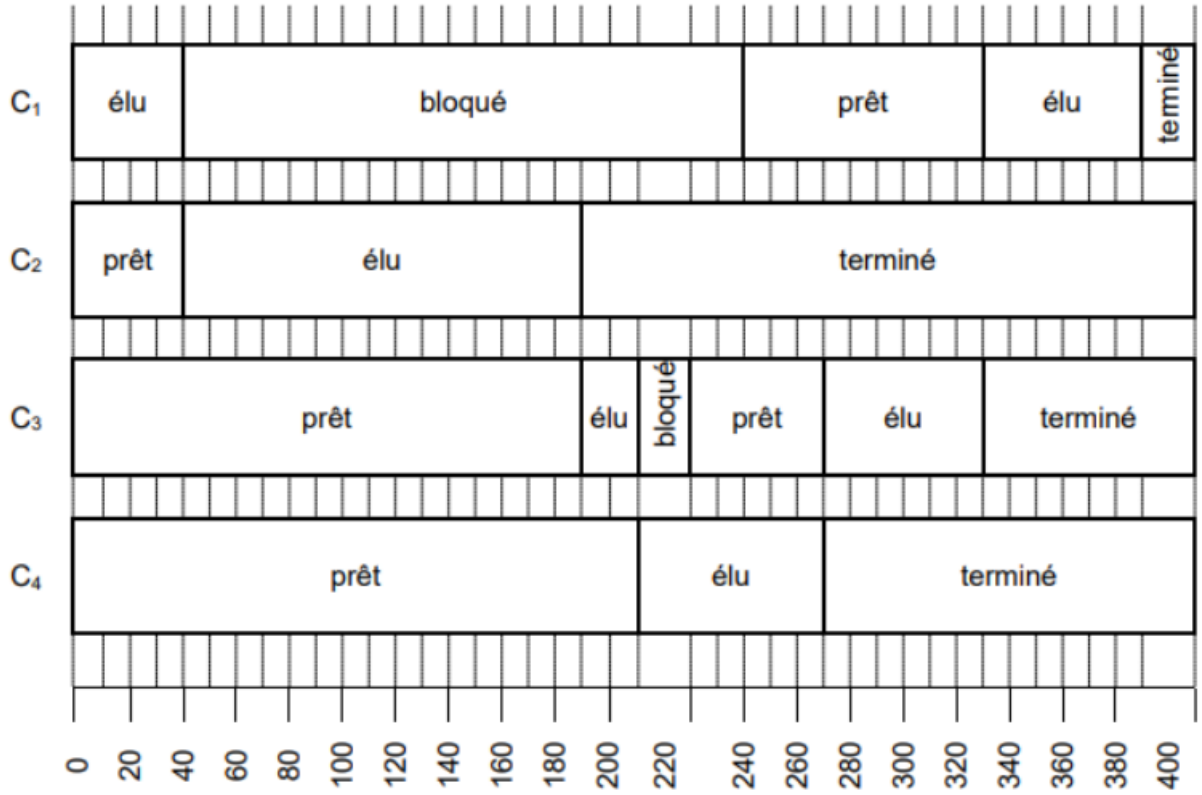
1. Décrire le fonctionnement des entrées/sorties dans une file de type FIFO.

   FIFO signifie Firts In First Out, autrement dit le premier élément entré dans la file est le premier à en sortir.

2. On suppose que les quatre processus arrivent dans la file et y sont placés dans l'ordre C_1, C_2, C_3 et C_4.

- Les temps d'exécution totaux de C_1, C_2, C_3 et C_4 sont respectivement 100 ms, 150 ms, 80 ms et 60 ms.
- Après 40 ms d'exécution, le processus C_1 demande une opération d'écriture disque, opération qui dure 200 ms. Pendant cette opération d'écriture, le processus C_1 passe à l'état bloqué.
- Après 20 ms d'exécution, le processus C_3 demande une opération d'écriture disque, opération qui dure 10 ms. Pendant cette opération d'écriture, le processus C_3 passe à l'état bloqué.

Sur le chronogramme ci-dessous, les états du processus C_2 sont donnés. Compléter avec les états des processus C_1, C_3 et C_4.



Exercice 6 *Amérique du Nord 2022 sujet 2*

On considère trois processus P_1, P_2 et P_3, tous soumis à l'instant 0 dans l'ordre 1, 2 et 3.

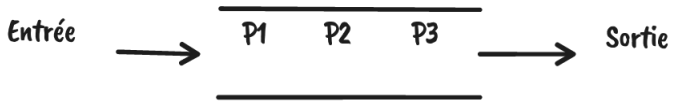
Nom du processus	Durée d'exécution en unités de temps	Ordre de soumission
P_1	3	1
P_2	1	2
P_3	4	3

1. Dans cette question, on considère que les processus sont exécutés de manière concurrente selon la politique du tourniquet : le temps est découpé en tranches nommées *quantums de temps*.

Les processus prêts à être exécutés sont placés dans une file d'attente selon leur ordre de soumission.

Lorsqu'un processus est élu, il s'exécute au plus durant un quantum de temps. Si le processus n'a pas terminé son exécution à l'issue du quantum de temps, il réintègre la file des processus prêts

(côté entrée). Un autre processus, désormais en tête de la file (côté sortie) des processus prêts, est alors à son tour élu pour une durée égale à un quantum de temps maximum.



Compléter le tableau ci-dessous en indiquant dans chacune des cases le processus exécuté à chaque cycle. Le quantum correspond à une unité de temps.

P1	P2	P3	P1	P3	P1	P3	P3

2. Dans cette question, on considère que les processus sont exécutés en appliquant la politique du *plus court d'abord* : les processus sont exécutés complètement dans l'ordre croissant de leurs temps d'exécution, le plus court étant exécuté en premier.

Compléter le tableau ci-dessous en indiquant dans chacune des cases le processus exécuté à chaque cycle.

P2	P1	P1	P1	P3	P3	P3	P3

Exercice 7 ***Implémentation d'algorithmes d'ordonnancement***

Exercice 28 page 261 du manuel à faire dans Capytale :
https://capytale2.ac-paris.fr/web/c/add5-626257/mcer.

3 Ressource en accès exclusif et interblocage

3.1 Accès concurrent à une ressource partagée

Exercice 8 *Ressource partagée et processus concurrents*

On considère le programme `prog_concurrente.py`. Une instance d'exécution de ce programme crée un processus mais ce processus est constitué lui même de deux *processus légers* ou thread qui sont deux fils d'exécution concurrents de la même fonction `processus`. En général les threads permettent de faire exécuter deux tâches en parallèle par un même processus : par exemple dans un programme d'interface graphique, un thread gère les attentes d'événements et un autre les calculs lourds. Dans notre programme `prog_concurrente.py`, les threads `p1` et `p2` vont écrire dans le même fichier `"test_concurrence_verrou.txt"`.

```
import threading


def processus(pid):
    f = open("test_concurrence_verrou.txt", "w")
    for i in range(1, 10001):
        f.write("pid :" + str(pid) + " : " + str(i) + "\n")
        f.flush()
    f.close()

p1 = threading.Thread(target=processus, args=(1,))
p2 = threading.Thread(target=processus, args=(2,))
p1.start()
p2.start()
```

1. Quel nombre de lignes peut-on attendre dans le fichier `test_concurrence.txt` après exécution de ce programme ?

   Puisque les deux processus légers s'exécutent en parallèle et que chaque thread écrit 10000 lignes dans `"test_concurrence_verrou.txt"`, on peut s'attendre à ce que ce fichier contienne finalement $2 \times 10000 = 20000$ lignes.

2. Exécuter le programme puis éditer le fichier `test_concurrence.txt`, quel est le nombre de lignes du programme ? Après exécution du programme, on constate que `test_concurrence.txt` contient seulement 10000 lignes.

3. Exécuter de nouveau le programme puis éditer le fichier `test_concurrence.txt`. Le contenu du fichier est-il prévisible ? Donner une explication.

   Après chaque exécution du programme, on constate que `test_concurrence.txt` contient seulement 10000 lignes. De plus les numéros de ligne se suivent bien. On peut conjecturer que chaque processus légers a écrit dans le fichier `test_concurrence.txt` en écrasant la ligne écrite par l'autre processus si ce dernier avait déjà atteint cette ligne dans sa progression. Les deux processus légers accèdent à la ressource de façon concurrente et cela créé un résultat imprédictible car l'ordre d'apparition des PID va changer si on renouvelle l'expérience. Ce n'est pas acceptable, il faudrait verrouiller l'accès à la ressource lorsqu'un processus l'utilise.

Exercice 9 *Verrou sur une ressource en accès exclusif*

Le programme `prog_concurrente2.py` est une modification du programme `prog_concurrente.py` avec un verrou créé à l'aide du module filelock.

```
import threading
from filelock import FileLock


def processus(pid):
    verrou = FileLock(f"test_concurrence_verrou.txt.lock")
    verrou.acquire()
    print(
        "Acquisition du verrou sur la ressource par le processus de pid ",
            pid
    )
    f = open("test_concurrence_verrou.txt", "w")
    for i in range(1, 10001):
        f.write("pid :" + str(pid) + " : " + str(i) + "\n")
        f.flush()
    f.close()
    print(
        "Libération du verrou sur la ressource par le processus de pid ",
            pid
    )
    verrou.release()


p1 = threading.Thread(target=processus, args=(1,))
p2 = threading.Thread(target=processus, args=(2,))
p1.start()
p2.start()
```

1. Éxécuter ce programme et vérifier le contenu du fichier `"test_concurrence_verrou.txt"`. Obtient-on les mêmes résultats que dans l'exercice précédent?

   Cette fois un seul processus léger a pu accéder en écriture de façon exclusive au fichier `"test_concurrenc`
   L'autre processus n'a pas pu accéder au fichier, comme s'il avait été verrouillé.

2. Permuter les instructions `p1.start()` et `p2.start()` puis exécuter. Quel est le changement pour le fichier `"test_concurrence_verrou.txt"`?

   Si on permute l'ordre dans lequel débutent les processus légers `p1` et `p2`, on observe que le processus qui a pu accéder de façon exclusive au fichier en écriture, a changé.

3. D'après l'article Wikipedia quel est le rôle d'un verrou informatique et quels risques permet-il de prévenir? Dans le programme `prog_concurrente2.py`, quelle ressources est verrouillée? Donner les numéros de ligne où le verrou est posé puis relâché.

   Un verrou informatique permet de s'assurer qu'une seule personne, ou un seul processus accède à une ressource à un instant donné. Ceci est souvent utilisé dans le domaine des accès à des fichiers

sur des systèmes d'exploitation multi-utilisateur, car si deux programmes modifient un même fichier au même moment, le risque est de :

- provoquer des erreurs dans un des deux programmes, voire dans les deux ;
- laisser le fichier en fin de traitement dans une complète incohérence ;
- endommager le fichier manipulé

Méthode *Verrou, voir manuel p. 246.*

Prenons l'exemple de la réservation d'une place de concert sur un site Web. Imaginons qu'un processus A commence la réservation à un instant t_A et qu'un processus B peut accéder à la même place et finaliser sa réservation après t_A et avant la fin du processus A. C'est un problème posé par l'accès concurrent à une ressource partagée.
Pour assurer l'**accès exclusif** d'une ressource à un seul processus à la fois, les systèmes d'exploitation offrent un service de **verrou d'exclusion mutuelle** ou *mutex*.
Dans le programme, la *section critique* qui doit être protégée, est placée entre l'acquisition du verrou et son relâchement. Si plusieurs instances du programme s'exécutent, la première qui accaparent le verrou, peut accéder à la ressource et poursuivre son exécution. Les autres processus passent en état bloqué tant que le verrou n'est pas relâché.

```
Code non protégé
Acquisition du verrou
    Accès exclusif à la ressource, section critique
Relachement du verrou
```

3.2 Interblocage

Exercice 10

Considérons un ordinateur où s'exécutent seulement deux processus P1 et P2 avec deux ressources en accès exclusif RA et RB protégées par des verrous verrouA et verrouB.
On fait l'hypothèse que le coût de chaque instruction est de 1 unité de temps (6 unités par processus) et que l'ordonnanceur applique un algorithme de tourniquet : un processus est élu pendant un quantum de temps puis il est préempté et l'autre processus est élu etc … Si un processus est bloqué, son quantum d'exécution est interrompu et l'autre processus est élu. Le processus P1 est prêt à l'instant 0 et le processus P2 est prêt à l'instant 1.
Nous allons faire varier la valeur du quantum en unités de temps puis observer l'effet sur l'ordonnancement.

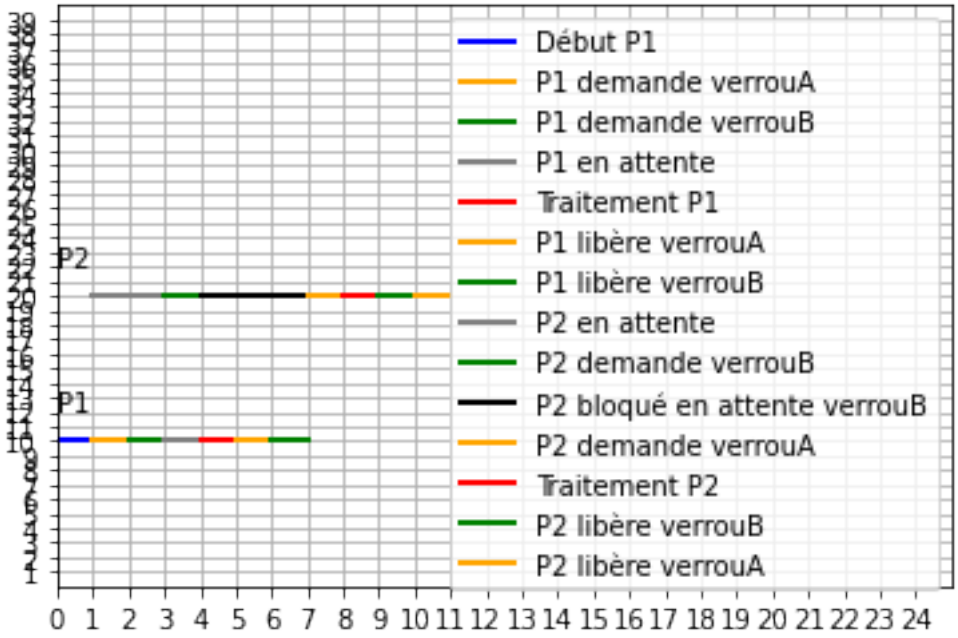
Programme du processus P1

```
Début de P1
Demande verrouA
Demande verrouB
Traitement de P1
Libère verrouA
Libère verrouB
```

Programme du processus P2

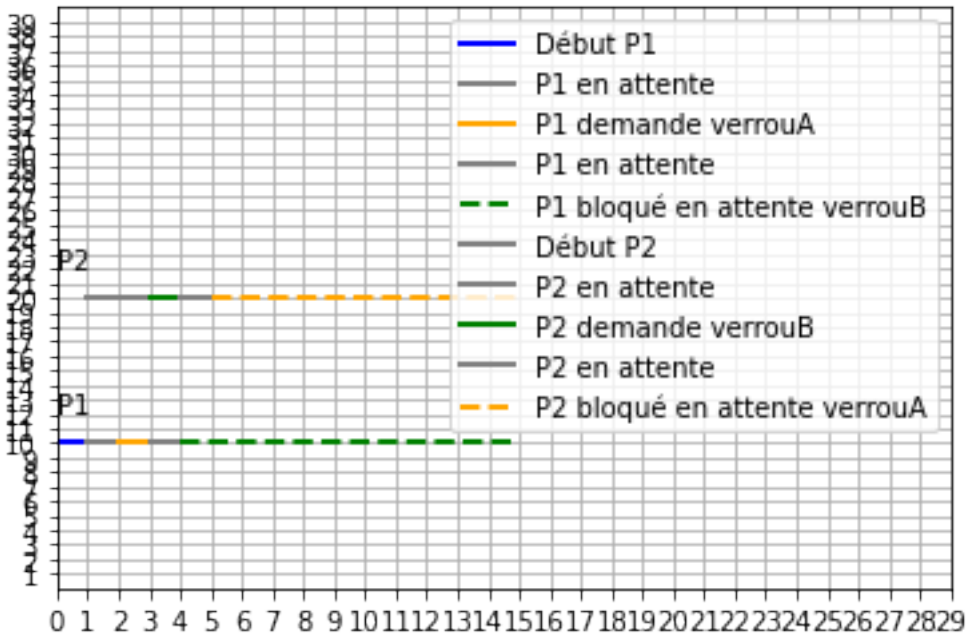
```
Début de P2
Demande verrouB
Demande verrouA
Traitement de P2
Libère verrouB
Libère verrouA
```

1. On choisit un quantum de 3 unité de temps, construire le chronogramme d'ordonnancement des deux processus.



2. On choisit un quantum de 1 unité de temps, construire le chronogramme d'ordonnancement des deux processus. Que va-t-il se passer ?

Au temps 3 le processus P1 détient le verrouA et au temps 4, le processus P2 détient le verrouB. Ensuite P1 et P2 passent en état bloqués puisqu'ils attendent que l'autre processus libère le verrou qu'il détient avant de libérer leur propre verrou : ils sont mutuellement bloqués, on parle d'**interblocage**.



3. Comment pourrait-on modifier le programme du processus P2 pour que le problème précédent ne puisse pas se produire ?

On peut échanger l'ordre d'acquisition des verrous dans le programme du processus P2 pour qu'il soit identique à celui du programme du processus P1. Ainsi si P1 acquiert le verrouA en premier, P2 restera bloqué et ne pourra pas acquérir le verrouB. P1 pourra s'exécuter complètement, acquérir le verrouB, libérer le verrouA puis le verrouB et ensuite P2 pourra être débloqué, acquérir le verrouA puis le verrouB et s'exécuter lui aussi complètement.

Exercice 11

Se placer dans un répertoire contenant le fichier `interblocage_jean_diraison.py` puis exécuter ce script écrit par un collègue dans deux lignes de commande distinctes.
Que peut-on observer au bout d'un certain temps ? Renouveler l'expérience.
Éditer le contenu du script et proposer une explication du comportement observé.
En lançant deux processus d'exécution de ce programme on aboutit assez vite à un interblocage : chaque processus détient un verrou sur un fichier et veut acquérir un verrou sur un fichier détenu par l'autre. L'interblocage n'est pas immédiat, il faut attendre que les deux processus essaient d'acquérir les mêmes verrous dans des ordres inverses.

```
administrateur@smob-ubuntu-p01:~$ python3 interblocage_jean_diraison.py
 8   verrouiller: C libérer: B
administrateur@smob-ubuntu-p01:~$ python3 interblocage_jean_diraison.py
 50   verrouiller: B libérer: C
```

Commentaires dans le script du programme :

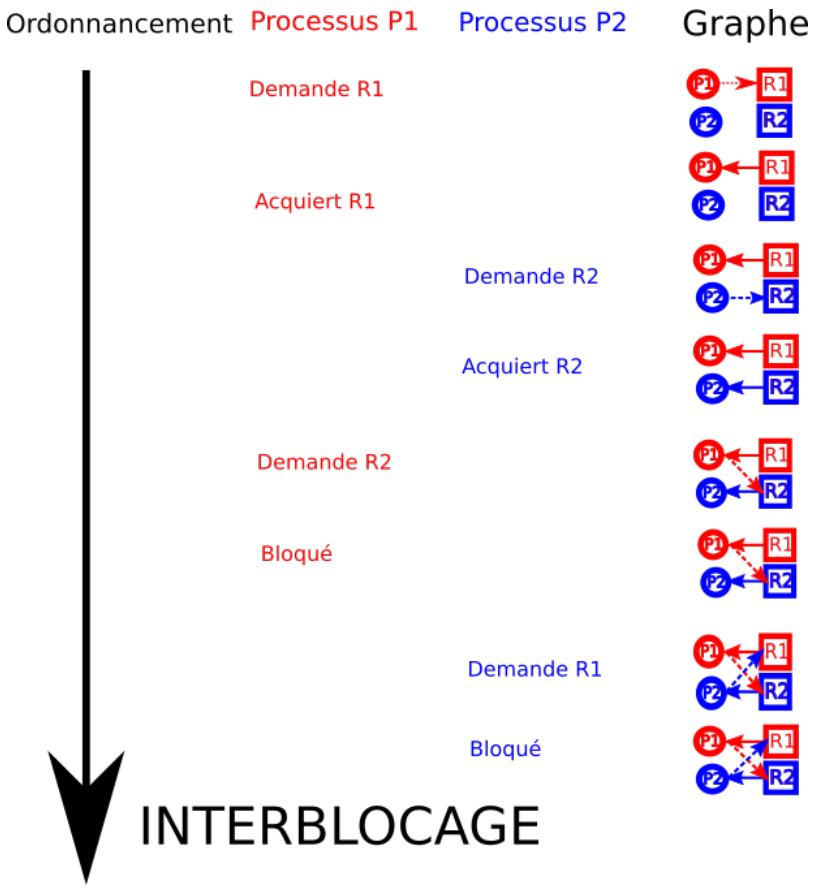
```
from random import choice
from time import sleep
from filelock import (
    FileLock,
)  # apt-get install python3-filelock ou pip/pip3 install filelock


def main(caracteres):
    """
    Boucle sans fin choisissant un caractere au hasard et l'écrivant dans
         le fichier
    apres verrouillage puis liberation de l'ancien verrou
    """
    compteur = 0
    caractere = ""
    verrou = None
    while True:
        precedent = caractere
        caractere = choice(caracteres)
        compteur = compteur + 1
        print(
            "\r",
            compteur,
            "  verrouiller:",
            caractere,
            "  libérer:",
            precedent,
            end="",
            flush=True,
        )
        if caractere != precedent:
            ancien_verrou = verrou
            verrou = FileLock(caractere + ".lock")
            # Demande de verrou => processus bloqué tant que verrou non lib
                 éré
            verrou.acquire()
            # Libération du verrou précédent
            if ancien_verrou is not None:
                ancien_verrou.release()
        fichier = open(caractere, "w")
        fichier.write(caractere)
        fichier.close()
        sleep(0.1)


main("ABCDEFGHIJ")
```

Définition 5 *Interblocage*

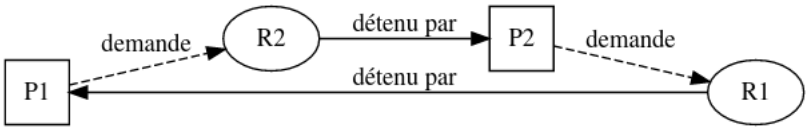
Lire l'explication sur l'interblocage dans le manuel entre les pages 246 et 249.



L'utilisation de verrous permet de bloquer l'accès à des ressources en accès exclusif mais présente le risque d'**interblocage**, par exemple si deux processus se bloquent mutuellement la ressource dont ils ont besoin . Si on représente les relations entre processus et ressources par un graphe orienté de dépendances, une situation d'interblocage correspond à la présence d'un cycle.
On peut aussi caractériser une situation d'interblocage par la vérification de certaines conditions. Par exemple avec deux processus P1 et P2 et deux ressources en accès exclusifs :

- *Ressources en accès exclusif :* le processus P1 possède un verrou sur la ressource A et le processus P2 un verrou sur la ressource B ;
- *Dépendance circulaire :* P1 demande B avant de libérer A et P2 demande A avant de libérer B ;
- *Non préemption :* il est impossible de préempter la ressource détenue par un processus.

Si plusieurs processus utilisent des verrous sur les mêmes ressources, une situation d'interblocage peut être prévenue par le programmeur si ***le même ordre de demande des verrous*** *est défini dans tous les programmes.*



Exercice 12 *Candidats libres 2021 sujet 2*

On trouvera ci- dessous deux programmes rédigés en pseudo-code.
Verrouiller un fichier signifie que le programme demande un accès exclusif au fichier et l'obtient si le fichier est disponible.

Programme 1

```
Verrouiller fichier_1
Calculs sur fichier_1
Verrouiller fichier_2
Calculs sur fichier_1
Calculs sur fichier_2
Calculs sur fichier_1
Déverrouiller fichier_2
Déverrouiller fichier_1
```

Programme 2

```
Verrouiller fichier_2
Verrouiller fichier_1
Calculs sur fichier_1
Calculs sur fichier_2
Déverrouiller fichier_1
Déverrouiller fichier_2
```

1. En supposant que les processus correspondant à ces programmes s'exécutent de façon concurrente et que l'ordonnancement est préemptif, on peut aboutir à une situation d'interblocage.

   Exemple de chronogramme aboutissant à un interblocage :

Quantum de temps	Processus P1	Processus P2
1	Elu, verrouiller fichier 1	Pas prêt
2	Elu, calculs sur fichier 1	Pas prêt
3	Prêt, en attente	Elu, verrouiller fichier 2
4	Elu, demande fichier 2 qui est verrouillé par P2	Prêt, en attente
5	Bloqué en attente déverrouillage fichier 2	Elu, demande fichier 1 qui est verrouillé par P1
6	Bloqué en attente déverrouillage fichier 2 par P2	Bloqué en attente déverrouillage fichier 1 par P2

2. Proposer une modification du programme 2 permettant d'éviter ce problème.

   On peut permuter les deux premières lignes du programme du processus P2 pour que l'ordre de demande des verrous soit le même dans les deux processus. Ainsi le premier processus qui détient le verrou sur le fichier 1 bloque l'autre processus et peut acquérir le verrou sur le fichier 2 et s'exécuter complètement sans provoquer d'interblocage.

Exercice 13 *Amérique du nord 2022 sujet 2*

On considère trois ressources R_1, R_2 et R_3 et trois processus P_1, P_2 et P_3 dont les programmes (une instruction exécutable en un quantum d'unité de temps processeur) sont indiqués ci-dessous :

Processus P1

```
Demande R1
Demande R2
Libère R1
Libère R2
```

Processus P2

```
Demande R2
Demande R3
Libère R2
Libère R3
```

Processus P3

```
Demande R3
Demande R1
Libère R3
Libère R1
```

1. Rappeler les différents états d'un processus. On suppose un ordonnancement préemptif. Illustrer le risque d'interblocage, en proposant un chronogramme d'ordonnancement le provoquant.

   Exemple de chronogramme aboutissant à un interblocage :

Quantum de temps	Processus P1	Processus P2	Processus P3
1	Elu, acquiert R1	Prêt, en attente	Prêt, en attente
2	Prêt, en attente	Elu, acquiert R2	Prêt, en attente
3	Prêt, en attente	Prêt, en attente	Elu, acquiert R3
4	Elu, demande R2	Prêt, en attente	Prêt, en attente
5	Bloqué, attend R2	Elu, demande R3	Prêt, en attente
6	Bloqué, attend R2	Bloqué attend R3	Elu demande R1
7	Bloqué, attend R2	Bloqué attend R3	Bloqué attend R1

2. Proposer un chronogramme d'ordonnancement sans interblocage.

   Pour lever l'interblocage, il suffit que les trois processus ne soient pas concurrents et s'exécutent l'un après l'autre.

   Elu libère R3 Exemple de chronogramme aboutissant à un interblocage :

Quantum de temps	Processus P1	Processus P2	Processus P3
1	Elu, acquiert R1	Pas prêt	Pas prêt
2	Elu, acquiert R2	Pas prêt	Pas prêt
3	Elu, libère R1	Pas prêt	Pas prêt
4	Elu, libère R2	Pas prêt	Pas prêt
5	Terminé	Elu demande R2	Pas prêt
6	Terminé	Elu demande R3	Pas prêt
7	Terminé	Elu libère R2	Pas prêt
8	Terminé	Elu libère R3	Pas prêt
9	Terminé	Terminé	Elu demande R3
10	Terminé	Terminé	Elu demande R1
11	Terminé	Terminé	Elu, libère R3
12	Terminé	Terminé	Elu, libère R1

3. Proposer une modification du programme du processus P2 qui permettrait de prévenir tout interblocage.

   Il suffit d'inverser l'ordre de demande des verrous sur les ressources R2 et R3 dans le programme de P2 pour qu'il demande d'abord la ressource R3 comme le processus P3. Ainsi la ressource R3 peut être détenue par P2 ou P3, le premier qui la prend, indépendamment de l'exécution de P1. On distingue alors deux cas :

   - Si P2 détient R3 en premier il bloque d'abord P3 pour s'exécuter complètement sans interblocage avec P1 puisque P1 et P2 demandent une seule ressource commune R2 : le premier à la verrouiller pourra s'exécuter sans interblocage et l'autre s'exécutera ensuite. Quand P2 est terminé, P3 peut s'exécuter sans interblocage avec P1 puisque sur les deux ressources demandées par P1 et P3 au moins une est différente donc le premier à verrouiller la ressource pourra s'exécuter jusqu'au bout.
   - Sinon P3 détient R3 en premier et bloque P2 pour s'exécuter complètement sans interblocage avec P1 selon le même argument que précédemment. Ensuite P2 peut s'exécuter complètement là aussi sans interblocage avec P1 puisque P1 et P2 ne demandent qu'une seule ressource commune.

Table des matières