Introduction

Les ordinateurs modernes sont multitâches : traitement de texte, navigateur web, environnement de programmation Python peuvent s'exécuter en même temps. De plus de nombreux services s'exécutent de façon invisible, en arrière plan : pour la gestion des impressions, du courrier électronique, des fenêtres. L'ordinateur donne l'impression de pouvoir exécuter des programmes en parallèle. Pourtant on a vu en classe de première, que dans le modèle de Von Neumann, l'unité centrale exécutait une seule instruction à la fois. Par ailleurs, si on clique deux fois sur le raccourci du navigateur Web on peut lancer deux fils d'exécution indépendants de ce programme. Il semble donc nécessaire de distinguer un programme, constitué d'une série d'instructions, d'une instance d'exécution de ce programme.

Sources :

- *« Systèmes d'exploitation » de Andrew Tanenbaum.*
- *« Histoire illustrée de l'informatique » de E.Lazard et P.Mounier-Kuhn aux éditions EDPSciences.*
- *« Manuel de Terminale NSI » de Michel Baudouin-Lafon aux éditions Hachette.*
- *« Manuel de Terminale NSI » de T. Balabonski, S. Conchon, JC. Filliâtre, K.Nguyen aux éditions Ellipse.*
- *« Cours de Terminale NSI » de Gilles Lassus : https://glassus.github.io/terminale_nsi/.*
- *« Cours de Terminale NSI » de Laurent Cooper :* *http://lycee.educinfo.org/index.php?page=introduction&activite=processus.*

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 ?

....

....

4. Exécuter la commande `ps -eo pid,stat,command` (*pas d'espace dans* `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 `STAT` ?

....

....

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 ?

. . . .

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 ?

. . . .

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 ?

. . . .

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 ?

. . . .

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`.

. . . .

. . . .

. . . .

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`.

```
fjunier@fjunier:~$ pstree -p
systemd(1)─┬─ModemManager(1480)─┬─{ModemManager}(1533)
           │                    └─{ModemManager}(1536)
           ├─NetworkManager(1371)─┬─{NetworkManager}(1488)
           │                      └─{NetworkManager}(1489)
           ├─accounts-daemon(1362)─┬─{accounts-daemon}(1386)
           │                       └─{accounts-daemon}(1471)
           ├─acpid(1363)
           ├─anydesk(1500)─┬─{anydesk}(1552)
           │               ├─{anydesk}(1553)
           │               └─{anydesk}(3354)
           ├─apache2(1626)─┬─apache2(1635)
           │               ├─apache2(1636)
           │               ├─apache2(1637)
           │               ├─apache2(1638)
           │               └─apache2(1639)
           ├─atd(1403)
           ├─avahi-daemon(1367)───avahi-daemon(1408)
           ├─colord(1477)─┬─{colord}(1493)
           │              └─{colord}(1497)
```

Quelle structure reconnaît-on dans cet affichage ?

. . . .

. . . .

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 `PPID 1` ? Reconstruire le chemin de ce processus vers le premier processus lancé avec la commande `geany`.

. . . .

. . . .

. . . .

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.

   ....

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 vont faire ces programmes si on les exécute ?

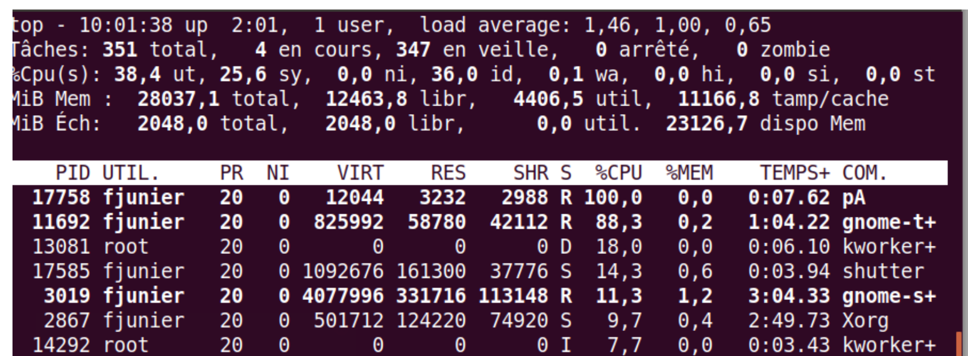
   ....

   ....

. . . .

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` .



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

. . . .

. . . .

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

. . . .

. . . .

. . . .

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

. . . .

. . . .

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.

. . . .

. . . .

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.

Que signifient ces priorités ? nous allons le découvrir.

Afficher le contenu des dix premières lignes de `output` dans la console avec `head -10 output` puis des dix dernières lignes avec `tail -10 output`. Observer la répartition des caractères `"B"` et `"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 ?

....

....

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

....

....

....

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

....

....

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

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

Que peut-on observer ?

....

....

....

d. Reprendre la question **a.** avec la commande `nice -n 0 ./pC & nice -n 0 ./pB & ps`.

Que peut-on observer ?

....

....

....

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

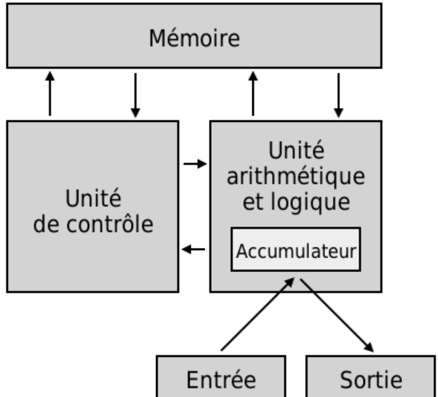
....

....

....

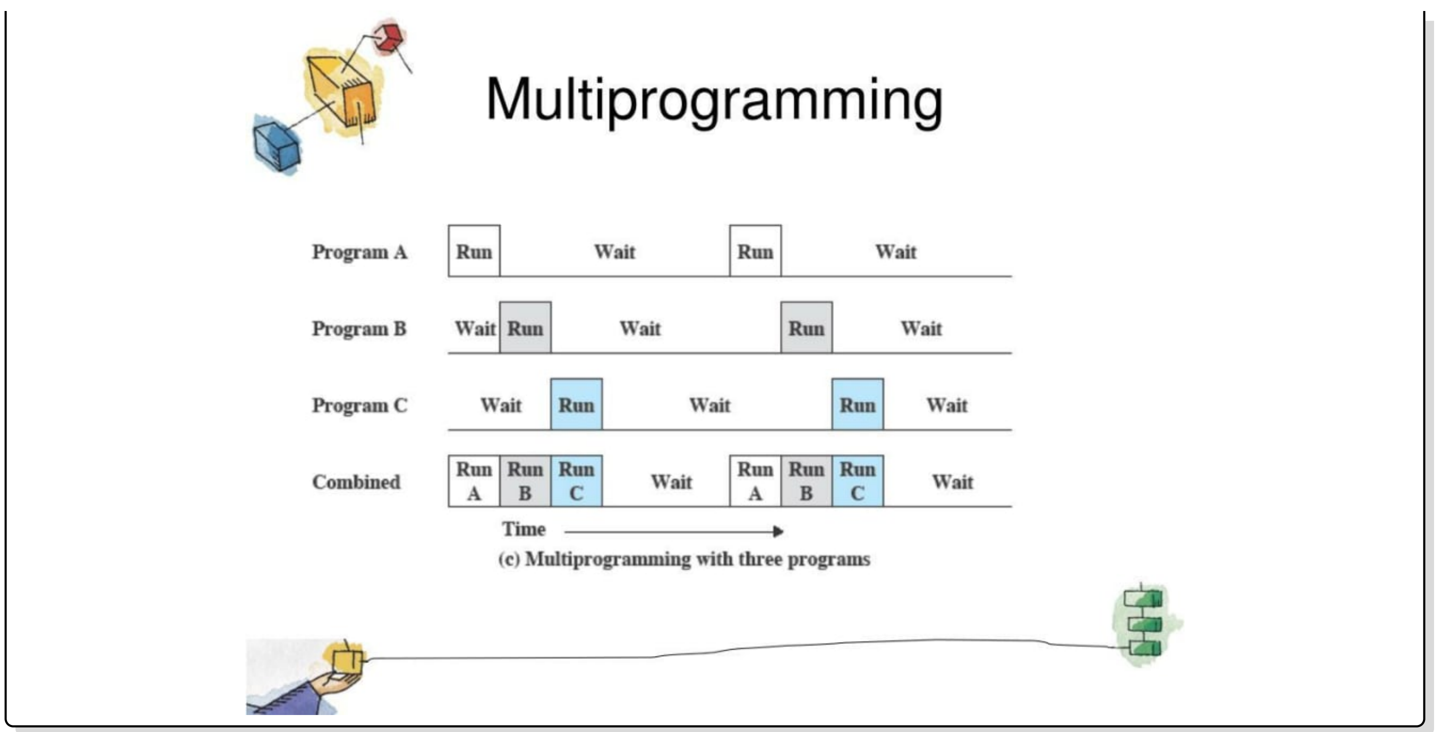
Définition 2 *Multiprogrammation*

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 ...



Histoire 1 ***De Multics à Linux en passant par Unix***

Le projet d'un système d'exploitation multiutilisateurs en temps partagé Multics, a été lancé en 1964 par le MIT, General Electric et les laboratoires Bell d'AT&T.

Introduisant de nombreuses innovations, ce projet ambitieux prend du retard. En raison de son coût, les laboratoires Bell abandonnent le projet en 1969. Ken Thompson et Dennis Ritchie de l'équipe Multics chez Bell, poursuivent le projet de leur propre initiative. Ils écrivent une version simplifiée, un système d'exploitation multitâches et multiutilisateurs (mais avec un seul utilisateur en même temps), qu'ils nomment Unics, rapidement transformé en Unix. D'abord développé sur PDP-7, Unix est porté sur le mini-ordinateur le plus populaire de l'époque, le PDP-11.

Ken Thompson et Dennis Ritchie réécrivent Unix en 1972 dans le langage C qu'ils ont créé spécialement. Démontrant son efficacité, Unix est adopté par les autres services de Bell, mais la loi anti-trust empêche AT&T de le commercialiser.

Il est alors diffusé dans les universités et en particulier à Berkeley, à des fins pédagogiques, sous une licence à prix modique. Dès 1977, des chercheurs de Berkeley apportent de nombreuses améliorations à Unix et le diffusent sous le nom de Berkeley Software Distribution.

Au début des années 1980, le département américain de recherche DARPA finance le développement de nouvelles versions de la BSD pour implémenter la pile TCP/IP à la base d'Internet.

Des procès opposent AT&T et BSD sur la propriété des codes sources utilisés dans les différentes versions d'UNIX, la première version de BSD libérée de tout code propriétaire AT&T est publiée en 1989.

Unix connaît une large descendance avec le développement du noyau Linux (logiciel libre) en 1991 par Linus Torvalds et celui de MacOS (semi-propriétaire) vers l'an 2000.

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** :

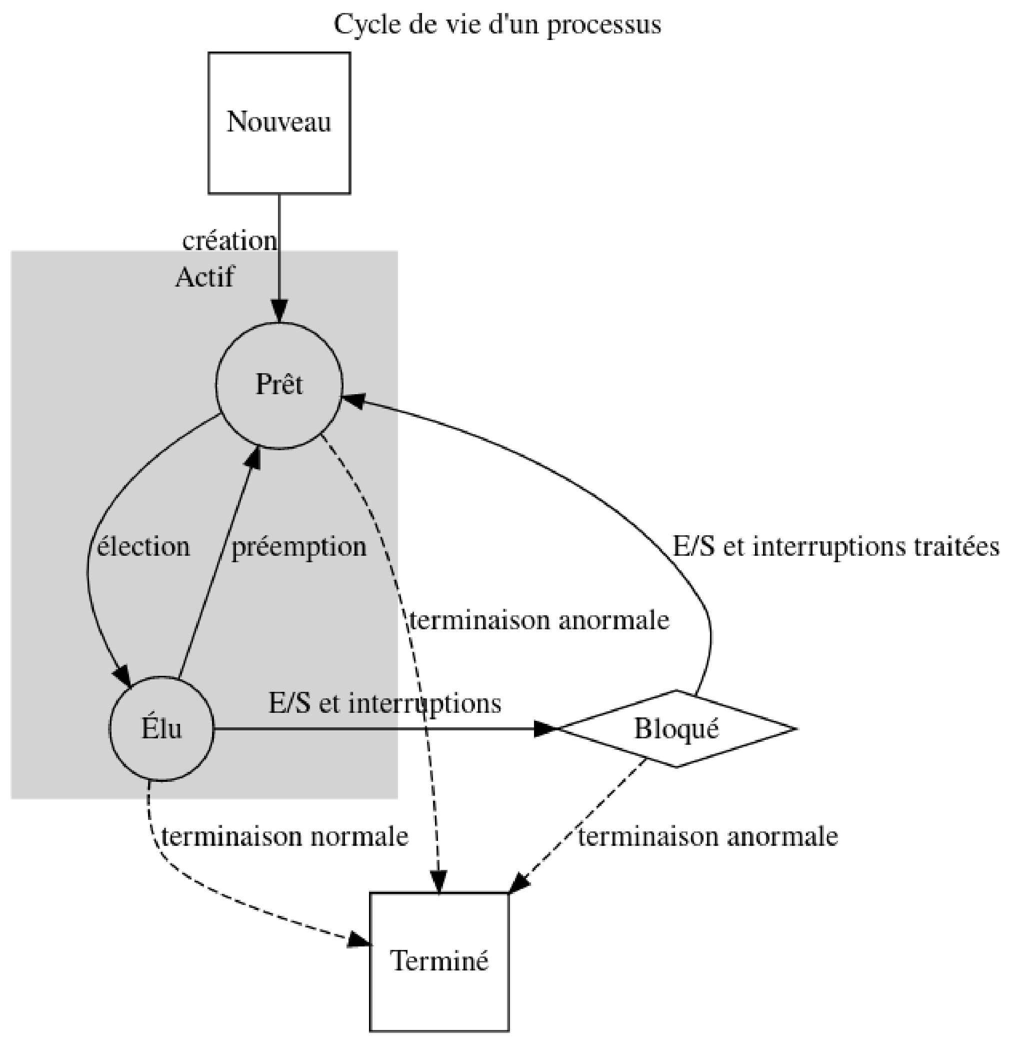
- choix du plus récent (FIFO) ;
- choix de celui avec le temps d'exécution le plus court;
- choix par priorité.

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.

   ....

   ....

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

   ....

   ....

   ....

   ....

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

   ....

   ....

   ....

   ....

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

   ....

   ....

   ....

   ....

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.

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.

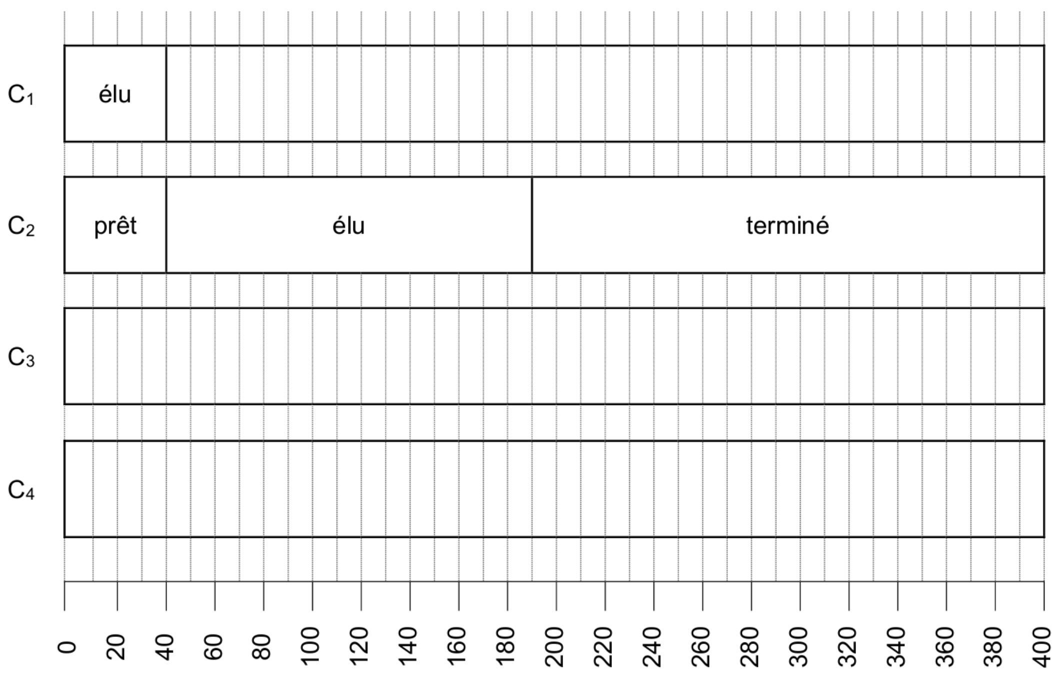
1. Décrire le fonctionnement des entrées/sorties dans une file de type FIFO.

…………………………………………………………………………………………………

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*

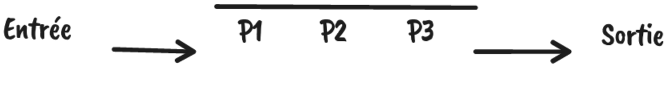
On considère trois processus P_1, P_2 et P_3, tous soumis à l'instant 0 dans l'ordre 1, 2 et 3.

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

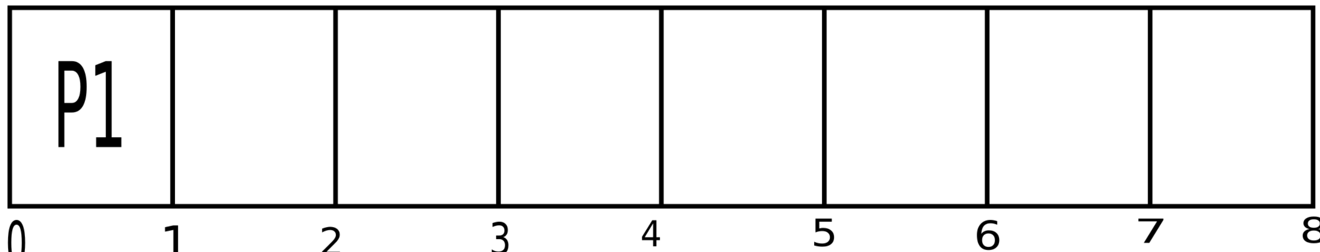
1. Dans cette question, on considère que les processus sont exécutés de manière concurrente selon la politique du tourniquet : le temps est découpé en tranches nommées *quantums de temps.*

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.

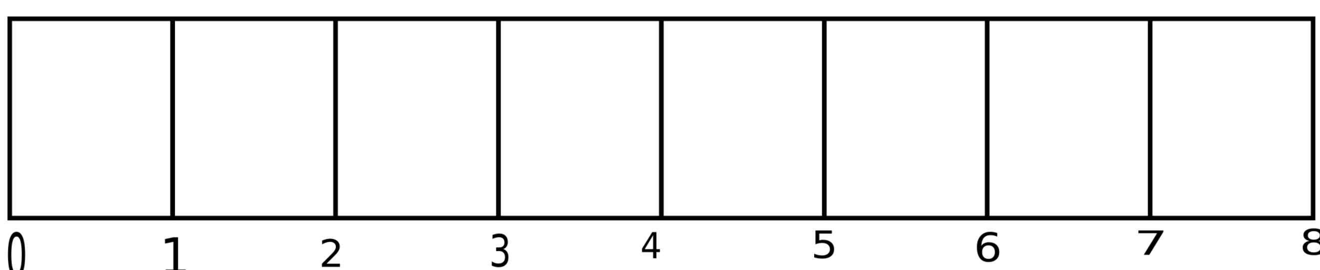


Compléter le tableau ci-dessous en indiquant dans chacune des cases le processus exécuté à chaque cycle. Le quantum correspond à une unité de temps.



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.



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 en parallèle dans le même fichier `"test_concurrence_verrou.txt"`.

```python
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 ?

. . . .

2. Exécuter le programme puis éditer le fichier `test_concurrence.txt`, quel est le nombre de lignes du programme ?

. . . .

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

. . . .

. . . .

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.

```
1  import threading
2  from filelock import FileLock
3
4
5  def processus(pid):
6      verrou = FileLock(f"test_concurrence_verrou.txt.lock")
7      verrou.acquire()
8      print(
9          "Acquisition du verrou sur la ressource par le processus de pid ",
               pid
10     )
11     f = open("test_concurrence_verrou.txt", "w")
12     for i in range(1, 10001):
13         f.write("pid :" + str(pid) + " : " + str(i) + "\n")
```

```
14          f.flush()
15      f.close()
16      print(
17          "Libération du verrou sur la ressource par le processus de pid ",
                pid
18      )
19      verrou.release()
20
21
22  p1 = threading.Thread(target=processus, args=(1,))
23  p2 = threading.Thread(target=processus, args=(2,))
24  p1.start()
25  p2.start()
```

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

....

....

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

....

....

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.

....

....

....

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 ?

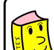
   . . . .

   . . . .

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

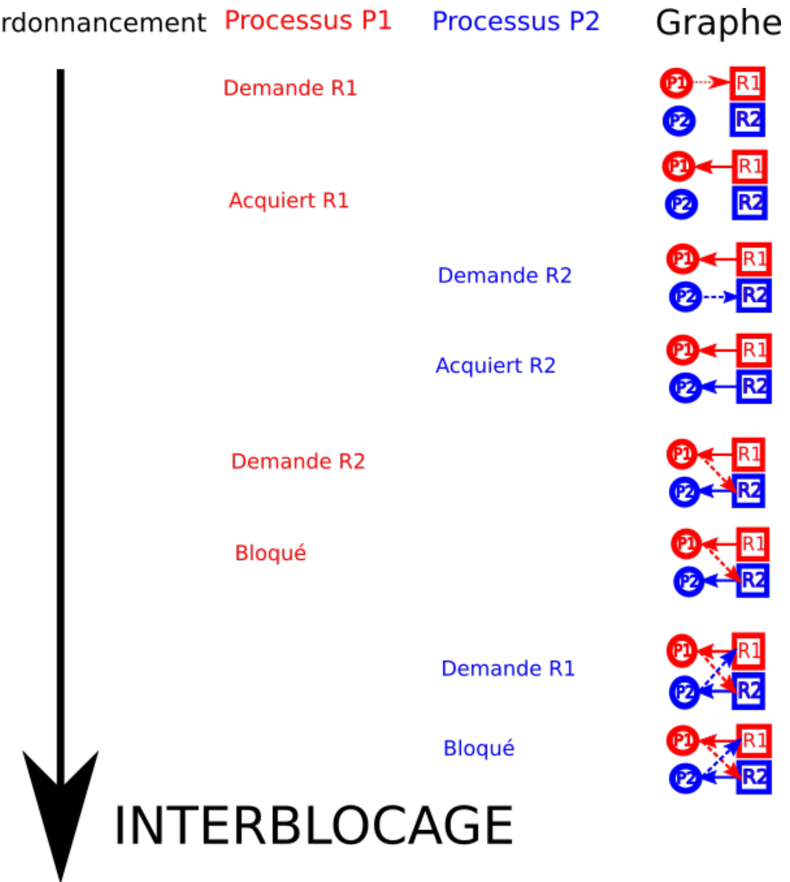
Exercice 11

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.



Définition 5 *Interblocage*

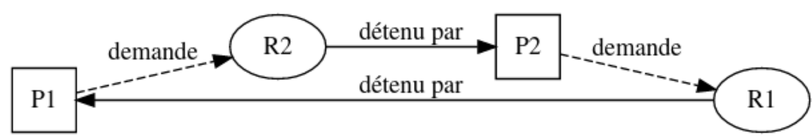
Lire l'explication sur l'interblocage dans le manuel entre les pages 246 et 249.



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, expliquer le problème qui peut être rencontré.

   ....
   ....
   ....

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

   ....
   ....
   ....

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.

   ....
   ....

2. Proposer un chronogramme d'ordonnancement sans interblocage.

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

Table des matières

1 Processus **1**
1.1 Différence entre programme et processus 1
1.2 Multiprogrammation et pseudo-parallélisme 5

2 Cycle de vie d'un processus et ordonnancement **10**
2.1 Cycle de vie d'un processus 10
2.2 Ordonnancement 12

3 Ressource en accès exclusif et interblocage **16**
3.1 Accès concurrent à une ressource partagée 16
3.2 Interblocage 19